IBM ILOG Dispatcher

Reference Manual

June 2009

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

About This Manual

This reference manual provides you with a complete description of the components of IBM® ILOG® Dispatcher.

Group Summary	
optim.dispatcher	The IBM® ILOG® Dispatcher API.

What is Dispatcher?

Dispatcher is an extension of the IBM® ILOG® Solver C++ constraint-programming library, especially adapted to problems in *vehicle routing* and *maintenance dispatching*.

This library is not a new programming language; it lets you use data structures and control structures provided by C++. Thus, the Dispatcher part of an application can be completely integrated with the rest of that application (for example, the graphic interface, connections to databases, etc.) because it can share the *same* objects.

What You Need to Know

This manual assumes that you are familiar with the operating system on which you are using Dispatcher. Since Dispatcher is written for C++ developers, this manual assumes that you can write C++ code and that you have a working knowledge of your C++ development environment.

IBM® ILOG® Dispatcher works with IBM ILOG Solver. This manual assumes that you have a working knowledge of IBM ILOG Solver and Solver concepts, such as domains, constraints, goals, handles, choice points, propagation, reversibility, and local search methods.

Notation

Throughout this manual, the following typographic conventions apply:

- Samples of code are written in this `typeface`.
- The names of constructors and member functions appear in this `typeface` in the section where they are documented.
- Important ideas are emphasized like *this, in italics*.

Naming Conventions

The names of types, classes, and functions defined in the Dispatcher library begin with `Ilo`. The names of classes are written as concatenated, capitalized words. For example:

`IloDispatcher` or `IloVisit`.

A lowercase letter begins the first word in names of arguments, instances, and member functions. Other words in such a name begin with an uppercase (that is, capital) letter. For example,

```
aVisit
IloVisit::getName()
```

There are no public data members in Dispatcher.

Accessors begin with the keyword `get` followed by the name of the data member. Accessors for Boolean members begin with `is` followed by the name of the data member. Modifiers begin with the keyword `set` followed by the name of the data member. Like other member functions, the first word in such a name begins with a lowercase letter, and any other words in the name begin with an uppercase (that is, capital) letter. The following example shows samples of accessors and modifiers in Dispatcher:

```
class IloVisit {
public:
    IloVisit(IloNode node, const char* name = 0);
    IloVisitVar getNextVar() const;
    IloNumVar getRankVar() const;
    IloBool isBreakable() const;
    void setPenaltyCost(IloNum val);
};
```

Include Files

In this reference manual, the documentation of a class uses the caption "Include File" to indicate which header file you need to include in your application. The caption "Definition File" indicates the header file where the class is actually defined.

Concepts

Construction Heuristics

To help you build a preliminary solution to a problem, Dispatcher provides predefined functions that return a goal. Each of these goals implements a *construction heuristic* to generate a preliminary solution to a problem. You can use these goals to generate a first solution to a routing problem. Once you have built a first solution, you can use neighborhoods and search heuristics to improve that solution.

See Also

IloDispatcherGenerate,IloInsertionGenerate,
IloNearestAdditionGenerate,IloNearestDepotGenerate,IloSavingsGenerate,IloSweepGenerate

Cost Function

Description

IBM® ILOG® Dispatcher has a built-in cost function which is composed of a fixed and a variable cost for each vehicle and of a cost for not performing visits (this cost is referred to as penalty cost).

This cost function is taken into account by the predefined first solution goals (`IloSavingsGenerate`, `IloInsertionGenerate`, etc.) and is the objective variable attached to routing solutions (instances of the `IloRoutingSolution` class); it will therefore be used by local search algorithms.

The fixed cost of a vehicle represents the cost of using that vehicle. (A vehicle is used if its route is not empty.) It is specified using

```
IloVehicle::setCost(IloNum value)
```

The variable cost of a vehicle is proportional to the total amount of dimension used by the vehicle and is specified using

```
IloVehicle::setCost(IloDimension dim, IloNum coef)
```

The penalty cost can be set via

```
IloVisit::setPenaltyCost(IloNum penaltyCost)
```

The cost variable of a vehicle can be obtained with the member function `IloVehicle::getCostVar()`, and that of the entire routing plan with `IloDispatcher::getTotalCost()` or `IloDispatcher::getCostVar()`.

Costs on Vehicles

Vehicle cost is bound as follows: the transit variables of all visits performed by the vehicle (including the first and last visits representing the start and end points of the vehicle), are added together to give the usage of the dimension for the vehicle. This usage is also reinforced by bounds computed from the *cumulative variable* at the last visit plus the transit variable at the last visit, minus the cumulative variable at the first visit as follows. Suppose `time` is an `IloDimension2`:

```
  IloVisit first = vehicle.getFirstVisit();
  IloVisit last = vehicle.getLastVisit();
  IloNumVar usage = last.getCumulVar(time)
                  + last.getTransitVar(time)
                  - first.getCumulVar(time);
```

The statement `vehicle.setCost(time, 40.0)` means that for each unit of time that the vehicle works, 40 units of cost are accrued. Cost for the vehicle is computed by multiplying the usage variable of the dimension (in this case, `time`) by the coefficient (40.0 in this case). If cost is specified in more than one dimension (for instance, `time` and `distance`), the cost for each dimension specified for the vehicle is computed in the same fashion.

The total cost of a vehicle is determined by summing the cost for each dimension specified for the vehicle and adding any fixed costs for that vehicle. If a fixed cost is specified, it is added to the cost of the vehicle if the vehicle is in use (that is, if the vehicle performs any visits other than its first and last visits).

Costs on Routing Plans

The total cost of a routing plan is determined by summing the total costs for all vehicles and adding any costs related to unperformed visits.

For each visit that is unperformed, an amount equal to the penalty cost `penCost` set on the visit via `visit.setPenaltyCost(penCost)` is added to the cost of the plan.

Instantiation of Plan Variables

In cases where the bounds on the cost function may not be an accurate reflection of true cost (for instance, when there are complex constraints on the transit or cumulative variables), the transit and/or cumulative variables of a problem can be fully instantiated to tighten the bounds to a value.

The `IloInstantiateTransits` goals are provided for this purpose. An `IloInstantiateTransits` goal can be added to the search, along with a goal to bind the first (or last) cumulative variable for each dimension and vehicle, to completely bind the transit and cumulative variables of the problem.

See Also

IloDimension, IloVehicle, IloVisit

Dimensions

Description

A given routing problem involves vehicles that travel routes to make visits to designated nodes. Those vehicles may have different capacities in terms of weight (for solid goods, for example) or volume (for liquids, say); those routes may entail different costs or distances traveled; those service visits may require different amounts of time (perhaps for waiting, unloading, reloading, etc.). In short, there may be many different *dimensions* (such as weights, volumes, costs, distances, times) in a given routing problem.

The class `IloDimension` makes it possible to model the various dimensions that occur in a problem. When you create an instance of `IloDimension` , you can associate a constrained variable with this dimension for each object (if needed). Constraints can subsequently be posted on those variables.

The class `IloDimension` has two subclasses to represent the *intrinsic* and *extrinsic* dimensions of an object. An intrinsic dimension depends only on the single object with which it is associated. An extrinsic dimension depends on two objects.

The subclass `IloDimension1` represents the dimensions that are intrinsic to an object. For example, weight is represented by `IloDimension1` because the weight of an object depends only on that object.

`IloDimension2` represents the dimensions that are extrinsic to an object; that is, those dimensions that depend on something outside the object itself. For example, time is usually represented as an instance of `IloDimension2` because the time to travel from one visit to another depends on both those visits.

By definition, `IloDimension2` is closely linked to the concept of *distance* . For that reason, one of the data members of `IloDimension2` is in fact an instance of `IloDistance`. Objects of `IloDistance` define how distances are computed between nodes (see the class `IloNode`) with respect to a dimension. Distance functions include `IloEuclidean`, which computes the Euclidean distance between nodes according to their coordinates, and `IloManhattan`, which computes the grid pattern distance between nodes. Dispatcher also allows you to define your own distance function.

Expressing Capacity

Vehicles usually have a *capacity*; that is, they cannot hold more than a certain weight or a given number of pallets or a particular volume. To express this idea, it is necessary to define the dimension in which the capacity will be expressed and to set its value. Likewise, the demand of the visits must also be defined. For example, the code expressing the capacity of a truck and the demand of a visit in terms of weight looks like this:

```
IloEnv env;
IloModel mdl(env);
IloDimension1 weight(env);
mdl.add(weight);
IloVehicle vehicle(env);
vehicle.setCapacity(weight,1500);
// vehicle capacity 1500 kg
mdl.add(vehicle);
IloNode node(env);
IloVisit visit(node);
mdl.add(visit.getTransitVar(weight) == 12);
// visit weighs 12 kilos
mdl.add(visit);
```

In this way, it is easy to express different capacities in different dimensions.

Expressing Cost

Like capacity, *cost* can vary in many different dimensions. To cite but a few, the cost of a routing problem may vary according to the number of vehicles (fixed cost for renting or owning them), the drivers (their salaries), the distance traveled, the time spent (by the vehicles or the drivers), the number of stops, and so forth.

The way of expressing cost is very similar to the way of expressing capacity:

```
IloEnv env;
IloModel mdl(env);
IloDimension2 distance(env, IloEuclidean);
mdl.add(distance);
IloVehicle vehicle(env);
vehicle.setCost(distance, 3.2);
// $ 3.2 per mile
mdl.add(vehicle);
```

For simplicity, we consider only costs that are linear with respect to the dimensions. It is possible to handle non-linear costs, but at a much higher implementation expense. This version of Dispatcher handles only linear costs.

Fixed costs can also be expressed with the member function `IloVehicle::setCost(fixedCost)`. When the vehicle is used, this *fixed cost* is added.

Expressing Service Time

The same technique applies to *service times* or *delays*. The following code shows how the service time at a customer's location can be defined:

```
IloEnv env;
IloModel mdl(env);
IloDimension1 volume(env);
mdl.add(volume);
```

```
IloDimension2 time(env, IloEuclidean);
mdl.add(time);
IloNode node(env);
IloVisit visit(node);
mdl.add(visit.getTransitVar(volume) == 10);
// 10 m3
mdl.add(visit.getDelayVar(time) ==
              .5 * visit.getTransitVar(volume));
// .5 minute / m3
mdl.add(visit);
```

The delay is equal to the absolute value of the calculation unit times volume.

See Also

IloDimension, IloDimension1, IloDimension2, IloDistance

Extraction

Description

Dispatcher extends the extraction capabilities of Solver. For more information on the extraction mechanism, see the *IBM ILOG Solver User's Manual*.

The extraction of certain Dispatcher objects triggers the extraction of other objects. This section describes these dependencies. Note that an extractable directly added to a model will always be extracted. Since `IloSolution` objects will not store objects that have not been extracted, it is important to explicitly add any object to the model that you want to be extracted and that is not included in the following dependency relationships.

IloVisit

If an instance of `IloVisit` is added to the model, its extraction will trigger the extraction of the following objects related to that instance of `IloVisit`:

- start and end nodes (instances of `IloNode`)
- rank variable (an instance of `IloNumVar`)
- next visit variable (an instance of `IloVisitVar`)
- previous visit variable (an instance of `IloVisitVar`)
- vehicle variable (an instance of `IloVehicleVar`)
- dimension variables corresponding to dimensions that have been extracted (instances of `IloNumVar`). These dimension variables are cumul and transit variables for all dimensions and delay, wait, and travel variables for instances of `IloDimension2`.

IloVehicle

If an instance of `IloVehicle` is added to the model, its extraction will trigger the extraction of its first and last visits. Speed, capacity and cost related to dimensions will only be considered for dimensions that have been extracted.

IloDimension1 and IloDimension2

If an instance of `IloDimension1` or `IloDimension2` is added to the model, its extraction will trigger the extraction of dimension variables corresponding to visits that have been extracted. It will also update the corresponding speed, capacity, and cost of extracted vehicles.

Note

Instances of `IloDimension1` and `IloDimension2` are not automatically extracted when an instance of `IloVisit` is added to the model. In fact, the extraction of an instance of `IloVehicleBreakCon` is the only event that can trigger the extraction of an instance of `IloDimension2` and an instance of `IloDimension1` can only be extracted if added explicitly to a model. Therefore, it is recommended to always explicitly add dimensions of both types to a model.

Note that instances of `IloDimension1` and `IloDimension2` are not automatically extracted when an instance of `IloVisit` is added to the model. In fact, the extraction of an instance of `IloVehicleBreakCon` is the only event that can trigger the extraction of an instance of `IloDimension2` and an instance of `IloDimension1` can only be extracted if added explicitly to a model. Therefore, it is recommended to always explicitly add dimensions of both types to a model.

Cost Variable

If any Dispatcher extractable is present in the model, the cost variable representing the sum of the cost of the vehicles and of unperformed visits is extracted.

Functions

The extraction of a variable created from the `()` operators of `IloVisitToNumFunction` triggers the extraction of the visits included in the `IloVisitArray` passed to the constructor of the function.

The extraction of a variable created from the `()` operator of `IloVehicleToNumFunction` triggers the extraction of the vehicles included in the `IloVehicleArray` passed to the constructor of the function.

Constraints

The extraction of constraints involving visits will extract all the variables and visits involved. For example, the extraction of equalities between next or previous visit variables and visits or visit arrays will extract all the variables and visits involved. This behavior is the same for the following constraints:

- constraints on vehicles and vehicle variables
- constraints on vehicle break constraints (instances of `IloVehicleBreakCon`)
- constraints on visits (`IloVehicleBreakCon::justAfter()` constraints).

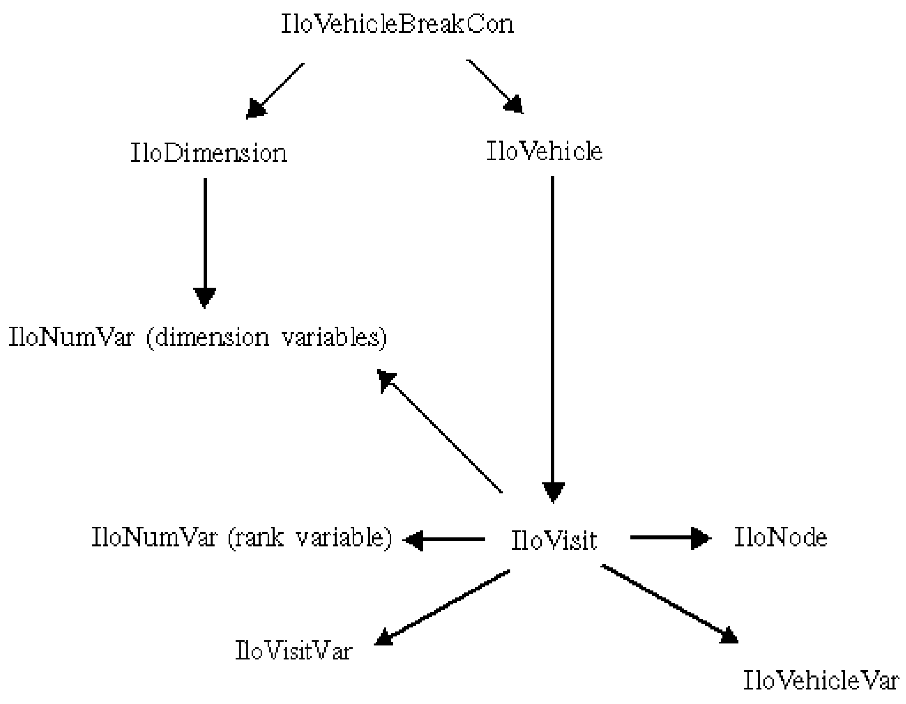
The extraction of vehicle break constraints triggers the extraction of the corresponding start and duration variables (instances of `IloNumVar`), as well as the extraction of the related vehicle and dimension.

Variables

The extraction of dimension, next visit, previous visit, vehicle, vehicle break start or vehicle break duration variables does not trigger the extraction of any other extractable object.

Interdependency Chart

The following figure summarizes the interdependencies that exist among the main extractables in Dispatcher. The arrows denote the direction of the extraction dependency relationship. For example, if you add an instance of `IloVehicle` to a model, this will automatically extract the first and last visits to be performed by that vehicle (instances of `IloVisit`). The extraction of the instance of `IloVisit` will trigger the extractions of the visit's rank, previous visit, next visit, vehicle and node variables. The extraction of the instance of `IloVisit` will also trigger the extraction of all dimension variables for dimensions already explicitly extracted in the model.



See Also

IloNode, IloDimension, IloVisit, IloVehicle, IloVehicleBreakCon, IloVisitVar, IloVehicleVar, IloNumVar documented in the *IBM ILOG Concert Technology Reference Manual*.

Iterators

An iterator is an object that traverses an underlying data structure of other objects. The iterator contains a traversal state of this data structure. Besides its constructors and destructors, an iterator has member functions to access the element at the current position, to check whether the iterator has passed beyond the end position, and to shift the iterator to the next position.

In order to help you implement selection algorithms in a search procedure or to display data, Dispatcher offers iterators as a way to find all the objects needed by such a calculation.

Moreover, both the container to scan and the data to access are Dispatcher handles to implementation classes. For example, an instance of the nested class `IloRoutingSolution::RouteIterator` scans all the visits served by an instance of `IloVehicle`. Described in these terms, an iterator is very similar to a C string; like an array (the container) of characters (the handles to data) ending with the null pointer (the handle to the null implementation object).

Here's a typical skeleton for iterators in Dispatcher.

```
class IloHandleDataIterator {
public:
  IloHandleDataIterator(const IloHandleContainer);
  ~IloHandleDataIterator();
  IloBool ok();
  IloHandleData operator*();
  IloHandleDataIterator& operator++();
};
```

The member function `ok()` returns `IloTrue` if the current position of the iterator is valid. It returns `IloFalse` if the container has been entirely scanned.

The dereference `operator*` accesses the handle at the current position of the iterator.

The left increment `operator++` shifts the current position of the iterator.

Example

As an example, we could display the routes served by a list of vehicles with the following code.

```
  void show(IloDispatcher dispatcher) {
    IloEnv env = dispatcher.getEnv();
    IloSolver solver = dispatcher.getSolver();
    for(IloIterator<IloVehicle> wi(env); wi.ok(); ++wi) {
      IloVehicle vehicle = *wi;
      if(dispatcher.getRouteSize(vehicle) != 0) {
        for(IloDispatcher::RouteIterator ri(dispatcher,vehicle); ri.ok();
++ri) {
          IloVisit visit = *ri;
          env.out() << " -> " << visit.getName() << " ("
                    << visit.getId() << ") ";
          for(IloIterator<IloDimension> di(env); di.ok(); ++di) {
            env.out() << solver.getFloatVar(visit.getCumulVar(*di));
          }
        }
        env.out() << endl;
      }
    }
    env.out() << "Cost : " << dispatcher.getTotalCost() << endl;
    env.out() << "Vehicles used : "
              << dispatcher.getNumberOfVehiclesUsed() << endl;
  }
```

See Also

IloDimensionIterator, IloDimension1Iterator, IloDimension2Iterator, IloVehicleBreakConIterator, IloVehicleIterator, IloVisitIterator, IloDispatcher::RouteIterator, IloDispatcher::UnperformedVisitIterator, IloNode::Iterator, IloRoutingSolution::RouteIterator, IloRoutingSolution::UnperformedVisitIterator, IloRoutingSolution::VehicleIterator, IloRoutingSolution::VisitIterator

Neighborhoods

Dispatcher offers several predefined *neighborhoods* (that is, subclasses of `IloNHoodI`): `IloCross`, `IloExchange`, `IloMakePerformed`, `IloMakePerformedPair`, `IloMakeUnperformed`, `IloOrOpt`, `IloRelocate`, `IloFPRelocate`, `IloSwapPerform`, `IloTwoOpt`. These neighborhoods can be used to modify or improve routing solutions using local search goals provided by Dispatcher. All neighborhoods have state and you should call `reset()` on them when the neighborhood is to be reused in a new local search. (See the entry for `IloNHood::reset` in the *IBM ILOG Solver Reference Manual.*)

Please refer to the *IBM ILOG Solver User's Manual* for more information on neighborhoods.

See Also

IloCross, IloExchange, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloRelocate, IloFPRelocate, IloSwapPerform, IloTwoOpt

Group optim.dispatcher

The IBM® ILOG® Dispatcher API.

Class Summary
IloArrayVehicleToNumFunctionI
IloArrayVisitToNumFunctionI
IloComposedDistance
IloComposedVisitDistance
IloDefaultDecisionTracerI
IloDefaultFSDecisionMakerI
IloDefaultVisitVehicleFSDecisionI
IloDelaySumVar
IloDimension
IloDimension1
IloDimension1Iterator
IloDimension2
IloDimension2Iterator
IloDimensionIterator
IloDimensionWindows
IloDimensionWindows::ForbiddenIterator
IloDimensionWindows::Iterator
IloDispatcher
IloDispatcherFSParameters
IloDispatcherGLS
IloDispatcherGoalFactory
IloDispatcherGoalFactoryI
IloDispatcherGraph
IloDispatcherGraph::AdjacencyListIterator
IloDispatcherGraph::Arc
IloDispatcherGraph::Node
IloDispatcherGraph::PathIterator
IloDispatcherNHoodParameters
IloDispatcher::RouteIterator
IloDispatcherTabuSearch
IloDispatcher::UnperformedVisitIterator
IloDispatcher::VehicleBreakConIterator
IloDistance
IloDistanceEvalI
IloDistanceI
IloEvalVehicleToNumFunctionI
IloEvalVisitToNumFunctionI

IloEverywhereNode
IloExecutionWindowsToVisitCon
IloExplicitArcPredicate
IloExplicitDistance
IloExplicitVisitDistance
IloFSDecisionI
IloFSDecisionMakerI
IloFSDecisionTracerI
IloNADecisionI
IloNADecisionMakerI
IloNode
IloNode::Iterator
IloOutOfRouteConstraint
IloOutputManip
IloPairDecisionI
IloProductDimension
IloRoutingSolution
IloRoutingSolution::RouteIterator
IloRoutingSolution::UnperformedVisitIterator
IloRoutingSolution::VehicleIterator
IloRoutingSolution::VisitIterator
IloSimpleDistanceEvalI
IloSimpleVisitDistanceEvalI
IloSingleVehicleFSDecisionI
IloSparseExplicitDistance
IloSparseExplicitVisitDistance
IloTravelSumVar
IloVehicle
IloVehicleBreakCon
IloVehicleBreakConIterator
IloVehicleEquiv
IloVehicleEquivEvalI
IloVehicleEquivI
IloVehicleIterator
IloVehicleLIFOConstraint
IloVehiclePair
IloVehicleToNumFunction
IloVehicleToNumFunctionI
IloVehicleVar
IloVisit
IloVisitAlternativeConstraint

IloVisitDistance
IloVisitDistanceEvalI
IloVisitDistanceI
IloVisitIterator
IloVisitPair
IloVisitToNumFunction
IloVisitToNumFunctionI
IloVisitVar
IloVisitVehicleCompat
IloVisitVehicleCompatI
IloVisitVehiclePredicateCompatI

Typedef Summary
IloArcPredicate
IloDistanceFunction
IloSimpleDistanceFunction
IloSimpleVehicleToNumFunction
IloSimpleVisitDistanceFunction
IloSimpleVisitToNumFunction
IloVehicleArray
IloVehicleEquivFunction
IloVehiclePairPredicate
IloVisitArray
IloVisitDistanceFunction
IloVisitVehicleCompatPredicate

Enumeration Summary
IloFSDecisionRejectCause
IloNearestAdditionBehavior
IloNearestAdditionExtension
IloOutOfRouteReference

Function Summary
IloAffineFunction
IloAllUnperformedGenerate
IloAllVehiclesDifferent
IloAllVehiclesEquivalent
IloBoxVehiclePairPredicate
IloCompatible
IloCouple
IloCross
IloDecouple

IloDispatcherGenerate
IloDistanceThresholdArcPredicate
IloDistanceThresholdArcPredicate
IloDistMax
IloEarlinessFunction
IloEuclidean
IloExchange
IloFinalizePlan
IloFPRelocate
IloFunctionDistance
IloGenerateRoute
IloGeographical
IloGetDispatcherDefaultVehicleEquivalence
IloGraphDistance
IloInsertionGenerate
IloInsertVisit
IloInsertVisit
IloInsertVisit
IloInsertVisit
IloInstantiateTransits
IloInstantiateTransits
IloInstantiateTransits
IloInstantiateTransits
IloInstantiateVehicleBreak
IloInstantiateVehicleBreakDuration
IloInstantiateVehicleBreakPosition
IloInstantiateVehicleBreaks
IloInstantiateVehicleBreakStart
IloIntraRelocate
IloMakePerformed
IloMakePerformedPair
IloMakeUnperformed
IloManhattan
IloMax
IloMergeAndRelocateTours
IloMin
IloNearestAdditionGenerate
IloNearestDepotGenerate
IloOrderedVisitPair
IloOrOpt
IloRejectNeighbor

IloRelocate
IloSameNodeArcPredicate
IloSavingsGenerate
IloSetVehicleVisitCumuls
IloSetVisitCumuls
IloSolutionValueComparator
IloSortedNHood
IloSortedNHood
IloSwapPerform
IloSweepGenerate
IloTardinessFunction
IloTerse
IloTwoOpt
IloVehicleDependentDelayConstraint
IloVerbose
IloVisitAlternativeSwap
operator!=
operator!=
operator!=
operator!=
operator!=
operator*
operator*
operator*
operator*
operator+
operator+
operator+
operator<<
operator<<
operator<<
operator==
operator==
operator==
operator==
operator==

Variable Summary
IloEarthRadiusInKm
IloEarthRadiusInMiles

The IBM® ILOG® Dispatcher API.

Class IloDispatcherGraph::AdjacencyListIterator

Definition file: ildispat/ilographdist.h
Include file: <ildispat/ilodispatcher.h>

`AdjacencyListIterator` is a class nested in the class `IloDispatcherGraph`. This class is used to access the arcs in the adjacency list of (i.e. all directed arcs emanating from) an object of type `IloDispatcherGraph::Node`.

See Also: IloDispatcherGraph, IloDispatcherGraph::Node, IloDispatcherGraph::Arc, IloDispatcherGraph::AdjacencyListIterator, IloGraphDistance

Constructor Summary	
`public`	`AdjacencyListIterator(IloDispatcherGraph g, IloDispatcherGraph::Node n)`

Method Summary	
`public IloBool`	`ok() const`
`public IloDispatcherGraph::Arc`	`operator*() const`
`public AdjacencyListIterator &`	`operator++()`

Constructors

`public` **`AdjacencyListIterator`**`(IloDispatcherGraph g, IloDispatcherGraph::Node n)`

This constructor creates an iterator to access all instances of `IloDispatcherGraph::Arc` emanating from `n`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the adjacent arcs have been scanned by the iterator.

`public IloDispatcherGraph::Arc` **`operator*`**`() const`

This operator returns the current instance of `IloDispatcherGraph::Arc`, the one to which the invoking iterator points.

`public AdjacencyListIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloDispatcherGraph::Arc` in the adjacency list.

Class IloDispatcherGraph::Arc

Definition file: ildispat/ilographdist.h
Include file: <ildispat/ilodispatcher.h>

`Arc` is a class nested in the class `IloDispatcherGraph`. Objects of this class represent directed arcs in an `IloDispatcherGraph` object. Each arc is uniquely associated to an identifier of type `IloInt`, and is necessarily associated to two end nodes of type `IloDispatcherGraph::Node`. Only one arc may join two nodes in the same sense. When an object of class `Arc` is created (either by using one of the provided constructors or by using the member function `IloDispatcherGraph::createArcsFromFile`), all turns out of the arc into subsequent arcs are allowed at no penalty. Similarly, all turns into the arc from a preceding arc are allowed at no penalty. The cost of traversing an arc may be expressed in terms of multiple dimensions. These costs are manipulated using the member function `IloDispatcherGraph::setArcCost`.

See Also: IloDispatcherGraph, IloDispatcherGraph::PathIterator, IloDispatcherGraph::Node, IloDispatcherGraph::AdjacencyListIterator, IloGraphDistance

Constructor and Destructor Summary	
`public`	`Arc(IloDispatcherGraph g, const IloInt id, IloDispatcherGraph::Node fromNode, const IloDispatcherGraph::Node toNode)`
`public`	`Arc(const Arc & arc)`
`public`	`Arc(IloDispatcherGraphI::ArcI * impl=0)`

Method Summary	
`public IloDispatcherGraph::Node`	`getFromNode() const`
`public IloDispatcherGraphI::ArcI *`	`getImpl() const`
`public IloInt`	`getIndex() const`
`public IloDispatcherGraph::Node`	`getToNode() const`

Constructors and Destructors

`public` **`Arc`**`(IloDispatcherGraph g, const IloInt id, IloDispatcherGraph::Node fromNode, const IloDispatcherGraph::Node toNode)`

This constructor creates an arc in the graph `g`, with identifier `id`, and joining `fromNode` and `toNode`.

`public` **`Arc`**`(const Arc & arc)`

This copy constructor creates a handle from a reference to an `Arc` object. That new `Arc` object and `arc` both point to the same implementation object.

`public` **`Arc`**`(IloDispatcherGraphI::ArcI * impl=0)`

This constructor creates a handle object (an instance of `IloDispatcherGraph::Arc`) from a pointer to an implementation object (an instance of the class `IloDispatcherGraphI::ArcI`).

Methods

`public IloDispatcherGraph::Node` **`getFromNode`**`() const`

This member function returns the arc's origin node.

`public IloDispatcherGraphI::ArcI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking `Arc` object.

`public IloInt` **`getIndex`**`() const`

This member function returns the unique identifier associated with the `Arc` object.

`public IloDispatcherGraph::Node` **`getToNode`**`() const`

This member function returns the arc's destination node.

Class IloDimensionWindows::ForbiddenIterator

Definition file: ildispat/iloproto.h
Include file: <ildispat/ilodispatcher.h>

`ForbiddenIterator` is a class nested in the class `IloDimensionWindows`. It allows you to step through the forbidden intervals of a dimension windows constraint, in increasing interval lower bound order.

See Also: IloDimensionWindows, IloDimensionWindows::Iterator

Constructor Summary	
`public`	`ForbiddenIterator(IloDimensionWindows win)`

Method Summary	
`public IloNum`	`getLB() const`
`public IloNum`	`getUB() const`
`public IloBool`	`ok() const`
`public ForbiddenIterator &`	`operator++()`

Constructors

`public` **`ForbiddenIterator`**`(IloDimensionWindows win)`

This constructor creates an iterator to traverse the forbidden intervals contained in `win`.

Methods

`public IloNum` **`getLB`**`() const`

This member function returns the lower bound of the forbidden interval to which the invoking iterator points.

`public IloNum` **`getUB`**`() const`

This member function returns the upper bound of the forbidden interval to which the invoking iterator points.

`public IloBool` **`ok`**`() const`

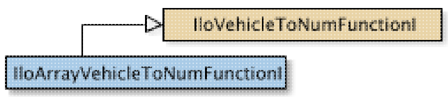
This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the forbidden intervals have been scanned.

`public ForbiddenIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next forbidden interval (the first one starting after the current forbidden interval).

Class IloArrayVehicleToNumFunctionI

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>



This class is an implementation class, a predefined subclass of `IloVehicleToNumFunctionI`, that you use to define a new vehicle to `IloNum` function expressed using two arrays, an array of vehicles and an array of corresponding values.

Constructor and Destructor Summary	
public	`IloArrayVehicleToNumFunctionI(IloEnv env, IloVehicleArray vehicles, IloNumArray values, IloNum unperfValue, IloNum defaultValue)`
public	`IloArrayVehicleToNumFunctionI(IloEnv env, IloVehicleArray vehicles, IloNumArray values, IloNum unperfValue)`

Method Summary	
public void	`addVehicleValue(IloVehicle vehicle, IloNum value)`
public IloNum	`getDefaultValue() const`
public virtual IloNum	`getUnperformedValue()`
public virtual IloNum	`getValue(IloVehicle vehicle)`
public void	`setUnperformedValue(IloNum unperformedValue)`

Inherited Methods from `IloVehicleToNumFunctionI`
`getUnperformedValue, getValue`

Constructors and Destructors

`public` **`IloArrayVehicleToNumFunctionI`**`(IloEnv env, IloVehicleArray vehicles, IloNumArray values, IloNum unperfValue, IloNum defaultValue)`

This constructor creates a new vehicle to `IloNum` function from an array of vehicles `vehicles` and an array of values `values`. For a given index `i`, `values[i]` is the value of the function for vehicle `vehicles[i]`. Duplicate vehicles in `vehicles` are forbidden. Both arrays should have the same size. The value `unperfValue` is the value returned by the function for the visit unperformed state. The value `defaultValue` is the value taken by the function for a vehicle for which no value has been specified.

`public` **`IloArrayVehicleToNumFunctionI`**`(IloEnv env, IloVehicleArray vehicles, IloNumArray values, IloNum unperfValue)`

This constructor creates a new vehicle to `IloNum` function from an array of vehicles `vehicles` and an array of values `values`. For a given index `i`, `values[i]` is the value of the function for vehicle `vehicles[i]`. Duplicate vehicles in `vehicles` are forbidden. Both arrays should have the same size. The value `unperfValue` is the value returned by the function for the visit unperformed state.

Methods

`public void **addVehicleValue**(IloVehicle vehicle, IloNum value)`

This member function adds the value `value` for `vehicle` to the function. If `vehicle` already has a value specified in the function, it will be overwritten.

`public IloNum **getDefaultValue**() const`

This member function returns the default value of the function. This is the value returned by the function for vehicles for which no value has been specified .

`public virtual IloNum **getUnperformedValue**()`

This member function is redefined to return the value, corresponding to the unperformed state, passed in the constructor.

`public virtual IloNum **getValue**(IloVehicle vehicle)`

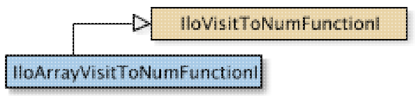
This member function returns a numeric value corresponding to `vehicle`. If `vehicles` is the array of vehicles and `values` the array of values passed to the constructor and if `i` is the index for which `vehicles[i] = vehicle`, then this member function will return `values[i]`.

`public void **setUnperformedValue**(IloNum unperformedValue)`

This member function modifies the value returned by the function for the visit unperformed state.

Class IloArrayVisitToNumFunctionI

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>



This class is an implementation class, a predefined subclass of `IloVisitToNumFunctionI`, that you use to define a new visit to `IloNum` function expressed using two arrays, an array of visits and an array of corresponding values.

Constructor and Destructor Summary	
`public`	`IloArrayVisitToNumFunctionI(IloEnv env, IloVisitArray visits, IloNumArray values, IloNum unperfValue)`

Method Summary	
`public virtual IloNum`	`getUnperformedValue()`
`public virtual IloNum`	`getValue(IloVisit visit)`

Inherited Methods from `IloVisitToNumFunctionI`
`getUnperformedValue, getValue`

Constructors and Destructors

`public` **`IloArrayVisitToNumFunctionI`**`(IloEnv env, IloVisitArray visits, IloNumArray values, IloNum unperfValue)`

This constructor creates a new visit to `IloNum` function from an array of visits `visits` and an array of values `values`. For a given index `i`, `values[i]` is the value of the function for visit `visits[i]`. Duplicate visits in `visits` are forbidden. Both arrays should have the same size. The value `unperfValue` is the value taken by the function for the visit unperformed state.

Methods

`public virtual IloNum` **`getUnperformedValue`**`()`

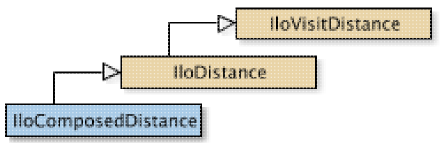
This member function is redefined to return the value, corresponding to the unperformed state, passed in the constructor.

`public virtual IloNum` **`getValue`**`(IloVisit visit)`

This member function returns a numeric value corresponding to `visit`. If `visits` is the array of visits and `values` the array of values passed to the constructor and if `i` is the index for which `visits[i] = visit`, then this member function will return `values[i]`.

Class IloComposedDistance

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>



This distance class allows the user to combine different instances of `IloDistance` by assigning them to a specific vehicle using the member function IloComposedDistance::setDistance. If no instance of `IloDistance` has been assigned to a given vehicle, all distance values returned for that vehicle are `IloInfinity`.

See Also: IloDistance, IloVehicle, IloExplicitDistance, IloSparseExplicitDistance

Constructor and Destructor Summary	
`public`	`IloComposedDistance(IloEnv env)`
`public`	`IloComposedDistance(IloEnv env, IloNum defaultValue)`
`public`	`IloComposedDistance(IloEnv env, IloDistance defaultDistance)`

Method Summary	
`public IloDistance`	`getDefaultDistance() const`
`public IloNum`	`getDefaultValue() const`
`public void`	`setDefaultDistance(IloDistance distance)`
`public void`	`setDefaultValue(IloNum defaultValue)`
`public void`	`setDistance(IloVehicle vehicle, IloDistance distance)`

Inherited Methods from **`IloDistance`**
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setCache, setKey, unsetCache`

Inherited Methods from **`IloVisitDistance`**
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setKey`

Constructors and Destructors

`public` **`IloComposedDistance`**`(IloEnv env)`

This constructor creates a composed distance object in the environment `env`.

`public` **`IloComposedDistance`**`(IloEnv env, IloNum defaultValue)`

This constructor creates a composed distance object in the environment `env` with the value `defaultValue`. If no instance of `IloDistance` has been assigned to a given vehicle, all distance values returned for that vehicle will be `defaultValue`.

`public **IloComposedDistance**(IloEnv env, IloDistance defaultDistance)`

This constructor creates a composed distance object in the environment `env` using the distance function `defaultDistance`. If no instance of `IloDistance` has been assigned to a given vehicle, all distance values computed for that vehicle will be computed using `defaultDistance`.

Methods

`public IloDistance **getDefaultDistance**() const`

This member function returns the instance of `IloDistance` used as a default distance in the composed distance object.

`public IloNum **getDefaultValue**() const`

This member function returns the default value associated with the composed distance object.

`public void **setDefaultDistance**(IloDistance distance)`

This member function assigns the default distance function `distance`. If no instance of `IloDistance` has been assigned to a given vehicle, all distance values computed for that vehicle will be computed using `defaultDistance`.

`public void **setDefaultValue**(IloNum defaultValue)`

This member function assigns the default distance value `defaultValue`. If no instance of `IloDistance` has been assigned to a given vehicle, all distance values returned for that vehicle will be `defaultValue`.

`public void **setDistance**(IloVehicle vehicle, IloDistance distance)`

This member function assigns the distance `distance` to vehicle `vehicle`.

Class IloComposedVisitDistance

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>

IloVisitDistance
IloComposedVisitDistance

This distance class allows the user to combine different instances of `IloVisitDistance` by assigning them to a specific vehicle using the member function IloComposedVisitDistance::setDistance. If no instance of `IloVisitDistance` has been assigned to a given vehicle, a default value or a default distance is returned for that vehicle.

See Also: IloVisitDistance, IloSparseExplicitVisitDistance, IloExplicitVisitDistance, IloVehicle

Constructor and Destructor Summary	
`public`	`IloComposedVisitDistance(IloEnv env)`
`public`	`IloComposedVisitDistance(IloEnv env, IloNum defaultValue)`
`public`	`IloComposedVisitDistance(IloEnv env, IloVisitDistance defaultDistance)`

Method Summary	
`public IloVisitDistance`	`getDefaultDistance() const`
`public IloNum`	`getDefaultValue() const`
`public void`	`setDefaultDistance(IloVisitDistance distance)`
`public void`	`setDefaultValue(IloNum defaultValue)`
`public void`	`setDistance(IloVehicle vehicle, IloVisitDistance distance)`

Inherited Methods from `IloVisitDistance`
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setKey`

Constructors and Destructors

`public` **`IloComposedVisitDistance`**`(IloEnv env)`

This constructor creates a composed distance object in the environment `env`.

`public` **`IloComposedVisitDistance`**`(IloEnv env, IloNum defaultValue)`

This constructor creates a composed distance object in the environment `env` with the value `defaultValue`. If no instance of `IloDistance` has been assigned to a given vehicle, all distance values returned for that vehicle will be `defaultValue`.

`public` **`IloComposedVisitDistance`**`(IloEnv env, IloVisitDistance defaultDistance)`

This constructor creates a composed distance object in the environment `env` using the distance function `defaultDistance`. If no instance of `IloVisitDistance` has been assigned to a given vehicle, all distance values computed for that vehicle will be computed using `defaultDistance`.

Methods

`public IloVisitDistance` **`getDefaultDistance`**`() const`

This member function returns the instance of `IloVisitDistance` used as a default distance in the composed distance object.

`public IloNum` **`getDefaultValue`**`() const`

This member function returns the default value associated with the composed distance object.

`public void` **`setDefaultDistance`**`(IloVisitDistance distance)`

This member function assigns the default distance function `distance`. If no instance of `IloVisitDistance` has been assigned to a given vehicle, all distance values computed for that vehicle will be computed using `distance`.

`public void` **`setDefaultValue`**`(IloNum defaultValue)`

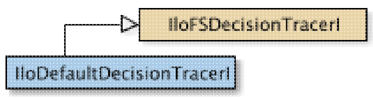
This member function assigns the default distance value `defaultValue`. If no instance of `IloVisitDistance` has been assigned to a given vehicle, all distance values returned for that vehicle will be `defaultValue`.

`public void` **`setDistance`**`(IloVehicle vehicle, IloVisitDistance distance)`

This member function assigns the distance `distance` to vehicle `vehicle`.

Class IloDefaultDecisionTracerI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>



This class is a concrete subclass of the abstract `IloDefaultFSDecisionTracerI` class. It provides empty implementations for all virtual member functions of the abstract tracer. This class is not meant to be used directly, but as a convenient shorthand, to be subclassed by defining only the appropriate trace member functions.

Constructor Summary	
`public`	`IloDefaultDecisionTracerI(IloDispatcher dsp)`

Method Summary	
`public virtual void`	`beginDecisionCommit(const IloFSDecisionI *)`
`public virtual void`	`beginDecisionTest(const IloFSDecisionI *)`
`public virtual void`	`beginExecute(const IloFSDecisionMakerI *)`
`public virtual void`	`endDecisionCommit(const IloFSDecisionI *)`
`public virtual void`	`endDecisionTest(const IloFSDecisionI *)`
`public virtual void`	`endExecute(const IloFSDecisionMakerI *)`
`public virtual void`	`notifyChosen(const IloFSDecisionI *)`
`public virtual void`	`notifyInfeasible(const IloFSDecisionI *)`
`public virtual void`	`notifyRejected(const IloFSDecisionI *, IloFSDecisionRejectCause)`
`public virtual void`	`notifyValidated(const IloFSDecisionI *)`
`public virtual void`	`registerDecision(const IloFSDecisionI *)`

Inherited Methods from `IloFSDecisionTracerI`
`beginDecisionCommit, beginDecisionTest, beginExecute, endDecisionCommit, endDecisionTest, endExecute, getDispatcher, notifyChosen, notifyInfeasible, notifyRejected, notifyValidated, registerDecision`

Constructors

`public` **`IloDefaultDecisionTracerI`**`(IloDispatcher dsp)`

This is the constructor for the `IloDefaultDecisionTracerI` class. This constructor builds an instance of a tracer that does nothing, and, as such, should not be used directly, but in the constructor of a derived class. This class can be useful in defining custom tracer objects, for which only a small subset of methods should actually do something. Deriving from this default tracer allows you to define only those methods that do something, without needing to define others with an empty behavior.

Methods

`public virtual void` **`beginDecisionCommit`**`(const IloFSDecisionI *)`

This member function is called before calling the `commit` member function of the decision maker with the decision as argument. The decision has been tested as a legal one.

`public virtual void` **`beginDecisionTest`**`(const IloFSDecisionI *)`

This virtual member function is called before a decision is tested for legality using the `make` member function of the decision.

`public virtual void` **`beginExecute`**`(const IloFSDecisionMakerI *)`

This member function is called at the beginning of the execution of a decision maker, before any decision has been created and registered.

`public virtual void` **`endDecisionCommit`**`(const IloFSDecisionI *)`

This member function is called after calling the `commit` member function of the decision maker on the decision.

`public virtual void` **`endDecisionTest`**`(const IloFSDecisionI *)`

This virtual member function is called after executing the `make` member function of the decision within the testing of the decision. If the making of the decision fails, this member function may not be called.

`public virtual void` **`endExecute`**`(const IloFSDecisionMakerI *)`

This member function is called at the end of the execution of a decision maker, after all legal visits have been considered.

`public virtual void` **`notifyChosen`**`(const IloFSDecisionI *)`

This member function is called when a decision has been selected as the best legal decision that can be performed. This method is called before the decision maker attempts to execute and commit the decision.

`public virtual void` **`notifyInfeasible`**`(const IloFSDecisionI *)`

This member function is called whenever a decision has been statically computed as not feasible, before it has been tested. This can happen for nearest addition decisions, when the current route is already too long to accept the candidate visit.

`public virtual void` **`notifyRejected`**`(const IloFSDecisionI *,`
`IloFSDecisionRejectCause)`

This virtual member function is called when the decision has been tested using the `isLegal` member function of the decision maker, and has been rejected. The rejection can be caused by any of three scenarios:

- the routing assignment, created by the decision's `make`, fails
- the route completion goal, used to check that the decision is consistent with the closing of the route, fails
- the justifier goal of the visit (typically a time-placement goal that tries to find a justifying set of starting times and dates and breaks, if any, along the route), fails

The cause of the rejection is identified by the `cause` enumerated value, which is passed to the method.

```
public virtual void notifyValidated(const IloFSDecisionI *)
```

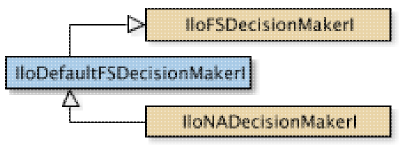
This member function is called when a decision has been accepted by the `isLegal` member function of the decision maker. The best decision will be selected from among the validated decisions.

```
public virtual void registerDecision(const IloFSDecisionI *)
```

This virtual member function is called when a decision is registered. A decision has to be registered to be taken into account by the first solution framework.

Class IloDefaultFSDecisionMakerI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>



This class is a subclass of the abstract decision maker class, specialized for decisions that place one visit at a time. This class defines a specialized behavior of the `init()` member function, which iterates on all visits to create the decisions stored in the decision maker.

Constructor Summary	
`protected`	`IloDefaultFSDecisionMakerI(IloDispatcher dispatcher, IloDispatcherFSParameters param)`

Method Summary	
`public virtual IloFSDecisionI *`	`createDecision(IloVisit visit, IloVehicle vehicle)`
`public virtual void`	`init()`
`public void`	`registerVisitVehicleDecision(IloVisit visit, IloVehicle vehicle)`

Inherited Methods from `IloFSDecisionMakerI`
`commit, getBestDecision, getDispatcher, getEnv, getTracer, init, isLegal, registerGlobalDecision, registerVehicleDecision, registerVisitDecision, setTracer, storeDecision`

Constructors

`protected` **`IloDefaultFSDecisionMakerI`**`(IloDispatcher dispatcher, IloDispatcherFSParameters param)`

This constructor creates an `IloDefaultFSDecisionMakerI` from an `IloDispatcher`. As this class is an abstract class, this constructor is defined as protected.

Methods

`public virtual IloFSDecisionI *` **`createDecision`**`(IloVisit visit, IloVehicle vehicle)`

This virtual member function is a virtual defined by this class. It is responsible for the creation of an instance of `IloDefaultVisitVehicleFSDecisionI*`, which models the assignment of visit to vehicle. The precise semantics of this decision will be defined by the concrete decision class that is actually built. The decision object must be allocated on an `IloEnv` memory.

`public virtual void` **`init`**`()`

This member function implements the virtual `init` member function of the `IloFSDecisionMakerI` class. This method iterates on all visits that are extracted in the dispatcher and, for each visit, looks for all vehicles that can be assigned to this visit. For each compatible couple (visit, vehicle) it then does two things.

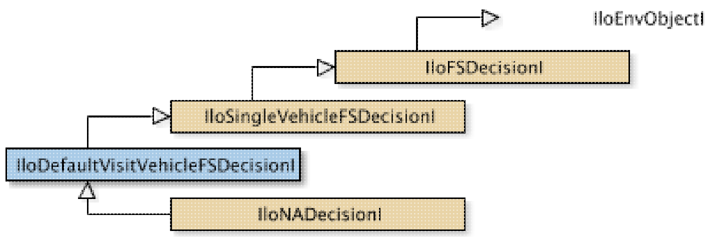
First, it calls the `createDecision(IloVisit, Ilovehicle)` virtual method, which returns a pointer to an `IloFSDecisionI`, possibly zero. This virtual method is responsible for deciding whether or not it is worthwhile to create a decision for this couple. If it is not worthwhile, then no decision is created, the method returns 0, and this possibility will not be considered by the first solution framework. If it returns a non-zero decision, then this decision must be stored using the `storeDecision` method.

```
public void registerVisitVehicleDecision(IloVisit visit, IloVehicle vehicle)
```

This member function is responsible for registering the decision. A decision has to be registered to be considered in the decision maker's main decision-handling loop. Otherwise, it will never be considered.

Class IloDefaultVisitVehicleFSDecisionI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>


IloEnvObjectI
IloFSDecisionI
IloSingleVehicleFSDecisionI
IloDefaultVisitVehicleFSDecisionI
IloNADecisionI

This abstract class models the decision to assign a visit on a vehicle. As such, it derives from `IloSingleVehicleFSDecisionI` class. As the precise location of the visit is not defined at this stage, this decision class does not define the `make` member function, and hence is abstract.

However, the decision comparison process is redefined to compare the costs associated with the decision. Subclassing from this class requires you to define an `evaluate` member function, but the `isBetterThan` member function has a default behavior for this class. This class inherits from `IloFSDecisionI`.

Constructor Summary	
`public`	`IloDefaultVisitVehicleFSDecisionI(IloVisit visit, IloVehicle vehicle)`

Method Summary	
`public virtual IloBool`	`calcFeasibility(IloFSDecisionMakerI * dm) const`
`public virtual void`	`display(ostream & out) const`
`public IloVisit`	`getVisit() const`
`public virtual IloBool`	`isBetterThan(IloFSDecisionI * dec, const IloFSDecisionMakerI * dm) const`
`public virtual void`	`store(IloFSDecisionMakerI * dm)`

Inherited Methods from `IloSingleVehicleFSDecisionI`
`calcFeasibility, display, evaluate, getCost, getInChainStart, getOutChainEnd, getRouteCompletionGoal, getVehicle, isArcFeasible, isFeasible, isPossible`

Inherited Methods from `IloFSDecisionI`
`Compare, Compare, Compare, display, getEnv, getJustifierGoal, getRouteCompletionGoal, IsArcFeasibleOnDimension, isBetterThan, make, store`

Constructors

`public` **`IloDefaultVisitVehicleFSDecisionI`**`(IloVisit visit, IloVehicle vehicle)`

This constructor builds a default visit vehicle decision from a visit and a vehicle.

Methods

`public virtual IloBool` **`calcFeasibility`**`(IloFSDecisionMakerI * dm) const`

This member function computes a feasibility predicate for the decision. As the precise location of the visit on the vehicle is not defined at this class level, this predicate only tests that the vehicle can be assigned to the visit. Subclasses should redefine this member function.

Decisions that are proven infeasible by the `calcFeasibility` predicate will be discarded when searching for the best decision. Feasibility status is automatically refreshed after each decision is taken.

`public virtual void` **`display`**`(ostream & out) const`

This member function displays the decision.

`public IloVisit` **`getVisit`**`() const`

This member function returns the visit associated with the decision. Note that the base class has no `getVisit` member function, as it could involve several visits, as in a decision that would place a pickup and delivery pair.

`public virtual IloBool` **`isBetterThan`**`(IloFSDecisionI * dec, const IloFSDecisionMakerI * dm) const`

This member function is an implementation of the pure virtual member function of the `IloFSDecisionI` class. It assumes that the two decisions are of the `IloDefaultVisitVehicleFSDecisionI` type. This member function tests the two costs and if the cost of the invoking decision is lower, returns true.

If the costs are equal, it performs a tie-breaking on the two decision vehicles, if they are different. Otherwise, it performs a tie-breaking on the decision visits.

`public virtual void` **`store`**`(IloFSDecisionMakerI * dm)`

This member function implements the pure virtual method of class `IloFSDecisionI`. It registers the decision with its visit and its vehicle and also for global searches.

Class IloDelaySumVar

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

A delay sum variable is a constrained variable representing the sum of the delay variables of the visits belonging to the route of a vehicle for a given extrinsic dimension.

If the extrinsic dimension represents time, this variable can be used to limit the total service time spent by a vehicle.

See Also: IloDimension2, IloTravelSumVar, IloVehicle, IloVisit, operator+, IloVehicle::getDelaySumVar

Constructor Summary	
`public`	`IloDelaySumVar(IloVehicle vehicle, IloDimension2 dim2)`

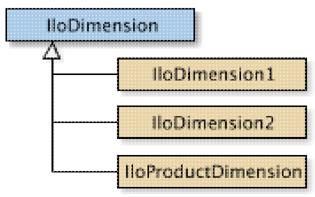
Constructors

`public` **`IloDelaySumVar`**`(IloVehicle vehicle, IloDimension2 dim2)`

This constructor creates a delay sum variable from a vehicle and an extrinsic dimension.

Class IloDimension

Definition file: ildispat/ilodim.h
Include file: <ildispat/ilodispatcher.h>



This is the parent class of the two classes, `IloDimension1` and `IloDimension2`, for representing *dimensions*.

See Also: IloDimension1, IloDimension2

Constructor Summary	
`public`	`IloDimension()`

Method Summary	
`public void`	`assumeTriangleInequality(IloBool assume)`

Constructors

`public` **`IloDimension`**`()`

This constructor creates a dimension whose handle pointer is null.

Methods

`public void` **`assumeTriangleInequality`**`(IloBool assume)`

This member function indicates to the invoking dimension whether it can assume that the triangle inequality holds on all transit triples of the problem. The triangle inequality holds if, for distinct visits *a*, *b*, and *c*, it is true that *transit(a,c) <= transit(a,b) + transit(b,c)*. Use `assumeTriangleInequality` to indicate to the dimension that this condition holds by passing a value of `IloTrue`. By default, the triangle inequality is not assumed.

The triangle inequality can normally be assumed in the following situations:

- instances of `IloDimension2` in node routing problems
- instances of `IloDimension1` where all weights are non-negative.

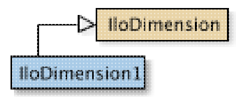
The triangle inequality does not hold in the following situations:

- en route pickup and delivery problems
- problems where weights can be negative.

In Dispatcher 3.2 and later versions, only the propagation activated by `IloDispatcher::setFilterLevel(IlcMedium)` makes use of the value of this flag. When this filtering level is selected, more propagation will be performed on dimensions for which the triangle inequality can be assumed.

Class IloDimension1

Definition file: ildispat/ilodim.h
Include file: <ildispat/ilodispatcher.h>



Instances of the class `IloDimension1` represent the dimensions that are *intrinsic* to an object. For example, weight is represented by `IloDimension1` because the weight of an object depends only on that object, not on any others.

See Also: IloDimension, IloDimension2

Constructor Summary	
`public`	`IloDimension1(IloEnv env, const char * name=0)`
`public`	`IloDimension1(IloEnv env, IloBool postIt, const char * name=0)`

Method Summary	
`public static IloBool`	`Exists(IloEnv env, const char * key)`
`public static IloDimension1`	`Find(IloEnv env, const char * key)`
`public const char *`	`getKey() const`
`public void`	`removeKey()`
`public void`	`setKey(const char * key)`

Inherited Methods from `IloDimension`
`assumeTriangleInequality`

Constructors

`public` **`IloDimension1`**`(IloEnv env, const char * name=0)`

This constructor creates an instance of the class `IloDimension1`, associated with the environment indicated by `env`. The optional argument `name`, if provided, becomes the name of the dimension.

`public` **`IloDimension1`**`(IloEnv env, IloBool postIt, const char * name=0)`

This constructor creates an instance of the class `IloDimension1`, associated with the environment indicated by `env`. The optional argument `name`, if provided, becomes the name of the dimension.

The parameter `postIt` indicates whether the underlying path constraint associated with the dimension is posted or not. Setting `postIt` to `IloFalse` speeds up the search but should only be done if no constraints are posted on variables related to the invoking dimension (using `IloVehicle::setCapacity` with the invoking dimension as a parameter is the same as adding a constraint). However, a dimension created with `postIt=IloFalse` may be safely used in the cost function.

Methods

`public static IloBool` **`Exists`**`(IloEnv env, const char * key)`

This static member function returns `IloTrue` if an `IloDimension1` object having key `key` exists and `IloFalse` if not.

`public static IloDimension1` **`Find`**`(IloEnv env, const char * key)`

This static member function returns the object corresponding to the key `key` set using `IloDimension1::setKey`. If there is no object corresponding to `key` an `IloException` is thrown.

`public const char *` **`getKey`**`() const`

This member function returns the key set on the invoking object

`public void` **`removeKey`**`()`

This member function allows the user to remove the key set on the invoking object.

`public void` **`setKey`**`(const char * key)`

This member function allows the user to set `key` on the invoking object. This key is unique. Each intrinsic dimension must have a different key; otherwise, an exception is thrown.

Class IloDimension1Iterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

An instance of the class `IloDimension1Iterator` is an iterator that traverses all instances of the class `IloDimension1` in a model.

See Also: IloDimension, IloDimension1

Constructor Summary	
public	IloDimension1Iterator(IloModel mdl, IloBool deep=IloTrue)
public	IloDimension1Iterator(const IloDimension1Iterator & iter)

Method Summary	
public IloBool	ok() const
public IloDimension1	operator*() const
public const IloDimension1Iterator &	operator++()
public const IloDimension1Iterator &	operator=(const IloDimension1Iterator & iter)

Constructors

`public` **`IloDimension1Iterator`**`(IloModel mdl, IloBool deep=IloTrue)`

This constructor creates an iterator which will iterate over all instances of `IloDimension1` in model `mdl`. If the parameter `deep` has the value `IloTrue`, all submodels of `mdl` will form part of the iteration. If `deep` has the value `IloFalse`, submodels will not be investigated by the iterator.

`public` **`IloDimension1Iterator`**`(const IloDimension1Iterator & iter)`

This copy constructor creates an iterator from another iterator `iter`. After execution, both the newly created iterator and `iter` will be at the same position within the model.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloFalse` if the iterator has scanned all instances of `IloDimension1` in the model, otherwise it returns `IloTrue`.

`public IloDimension1` **`operator*`**`() const`

This operator returns the instance of `IloDimension1` at which the iterator is currently pointing.

`public const IloDimension1Iterator &` **`operator++`**`()`

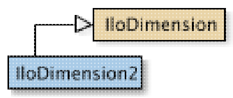
This operator moves the iterator on to the next instance of `IloDimension1` within the model, providing one exists. The operator returns the invoking iterator at its new position.

`public const IloDimension1Iterator &` **`operator=`**`(const IloDimension1Iterator & iter)`

This assignment operator copies the state of `iter` to the iterator on the left-hand side of the operator. After execution, both iterators will be at the same position within the model.

Class IloDimension2

Definition file: ildispat/ilodim.h
Include file: <ildispat/ilodispatcher.h>



Instances of the class `IloDimension2` represent the dimensions that are extrinsic to an object. That is, extrinsic dimensions do not depend only on that single object; in fact, they depend on at least one other factor as well, such as another node. For instance, time is represented by `IloDimension2` because the time to travel from one visit to another depends on *both* of those two visits. By definition, `IloDimension2` is closely linked to the concept of distance. Objects of the class `IloDistance` define how distances are computed between nodes.

See Also: IloDimension, IloDimension1, IloDistance, IloDistanceEvalI, IloDistanceFunction, IloDistanceI, IloSimpleDistanceEvalI, IloSimpleDistanceFunction

Constructor Summary	
public	`IloDimension2(IloEnv env, IloDistance distance, const char * name=0)`
public	`IloDimension2(IloEnv env, IloVisitDistance distance, const char * name=0)`
public	`IloDimension2(IloEnv env, IloDistance distance, IloBool postIt, const char * name=0)`
public	`IloDimension2(IloEnv env, IloVisitDistance distance, IloBool postIt, const char * name=0)`
public	`IloDimension2(IloEnv env, IloDistanceFunction distFunction, const char * name=0)`
public	`IloDimension2(IloEnv env, IloDistanceFunction distFunction, IloBool postIt, const char * name=0)`
public	`IloDimension2(IloEnv env, IloSimpleDistanceFunction distFunction, const char * name=0)`
public	`IloDimension2(IloEnv env, IloSimpleDistanceFunction distFunction, IloBool postIt, const char * name=0)`

Method Summary	
`public static IloBool`	`Exists(IloEnv env, const char * key)`
`public static IloDimension2`	`Find(IloEnv env, const char * key)`
`public const char *`	`getKey() const`
`public IloBool`	`isCached() const`
`public void`	`removeKey()`
`public void`	`setCached(IloBool cached)`
`public void`	`setKey(const char * key)`

Inherited Methods from `IloDimension`
`assumeTriangleInequality`

Constructors

`public **IloDimension2**(IloEnv env, IloDistance distance, const char * name=0)`

This constructor creates an instance of the class `IloDimension2`, associated with the environment indicated by `env`, with distances defined by the argument `distance`. The optional argument `name`, if provided, becomes the name of the dimension.

`public **IloDimension2**(IloEnv env, IloVisitDistance distance, const char * name=0)`

This constructor creates an instance of the class `IloDimension2`, associated with the environment indicated by `env`, with distances defined by the argument `distance`. The optional argument `name`, if provided, becomes the name of the dimension.

`public **IloDimension2**(IloEnv env, IloDistance distance, IloBool postIt, const char * name=0)`

This constructor creates an instance of the class `IloDimension2`, associated with the environment indicated by `env`, with distances defined by the argument `distance`. The optional argument `name`, if provided, becomes the name of the dimension.

The parameter `postIt` indicates whether the underlying constraint associated with the new instance is posted or not. Setting `postIt` to `IloFalse` speeds up the search but should only be done if no constraints are posted on variables related to the invoking dimension. However, a dimension created with `postIt=IloFalse` may be safely used in the cost function.

`public **IloDimension2**(IloEnv env, IloVisitDistance distance, IloBool postIt, const char * name=0)`

This constructor creates an instance of the class `IloDimension2`, associated with the environment indicated by `env`, with distances defined by the argument `distance`. The optional argument `name`, if provided, becomes the name of the dimension.

The parameter `postIt` indicates whether the underlying constraint associated with the new instance is posted or not. Setting `postIt` to `IloFalse` speeds up the search but should only be done if no constraints are posted on variables related to the invoking dimension. However, a dimension created with `postIt=IloFalse` may be safely used in the cost function.

`public **IloDimension2**(IloEnv env, IloDistanceFunction distFunction, const char * name=0)`

This constructor creates an instance of the class `IloDimension2`, associated with the environment indicated by `env`, with distances defined by the function `distFunction`. The optional argument `name`, if provided, becomes the name of the dimension.

`public **IloDimension2**(IloEnv env, IloDistanceFunction distFunction, IloBool postIt, const char * name=0)`

This constructor creates an instance of the class `IloDimension2`, associated with the environment indicated by `env`, with distances defined by the function `distFunction`. The optional argument `name`, if provided, becomes the name of the dimension.

The parameter `postIt` indicates whether the underlying constraint associated with the new instance is posted or not. Setting `postIt` to `IloFalse` speeds up the search but should only be done if no constraints are posted on variables related to the invoking dimension. However, a dimension created with `postIt=IloFalse` may be safely used in the cost function.

```
public IloDimension2(IloEnv env, IloSimpleDistanceFunction distFunction, const char
* name=0)
```

This constructor creates an instance of the class `IloDimension2`, associated with the environment indicated by `env`, with distances defined by the function `distFunction`. The optional argument `name`, if provided, becomes the name of the dimension.

```
public IloDimension2(IloEnv env, IloSimpleDistanceFunction distFunction, IloBool
postIt, const char * name=0)
```

This constructor creates an instance of the class `IloDimension2`, associated with the environment indicated by `env`, with distances defined by the function `distFunction`. The optional argument `name`, if provided, becomes the name of the dimension.

The parameter `postIt` indicates whether the underlying constraint associated with the new instance is posted or not. Setting `postIt` to `IloFalse` speeds up the search but should only be done if no constraints are posted on variables related to the invoking dimension. However, a dimension created with `postIt=IloFalse` may be safely used in the cost function.

Methods

```
public static IloBool Exists(IloEnv env, const char * key)
```

This static member function returns `IloTrue` if an `IloDimension2` object having key `key` exists and `IloFalse` if not.

```
public static IloDimension2 Find(IloEnv env, const char * key)
```

This static member function returns the object corresponding to the key `key` set using `IloDimension2::setKey`. If there is no object corresponding to `key` an `IloException` is thrown.

```
public const char * getKey() const
```

This member function returns the key set on the invoking object

```
public IloBool isCached() const
```

This member function returns `IloTrue` if the status of the invoking extrinsic dimension is cached. Otherwise, it returns `IloFalse`. See the member function `IloDimension2::setCached` for more information.

```
public void removeKey()
```

This member function allows the user to remove the key set on the invoking object.

```
public void setCached(IloBool cached)
```

This member function modifies the `cached` status of the invoking extrinsic dimension. When `cached` is `IloTrue`, it calls `IloDistance::setCache` with `log2rows` = 18 and `log2cols` = 0. This is very useful if distance computations are slow.

When `cached` is `IloFalse`, calls to the distance function of the invoking dimension are not cached. By default, caching is off.

```
public void setKey(const char * key)
```

This member function allows the user to set `key` on the invoking object. This key is unique. Each extrinsic dimension must have a different key; otherwise, an exception is thrown.

Class IloDimension2Iterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

An instance of the class `IloDimension2Iterator` is an iterator that traverses all instances of the class `IloDimension2` in a model.

See Also: IloDimension, IloDimension2

Constructor Summary	
`public`	`IloDimension2Iterator(IloModel mdl, IloBool deep=IloTrue)`
`public`	`IloDimension2Iterator(const IloDimension2Iterator & iter)`

Method Summary	
`public IloBool`	`ok() const`
`public IloDimension2`	`operator*() const`
`public const IloDimension2Iterator &`	`operator++()`
`public const IloDimension2Iterator &`	`operator=(const IloDimension2Iterator & iter)`

Constructors

`public` **`IloDimension2Iterator`**`(IloModel mdl, IloBool deep=IloTrue)`

This constructor creates an iterator which will iterate over all instances of `IloDimension2` in model `mdl`. If the parameter `deep` has the value `IloTrue`, all submodels of `mdl` will form part of the iteration. If `deep` has the value `IloFalse`, submodels will not be investigated by the iterator.

`public` **`IloDimension2Iterator`**`(const IloDimension2Iterator & iter)`

This copy constructor creates an iterator from another iterator `iter`. After execution, both the newly created iterator and `iter` will be at the same position within the model.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloFalse` if the iterator has scanned all instances of `IloDimension2` in the model. Otherwise, it returns `IloTrue`.

`public IloDimension2` **`operator*`**`() const`

This operator returns the instance of `IloDimension2` at which the iterator is currently pointing.

`public const IloDimension2Iterator &` **`operator++`**`()`

This operator moves the iterator on to the next instance of `IloDimension2` within the model, providing one exists. The operator returns the invoking iterator at its new position.

`public const IloDimension2Iterator &` **`operator=`**`(const IloDimension2Iterator & iter)`

This assignment operator copies the state of `iter` to the iterator on the left-hand side of the operator. After execution, both iterators will be at the same position within the model.

Class IloDimensionIterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

An instance of the class `IloDimensionIterator` is an iterator that traverses all instances of the class `IloDimension` in a model (that is, all instances of `IloDimension1` and `IloDimension2`).

See Also: IloDimension, IloDimension1, IloDimension2

Constructor Summary	
`public`	`IloDimensionIterator(IloModel mdl, IloBool deep=IloTrue)`
`public`	`IloDimensionIterator(const IloDimensionIterator & iter)`

Method Summary	
`public IloBool`	`ok() const`
`public IloDimension`	`operator*() const`
`public const IloDimensionIterator &`	`operator++()`
`public const IloDimensionIterator &`	`operator=(const IloDimensionIterator & iter)`

Constructors

`public` **`IloDimensionIterator`**`(IloModel mdl, IloBool deep=IloTrue)`

This constructor creates an iterator which will iterate over all instances of `IloDimension` in model `mdl`. If the parameter `deep` has the value `IloTrue`, all submodels of `mdl` will form part of the iteration. If `deep` has the value `IloFalse`, submodels will not be investigated by the iterator.

`public` **`IloDimensionIterator`**`(const IloDimensionIterator & iter)`

This copy constructor creates an iterator from another iterator `iter`. After execution, both the newly created iterator and `iter` will be at the same position within the model.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloFalse` if the iterator has scanned all instances of `IloDimension` in the model. Otherwise it returns `IloTrue`.

`public IloDimension` **`operator*`**`() const`

This operator returns the instance of `IloDimension` at which the iterator is currently pointing.

`public const IloDimensionIterator & `**`operator++`**`()`

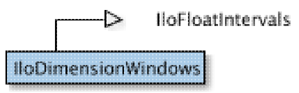
This operator moves the iterator on to the next instance of `IloDimension` within the model, providing one exists. The operator returns the invoking iterator at its new position.

`public const IloDimensionIterator & `**`operator=`**`(const IloDimensionIterator & iter)`

This assignment operator copies the state of `iter` to the iterator on the left-hand side of the operator. After execution, both iterators will be at the same position within the model.

Class IloDimensionWindows

Definition file: ildispat/iloproto.h
Include file: <ildispat/ilodispatcher.h>



A dimension window represents an interval during which a visit can be performed. This means that the visit must start after the beginning of the window and end before the end of the window. Therefore, if the window is related to an instance `dim` of `IloDimension2` and if the window starts at `a` and ends at `b`, the following relation holds: `a <= visit.getCumulVar(dim) && visit.getEndCumulVar(dim) <= b`. Dimension windows are represented by permitted and forbidden intervals. Permitted intervals are the intervals when a visit can occur and forbidden intervals are the intervals when a visit cannot occur. Dimension on windows are exclusively related to extrinsic dimensions (instances of `IloDimension2`).

The constraint `IloExecutionWindowsToVisitCon` relates an instance of `IloDimensionWindows` to a specific `IloVisit`.

See Also: IloExecutionWindowsToVisitCon

Constructor Summary	
`public`	`IloDimensionWindows()`
`public`	`IloDimensionWindows(IloDimensionWindows::ImplClass * impl)`
`public`	`IloDimensionWindows(IloEnv env, IloDimension2 dim, const char * name=0)`

Method Summary	
`public IloDimension2`	`getDimension()`
`public IloDimensionWindows::ImplClass *`	`getImpl() const`
`public IloNum`	`getLB() const`
`public IloNum`	`getUB() const`
`public void`	`setBounds(IloNum lb, IloNum ub)`
`public void`	`setForbiddenInterval(IloNum lb, IloNum ub)`
`public void`	`setLB(IloNum lb)`
`public void`	`setPermittedInterval(IloNum lb, IloNum ub)`
`public void`	`setUB(IloNum ub)`

Inner Class
IloDimensionWindows::ForbiddenIterator
IloDimensionWindows::Iterator

Constructors

`public` **`IloDimensionWindows`**`()`

This constructor creates a dimension window whose handle pointer is null. This object must be assigned before it can be used.

`public **IloDimensionWindows**(IloDimensionWindows::ImplClass * impl)`

This constructor creates a handle object (an instance of `IloDimensionWindows`) from a pointer to an implementation object (an instance of the class `IloDimensionWindowsI`).

`public **IloDimensionWindows**(IloEnv env, IloDimension2 dim, const char * name=0)`

This constructor creates a dimension window on the extrinsic dimension `dim` on the environment `env`. The optional argument `name`, if provided, becomes the name of the dimension window.

Initially, the interval `[0, IloInfinity)` is permitted on the window.

Methods

`public IloDimension2 **getDimension**()`

This member function returns the dimension associated with the dimension window.

`public IloDimensionWindows::ImplClass * **getImpl**() const`

This member function returns a pointer to the implementation object corresponding to the invoking dimension window.

`public IloNum **getLB**() const`

This member function returns the lower bound of the first permitted interval. It is the earliest starting value for the visit attached to the constraint.

`public IloNum **getUB**() const`

This member function returns the upper bound of the last permitted interval. It is the latest ending value for the visit attached to the constraint.

`public void **setBounds**(IloNum lb, IloNum ub)`

This member function sets the earliest starting and latest ending values of the visit attached to the constraint to `lb` and `ub`. This function can remove permitted intervals.

`public void **setForbiddenInterval**(IloNum lb, IloNum ub)`

This member function sets the interval between `lb` and `ub` as forbidden, potentially removing permitted intervals.

`public void **setLB**(IloNum lb)`

This member function sets the earliest starting value of the visit attached to the constraint to `lb`. This function can remove permitted intervals.

```
public void setPermittedInterval(IloNum lb, IloNum ub)
```

This member function sets the interval between `lb` and `ub` as permitted, potentially removing forbidden intervals.

```
public void setUB(IloNum ub)
```

This member function sets the latest ending value of the visit attached to the constraint to `ub`. This function can remove permitted intervals.

Class IloDispatcher

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

An instance of `IloDispatcher` organizes all the details of a routing problem.

See Also: IloVisit, IloVehicle, IloVehicleBreakCon

Constructor Summary	
`public`	`IloDispatcher(IloSolver solver, const char * name)`
`public`	`IloDispatcher()`
`public`	`IloDispatcher(IloDispatcherI * impl)`
`public`	`IloDispatcher(const IloDispatcher & disp)`

Method Summary	
`public void`	`alwaysRecomputeCost(IloBool recompute) const`
`public IloNum`	`getCost(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const`
`public IlcFloatVar`	`getCostVar(IloVehicle vehicle, IloDimension dim) const`
`public IloNumVar`	`getCostVar() const`
`public IlcFloatVar`	`getCumulVar(IloVisit visit, IloDimension dim) const`
`public IlcFloatVar`	`getDelayVar(IloVisit visit, IloDimension2 dim) const`
`public IlcFloatVar`	`getDurationVar(IloVehicleBreakCon brk) const`
`public IlcFloatVar`	`getDurationVar(IloVisit visit, IloDimension2 dim) const`
`public IlcFloatExp`	`getEndCostVar(IloVisit visit, IloDimension2 dim) const`
`public IlcFloatVar`	`getEndCumulVar(IloVisit visit, IloDimension2 dim) const`
`public IloEnv`	`getEnv() const`
`public IlcFilterLevel`	`getFilterLevel() const`
`public IloDispatcherI *`	`getImpl() const`
`public IloInt`	`getIndex(IloVehicle vehicle) const`
`public IloInt`	`getIndex(IloVisit visit) const`
`public IlcIntVar`	`getIntVar(IloVehicleVar var) const`
`public IlcIntVar`	`getIntVar(IloVisitVar var) const`
`public IloModel`	`getModel() const`
`public const char *`	`getName() const`
`public IlcIntVar`	`getNextVar(IloVisit visit) const`
`public IloInt`	`getNumberOfDimensions() const`
`public IloInt`	`getNumberOfNodes() const`
`public IloNum`	`getNumberOfSuccesses() const`
`public IloInt`	`getNumberOfUnperformedVisits() const`
`public IloInt`	

	getNumberOfVehicleBreakConstraints(IloVehicle vehicle, IloDimension2 dim) const
public IloInt	getNumberOfVehicleBreakConstraints(IloVehicle vehicle) const
public IloInt	getNumberOfVehicles() const
public IloInt	getNumberOfVehiclesUsed() const
public IloInt	getNumberOfVisits() const
public IloNumVar	getPenalizedCostVar() const
public IloVisit	getPosition(IloVehicleBreakCon brk) const
public IlcIntVar	getPositionVar(IloVehicleBreakCon brk) const
public IlcIntVar	getPrevVar(IloVisit visit) const
public IlcIntVar	getRankVar(IloVisit visit) const
public IloInt	getRouteSize(IloVehicle vehicle) const
public IloSolver	getSolver() const
public IlcFloatExp	getStartCostVar(IloVisit visit, IloDimension2 dim) const
public IlcFloatVar	getStartVar(IloVehicleBreakCon brk) const
public IloNum	getTotalCost() const
public IlcFloatVar	getTransitVar(IloVisit visit, IloDimension dim) const
public IlcFloatVar	getTravelVar(IloVisit visit, IloDimension2 dim) const
public IloVehicle	getVehicle(IloInt index) const
public IlcIntVar	getVehicleVar(IloVisit visit) const
public IloVisit	getVisit(IloInt index) const
public IlcFloatVar	getWaitVar(IloVisit visit, IloDimension2 dim) const
public IlcConstraint	interrupting(IloVehicleBreakCon brk) const
public IloBool	isInterrupting(IloVehicleBreakCon brk) const
public IloBool	isNonInterrupting(IloVehicleBreakCon brk) const
public IloBool	isPerformed(IloVisit visit) const
public IloBool	isPerformed(IloVehicleBreakCon brk) const
public IloBool	isRouteComplete(IloVehicle vehicle) const
public IloBool	isUnperformed(IloVisit visit) const
public IloBool	isUnperformed(IloVehicleBreakCon brk) const
public IlcConstraint	nonInterrupting(IloVehicleBreakCon brk) const
public IlcConstraint	performed(IloVisit visit) const
public IlcConstraint	performed(IloVehicleBreakCon brk) const
public void	printInformation() const
public void	setFilterLevel(IlcFilterLevel level) const
public void	setInterrupting(IloVehicleBreakCon brk) const
public void	setName(const char * name) const
public void	setNext(IloVisit visit, IloVisit next) const
public void	setNonInterrupting(IloVehicleBreakCon brk) const
public void	setPerformed(IloVehicleBreakCon brk) const
public void	setPosition(IloVehicleBreakCon brk, IloVisit visit) const

public void	setPrev(IloVisit visit, IloVisit prev) const
public void	setUnperformed(IloVehicleBreakCon brk) const
public void	setVehicle(IloVisit visit, IloVehicle vehicle) const
public IlcConstraint	unperformed(IloVisit visit) const
public IlcConstraint	unperformed(IloVehicleBreakCon brk) const
public void	whenComplete(IloVehicle vehicle, const IlcGoal goal) const
public void	whenComplete(IloVehicle vehicle, const IlcDemon demon) const
public void	whenInterrupt(IloVehicleBreakCon con, const IlcGoal goal) const
public void	whenPerformed(IloVehicleBreakCon con, const IlcGoal goal) const

Inner Class
IloDispatcher::RouteIterator
IloDispatcher::UnperformedVisitIterator
IloDispatcher::VehicleBreakConIterator

Constructors

`public` **`IloDispatcher`**`(IloSolver solver, const char * name)`

This constructor creates a dispatcher object associated with the solver `solver`. The optional argument `name`, if provided, becomes the name of the dispatcher object.

`public` **`IloDispatcher`**`()`

This constructor creates a dispatcher object whose handle pointer is null. This object must be assigned before it can be used.

`public` **`IloDispatcher`**`(IloDispatcherI * impl)`

This constructor creates a handle object (an instance of `IloDispatcher`) from a pointer to an implementation object (an instance of the class `IloDispatcherI`).

`public` **`IloDispatcher`**`(const IloDispatcher & disp)`

This copy constructor creates a handle from a reference to a dispatcher object. That dispatcher object and `disp` both point to the same implementation object.

Methods

`public void` **`alwaysRecomputeCost`**`(IloBool recompute) const`

This member function forces Dispatcher to always recompute the cost between visits. Dimension costs set on vehicles result in a cost value for each pair of visits. By default, Dispatcher caches these costs, which can be memory consuming.

`public IloNum` **`getCost`**`(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const`

This member function returns the cost incurred by `vehicle` of the invoking dispatcher object between `visit1` and `visit2`.

`public IlcFloatVar` **`getCostVar`**`(IloVehicle vehicle, IloDimension dim) const`

This member function returns the extracted constrained variable corresponding to the cost coefficient variable of `vehicle` for dimension `dim`. If completing the route of vehicle does not instantiate this variable, you may need to add a goal which will do so.

`public IloNumVar` **`getCostVar`**`() const`

This member function returns the cost variable. This variable is the sum of the cost of the vehicles and of the unperformed visits.

`public IlcFloatVar` **`getCumulVar`**`(IloVisit visit, IloDimension dim) const`

This member function returns the extracted constrained variable corresponding to the cumul variable of `visit` for dimension `dim`.

`public IlcFloatVar` **`getDelayVar`**`(IloVisit visit, IloDimension2 dim) const`

This member function returns the extracted constrained variable corresponding to the delay variable of `visit` for the extrinsic dimension `dim`.

`public IlcFloatVar` **`getDurationVar`**`(IloVehicleBreakCon brk) const`

This member function returns the extracted constrained variable corresponding to the duration variable of the vehicle break constraint `brk`.

`public IlcFloatVar` **`getDurationVar`**`(IloVisit visit, IloDimension2 dim) const`

This member function returns the duration variable for visit `visit` on dimension `dim`. Its semantics are identical to the following: `getSolver().getFloatVar(visit.getDurationVar(dim));`

`public IlcFloatExp` **`getEndCostVar`**`(IloVisit visit, IloDimension2 dim) const`

This member function returns the extracted constrained variable corresponding to the cost of performing `visit` according to the value of its end-cumul variable for the extrinsic dimension `dim`.

`public IlcFloatVar` **`getEndCumulVar`**`(IloVisit visit, IloDimension2 dim) const`

This member function returns the end-cumul variable for visit `visit` on dimension `dim`. Its semantics are identical to the following: `getSolver().getFloatVar(visit.getEndCumulVar(dim));`

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment of the invoking dispatcher object.

`public IlcFilterLevel` **`getFilterLevel`**`() const`

This member function returns the current filter level of the underlying path constraints on extracted dimensions.

`public IloDispatcherI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking dispatcher object.

`public IloInt` **`getIndex`**`(IloVehicle vehicle) const`

This member function returns the index of `vehicle` after extraction.

`public IloInt` **`getIndex`**`(IloVisit visit) const`

This member function returns the index of `visit` after extraction.

`public IlcIntVar` **`getIntVar`**`(IloVehicleVar var) const`

This member function returns the extracted constrained variable corresponding to the vehicle variable `var`. The domain of this variable represents the indices of extracted vehicles.

`public IlcIntVar` **`getIntVar`**`(IloVisitVar var) const`

This member function returns the extracted constrained variable corresponding to the visit variable `var`. The domain of this variable represents the indices of extracted visits.

`public IloModel` **`getModel`**`() const`

This member function returns the model attached to the solver from which the invoking `IloDispatcher` object was created.

`public const char *` **`getName`**`() const`

This member function returns the name of the invoking dispatcher object.

`public IlcIntVar` **`getNextVar`**`(IloVisit visit) const`

This member function returns the extracted constrained variable corresponding to the next variable of `visit`. The domain of this variable represents the indices of extracted visits.

`public IloInt` **`getNumberOfDimensions`**`() const`

This member function returns the number of dimensions in the invoking dispatcher object.

`public IloInt` **`getNumberOfNodes`**`() const`

This member function returns the number of nodes associated with the invoking dispatcher object.

`public IloNum` **`getNumberOfSuccesses`**`() const`

This member function returns the number of moves that have succeeded in the invoking dispatcher object.

`public IloInt` **`getNumberOfUnperformedVisits`**`() const`

This member function returns the number of unperformed visits in the invoking dispatcher object.

`public IloInt` **`getNumberOfVehicleBreakConstraints`**`(IloVehicle vehicle, IloDimension2 dim) const`

This member function returns the number of extracted vehicle constraints on vehicle `vehicle` and dimension `dim`. This number includes vehicle constraints that are added directly as hard constraints to the model, and those that are involved only in metaconstraints.

`public IloInt` **`getNumberOfVehicleBreakConstraints`**`(IloVehicle vehicle) const`

This member function returns the number of extracted vehicle constraints on vehicle `vehicle`. This number includes vehicle constraints that are added directly as hard constraints to the model, and those that are involved only in metaconstraints. This member function returns the total number of break constraints for vehicle created for all dimensions.

`public IloInt` **`getNumberOfVehicles`**`() const`

This member function returns the number of vehicles associated with the invoking dispatcher object.

`public IloInt` **`getNumberOfVehiclesUsed`**`() const`

This member function returns the total number of vehicles used in the invoking dispatcher object.

`public IloInt` **`getNumberOfVisits`**`() const`

This member function returns the number of visits associated with the invoking dispatcher object.

`public IloNumVar` **`getPenalizedCostVar`**`() const`

This member function returns the penalized cost variable used in guided local search.

`public IloVisit` **`getPosition`**`(IloVehicleBreakCon brk) const`

This member function returns the position of the vehicle break `brk` of the invoking dispatcher object. The position is the visit after which the break occurs. If the position variable is unbound, an error occurs.

`public IlcIntVar` **`getPositionVar`**`(IloVehicleBreakCon brk) const`

This member function returns the extracted constrained variable corresponding to the position variable of the vehicle break constraint `brk`.

`public IlcIntVar` **`getPrevVar`**`(IloVisit visit) const`

This member function returns the extracted constrained variable corresponding to the prev variable of `visit`. The domain of this variable represents the indices of extracted visits.

`public IlcIntVar` **`getRankVar`**`(IloVisit visit) const`

This member function returns the extracted constrained variable corresponding to the rank variable of `visit`.

`public IloInt` **`getRouteSize`**`(IloVehicle vehicle) const`

This member function returns the number of visits in the route of vehicle, not including the vehicle's first and last (depot) visits. This function can only be used in a dispatcher object which is in an instantiated state.

`public IloSolver` **`getSolver`**`() const`

This member function returns the solver associated with the invoking dispatcher object.

`public IlcFloatExp` **`getStartCostVar`**`(IloVisit visit, IloDimension2 dim) const`

This member function returns the extracted constrained variable corresponding to the cost of performing `visit` according to the value of its cumul variable for the extrinsic dimension `dim`.

`public IlcFloatVar` **`getStartVar`**`(IloVehicleBreakCon brk) const`

This member function returns the extracted constrained variable corresponding to the start variable of the vehicle break constraint `brk`.

```
public IloNum getTotalCost() const
```

This member function returns the same value that is obtained by calling `getCostVar().getMin()`.

```
public IlcFloatVar getTransitVar(IloVisit visit, IloDimension dim) const
```

This member function returns the extracted constrained variable corresponding to the transit variable of `visit` for dimension `dim`.

```
public IlcFloatVar getTravelVar(IloVisit visit, IloDimension2 dim) const
```

This member function returns the extracted constrained variable corresponding to the travel variable of `visit` for the extrinsic dimension `dim`.

```
public IloVehicle getVehicle(IloInt index) const
```

This member function returns the vehicle corresponding to `index`.

Note

The following is always true (`i` is an `IloInt` and `v` an `IloVehicle`):

`dispatcher.getIndex(dispatcher.getVehicle(i)) == i` and
`dispatcher.getVehicle(dispatcher.getIndex(v)) == v`.

```
public IlcIntVar getVehicleVar(IloVisit visit) const
```

This member function returns the extracted constrained variable corresponding to the vehicle variable of `visit`. The domain of this variable represents the indices of extracted vehicles.

```
public IloVisit getVisit(IloInt index) const
```

This member function returns the visit corresponding to `index`.

Note

The following is always true (`i` is an `IloInt` and `v` an `IloVisit`):

`dispatcher.getIndex(dispatcher.getVisit(i)) == i` and
`dispatcher.getVisit(dispatcher.getIndex(v)) == v`.

```
public IlcFloatVar getWaitVar(IloVisit visit, IloDimension2 dim) const
```

This member function returns the extracted constrained variable corresponding to the waiting variable of `visit` for the extrinsic dimension `dim`.

```
public IlcConstraint interrupting(IloVehicleBreakCon brk) const
```

This member function returns a constraint that may be added within search to assert that `brk` must interrupt a visit. This member function may be used only inside search.

```
public IloBool isInterrupting(IloVehicleBreakCon brk) const
```

This member function returns `IloTrue` if and only if `brk` must interrupt a visit. A break is said to interrupt a visit if it executes between the start and the end of the visit (see `IloVisit::getCumulVar()` and `IloVisit::getEndCumulVar()`).

This condition may either be deduced by propagation (using start times and duration of visits and breaks), or asserted through the use of the member function `setInterrupting(IloVehicleBreakConstraint)`. Note that if `IloFalse` is returned, this does not indicate that `brk` will definitely not interrupt a visit (for this condition use `isNonInterrupting(brk)`), but only that it is not certain that the `brk` will interrupt a visit.

```
public IloBool isNonInterrupting(IloVehicleBreakCon brk) const
```

This member function returns `IloTrue` if and only if `brk` must not interrupt a visit. A break is said to interrupt a visit if it executes between the start and the end of the visit (see `IloVisit::getCumulVar()` and `IloVisit::getEndCumulVar()`). This condition may either be deduced by propagation (using start times and duration of visits and breaks), or asserted through the use of the member function `setNonInterrupting(IloVehicleBreakConstraint)`.

Note that if `IloFalse` is returned, this does not indicate that `brk` will definitely interrupt a visit (for this condition use `isInterrupting(brk)`), but only that it is not certain that the `brk` will not interrupt a visit.

```
public IloBool isPerformed(IloVisit visit) const
```

This member function returns `IloTrue` if `visit` is performed. Otherwise, it returns `IloFalse`.

```
public IloBool isPerformed(IloVehicleBreakCon brk) const
```

This member function returns `IloTrue` if and only if `brk` must be performed. A break is optional if it not posted as a hard constraint, but instead occurs as part of a metaconstraint. This performed status may be asserted through the use of the member function `setPerformed(IloVehicleBreakConstraint)`. Note that if `IloFalse` is returned, this does not indicate that `brk` will definitely not be performed (for this condition use `isUnperformed(brk)`), but only that it is not certain that the `brk` will be performed.

```
public IloBool isRouteComplete(IloVehicle vehicle) const
```

This member function returns `IloTrue` if `vehicle` has a complete route associated with it. This means that the route is completely connected from the first to the last visit. Otherwise, it returns `IloFalse`.

```
public IloBool isUnperformed(IloVisit visit) const
```

This member function returns `IloTrue` if `visit` is unperformed. Otherwise, it returns `IloFalse`.

`public IloBool` **`isUnperformed`**`(IloVehicleBreakCon brk) const`

This member function returns `IloTrue` if and only if `brk` must not be performed. A break is optional if it is not posted as a hard constraint, but instead occurs as part of a metaconstraint. This unperformed status may either be deduced by propagation (using start times and duration of visits and breaks), or asserted through the use of the member function `setUnperformed(IloVehicleBreakConstraint)`. Note that if `IloFalse` is returned, this does not indicate that `brk` will definitely be performed (for this condition use `isPerformed(brk)`), but only that it is not certain that the `brk` will not be performed.

`public IlcConstraint` **`nonInterrupting`**`(IloVehicleBreakCon brk) const`

This member function returns a constraint that may be added within search to assert that `brk` must not interrupt any visit. This member function may be used only inside search.

`public IlcConstraint` **`performed`**`(IloVisit visit) const`

This member function returns a constraint stating that `visit` must be performed by a vehicle. As an `IlcConstraint` is returned, this method is useful inside a Solver search for deciding if a visit should be performed or not. If no penalty cost has been set on `visit`, this constraint is always satisfied.

`public IlcConstraint` **`performed`**`(IloVehicleBreakCon brk) const`

This member function returns a constraint that may be added within search to assert that `brk` must be performed. This member function may be used only inside search.

`public void` **`printInformation`**`() const`

This member function displays information about the invoking dispatcher object to standard output.

`public void` **`setFilterLevel`**`(IlcFilterLevel level) const`

This member function sets the filter level of the underlying path constraints on extracted dimensions to `level`. The available levels are `IlcLow` (default), `IlcBasic`, and `IlcMedium`. When the level is `IlcLow`, the constraints do not propagate until the route of a vehicle is closed. When the level is `IlcBasic`, propagation is triggered by `whenValue` events for next variables and by `whenRange` events for cumulative variables and transit variables. When the level is `IlcMedium`, propagation is triggered by `whenDomain` events for next variables and `whenRange` events for cumulative and transit variables.

The `IlcMedium` level takes considerably longer to propagate than the other two levels. It is recommended only during execution of a first solution method you have written, when the `IlcBasic` level has proved insufficient and resulted in a large number of backtracks.

`public void` **`setInterrupting`**`(IloVehicleBreakCon brk) const`

This member function asserts that `brk` must interrupt a visit; it is normally used only in user code as part of a custom-written goal to instantiate vehicle breaks. This member function may only be used inside search. The effects of this member function are reversible.

`public void setName(const char * name) const`

This member function sets the name of the invoking dispatcher object to a copy of `name`.

`public void setNext(IloVisit visit, IloVisit next) const`

This member function sets `next` to be the visit just after `visit` in the extracted model. This method should only be used during search.

`public void setNonInterrupting(IloVehicleBreakCon brk) const`

This member function asserts that `brk` must not interrupt any visit; it is normally used only in user code as part of a custom-written goal to instantiate vehicle breaks. This member function may be used only inside search. The effects of this member function are reversible.

`public void setPerformed(IloVehicleBreakCon brk) const`

This member function asserts that `brk` must be performed; it is normally used only in user code as part of a custom-written goal to instantiate vehicle breaks. This member function may be used only inside search. The effects of this member function are reversible.

`public void setPosition(IloVehicleBreakCon brk, IloVisit visit) const`

This member function sets `visit` to be the position of `brk` in the extracted model. This method should only be used during search.

`public void setPrev(IloVisit visit, IloVisit prev) const`

This member function sets `prev` to be the visit just before `visit` in the extracted model. This method should only be used during search.

`public void setUnperformed(IloVehicleBreakCon brk) const`

This member function asserts that `brk` must not be performed; it is normally used only in user code as part of a custom-written goal to instantiate vehicle breaks. This member function may be used only inside search. The effects of this member function are reversible.

`public void setVehicle(IloVisit visit, IloVehicle vehicle) const`

This member function sets `vehicle` to be the vehicle performing `visit` in the extracted model. This method should only be used during search.

`public IlcConstraint unperformed(IloVisit visit) const`

This member function returns a constraint stating that `visit` must not be performed by a vehicle. As an `IlcConstraint` is returned, this method is useful inside a Solver search for deciding if a visit should be performed or not. If no penalty cost has been set on `visit`, this constraint is always violated.

`public IlcConstraint` **`unperformed`**`(IloVehicleBreakCon brk) const`

This member function returns a constraint that may be added within search to assert that `brk` must not be performed. This member function may be used only inside search.

`public void` **`whenComplete`**`(IloVehicle vehicle, const IlcGoal goal) const`

This member function associates `goal` with the “complete” event of `vehicle`. Whenever the route of `vehicle` becomes complete, the `goal` will be executed immediately. A route is complete when the route performed by the vehicle is closed (in other words, when a route starting at the first visit of the vehicle and ending at the last one has been created).

`public void` **`whenComplete`**`(IloVehicle vehicle, const IlcDemon demon) const`

This member function associates `demon` with the “complete” event of `vehicle`. Whenever the route of `vehicle` becomes complete, the `demon` will be executed immediately. A route is complete when the route performed by the vehicle is closed (in other words, when a route starting at the first visit of the vehicle and ending at the last one has been created).

`public void` **`whenInterrupt`**`(IloVehicleBreakCon con, const IlcGoal goal) const`

This member function may be used for writing constraints that react to changes in the state of a vehicle break constraint. When the interrupt status of `con` is known (either interrupting or noninterrupting), then `goal` will be executed. This member function may be used only inside search. The effects of this member function are reversible.

`public void` **`whenPerformed`**`(IloVehicleBreakCon con, const IlcGoal goal) const`

This member function may be used for writing constraints that react to changes in the performed state of a vehicle break constraint. When the performed status of `con` is known (either performed or unperformed), then `goal` will be executed. This member function may be used only inside search. The effects of this member function are reversible.

Class IloDispatcherFSParameters

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Dispatcher's built-in first solution methods can be parameterized in a variety of ways. Owing to the various different parameters that can be passed, passing these directly in the constructor of the first solution method can be cumbersome. This handle class encapsulates the different types of parameters that can be passed to Dispatcher's built-in first solution heuristics. Not all of the first solution heuristics can make use of all the parameters.

The parameters which comprise the `IloDispatcherFSParameters` class are the following:

- A goal factory (an instance of `IloDispatcherGoalFactory`). This goal factory is used to generate goals which perform some action each time a vehicle's route is tested for legality at each stage in the first solution construction method. Normally, the goal performs any scheduling of the cumul vars along the route.
- A number of proximate visits. This integer value (called *k*) is used by first solution methods (where appropriate) to determine, at each stage in the construction process, how many visits are to be considered for routing next to the visit under consideration. Dispatcher always considers up to the *k* closest visits to the one under consideration, with proximity in this respect being defined by Dispatcher's cost function.
- A partial solution (an instance of `IloRoutingSolution`). This solution is used to maintain a partial solution during the first solution construction. If the first solution method fails, then this solution will hold the visits and vehicles which were scheduled immediately before the failure.
- A search limit (an instance of `IloSearchLimit`). During the construction of a first solution, routes are extended. Each time a route is extended, the validity of the current route is tested by "closing" the route; that is, the current route runs from the vehicle's first to last visits. In order to close the route, Dispatcher executes an internal goal. For simple problems, this succeeds easily, but for more complex ones, this can take longer. For example, this can occur in the presence of "same-vehicle" constraints that mean that some visits not already scheduled must be scheduled on the same route as their partner in order to close the route. This goal, if not limited, can take a prohibitive amount of time to execute. The search limit serves to limit the execution of this "vehicle closing" goal.

Note

This limit does not limit the goal which is generated via the goal factory, but merely the route closing goal which is executed before the goal generated by the goal factory. If a limit is needed on the goal generated by the goal factory, this should be done explicitly at that point.

Constructor Summary	
`public`	`IloDispatcherFSParameters(IloEnv env)`

Method Summary	
`public IloDispatcherGoalFactory`	`getGoalFactory() const`
`public IloInt`	`getNumberOfProximateVisits() const`
`public IloRoutingSolution`	`getPartialSolution() const`
`public IloSearchLimit`	`getSearchLimit() const`
`public void`	`setGoalFactory(IloDispatcherGoalFactory goalFactory)`
`public void`	`setNumberOfProximateVisits(IloInt nbOfProximateVisits)`

public void	setPartialSolution(IloRoutingSolution solution)
public void	setSearchLimit(IloSearchLimit limit)

Constructors

public **IloDispatcherFSParameters**(IloEnv env)

This constructor creates an instance of the `IloDispatcherFSParameters` class. The class will be allocated on the environment `env`.

Methods

public IloDispatcherGoalFactory **getGoalFactory**() const

This member function returns the goal factory previously set using a call to `setGoalFactory`, or an empty handle if no such call has been made.

public IloInt **getNumberOfProximateVisits**() const

This member function returns the value previously set using a call to `setNbOfProximateVisits`, or `IloIntMax` if no such call has been made.

public IloRoutingSolution **getPartialSolution**() const

This member function returns the routing solution previously set using a call to `setPartialSolution`, or an empty handle if no such call has been made.

public IloSearchLimit **getSearchLimit**() const

This member function returns the search limit previously set using a call to `setSearchLimit`, or an empty handle if no such call has been made.

public void **setGoalFactory**(IloDispatcherGoalFactory goalFactory)

This member function sets the goal factory parameter to `goalFactory`, which can be later retrieved using `getGoalFactory`.

public void **setNumberOfProximateVisits**(IloInt nbOfProximateVisits)

This member function sets the number of proximate visits parameter to `nbOfProximateVisits`, which can be later retrieved using `getNumberOfProximateVisits`.

public void **setPartialSolution**(IloRoutingSolution solution)

This member function sets the partial solution parameter to `solution`, which can be later retrieved using `getPartialSolution`. Note that it is sufficient to pass an empty instance of `IloRoutingSolution`, created using only `IloRoutingSolution(IloEnv)`, as the first solution method will fill in the details of the solution as it goes.

```
public void setSearchLimit(IloSearchLimit limit)
```

This member function sets the search limit parameter to `limit`, which can be later retrieved using `getSearchLimit()`.

Class IloDispatcherGLS

Definition file: ildispat/ilometa.h
Include file: <ildispat/ilodispatcher.h>

This class implements a metaheuristic for routing problems based on the Guided Local Search metaheuristic, which is based upon the idea of optimizing an augmented cost function.

An augmented cost function is created by adding a penalty term to the true cost function. The penalty term is the sum of all penalties for possible arcs in the routing problem. The penalty for each possible arc starts at zero.

Guided Local Search (GLS) works over a number of iterations. At each iteration, a greedy search to a local minimum, reducing the *penalized cost*, is carried out. (In the first iteration, since the penalty term is zero, this is equivalent to a greedy search on the true cost.) When a local minimum is reached, GLS increases the penalty term by choosing an arc of the solution and penalizing it. This increases the penalty term such that in subsequent iterations, cost can be reduced by removing that arc from the solution. In this way GLS allows search to roam, moving out of local minima.

Choosing an Arc to Penalize

GLS tries to choose a *bad* or costly arc in the solution to penalize, as removing costly arcs should lead to finding better solutions. GLS penalizes an arc for which *cost(a)/(1 + pentimes(a))* is a maximum of all arcs in the current solution.

- *cost(a)* is the cost of arc *a*. This is derived from `IloDispatcher::getCost(IloVisit v1, IloVisit v2, IloVehicle veh)` where `v1` is the first visit of arc *a*, `v2` is the second visit of arc *a*, and `veh` is the vehicle which performs `v1->` `v2` in the current solution.
- *pentimes(a)* is the number of times arc *a* has already been penalized.

Thus GLS tries to penalize arcs with high cost. However, if an arc has been penalized a number of times, the importance of cost reduces. This is due to the fact that, if an arc has been penalized a large number of times and is still in the solution, there may be no better arc with which to replace it and it is probably best to start looking elsewhere to place penalties.

Penalizing an Arc

An arc is penalized by increasing its penalty cost by an amount equal to a penalty factor (specified to the metaheuristic) times the cost of the arc, as described above.

Note

If an arc is penalized on one vehicle, it will also be penalized on all equivalent vehicles.

The following example shows how `IloDispatcherGLS` can be used to search for a solution to a routing problem:

```
void ImproveWithGLS(IloDispatcher dispatcher,
                    IloRoutingSolution solution,
                    IloNHood nhood) {
  IloNumVar cost = dispatcher.getCostVar();
  IloEnv env = dispatcher.getEnv();
  IloGoal instantiateCost = IloDichotomize(env, cost, IloFalse);
  IloRoutingSolution rsol = solution.makeClone(env);
  IloRoutingSolution best = solution.makeClone(env);
  IloDispatcherGLS dgls(env, 0.2);

  IloSearchSelector sel = IloMinimizeVar(env, dgls.getPenalizedCostVar());
  IloGoal move = IloSingleMove(env, rsol, nhood, dgls, sel,
instantiateCost);
  move = move && IloStoreBestSolution(env, best);
```

```
    IloSolver solver = dispatcher.getSolver();
    IloCouple(nhood, dgls);
    for (IloInt i = 0; i < 150; i++) {
      if (solver.solve(move)) {
        cout << "Cost = " << solver.getMax(cost) << endl;
      }
      else {
        cout << "---" << endl;
        if (dgls.complete()) break;
      }
    }
    IloDecouple(nhood, dgls);
    IloGoal restoreSolution = IloRestoreSolution(env, best) &&
                              instantiateCost;
    solver.solve(restoreSolution);
    rsol.end();
    best.end();
  }
```

For more information, see the class `IloMetaHeuristic` in the *IBM ILOG Solver Reference Manual*.

See Also: IloCouple, IloDecouple, IloDispatcherTabuSearch

Constructor Summary	
`public`	`IloDispatcherGLS(IloEnv env, IloNum penfactor=0.15)`

Method Summary	
`public IloBool`	`complete()`
`public IloNum`	`getImprovementStep() const`
`public IloNumVar`	`getPenalizedCostVar() const`
`public IloNum`	`getPenalty() const`
`public IloNumVar`	`getPenaltyCostVar() const`
`public IloNum`	`getPenaltyFactor() const`
`public IloBool`	`isFeasible(IloSolver solver, IloSolution delta) const`
`public void`	`notify(IloSolver solver, IloSolution delta)`
`public void`	`reset()`
`public void`	`setImprovementStep(IloNum step)`
`public void`	`setPenaltyFactor(IloNum penFactor)`
`public IloBool`	`start(IloSolver solver, IloSolution solution)`
`public IloBool`	`test(IloSolver solver, IloSolution delta)`

Constructors

`public` **`IloDispatcherGLS`**`(IloEnv env, IloNum penfactor=0.15)`

This constructor builds a metaheuristic that performs a guided local search for a routing problem. The penalty factor indicates how strongly arcs should be penalized.

Methods

`public IloBool` **`complete`**`()`

This member function penalizes the cost of some arcs according to the rule specified in the description. It is important that this member function be called when a local minimum is reached, so that the GLS can remove itself using penalizing arcs. The function returns `IloTrue` if no arcs could be penalized (all have cost 0). Otherwise, it returns `IloFalse` (the usual case).

`public IloNum` **`getImprovementStep`**`() const`

This member function returns the step specified to the invoking metaheuristic at the previous call to `setImprovementStep`. If no such call has been made, 1e-4 is returned.

`public IloNumVar` **`getPenalizedCostVar`**`() const`

This member function returns the cost variable representing the objective to be optimized plus the penalty variable returned from `getPenaltyCostVar`. If a guided local search minimizing the penalized cost at each step is desired, a search selector that minimizes this variable should be used.

`public IloNum` **`getPenalty`**`() const`

This member function returns the penalty of the last solution instantiated.

`public IloNumVar` **`getPenaltyCostVar`**`() const`

This member function returns the variable representing the initial penalty to be added to the true cost variable to be optimized. Its domain is in the range `[0..IloInfinity)`.

`public IloNum` **`getPenaltyFactor`**`() const`

This member function returns the penalty factor passed in the constructor, or the most recently mentioned one in a call to `setPenaltyFactor`.

`public IloBool` **`isFeasible`**`(IloSolver solver, IloSolution delta) const`

This member function performs a pre-filter of solution deltas according to their penalized cost. If the neighbor specified by `delta` has a penalized cost of at least the upper bound of the penalized cost function, `IloFalse` is returned. In all other cases, `IloTrue` is returned.

`public void` **`notify`**`(IloSolver solver, IloSolution delta)`

This member function stores the value of the penalty variable (returned from `getPenaltyCostVar`) so that it can be retrieved with `getPenalty` later.

`public void` **`reset`**`()`

This member function sets the current penalty back to 0 for all arcs, so that GLS can be used for a new search.

```
public void setImprovementStep(IloNum step)
```

This member function indicates to guided local search that the penalized cost function must improve by at least `step` at each movement. By default, the step size is 1e-4.

```
public void setPenaltyFactor(IloNum penFactor)
```

This member function sets the penalty factor to `penFactor`, which allows you to adjust how strongly arcs are to be penalized during search.

```
public IloBool start(IloSolver solver, IloSolution solution)
```

This member function adds a constraint to the solver `solver` that enforces the relationship between the penalized cost variable, the penalty cost variable, and the objective variable in `solution`. The penalized cost variable is the sum of the remaining two. If no constraints are violated by doing this, `IloTrue` is returned. Otherwise, a failure occurs.

```
public IloBool test(IloSolver solver, IloSolution delta)
```

This member function causes a failure if the cost of the instantiated solution plus its penalty is greater than the upper bound of the penalized cost variable. Otherwise, it returns `IloTrue`.

Class IloDispatcherGoalFactory

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

This class is used by Dispatcher's first solution parameters and method to produce subgoals which are then used, for example, to schedule complete routes. First solution heuristics use these subgoals to validate their construction of a routing plan. As routes are built incrementally, it can be useful to run a validation subgoal only for those vehicles which have been modified since the last decision. The goal factory class provides a way to define which goal is to be used to validate a given vehicle. Note that validating one route independently of the others is not always possible. If, for example, a set of routes compete for a shared resource, then a routing change in one of these routes must be validated by a full rescheduling of the complete set. Rescheduling only the touched route cannot guarantee the feasibility of the final plan.

The `IloDispatcherGoalFactory` class is used both to produce goals for validation of individual vehicles and to validate whole routing plans, using the `getGoal()` method.

The functionalities provided by this class are limited to those of the basic `IloGoal` class. As a constructor exists to build a goal factory from an instance of `IloGoal`, instances of `IloGoal` can be passed where an `IloDispatcherGoalFactory` is expected.

See Also: IloDispatcherFSParameters, IloSavingsGenerate, IloNearestAdditionGenerate, IloSweepGenerate, IloNearestDepotGenerate, IloInsertionGenerate

Constructor Summary	
`public`	`IloDispatcherGoalFactory()`
`public`	`IloDispatcherGoalFactory(IloGoal goal)`
`public`	`IloDispatcherGoalFactory(IloDispatcherGoalFactoryI * impl)`
`public`	`IloDispatcherGoalFactory(const IloDispatcherGoalFactory & elem)`

Method Summary	
`public IlcGoal`	`getGoal(IloSolver solver) const`
`public IloGoal`	`getGoal(IloEnv env) const`
`public IloDispatcherGoalFactoryI *`	`getImpl() const`
`public IlcGoal`	`getVehicleGoal(IloSolver solver, IloVehicle vehicle) const`
`public IloGoal`	`getVehicleGoal(IloEnv env, IloVehicle vehicle) const`
`public IloBool`	`isSimpleGoal() const`
`public void`	`operator=(const IloDispatcherGoalFactory & h)`

Constructors

`public` **`IloDispatcherGoalFactory`**`()`

This constructor creates a goal factory object whose handle pointer is null. This object must be assigned before it can be used.

`public` **`IloDispatcherGoalFactory`**`(IloGoal goal)`

This constructor builds a goal factory from an instance of an `IloGoal` goal. When the newly constructed goal factory is passed to a first solution method, this goal factory makes sure that the subgoal `goal` is called each time a step in the first solution construction procedure needs to be validated.

`public` **`IloDispatcherGoalFactory`**`(IloDispatcherGoalFactoryI * impl)`

This constructor creates a handle object (an instance of `IloDispatcherGoalFactory`) from a pointer to an implementation object (an instance of the class `IloDispatcherGoalFactoryI`).

`public` **`IloDispatcherGoalFactory`**`(const IloDispatcherGoalFactory & elem)`

This copy constructor creates a handle from a reference to a goal factory object. That goal factory object and `elem` both point to the same implementation object.

Methods

`public IlcGoal` **`getGoal`**`(IloSolver solver) const`

This method returns an `IlcGoal` to validate a whole routing plan.

`public IloGoal` **`getGoal`**`(IloEnv env) const`

This method returns an `IloGoal` to validate a whole routing plan.

`public IloDispatcherGoalFactoryI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking goal factory.

`public IlcGoal` **`getVehicleGoal`**`(IloSolver solver, IloVehicle vehicle) const`

This method returns an `IlcGoal` to validate the route for vehicle `vehicle`.

`public IloGoal` **`getVehicleGoal`**`(IloEnv env, IloVehicle vehicle) const`

This method returns an `IloGoal` to validate the route for vehicle `vehicle`.

`public IloBool` **`isSimpleGoal`**`() const`

This predicate returns `IloTrue` when the goal factory has been built with one single goal. In this case, it always returns this goal to validate all routes, and also for global plan validation.

`public void` **`operator=`**`(const IloDispatcherGoalFactory & h)`

This operator assigns an address to the handle pointer of the invoking goal factory. This address is the location of the implementation object of the argument `h`. After the execution of this operator, the invoking goal factory and `h` both point to the same implementation object.

Class IloDispatcherGoalFactoryI

Definition file: ildispat/ilogoals.h

This class is the base implementation class for vehicle goal factories. A goal factory implements a mapping from vehicles to goals. Goal factories are used in all predefined first solution heuristics.

To define a new class of goal factory, you must define two virtual methods that return `IlcGoal` objects:

- The `getGoal()` method returns a global goal, used to validate a global plan.
- The `getVehicleGoal()` method return a goal that validates only the route of the vehicle, passed as an argument.

Method Summary	
`public virtual IlcGoal`	`getGoal(IloSolver solver) const`
`public IloGoal`	`getIloGoal(IloEnv env) const`
`public IloGoal`	`getIloVehicleGoal(IloEnv env, IloVehicle vehicle) const`
`public virtual IlcGoal`	`getVehicleGoal(IloSolver solver, IloVehicle vehicle) const`

Methods

`public virtual IlcGoal` **`getGoal`**`(IloSolver solver) const`

This pure virtual method returns the global validation goal.

`public IloGoal` **`getIloGoal`**`(IloEnv env) const`

This method builds and returns an `IloGoal` which, when extracted, returns the global validation goal, as defined by the virtual method `getGoal()`. This method is not virtual and should not be redefined.

`public IloGoal` **`getIloVehicleGoal`**`(IloEnv env, IloVehicle vehicle) const`

This method builds and returns an `IloGoal` that, when extracted, returns the vehicle validation goal, as defined by the virtual method `getVehicleGoal()`. This method is not virtual and should not be redefined.

`public virtual IlcGoal` **`getVehicleGoal`**`(IloSolver solver, IloVehicle vehicle) const`

This pure virtual method returns a goal to validate the route of vehicle `vehicle`.

Class IloDispatcherGraph

Definition file: ildispat/ilographdist.h
Include file: <ildispat/ilodispatcher.h>

The class `IloDispatcherGraph` allows you to create a graph representing a road network on which instances of `IloNode` can be positioned. `IloDispatcherGraph` maintains a directed graph representation on which the user may specify costs for traversing specific arcs, as well as penalties for turns between consecutive arcs. `IloDispatcherGraph` includes functionality to load network topology and costs from a `.csv` file, as well as for modifying the topology and costs by direct manipulation.

It computes and stores the cheapest paths between nodes for each vehicle. The cheapest path is the path whose value minimizes the cost function for a given vehicle.

See Also: IloDispatcherGraph::Node, IloDispatcherGraph::Arc, IloDispatcherGraph::PathIterator, IloDispatcherGraph::AdjacencyListIterator, IloGraphDistance, IloNode

Constructor and Destructor Summary	
`public`	`IloDispatcherGraph(IloEnv env, IloBool storePaths=IloTrue, IloInt preSizeArcs=1, IloInt preSizeNodes=1)`
`public`	`IloDispatcherGraph(IloEnv env, const char * file)`
`public`	`IloDispatcherGraph(const IloDispatcherGraph & g)`
`public`	`IloDispatcherGraph(IloDispatcherGraphI * impl=0)`

Method Summary	
`public void`	`addArc(IloDispatcherGraph::Node n1, IloDispatcherGraph::Node n2, IloDimension2 dim, IloNum value)`
`public void`	`addDimension2(IloDimension2 dim)`
`public IloBool`	`arcExists(const IloInt arcId) const`
`public IloBool`	`arcExistsWithEnds(const IloInt fromId, const IloInt toId) const`
`public void`	`associateByCoordsInFile(const IloNode node, const char * fileName)`
`public void`	`createArcsFromFile(const char * fileName, IloBool loadDims=IloTrue)`
`public void`	`end()`
`public void`	`forbidArcUse(IloDispatcherGraph::Arc a)`
`public void`	`forbidTurn(IloDispatcherGraph::Arc from, IloDispatcherGraph::Arc to)`
`public IloDispatcherGraph::Arc`	`getArc(const IloInt arcId) const`
`public IloDispatcherGraph::Arc`	`getArcByEnds(const IloInt fromNodeId, const IloInt toNodeId) const`
`public IloNum`	`getArcCost(IloDispatcherGraph::Arc a, IloDimension2 dim)`
`public void *`	`getArcObject(IloDispatcherGraph::Arc a)`
`public IloNum`	

	getDistance(IloNode n1, IloNode n2, IloVehicle v, IloDimension2 dim) const
public IloEnv	getEnv() const
public IloDispatcherGraphI *	getImpl() const
public IloDispatcherGraph::Node	getLocation(IloNode node) const
public IloDispatcherGraph::Node	getNode(const IloInt nodeId) const
public void *	getNodeObject(IloDispatcherGraph::Node n)
public IloNum	getOffsetCost(IloNode m, IloNode n, IloNum x, IloNode o, IloNode p, IloNum y, IloVehicle veh, IloDimension2 dim) const
public IloNum	getTurnPenalty(IloDispatcherGraph::Arc from, IloDispatcherGraph::Arc to, IloDimension2 dim)
public IloNum	getVisibility()
public IloDimension2	getVisibilityDim()
public void	loadArcDimensionDataFromFile(const char * filename, IloDimension2 dim)
public void	loadTurnDimensionDataFromFile(const char * fileName, IloDimension2 dim)
public IloBool	nodeExists(const IloInt nodeId) const
public void	setArcCost(IloDispatcherGraph::Arc a, IloDimension2 dim, const IloNum cost)
public void	setArcObject(IloDispatcherGraph::Arc a, void * o)
public void	setLocation(IloNode node, IloDispatcherGraph::Node n)
public void	setNodeObject(IloDispatcherGraph::Node n, void * o)
public void	setTurnPenalty(IloDispatcherGraph::Arc from, IloDispatcherGraph::Arc to, IloDimension2 dim, const IloNum penalty)
public void	setVisibility(IloDimension2 dim, const IloNum vis)
public void	unsetLocation(IloNode node)

Inner Class
IloDispatcherGraph::AdjacencyListIterator
IloDispatcherGraph::Arc
IloDispatcherGraph::Node
IloDispatcherGraph::PathIterator

Constructors and Destructors

`public` **`IloDispatcherGraph`**`(IloEnv env, IloBool storePaths=IloTrue, IloInt preSizeArcs=1, IloInt preSizeNodes=1)`

This constructor creates a graph which stores cheapest paths only if `storePaths` is set to `IloTrue`. Storing cheapest paths is only necessary if you intend to access them through the `IloDispatcherGraph::PathIterator` class. You will greatly reduce memory consumption if you do not store the paths.

The optional parameter `preSizeArcs` can be used to help reduce memory consumption. For example, if you have two arcs with indices of 1 and 13, you would call the constructor with `preSizeArcs=14`. The Dispatcher Graph then generates an array of size 14. If the parameter is left undefined, the array generated will be of size 16.

The optional parameter `preSizeNodes` can be used to help reduce memory consumption. For example, if you have two nodes with indices of 1 and 13, you would call the constructor with `preSizeNodes=14`. The Dispatcher Graph then generates an array of size 14. If the parameter is left undefined, the array generated will be of size 16.

```
public IloDispatcherGraph(IloEnv env, const char * file)
```

This constructor creates a graph that stores the cheapest paths in the file `file`. This constructor is useful if memory is at a premium as the shortest paths will be placed on the disk. However, this will mean that the distance computations and thus virtually all of Dispatcher's primary functions will proceed more slowly.

```
public IloDispatcherGraph(const IloDispatcherGraph & g)
```

This copy constructor creates a handle from a reference to a graph object. That graph object and `g` both point to the same implementation object.

```
public IloDispatcherGraph(IloDispatcherGraphI * impl=0)
```

This constructor creates a handle object (an instance of `IloDispatcherGraph`) from a pointer to an implementation object (an instance of the class `IloDispatcherGraphI`).

Methods

```
public void addArc(IloDispatcherGraph::Node n1, IloDispatcherGraph::Node n2,
IloDimension2 dim, IloNum value)
```

This member function adds an arc between nodes `n1` and `n2` to the graph. Its value according to the dimension `dim` is `value`. If an arc between `n1` and `n2` already exists, its value will be modified. Each dimension is assigned a value separately. If an arc between `n1` and `n2` exists but no value has been given for a dimension `dim`, then its value according to `dim` is `IloInfinity.`

```
public void addDimension2(IloDimension2 dim)
```

This member function adds a dimension to the graph so that arc cost information can be loaded from a `.csv` file using `IloDispatcherGraph::createArcsFromFile`. Each dimension that you want to load must be separately added to the graph.

If you use the member function `IloDispatcherGraph::loadArcDimensionDataFromFile` to load arc cost information for a dimension directly from a file, the member function `IloDispatcherGraph::addDimension2` is automatically called for this dimension.

```
public IloBool arcExists(const IloInt arcId) const
```

Given the identifier of a node, this member function returns `IloTrue` if an `IloDispatcherGraph::Arc` object with this identifier is already present in the graph, and `IloFalse` otherwise.

`public IloBool **arcExistsWithEnds**(const IloInt fromId, const IloInt toId) const`

Given the identifiers of the two endpoints of an arc, this member function returns `IloTrue` if an `IloDispatcherGraph::Arc` from node `fromId` to node `toId` exists in the graph. Note that all arcs in the graph are directed, and the two endpoints must be specified in the correct order, as a from-to pair.

`public void **associateByCoordsInFile**(const IloNode node, const char * fileName)`

When using even modestly sized networks, it may become quite difficult to determine which graph node corresponds to a given `IloNode`, and the use of individual calls to the member function `setLocation` becomes impractical. The member function `associateByCoordsInFile` looks up the coordinates of a given `IloNode` in a `.csv` file, and automatically associates it to the graph node with matching coordinates. The first line in the `.csv` file must contain the items "name", "x", and "y". Subsequent lines must contain an integer denoting a graph node, followed by two floating point numbers giving its coordinates. For more information, see the documentation of `IloCsvReader` in the IBM(R) ILOG(R) Concert Technology documentation.

`public void **createArcsFromFile**(const char * fileName, IloBool loadDims=IloTrue)`

This member function loads the topology of a network from a `.csv` file and creates all necessary arcs and nodes. The `.csv` file must contain the items "arcName", "from", and "to". Each following line must contain three integers to denote the arcName and the two nodes that the arc connects. By default, all turns between consecutive arcs are allowed, at a cost of 0. For more information, see the documentation of `IloCsvReader` in the IBM(R) ILOG(R) Concert Technology documentation.

If the optional parameter `loadDims` is set to the default `IloTrue`, the application will load dimension data as the topology is being read for all dimensions that have already been added to the graph using the member function `IloDispatcherGraph::addDimension2`. If the parameter `loadDims` is set to `IloFalse`, no dimension data will be loaded by this member function. In this case, you can use the member function `IloDispatcherGraph::loadArcDimensionDataFromFile` to directly load arc cost information for a specific dimension from a `.csv` file.

`public void **end**()`

This member function frees all resources used by the invoking graph object. You cannot use the invoking graph object after a call to this member function.

`public void **forbidArcUse**(IloDispatcherGraph::Arc a)`

This member function sets the cost of an arc to infinity for all dimensions currently defined in the graph to which the arc belongs.

`public void **forbidTurn**(IloDispatcherGraph::Arc from, IloDispatcherGraph::Arc to)`

This member function sets to infinity the cost of turning from arc `from` into arc `to` in all dimensions currently defined in the graph to which the arcs belong.

`public IloDispatcherGraph::Arc **getArc**(const IloInt arcId) const`

Given the identifier of an arc, this member function returns the corresponding `IloDispatcherGraph::Arc` object.

```
public IloDispatcherGraph::Arc getArcByEnds(const IloInt fromNodeId, const IloInt
toNodeId) const
```

Given the identifiers of the two endpoints of an arc, this member function returns the corresponding `IloDispatcherGraph::Arc` object. Note that all arcs in the graph are directed, and the two endpoints must be specified in the correct order.

```
public IloNum getArcCost(IloDispatcherGraph::Arc a, IloDimension2 dim)
```

This member function returns the cost of arc `a` along dimension `dim`.

```
public void * getArcObject(IloDispatcherGraph::Arc a)
```

This member function returns a pointer to the object associated to arc `a`, if it exists. Otherwise, it returns 0 (zero).

```
public IloNum getDistance(IloNode n1, IloNode n2, IloVehicle v, IloDimension2 dim)
const
```

This member function returns the value, expressed according to `dim`, of the cheapest path going from `n1` to `n2` using `v`. The value of a path is the sum of the values of the arcs composing the path according to dimension `dim`. The cheapest path is the path whose value minimizes the cost function of vehicle `v`.

```
public IloEnv getEnv() const
```

This member function returns the environment of the invoking graph object.

```
public IloDispatcherGraphI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking graph object.

```
public IloDispatcherGraph::Node getLocation(IloNode node) const
```

This member function retrieves the location of an instance of `IloNode` in the graph.

```
public IloDispatcherGraph::Node getNode(const IloInt nodeId) const
```

Given the identifier of a node, this member function returns the corresponding `IloDispatcherGraph::Node` object.

```
public void * getNodeObject(IloDispatcherGraph::Node n)
```

This member function returns a pointer to the object associated to node `n`, if it exists. Otherwise, it returns 0 (zero).

```
public IloNum getOffsetCost(IloNode m, IloNode n, IloNum x, IloNode o, IloNode p,
IloNum y, IloVehicle veh, IloDimension2 dim) const
```

The purpose of `IloDispatcherGraph::getOffsetCost` is to provide address-to-address costs. The member function `IloDispatcherGraph::getDistance` returns the distance between two `IloNodes`. The new member function `IloDispatcherGraph::getOffsetCost` returns the distance between two real addresses. An address would typically be considered a location within an arc, rather than a location directly at a node.

This member function returns the cost of the shortest path between the two addresses in terms of dimension `dim`, and for vehicle `veh`. The `IloNode m` and the `IloNode n` are the extreme points of the departure arc. The `IloNode o` and the `IloNode p` are the extreme points of the destination arc. `x` and `y` provide the offset coefficients that allow you to locate the exact address inside the origin and destination arcs.

For example, if the distance between `m` and `n` is 1 unit, the location of the departure address is `x*1` units away from `m` in the direction of `n`. If the distance between `o` and `p` is 12 units, the location of the departure address is `y*12` units away from `o` in the direction of `p`.

Note that if you use `getOffsetCost`, the `IloDispatcherGraph` must be created using `storePaths=IloTrue` in the constructor. This naturally increases the amount of memory that the graph will require.

```
public IloNum getTurnPenalty(IloDispatcherGraph::Arc from, IloDispatcherGraph::Arc
to, IloDimension2 dim)
```

This member function returns the turn penalty associated to the movement from arc `from` into arc `to`, along dimension `dim`.

```
public IloNum getVisibility()
```

This member function returns the value of the visibility parameter used in the shortest path computation.

```
public IloDimension2 getVisibilityDim()
```

This member function returns the dimension used to control visibility for the shortest path computation.

```
public void loadArcDimensionDataFromFile(const char * filename, IloDimension2 dim)
```

This member function allows the user to load the arc cost information from a `.csv` file. A single `.csv` file may contain the arc cost information for several dimensions, but the costs for each dimension must be loaded individually through a call to `loadArcDimensionDataFromFile`. The `.csv` file must contain a first line with the items "arcName" and the names of the dimensions in question. Each subsequent line must contain an integer to denote the arcNumber, and a floating point value for the arc cost in terms of each of the dimensions named in the first line.

If you use this member function to load arc cost information for a dimension directly from a file, the member function `IloDispatcherGraph::addDimension2` is automatically called for this dimension.

`public void` **`loadTurnDimensionDataFromFile`**`(const char * fileName, IloDimension2 dim)`

When new arcs are created, the default assumption is that turns to all contiguous arcs are permitted with no penalty. The user may override the turn penalties for specific arcs by loading turn penalty information from a `.csv` file. The first line of such file must contain the items "arc1Name", "arc2Name" and the dimension names. All subsequent lines must contain two integers denoting the arc the vehicle is turning from, and the arc the vehicle is turning into, as well as floating point values to specify the turning penalty in each of the dimensions named in the first line. For more information, see the documentation of `IloCsvReader` in the IBM(R) ILOG(R) Concert Technology documentation.

`public IloBool` **`nodeExists`**`(const IloInt nodeId) const`

Given the identifier of a node, this member function returns `IloTrue` if a `Node` object with this identifier is already present in the graph, and `IloFalse` otherwise.

`public void` **`setArcCost`**`(IloDispatcherGraph::Arc a, IloDimension2 dim, const IloNum cost)`

This member function sets the cost of arc `a`, along dimension `dim`. Negative costs are not accepted.

`public void` **`setArcObject`**`(IloDispatcherGraph::Arc a, void * o)`

This member function associates an object `o` with the arc `a` by passing a pointer to this object.

`public void` **`setLocation`**`(IloNode node, IloDispatcherGraph::Node n)`

This member function positions an instance of `IloNode` in the graph.

`public void` **`setNodeObject`**`(IloDispatcherGraph::Node n, void * o)`

This member function associates an object `o` with the node `n` by passing a pointer to this object.

`public void` **`setTurnPenalty`**`(IloDispatcherGraph::Arc from, IloDispatcherGraph::Arc to, IloDimension2 dim, const IloNum penalty)`

This member function sets the penalty of turning from arc `from` into arc `to`, along dimension `dim`. Negative penalties are not accepted.

`public void` **`setVisibility`**`(IloDimension2 dim, const IloNum vis)`

In a very large graph, it may be convenient to limit the area in which a shortest path computation is performed. When the visibility of the shortest path computation is set to `vis` on dimension `dim`, any nodes located more than `vis` units of `dim` away from the origin node are not considered in the computation. For a given shortest path computation, only one dimension can be used to control the visibility of nodes.

```
public void unsetLocation(IloNode node)
```

This member function removes an instance of `IloNode` from the graph.

Class IloDispatcherNHoodParameters

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This parameter class can be used to modify the default behavior of neighborhoods. Most of the parameters in this class will limit the scope of the neighborhoods in which they are used. This is usually done to improve the performance of the search and potentially the quality of the resulting solutions.

See Also: IloVisitAlternativeSwap, IloCross, IloExchange, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloMergeAndRelocateTours, IloOrOpt, IloRelocate, IloSwapPerform, IloTwoOpt, IloVehicleEquiv, IloVisitAlternativeSwap

Constructor Summary	
`public`	`IloDispatcherNHoodParameters(IloDispatcherNHoodParametersI * impl=0)`
`public`	`IloDispatcherNHoodParameters(const IloDispatcherNHoodParameters & param)`
`public`	`IloDispatcherNHoodParameters(IloEnv env)`

Method Summary	
`public void`	`end()`
`public IloArcPredicate`	`getArcFocusPredicate() const`
`public IloArcPredicate`	`getArcKeeperPredicate() const`
`public IloEnv`	`getEnv() const`
`public IloDispatcherNHoodParametersI *`	`getImpl() const`
`public IloVehicleEquiv`	`getVehicleEquivalence() const`
`public IloVehiclePairPredicate`	`getVehiclePairFocusPredicate() const`
`public IloVehicleArray`	`getVehicles() const`
`public IloVisitArray`	`getVisits() const`
`public IloBool`	`ignorePairs() const`
`public void`	`setArcFocusPredicate(IloArcPredicate arcPredicate)`
`public void`	`setArcKeeperPredicate(IloArcPredicate arcPredicate)`
`public void`	`setIgnorePairs(IloBool ignorePairs)`
`public void`	`setVehicleEquivalence(IloVehicleEquiv vehicleEquiv)`
`public void`	`setVehiclePairFocusPredicate(IloVehiclePairPredicate vehiclePredicate)`
`public void`	`setVehicles(IloVehicleArray vehicles)`
`public void`	`setVisits(IloVisitArray visits)`

Constructors

`public` **`IloDispatcherNHoodParameters`**`(IloDispatcherNHoodParametersI * impl=0)`

This constructor creates a handle object (an instance of `IloDispatcherNHoodParameters`) from a pointer to an implementation object (an instance of `IloDispatcherNHoodParametersI`).

`public` **`IloDispatcherNHoodParameters`**`(const IloDispatcherNHoodParameters & param)`

This copy constructor creates a handle from a reference to a parameter object.

`public` **`IloDispatcherNHoodParameters`**`(IloEnv env)`

This constructor creates a parameter object, allocated on the environment `env`.

Methods

`public void` **`end`**`()`

This member function frees all resources used by the invoking parameter object. You cannot use the invoking parameter object after a call to this member function.

`public IloArcPredicate` **`getArcFocusPredicate`**`() const`

This member function returns an arc predicate object which forbids neighborhoods from accepting new arcs which do not satisfy the predicate.

Currently the neighborhoods which take this parameter into account are `IloCross`, `IloExchange`, and `IloRelocate`.

`public IloArcPredicate` **`getArcKeeperPredicate`**`() const`

This member function returns an arc predicate object which forbids neighborhoods from removing arcs which satisfy the predicate.

Currently the neighborhoods which take this parameter into account are `IloCross`, `IloExchange`, `IloFPRelocate`, `IloOrOpt`, `IloRelocate`, and `IloTwoOpt`.

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment of the invoking parameter object.

`public IloDispatcherNHoodParametersI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking parameter object.

`public IloVehicleEquiv` **`getVehicleEquivalence`**`() const`

This member function returns a vehicle equivalence object which can be used to speed up the search. Neighborhoods using this equivalence object will only take into account non-empty vehicles and a single empty vehicle of each vehicle group according to this equivalence.

Currently the neighborhoods which take this parameter into account are `IloCross`, `IloExchange`, `IloFPRelocate`, `IloMergeAndRelocateTours`, and `IloRelocate`.

`public IloVehiclePairPredicate` **`getVehiclePairFocusPredicate`**`() const`

This member function returns a vehicle pair predicate object which forbids the neighborhoods from performing moves between pairs of vehicle which do not satisfy the predicate.

Currently the neighborhoods which take this parameter into account are `IloCross`, `IloExchange`, `IloFPRelocate`, `IloMergeAndRelocateTours`, and `IloRelocate`.

`public IloVehicleArray` **`getVehicles`**`() const`

This member function returns the array of vehicles on which neighborhoods can perform moves.

Currently the neighborhoods which take this parameter into account are `IloVisitAlternativeSwap`, `IloCross`, `IloExchange`, `IloFPRelocate`, `IloMakePerformed`, `IloMakePerformedPair`, `IloMakeUnperformed`, `IloMergeAndRelocateTours`, `IloOrOpt`, `IloRelocate`, `IloSwapPerform`, and `IloTwoOpt`.

`public IloVisitArray` **`getVisits`**`() const`

This member function returns the array of visits which neighborhoods can move.

Currently the neighborhoods which take this parameter into account are `IloMakePerformed`, `IloMakeUnperformed`, and `IloSwapPerform`.

`public IloBool` **`ignorePairs`**`() const`

If the Boolean value returned by this member function is true then neighborhoods will not move pickup and delivery visits together.

Currently the neighborhoods which take this parameter into account are `IloMakePerformed`, `IloMakeUnperformed`, `IloSwapPerform`, `IloExchange`, and `IloRelocate`.

`public void` **`setArcFocusPredicate`**`(IloArcPredicate arcPredicate)`

The arc predicate `arcPredicate` set by this member function forbids neighborhoods from accepting new arcs which do not satisfy the predicate.

Currently the neighborhoods which take this parameter into account are `IloCross`, `IloExchange`, and `IloRelocate`.

Example

The following code creates an arc predicate which can be passed to `setArcFocusPredicate` to focus move operators on visits which are not located at the same node:

```
ILOPREDICATE0(MyPredicate,
              IloVisitPair, arc) {
  return arc.getVisit1().getStartNode() != arc.getVisit2().getStartNode();
}
```

For more information, see `IloPredicate` and `ILOPREDICATE0`, documented in the *IBM ILOG Solver Reference Manual*.

`public void` **`setArcKeeperPredicate`**`(IloArcPredicate arcPredicate)`

The arc predicate `arcPredicate` set by this member function forbids neighborhoods from removing arcs which satisfy the predicate. The number of visits moved by the neighborhood may grow by including next and previous visits to satisfy the predicate.

Currently the neighborhoods which take this parameter into account are `IloCross`, `IloExchange`, `IloFPRelocate`, `IloOrOpt`, `IloRelocate`, and `IloTwoOpt`.

Example

The following code creates an arc predicate which can be passed to `setArcKeeperPredicate` to prevent move operators from breaking arcs of visits located at the same node:

```
ILOPREDICATE0(MyPredicate,
              IloVisitPair, arc) {
  return arc.getVisit1().getStartNode() == arc.getVisit2().getStartNode();
}
```

For more information, see `IloPredicate` and `ILOPREDICATE0`, documented in the *IBM ILOG Solver Reference Manual*.

`public void` **`setIgnorePairs`**`(IloBool ignorePairs)`

If `ignorePairs` is true, then neighborhoods will not move pickup and delivery visits together.

Currently the neighborhoods which take this parameter into account are `IloMakePerformed`, `IloMakeUnperformed`, `IloSwapPerform`, `IloExchange`, and `IloRelocate`.

`public void` **`setVehicleEquivalence`**`(IloVehicleEquiv vehicleEquiv)`

The vehicle equivalence object `vehicleEquiv` set by this member function can be used to speed up the search. Neighborhoods using this equivalence object will only take into account non-empty vehicles and a single empty vehicle of each vehicle group according to this equivalence.

Currently the neighborhoods which take this parameter into account are `IloCross`, `IloExchange`, `IloFPRelocate`, `IloMergeAndRelocateTours`, and `IloRelocate`.

`public void` **`setVehiclePairFocusPredicate`**`(IloVehiclePairPredicate vehiclePredicate)`

The vehicle pair predicate `vehiclePredicate` set by this member function forbids the neighborhoods from performing moves between pairs of vehicle which do not satisfy the predicate.

Currently the neighborhoods which take this parameter into account are `IloCross`, `IloExchange`, `IloFPRelocate`, `IloMergeAndRelocateTours`, and `IloRelocate`.

Example

The following code creates a vehicle pair predicate which can be passed to `setVehiclePairFocusPredicate` to only permit moves between vehicles which start at the same node:

```
ILOPREDICATE0(MyPredicate,
              IloVehiclePair, pair) {
  return pair.getVehicle1().getFirstVisit().getStartNode()
      == pair.getVehicle2().getFirstVisit().getStartNode();
}
```

For more information, see `IloPredicate` and `ILOPREDICATE0`, documented in the *IBM ILOG Solver Reference Manual*.

`public void` **`setVehicles`**`(IloVehicleArray vehicles)`

The array of vehicles `vehicle` set by this member function is the array of vehicles on which neighborhoods can perform moves.

Currently the neighborhoods which take this parameter into account are `IloVisitAlternativeSwap`, `IloCross`, `IloExchange`, `IloFPRelocate`, `IloMakePerformed`, `IloMakePerformedPair`, `IloMakeUnperformed`, `IloMergeAndRelocateTours`, `IloOrOpt`, `IloRelocate`, `IloSwapPerform`, and `IloTwoOpt`.

`public void` **`setVisits`**`(IloVisitArray visits)`

The array of visits `visits` set by this member function is the array of visits which neighborhoods can move.

Currently the neighborhoods which take this parameter into account are `IloMakePerformed`, `IloMakeUnperformed` and `IloSwapPerform`.

Class IloDispatcherTabuSearch

Definition file: ildispat/ilometa.h
Include file: <ildispat/ilodispatcher.h>

This class implements a metaheuristic for routing problems based upon tabu search. The principal idea of tabu search is that exploration of the search space should be encouraged by discouraging the re-visiting of previously explored areas. The most common way of doing this is by associating a “tabu status” to aspects of the solution.

Dispatcher's tabu search metaheuristic is based upon this simple idea. Each time a move is made, the arcs that have been added to the solution by the move are kept in a *keep list*, while the arcs that have been removed are added to a *forbid list*. They remain on these lists for a set number of moves, known as their tenure. This tenure can be controlled by the user. For Dispatcher, useful values for the tabu tenure start from 5 upwards, but can vary widely.

Whenever a move is examined by Dispatcher's tabu search, it looks at the new arcs added to the solution and at the old arcs that leave the solution. It then counts the number of new arcs that appear on the forbid list and the number of old (leaving) arcs that appear on the keep list. The sum of these is called the *tabu number* of the move. If the tabu number is above a certain value (which varies according to the move type), the move is declared tabu and rejected. This rejection can be overridden in one case: when the cost of this move is better than the best cost solution visited so far. This meta-rule to the tabu rule is known as the *aspiration criterion*.

It is worth noting that the tabu search metaheuristic does not work well when the first neighborhood move is taken at each stage in the search. (Normally, the `IloMinimizeVar` selector is used.) The `IloCross` neighborhood can also hamper the metaheuristic as it can provide a huge number of equal cost moves that can be performed, requiring large tabu tenures to avoid cycling.

The following example shows how `IloDispatcherTabuSearch` can be used to search for a solution to a routing problem:

```
void ImproveWithTabu(IloDispatcher dispatcher,
                     IloRoutingSolution solution,
                     IloNHood nhood) {
  IloNumVar cost = dispatcher.getCostVar();
  IloEnv env = dispatcher.getEnv();
  IloGoal instantiateCost = IloDichotomize(env, cost, IloFalse);
  IloRoutingSolution rsol = solution.makeClone(env);
  IloRoutingSolution best = solution.makeClone(env);
  IloDispatcherTabuSearch dts(env, 12);

  IloGoal move = IloSingleMove(env,
                               rsol,
                               nhood,
                               dts,
                               IloMinimizeVar(env, cost),
                               instantiateCost);
  move = move && IloStoreBestSolution(env, best);
  IloSolver solver = dispatcher.getSolver();
  for (IloInt i = 0; i < 150; i++) {
    if (solver.solve(move)) {
      cout << "Cost = " << solver.getMax(cost) << endl;
    }
    else {
      if (dts.complete()) break;
    }
  }
  IloGoal restoreSolution = IloRestoreSolution(env, best) &&
                            instantiateCost;
  solver.solve(restoreSolution);
  rsol.end();
  best.end();
}
```

For more information, see the class `IloMetaHeuristic` in the *IBM ILOG Solver Reference Manual*.

See Also: IloDispatcherGLS

Constructor Summary	
`public`	`IloDispatcherTabuSearch(IloEnv env, IloInt tenure)`

Method Summary	
`public IloBool`	`complete()`
`public IloInt`	`getTenure() const`
`public IloBool`	`isFeasible(IloSolver solver, IloSolution delta) const`
`public void`	`notify(IloSolver solver, IloSolution delta)`
`public void`	`reset()`
`public void`	`setTenure(IloInt tenure)`
`public void`	`start(IloSolver solver, IloSolution delta)`
`public IloBool`	`test(IloSolver solver, IloSolution delta)`

Constructors

`public` **`IloDispatcherTabuSearch`**`(IloEnv env, IloInt tenure)`

This constructor creates a metaheuristic that performs tabu search for a routing problem. You can use the parameter `tenure` to specify a certain number of moves; any individual arc will be held on either the keep list or the forbid list for the specified number of moves.

Methods

`public IloBool` **`complete`**`()`

This member function ages the tabu tenure of all tabu arcs by one iteration (removing the tabu status for any at 0) in the hope that this will allow some moves to be performed on the next iteration. It returns `IloFalse`.

`public IloInt` **`getTenure`**`() const`

This member function returns the tenure specified in the constructor set in the last call to `setTenure`.

`public IloBool` **`isFeasible`**`(IloSolver solver, IloSolution delta) const`

This member function performs pre-filtering of changes to the routing solution in accordance with the tabu arcs. If `delta` violates the tabu criterion and has a cost at least as great as the best solution seen, `IloFalse` is returned. `IloTrue` is returned in all other cases.

`public void` **`notify`**`(IloSolver solver, IloSolution delta)`

This member function first ages the tabu tenure of all tabu arcs by one iteration, removing the tabu status from any at 0. It then examines the new arcs added to the solution and the old arcs removed from the solution. The old arcs are placed on the forbid list with a tabu tenure dictated by `getTenure`. The new arcs are placed on the keep list with the same tenure.

`public void` **`reset`**`()`

This member function removes the tabu status of all arcs and resets the record of the best solution seen.

`public void` **`setTenure`**`(IloInt tenure)`

This member function sets the length of the tabu tenure to `tenure`.

`public void` **`start`**`(IloSolver solver, IloSolution delta)`

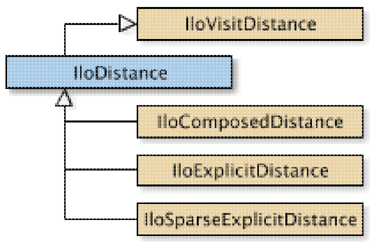
This member function updates its record of the best solution seen so far if `delta` has a cost lower than that record. The tabu status of any move is overridden if it leads to a solution of a cost less than that value.

`public IloBool` **`test`**`(IloSolver solver, IloSolution delta)`

This member function computes the difference in terms of arcs between the currently instantiated solution and the solution passed in `start`. If examination of the arcs that have appeared and the arcs that have left indicates that the move is tabu, and the cost of this solution is not less than the best solution seen, a failure results. Otherwise, this method returns `IloTrue`.

Class IloDistance

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher lets you define the distance function for a *dimension* (for example, the distance, the time, or the cost necessary for going from one node to the other).

This class is the handle class of the object that defines this distance function.

This handle class uses the virtual member function `IloDistanceI::computeDistance` to retrieve distance values.

See Also: IloDimension, IloDimension1, IloDimension2, IloDistanceEvalI, IloDistanceFunction, IloDistanceI, IloSimpleDistanceEvalI, IloSimpleDistanceFunction, IloComposedDistance, IloExplicitDistance, IloSparseExplicitDistance

Constructor and Destructor Summary	
`public`	`IloDistance(IloDistanceI * dist=0)`
`public`	`IloDistance(const IloDistance & dist)`
`public`	`IloDistance(IloEnv env, IloDistanceFunction distFunction)`
`public`	`IloDistance(IloEnv env, IloSimpleDistanceFunction distFunction)`
`public`	`IloDistance(IloDistanceFunction distFunction, IloVehicleEquiv equiv)`

Method Summary	
`public void`	`end()`
`public static IloBool`	`Exists(IloEnv env, const char * key)`
`public static IloDistance`	`Find(IloEnv env, const char * key)`
`public IloNum`	`getDistance(IloNode node1, IloNode node2, IloVehicle vehicle) const`
`public IloInt`	`getGroup(IloVehicle vehicle) const`
`public IloDistanceI *`	`getImpl() const`
`public const char *`	`getKey()`
`public void`	`refresh() const`
`public void`	`removeKey()`
`public void`	`setCache(IloEnv env, IloInt log2Rows, IloInt log2Cols) const`
`public void`	`setKey(const char * key)`
`public void`	`unsetCache() const`

Inherited Methods from `IloVisitDistance`
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setKey`

Constructors and Destructors

```
public IloDistance(IloDistanceI * dist=0)
```

This constructor creates a handle object (an instance of `IloDistance`) from a pointer to an object (an instance of the implementation class `IloDistanceI`).

```
public IloDistance(const IloDistance & dist)
```

This copy constructor creates a handle from a reference to a distance object. That distance object and `dist` both point to the same implementation object.

When distance functions are specified in Dispatcher, they can be cached, if distance computations are slow, through `IloDimension2::setCached`. (Normally, caching of distance functions is disabled.) When the distance depends not only on the starting and ending node, but the vehicle used to perform the trip, it becomes useful to introduce the notion of *vehicle equivalence*.

If two vehicles are specified as equivalent with respect to a particular distance metric and the distance between two nodes using one of the vehicles resides in the cache, the distance between the same two nodes using the other vehicle can be assumed to be the same. Thus, fewer cache slots are used. It is functionally unnecessary to specify a vehicle equivalence class in Dispatcher, but definition of such a class can lead to speed increases through better caching of distance data.

```
public IloDistance(IloEnv env, IloDistanceFunction distFunction)
```

This constructor creates a distance object in the environment `env`. The implementation object of the newly created handle is an instance of the class `IloDistanceEvalI` constructed with the distance function `distFunction`.

```
public IloDistance(IloEnv env, IloSimpleDistanceFunction distFunction)
```

This constructor creates a distance object in the environment `env`. The implementation object of the newly created handle is an instance of the class `IloSimpleDistanceEvalI` constructed with the distance function `distFunction`.

```
public IloDistance(IloDistanceFunction distFunction, IloVehicleEquiv equiv)
```

This constructor creates a handle to a distance object. The implementation object of this newly created handle is an instance of the class `IloDistanceEvalI` constructed with the distance function `distFunction` for the vehicle equivalence group equiv.

Methods

```
public void end()
```

This member function frees all resources used by the invoking distance object. You cannot use the invoking distance object after a call to this member function.

`public static IloBool` **`Exists`**`(IloEnv env, const char * key)`

This static member function returns `IloTrue` if an `IloDistance` object having key `key` exists and `IloFalse` if not.

`public static IloDistance` **`Find`**`(IloEnv env, const char * key)`

This static member function returns the object corresponding to the key `key` set using `IloDistance::setKey`. If there is no object corresponding to `key` an `IloException` is thrown.

`public IloNum` **`getDistance`**`(IloNode node1, IloNode node2, IloVehicle vehicle) const`

This member function returns the distance from `node1` to `node2` using vehicle `vehicle`.

`public IloInt` **`getGroup`**`(IloVehicle vehicle) const`

This member function returns the group as specified by the vehicle equivalence object associated with `vehicle`.

`public IloDistanceI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking distance.

`public const char *` **`getKey`**`()`

The following member function returns the key set on the invoking object

`public void` **`refresh`**`() const`

This member function flushes any internal caches on the distance function, and uses any vehicle equivalence class specified in the constructor to update the group of each vehicle.

This member function thus allows the distance function to be changed. This means that after a call to `refresh`, `IloDistanceI::computeDistance` can return a different value for the same three parameters than before the call to `refresh`. However, the new distance function must be as consistent as the old one in that successive calls using the same parameters must produce the same distance value.

`public void` **`removeKey`**`()`

This member function allows the user to remove the key set on the invoking object.

`public void` **`setCache`**`(IloEnv env, IloInt log2Rows, IloInt log2Cols) const`

This member function adds a cache to the invoking distance object so that distance computations can be cached. The parameter `env` indicates the environment within which the distance object is used. The cache is set-associative with *2log2Rows* rows, and a set-associative width of *2log2Cols*.

The method `IloDimension2::setCached` uses this member function to add a cache of size `log2rows` = 18 and `log2cols` = 0 to the distance object associated with the invoking dimension. No cache is added if one already exists.

```
public void setKey(const char * key)
```

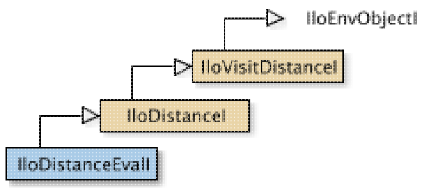
This member function allows the user to set `key` on the invoking object. This key is unique. Each distance must have a different key; otherwise, an exception is thrown.

```
public void unsetCache() const
```

This member function stops caching of distance values.

Class IloDistanceEvalI

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher lets you define the distance function for a dimension (for example, the distance, the time, or the cost necessary for going from one node to another).

This class is an implementation class, a predefined subclass of `IloDistanceI`, that you use to define a new distance function expressed by an evaluation function. This evaluation function is of type `IloDistanceFunction`.

See Also: IloDimension, IloDimension1, IloDimension2, IloDistance, IloDistanceFunction, IloDistanceI

Constructor and Destructor Summary	
`public`	`IloDistanceEvalI(IloEnv env, IloDistanceFunction distFunction)`
`public`	`IloDistanceEvalI(IloDistanceFunction distFunction, IloVehicleEquiv equiv)`

Method Summary	
`public virtual IloNum`	`computeDistance(IloNode node1, IloNode node2, IloVehicle vehicle) const`

Inherited Methods from `IloDistanceI`
`computeDistance, computeDistance, getDistance, getGroup, refresh, setCache, unsetCache, updateEquivalence`

Inherited Methods from `IloVisitDistanceI`
`computeDistance, getDistance, getGroup, refresh, setCache, unsetCache, updateEquivalence`

Constructors and Destructors

`public` **`IloDistanceEvalI`**`(IloEnv env, IloDistanceFunction distFunction)`

This constructor creates a new distance function from the evaluation function `distFunction`.

When distance functions are specified in Dispatcher, they can be cached, if distance computations are slow, through `IloDimension2::setCached`. (Normally, caching of distance functions is disabled.) When the distance depends not only on the starting and ending node, but the vehicle used to perform the trip, it becomes useful to introduce the notion of *vehicle equivalence*.

If two vehicles are specified as equivalent with respect to a particular distance metric and the distance between two nodes using one of the vehicles resides in the cache, the distance between the same two nodes using the other vehicle can be assumed to be the same. Thus, fewer cache slots are used. It is functionally unnecessary to specify a vehicle equivalence class in Dispatcher, but definition of such a class can lead to speed increases using better caching of distance data.

```
public IloDistanceEvalI(IloDistanceFunction distFunction, IloVehicleEquiv equiv)
```

This constructor creates a new distance function for the vehicle equivalence group `equiv` from the evaluation function `distFunction`.

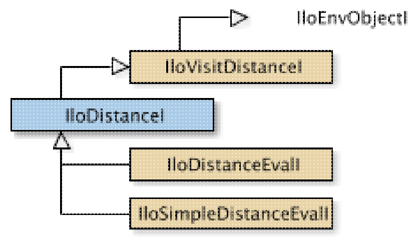
Methods

```
public virtual IloNum computeDistance(IloNode node1, IloNode node2, IloVehicle
vehicle) const
```

This member function returns a numeric value that represents the distance between `node1` and `node2` for the given `vehicle`. This is done using a call to `distFunction` passing `node1`, `node2`, and `vehicle` as parameters.

Class IloDistanceI

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher lets you define the distance function for a dimension (for example, the distance, the time, or the cost necessary for going from one node to another).

This class is the implementation class for `IloDistance`, the class of object that defines a distance function for a dimension. The virtual member function `IloDistanceI::computeDistance` returns the distance between two nodes.

To express new distance functions, you can define a subclass of `IloDistanceI`. If this distance can be expressed by an evaluation function, you can use the predefined subclasses `IloDistanceEvalI` or `IloSimpleDistanceEvalI` for that purpose.

See Also: IloDimension, IloDimension1, IloDimension2, IloDistance, IloVisitDistance

Constructor and Destructor Summary	
`public`	`IloDistanceI(IloEnv env, IloBool symmetric=IloFalse)`
`public`	`IloDistanceI(IloVehicleEquiv equiv, IloBool symmetric=IloFalse)`

Method Summary	
`public virtual IloNum`	`computeDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const`
`public virtual IloNum`	`computeDistance(IloNode node1, IloNode node2, IloVehicle vehicle) const`
`public virtual IloNum`	`getDistance(IloNode node1, IloNode node2, IloVehicle vehicle) const`
`public virtual IloInt`	`getGroup(IloVehicle vehicle) const`
`public virtual void`	`refresh()`
`public virtual void`	`setCache(IloEnv env, IloInt log2Rows, IloInt log2Cols)`
`public virtual void`	`unsetCache()`
`public virtual void`	`updateEquivalence()`

Inherited Methods from `IloVisitDistanceI`
`computeDistance, getDistance, getGroup, refresh, setCache, unsetCache, updateEquivalence`

Constructors and Destructors

`public **IloDistanceI**(IloEnv env, IloBool symmetric=IloFalse)`

This constructor creates an implementation distance object in the environment `env`.

When distance functions are specified in Dispatcher, they can be cached, if distance computations are slow, through `IloDimension2::setCached`. (Normally, caching of distance functions is disabled.) When the distance depends not only on the starting and ending node, but also on the vehicle used to perform the trip, it becomes useful to introduce the notion of *vehicle equivalence*.

If two vehicles are specified as equivalent with respect to a particular distance metric and the distance between two nodes using one of the vehicles resides in the cache, the distance between the same two nodes using the other vehicle can be assumed to be the same. Thus, fewer cache slots are used. It is functionally unnecessary to specify a vehicle equivalence class in Dispatcher, but definition of such a class can lead to speed increases through better caching of distance data.

`public **IloDistanceI**(IloVehicleEquiv equiv, IloBool symmetric=IloFalse)`

This constructor creates an implementation distance object for the vehicle equivalence group `equiv`.

Methods

```
public virtual IloNum computeDistance(IloVisit visit1, IloVisit visit2, IloVehicle
vehicle) const
```

You redefine this pure virtual member function to return a floating-point value that represents the distance from `visit1` to `visit2`. The return value of the function must depend only on `visit1` and `visit2`, and must produce the same value for each call with the same parameters.

```
public virtual IloNum computeDistance(IloNode node1, IloNode node2, IloVehicle
vehicle) const
```

You redefine this pure virtual member function to return a floating-point value that represents the distance from `node1` to `node2`. The return value of the function must depend only on `node1` and `node2`, and must produce the same value for each call with the same parameters.

```
public virtual IloNum getDistance(IloNode node1, IloNode node2, IloVehicle vehicle)
const
```

This member function returns the distance from `node1` to `node2`, using vehicle `vehicle`. If caching is enabled, this member function first searches the cache for the distance value. If the value is not found, this function calls `IloDistanceI::computeDistance`, returns the value obtained, and places that value into the cache.

```
public virtual IloInt getGroup(IloVehicle vehicle) const
```

This member function returns the vehicle equivalence group for `vehicle`. In the case where the invoking object was constructed with only an `IloEnv`, this function returns zero. Otherwise, it returns the group as specified by the vehicle equivalence object associated with the implementation.

```
public virtual void refresh()
```

This member function flushes any internal caches on the distance function, and uses any vehicle equivalence class specified in the constructor to update the group of each vehicle.

This member function thus allows the distance function to be changed. This means that after a call to `refresh`, `IloDistanceI::computeDistance` can return a different value for the same three parameters than before the call to `refresh`. However, the new distance function must be as consistent as the old one in that successive calls using the same parameters must produce the same distance value.

```
public virtual void setCache(IloEnv env, IloInt log2Rows, IloInt log2Cols)
```

This member function adds a cache to the invoking distance object so that distance computations can be cached. The parameter `env` indicates the environment upon which the distance object is allocated. The cache is set-associative with *2log2Rows* rows, and a set-associative width of *2log2Cols*.

The method `IloDimension2::setCached` uses this member function to add a cache to the distance object associated with the invoking dimension. No cache is added if one already exists.

```
public virtual void unsetCache()
```

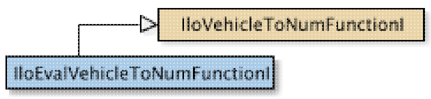
This member function stops caching of distance values.

```
public virtual void updateEquivalence()
```

This member function updates the vehicle equivalence group associated with the invoking distance object.

Class IloEvalVehicleToNumFunctionI

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>



This class is an implementation class, a predefined subclass of `IloVehicleToNumFunctionI`, that you use to define a new vehicle to `IloNum` function expressed by an evaluation function. This evaluation function is of type `IloSimpleVehicleToNumFunction`.

Constructor and Destructor Summary	
`public`	`IloEvalVehicleToNumFunctionI(IloEnv env, IloSimpleVehicleToNumFunction f)`

Method Summary	
`public virtual IloNum`	`getValue(IloVehicle vehicle)`

Inherited Methods from `IloVehicleToNumFunctionI`
`getUnperformedValue, getValue`

Constructors and Destructors

`public` **`IloEvalVehicleToNumFunctionI`**`(IloEnv env, IloSimpleVehicleToNumFunction f)`

This constructor creates a new vehicle to `IloNum` function from the evaluation function `f`.

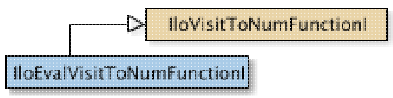
Methods

`public virtual IloNum` **`getValue`**`(IloVehicle vehicle)`

This member function returns a numeric value corresponding to `vehicle`. This is done via a call to `f` passing `vehicle` as a parameter.

Class IloEvalVisitToNumFunctionI

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>



This class is an implementation class, a predefined subclass of `IloVisitToNumFunctionI`, that you use to define a new visit to `IloNum` function expressed by an evaluation function. This evaluation function is of type `IloSimpleVisitToNumFunction`.

Constructor and Destructor Summary	
`public`	`IloEvalVisitToNumFunctionI(IloEnv env, IloSimpleVisitToNumFunction f)`

Method Summary	
`public virtual IloNum`	`getValue(IloVisit visit)`

Inherited Methods from **`IloVisitToNumFunctionI`**
`getUnperformedValue, getValue`

Constructors and Destructors

`public` **`IloEvalVisitToNumFunctionI`**`(IloEnv env, IloSimpleVisitToNumFunction f)`

This constructor creates a new visit to `IloNum` function from the evaluation function `f`.

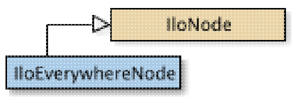
Methods

`public virtual IloNum` **`getValue`**`(IloVisit visit)`

This member function returns a numeric value corresponding to `visit`. This is done via a call to `f` passing `visit` as parameter.

Class IloEverywhereNode

Definition file: ildispat/ilonode.h
Include file: <ildispat/ilodispatcher.h>



This class represents an "everywhere" node, which has the special property that all distances measured to or from this node are zero.

For more information, see the concept Dimensions.

See Also: IloDimension2, IloDistance, IloNode

Inherited Methods from `IloNode`
`Exists, Find, getDistanceTo, getEnv, getKey, getName, getObject, getX, getY, getZ, isEverywhere, removeKey, setKey, setName, setObject`

Class IloExecutionWindowsToVisitCon

Definition file: ildispat/iloproto.h
Include file: <ildispat/ilodispatcher.h>

The constraint `IloExecutionWindowsToVisitCon` relates an instance of `IloDimensionWindows` to a specific `IloVisit`. It constrains a visit to be started after the lower bound of the dimension windows interval and to be ended before the upper bound of the dimension windows interval.

See Also: IloDimensionWindows

Constructor Summary	
`public`	`IloExecutionWindowsToVisitCon(IloVisit visit, IloDimensionWindows window, const char * name=0)`

Method Summary	
`public IloVisit`	`getVisit() const`
`public IloDimensionWindows`	`getWindow() const`

Constructors

`public` **`IloExecutionWindowsToVisitCon`**`(IloVisit visit, IloDimensionWindows window, const char * name=0)`

This constructor creates a dimension windows constraint `window` on visit `visit`. The optional argument `name`, if provided, becomes the name of the constraint.

Methods

`public IloVisit` **`getVisit`**`() const`

This member function returns the visit associated with the constraint `IloExecutionWindowsToVisitCon`.

`public IloDimensionWindows` **`getWindow`**`() const`

This member function returns the window associated with the constraint `IloExecutionWindowsToVisitCon`.

Class IloExplicitArcPredicate

Definition file: ildispat/iloarcpredicate.h
Include file: <ildispat/ilodispatcher.h>

This subclass of `IloArcPredicate` allows you to specify the satisfaction matrix explicitly using `IloExplicitArcPredicate::setValue` or `IloExplicitArcPredicate::setValues`. If the satisfaction value is not defined for an arc then a default is returned.

For more information, see the class `IloPredicate` documented in the *IBM ILOG Solver Reference Manual*.

See Also: IloArcPredicate

Constructor Summary	
`public`	`IloExplicitArcPredicate(IloEnv env, IloBool defaultValue=IloTrue)`

Method Summary	
`public void`	`setValue(IloVisitPair arc, IloBool value)`
`public void`	`setValues(IloArcPredicate predicate)`

Constructors

`public` **`IloExplicitArcPredicate`**`(IloEnv env, IloBool defaultValue=IloTrue)`

This constructor creates an explicit arc predicate in the environment `env`. The parameter `defaultValue` allows you to specify whether the predicate is satisfied or not when no satisfaction value has been specified for an arc.

Methods

`public void` **`setValue`**`(IloVisitPair arc, IloBool value)`

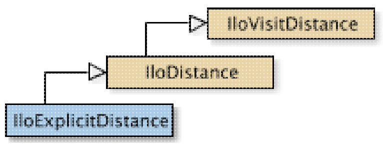
This member function specifies that the arc between visits `arc.getVisit1()` and `arc.getVisit2()` satisfies the invoking predicate if `value` is set to `IloTrue`. Otherwise, it specifies that this arc does not satisfy the predicate.

`public void` **`setValues`**`(IloArcPredicate predicate)`

This member function uses the invoking explicit arc predicate to cache the behavior of `predicate`. The invoking predicate and `predicate` then accept the same arcs. This member function can be used to speed up the calls to `IloArcPredicate::operator()` at the cost of an increased memory usage.

Class IloExplicitDistance

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>



This distance class allows the user to specify the distance matrix explicitly using the member function IloExplicitDistance::setValue. If a distance between two nodes is not specified, a default value is returned.

See Also: IloDistance, IloComposedDistance, IloSparseExplicitDistance

Constructor and Destructor Summary	
`public`	`IloExplicitDistance(IloEnv env, IloNum defaultValue=IloInfinity, IloInt size=0)`

Method Summary	
`public void`	`setValue(IloNode node1, IloNode node2, IloNum value)`

Inherited Methods from `IloDistance`
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setCache, setKey, unsetCache`

Inherited Methods from `IloVisitDistance`
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setKey`

Constructors and Destructors

```
public IloExplicitDistance(IloEnv env, IloNum defaultValue=IloInfinity, IloInt
size=0)
```

This constructor creates an explicit distance object in the environment `env`. The parameter `defaultValue` allows you to specify the value that will be returned when no actual distance between two nodes has been specified. Infinity is returned only if you do not override it with a value of your choice.

The parameter `size` is used to pre-size the distance matrix in order to save memory (automatic resizing consumes more memory than an appropriate pre-sizing).

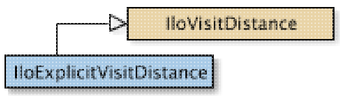
Methods

```
public void setValue(IloNode node1, IloNode node2, IloNum value)
```

This member function sets the explicit distance `value` between two nodes, `node1` and `node2`.

Class IloExplicitVisitDistance

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>



This distance class allows the user to specify the distance matrix explicitly using the member function IloExplicitVisitDistance::setValue. If a distance between two visits is not specified, then a default distance value is returned.

See Also: IloVisitDistance, IloComposedVisitDistance, IloSparseExplicitVisitDistance

Constructor and Destructor Summary	
`public`	`IloExplicitVisitDistance(IloEnv env, IloNum defaultValue=IloInfinity, IloInt size=0)`

Method Summary	
`public void`	`setValue(IloVisit visit1, IloVisit visit2, IloNum value)`

Inherited Methods from `IloVisitDistance`
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setKey`

Constructors and Destructors

`public` **`IloExplicitVisitDistance`**`(IloEnv env, IloNum defaultValue=IloInfinity, IloInt size=0)`

This constructor creates an explicit distance object in the environment `env`. The parameter `defaultValue` allows you to specify the value that will be returned when no actual distance between two visits has been specified. Infinity is returned only if you do not override it with a value of your choice.

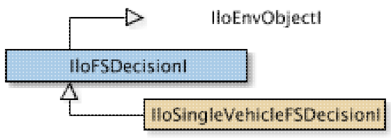
Methods

`public void` **`setValue`**`(IloVisit visit1, IloVisit visit2, IloNum value)`

This member function sets the explicit distance `value` between two visits, `visit1` and `visit2`.

Class IloFSDecisionI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher provides an open framework to define custom first solution heuristics. A first solution heuristic builds a routing plan by assigning visits (or groups of visits) to vehicles. It does so by considering elementary decisions, and either rejecting them as infeasible, or committing them into the solution that is being built.

The abstract class `IloFSDecisionI` is the base class for all first solution decisions that are handled by the framework. Dispatcher provides decision classes to support a nearest-addition type of first solution (FS) generation, but you can define your own class of decision to implement a custom FS algorithm.

Constructor and Destructor Summary	
`public`	`IloFSDecisionI(IloEnv env)`

Method Summary	
`public static IloInt`	`Compare(const char * s1, const char * s2)`
`public static IloInt`	`Compare(IloNum x1, IloNum x2, IloNum prec)`
`public static IloInt`	`Compare(IloInt x1, IloInt x2)`
`public virtual void`	`display(ostream & out) const`
`public IloEnv`	`getEnv() const`
`public virtual IlcGoal`	`getJustifierGoal(IloFSDecisionMakerI * dm)`
`public virtual IlcGoal`	`getRouteCompletionGoal(IloFSDecisionMakerI * dm)`
`public static IloBool`	`IsArcFeasibleOnDimension(IloDispatcher dispatcher, IloDimension dim, IloVehicle vehicle, IloVisit v1, IloVisit v2)`
`public virtual IloBool`	`isBetterThan(IloFSDecisionI * decision, const IloFSDecisionMakerI * dm) const`
`public virtual void`	`make(IloFSDecisionMakerI * dm)`
`public virtual void`	`store(IloFSDecisionMakerI * dm)`

Constructors and Destructors

`public` **`IloFSDecisionI`**`(IloEnv env)`

This constructor creates a decision in the environment `env`. As this class is an abstract class, the constructor is declared as protected.

Methods

`public static IloInt` **`Compare`**`(const char * s1, const char * s2)`

This static member function compares two strings in a lexicographic order, returning -1 if `s1` is lexicographically before `s2`, 1 if `s1` is lexicographically after `s2`, and 0 if they are identical. Non-null strings are always preferred to null strings.

This static comparison function is provided as a convenience for writing `isBetterThan` member functions.

```
public static IloInt Compare(IloNum x1, IloNum x2, IloNum prec)
```

This static member function compares two floating point numbers, with a given precision `prec`. It returns -1 when `x1` is less than or equal to `x2-prec`, 1 when `x1` is greater than or equal to `x2+prec`, and 0 otherwise.

This static comparison function is provided as a convenience for writing `isBetterThan` member functions.

```
public static IloInt Compare(IloInt x1, IloInt x2)
```

This static member function compares two integers and returns -1 when `x1` is less than `x2`, 1 when `x1` is greater than `x2`, or 0 if they are equal.

This static comparison function is provided as a convenience for writing `isBetterThan` member functions.

```
public virtual void display(ostream & out) const
```

This virtual member function displays the decision object.

```
public IloEnv getEnv() const
```

This member function returns the environment `env` on which the decision object was built.

```
public virtual IlcGoal getJustifierGoal(IloFSDecisionMakerI * dm)
```

This member function returns the justifier goal that is used to check the feasibility of the decision. This goal deals with nonrouting aspects of the route, typically scheduling start times of visits on the route, if scheduling is required. The default implementation of this member function returns 0 (empty goal).

```
public virtual IlcGoal getRouteCompletionGoal(IloFSDecisionMakerI * dm)
```

This pure virtual member function returns the goal used to close the route(s) touched by the decision. This goal is used in validating the decision. For simple decisions involving only one visit and one vehicle, the standard `IloGenerateRoute` Dispatcher goal is used. However, decisions could involve several visits and this goal might also involve several vehicles.

```
public static IloBool IsArcFeasibleOnDimension(IloDispatcher dispatcher,
IloDimension dim, IloVehicle vehicle, IloVisit v1, IloVisit v2)
```

This static function checks whether an arc `v1`, `v2` is feasible without violating the constraints linking the cumul and transit variables on `dim`.

For example, if a truck has to start between 5 AM and 6 AM and reach a destination where opening hours are between 8 AM and 10 AM, and if the driving time between the two locations is 6 hours, then this method will detect that the arc is unfeasible.

`public virtual IloBool` **`isBetterThan`**`(IloFSDecisionI * decision, const IloFSDecisionMakerI * dm) const`

This pure virtual member function is used in the building of the first solution to define the selection ordering of decisions (better decisions are considered first). When you define your own classes of decisions, you have to redefine this member function.

This member function returns `IloTrue` if the invoking decision is considered better than `decision`. Note that this result is valid in the current context of building a first solution. Accessing all dynamic values in the search can be done through the `IloDispatcher`, encapsulated by the `IloFSDecisionMaker` class (see `IloFSDecisionMakerI::getDispatcher` method).

Note that the semantics of this member function is similar to an `operator <`. The relation induced by this member function must be nonreflexive and antisymmetric. In other words, if `d1.isBetterThan(d2)` is true, then `d2.isBetterThan(d1)` must be false.

`public virtual void` **`make`**`(IloFSDecisionMakerI * dm)`

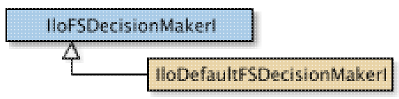
This pure virtual member function defines the actual semantics of the decision. It is called after a decision has been selected and checked for feasibility. If you define your own class of decision, you have to redefine this member function.

`public virtual void` **`store`**`(IloFSDecisionMakerI * dm)`

This pure virtual member function is responsible for storing the decision in a decision maker. Typically, it registers the decision with the visits and vehicles that are involved with the decision, and possibly as a global decision.

Class IloFSDecisionMakerI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>



This class is an abstract subclass of `IlcGoalI`, dedicated to the building of Dispatcher first solutions. A decision maker builds a Dispatcher solution by selecting, testing, and performing elementary decision objects (instances of the abstract class `IloFSDecisionI`). More precisely, the decision maker class is responsible for:

- creating decision objects at the beginning of the First Solution process
- storing, organizing, and selecting decision objects
- checking the feasibility of a particular decision

A decision maker is a goal, and can be used wherever a goal can. The `execute` member function of a decision maker calls first the `init` member function to initialize its decision objects, and then starts a decision-handling loop which is at the root of the FS framework. The decision-handling loop performs the following operations:

1. Search for the best decision among decisions to be considered. If none is found, the algorithm terminates.
2. Test whether this decision is feasible: if not, then remove it from the decisions to be considered and go back to Step 1.
3. Commit the decision. In other words, call its `make` member function, and call `commit` on the decision maker with the decision as an argument to perform side actions, if necessary. Then go back to Step 1.

Constructor Summary	
`public`	`IloFSDecisionMakerI(IloDispatcher dispatcher)`

Method Summary	
`public virtual void`	`commit(IloFSDecisionI * decision)`
`public virtual IloFSDecisionI *`	`getBestDecision()`
`public IloDispatcher`	`getDispatcher() const`
`public IloEnv`	`getEnv() const`
`public IloFSDecisionTracerI *`	`getTracer() const`
`public virtual void`	`init()`
`public virtual IloBool`	`isLegal(IloFSDecisionI * decision)`
`public void`	`registerGlobalDecision(IloFSDecisionI * decision)`
`public void`	`registerVehicleDecision(IloFSDecisionI * decision, IloVehicle vehicle)`
`public void`	`registerVisitDecision(IloFSDecisionI * decision, IloVisit visit)`
`public void`	`setTracer(IloFSDecisionTracerI * tracer)`
`public void`	`storeDecision(IloFSDecisionI * decision)`

Constructors

`public` **`IloFSDecisionMakerI`**`(IloDispatcher dispatcher)`

This is the constructor for the base decision maker class. This class is an abstract class, so this constructor should never be called directly, but from a derived class constructor. As decision makers are sub-classes of `IlcGoalI`, they are allocated on the underlying solver heap, not on the `IloEnv`.

Methods

`public virtual void` **`commit`**`(IloFSDecisionI * decision)`

This member function is called when a decision has been successfully selected and tested, and its `make` member function has been called. The default implementation does nothing.

`public virtual IloFSDecisionI *` **`getBestDecision`**`()`

This pure virtual member function returns the best decision among the decisions that have not yet been considered. It returns 0 if all decisions have been considered, and the first solution building process is complete.

`public IloDispatcher` **`getDispatcher`**`() const`

This member function returns the IloDispatcher object attached to the decision maker. Note that, as an `IlcGoal`, a decision maker inherits from a `getSolver` member function.

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment attached to the decision maker.

`public IloFSDecisionTracerI *` **`getTracer`**`() const`

This member function returns the associated tracer object, an instance of class `IloFSDecisionTracerI`, that is used to trace all events happening in the building of the first solution. By default no tracer object is associated to a decision maker, and this member function returns 0.

`public virtual void` **`init`**`()`

This virtual method is responsible for the creation and storage of the decision objects at the beginning of the execution of a decision maker. This method should be redefined in sub-classes of decision makers.

`public virtual IloBool` **`isLegal`**`(IloFSDecisionI * decision)`

This member function returns true if the decision is feasible. The default behavior of this member function is perform a nested `solve` that tests the feasibility of the decision. This nested `solve` calls first the `make` member function of the decision, which performs the actual links, the justifier goal and the route completion goal. If no goal fails, then the decision is assumed to be legal, and will actually be performed. If not, it will be removed from consideration, and a new decision will be selected.

`public void` **`registerGlobalDecision`**`(IloFSDecisionI * decision)`

This member function registers the decision for global searches for best decisions. Only registered decisions are taken into account when searching for the best global decision. This member function is useful only if you define your own decision classes.

```
public void registerVehicleDecision(IloFSDecisionI * decision, IloVehicle vehicle)
```

This member function registers the decision with the vehicle. It is necessary in algorithms that search for the best decision among those associated with a vehicle, as in the serial mode of nearest addition heuristics. This member function is useful only if you define your own decision classes.

```
public void registerVisitDecision(IloFSDecisionI * decision, IloVisit visit)
```

This member function registers the decision with the visit. This method should be used to register a decision with all associated visits. This member function is useful only if you define your own decision classes.

```
public void setTracer(IloFSDecisionTracerI * tracer)
```

This member function attaches a tracer object to a decision maker. The tracer object is notified of all events that happen in the building of the first solution.

```
public void storeDecision(IloFSDecisionI * decision)
```

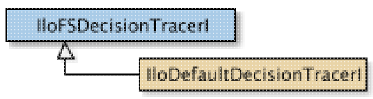
This member function is responsible for storing a newly created decision inside the decision maker.

It relies on the `store` virtual method of the class `IloFSDecisionI` to actually store the decision.

This member function should be used when redefining any `init` method of a decision maker class.

Class IloFSDecisionTracerI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>



This class is an abstract class used to monitor events happening while building the First Solution using decisions. A decision tracer can be attached to a decision maker object. The decision maker object will call the virtual member functions of the tracer objects to notify certain events. For example, each time a decision is chosen, the tracer object is notified. In addition to this abstract base class, Dispatcher provides a default tracer that does nothing. Subclassing from the default tracer only requires the definition of the member functions you want to trace, not all of them.

Constructor Summary	
`protected`	`IloFSDecisionTracerI(IloDispatcher)`

Method Summary	
`public virtual void`	`beginDecisionCommit(const IloFSDecisionI *)`
`public virtual void`	`beginDecisionTest(const IloFSDecisionI *)`
`public virtual void`	`beginExecute(const IloFSDecisionMakerI * dm)`
`public virtual void`	`endDecisionCommit(const IloFSDecisionI *)`
`public virtual void`	`endDecisionTest(const IloFSDecisionI *)`
`public virtual void`	`endExecute(const IloFSDecisionMakerI * dm)`
`public IloDispatcher`	`getDispatcher() const`
`public virtual void`	`notifyChosen(const IloFSDecisionI *)`
`public virtual void`	`notifyInfeasible(const IloFSDecisionI *)`
`public virtual void`	`notifyRejected(const IloFSDecisionI *, IloFSDecisionRejectCause cause)`
`public virtual void`	`notifyValidated(const IloFSDecisionI *)`
`public virtual void`	`registerDecision(const IloFSDecisionI *)`

Constructors

`protected` **`IloFSDecisionTracerI`**`(IloDispatcher)`

This constructor builds a decision tracer from a Dispatcher. As this class is an abstract class, the constructor is declared as protected.

Methods

`public virtual void` **`beginDecisionCommit`**`(const IloFSDecisionI *)`

This member function is called before calling the `commit` member function of the decision maker with the decision as argument. The decision has been tested as a legal one.

`public virtual void` **`beginDecisionTest`**`(const IloFSDecisionI *)`

This virtual member function is called before a decision is tested for legality using the `make` member function of the decision.

`public virtual void` **`beginExecute`**`(const IloFSDecisionMakerI * dm)`

This member function is called at the beginning of the execution of a decision maker, before any decision has been created and registered.

`public virtual void` **`endDecisionCommit`**`(const IloFSDecisionI *)`

This member function is called after calling the `commit` member function of the decision maker on the decision.

`public virtual void` **`endDecisionTest`**`(const IloFSDecisionI *)`

This virtual member function is called after executing the `make` member function of the decision within the testing of the decision. If the making of the decision fails, this member function may not be called.

`public virtual void` **`endExecute`**`(const IloFSDecisionMakerI * dm)`

This member function is called at the end of the execution of a decision maker, after all legal visits have been considered.

`public IloDispatcher` **`getDispatcher`**`() const`

This member function returns the dispatcher on which the tracer is built.

`public virtual void` **`notifyChosen`**`(const IloFSDecisionI *)`

This member function is called when a decision has been selected as the best legal decision that can be performed. This method is called before the decision maker attempts to execute and commit the decision.

`public virtual void` **`notifyInfeasible`**`(const IloFSDecisionI *)`

This member function is called whenever a decision has been statically computed as not feasible, before it has been tested. This can happen for nearest addition decisions, when the current route is already too long to accept the candidate visit.

`public virtual void` **`notifyRejected`**`(const IloFSDecisionI *, IloFSDecisionRejectCause cause)`

This virtual member function is called when the decision has been tested using the `isLegal` member function of the decision maker, and has been rejected. The rejection can be caused by any of three scenarios:

- the routing assignment, created by the decision's `make`, fails
- the route completion goal, used to check that the decision is consistent with the closing of the route, fails
- the justifier goal of the visit (typically a time-placement goal that tries to find a justifying set of starting times and dates and breaks, if any, along the route), fails

The cause of the rejection is identified by the `cause` enumerated value, which is passed to the method.

`public virtual void` **`notifyValidated`**`(const IloFSDecisionI *)`

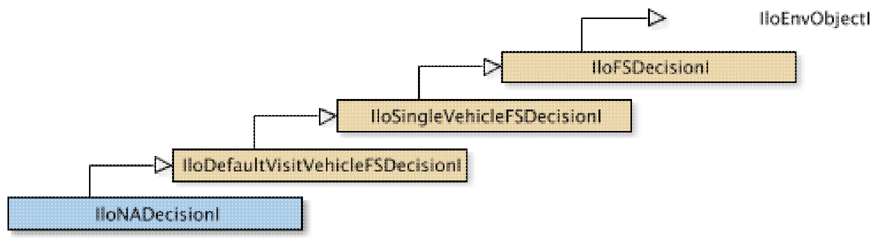
This member function is called when a decision has been accepted by the `isLegal` member function of the decision maker. The best decision will be selected from among the validated decisions.

`public virtual void` **`registerDecision`**`(const IloFSDecisionI *)`

This virtual member function is called when a decision is registered. A decision has to be registered to be taken into account by the first solution framework.

Class IloNADecisionI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>


IloEnvObjectI
IloFSDecisionI
IloSingleVehicleFSDecisionI
IloDefaultVisitVehicleFSDecisionI
IloNADecisionI

This abstract class is a subclass of `IloDefaultVisitVehicleFSDecisionI`, involving one visit and one vehicle. This class is dedicated to nearest-addition (NA) type of heuristics and is the base class of two other (concrete) decision classes modeling the two orientations of NA heuristics. As an abstract class, it defines new virtual member functions specific to nearest-addition.

Nearest-addition heuristics work by linking visits to the open ends of one or more open vehicles. An arc is defined as the couple of visits that will be linked by the decision; the candidate visit is always at one end of the arc. The getArc member function returns the oriented pair of visits that make up the arc.

Constructor Summary	
`public`	`IloNADecisionI(IloVisit visit, IloVehicle vehicle, IloNearestAdditionBehavior orientation)`

Method Summary	
`public virtual IloBool`	`calcFeasibility(IloFSDecisionMakerI * dm) const`
`public IloBool`	`closesVehicle() const`
`public void`	`display(ostream & out) const`
`public virtual IloNum`	`evaluate(IloFSDecisionMakerI * dm) const`
`public void`	`getArc(IloDispatcher dispatcher, IloVisit & from, IloVisit & to) const`
`public IloNearestAdditionBehavior`	`getOrientation() const`
`public virtual void`	`make(IloFSDecisionMakerI * dm)`
`public IloBool`	`opensVehicle() const`

Inherited Methods from `IloDefaultVisitVehicleFSDecisionI`
`calcFeasibility, display, getVisit, isBetterThan, store`

Inherited Methods from `IloSingleVehicleFSDecisionI`
`calcFeasibility, display, evaluate, getCost, getInChainStart, getOutChainEnd, getRouteCompletionGoal, getVehicle, isArcFeasible, isFeasible, isPossible`

Inherited Methods from `IloFSDecisionI`
`Compare, Compare, Compare, display, getEnv, getJustifierGoal, getRouteCompletionGoal, IsArcFeasibleOnDimension, isBetterThan, make, store`

Constructors

`public IloNADecisionI(IloVisit visit, IloVehicle vehicle, IloNearestAdditionBehavior orientation)`

This constructor builds a Nearest-Addition decision from a visit, a vehicle, and an orientation (forward or backward).

Methods

`public virtual IloBool calcFeasibility(IloFSDecisionMakerI * dm) const`

This member function is the implementation of the virtual member function of class `IloSingleVehicleFSDecisionI`. It returns `IloFalse` if some static computations can detect that the decision is infeasible. Otherwise, it returns `IloTrue`.

Note that this member function's filtering may be incomplete when complex side constraints are present. Decisions that pass this filtering, but which are infeasible, will be rejected later by the `isLegal` method.

`public IloBool closesVehicle() const`

This member function returns true when the decision closes a route. For example, a forward decision returns true when its visit is a last visit.

`public void display(ostream & out) const`

This member function displays the decision.

`public virtual IloNum evaluate(IloFSDecisionMakerI * dm) const`

This member function is the implementation of the virtual member function of class `IloSingleVehicleFSDecisionI`. It returns a cost that is used by the decision maker to select the best decision at a given point. The default implementation of this member function is to return the Dispatcher cost evaluation of linking the arc associated to the decision (see the `getArc` member function).

`public void getArc(IloDispatcher dispatcher, IloVisit & from, IloVisit & to) const`

This method returns the pair of visits (from, to), that the decision will try to link.

Note that the actual pair of visits linked by the decision depends both on the orientation of the decision, and on the current state of the vehicle route.

`public IloNearestAdditionBehavior getOrientation() const`

This method returns the orientation (forward, backward) associated with the decision.

`public virtual void make(IloFSDecisionMakerI * dm)`

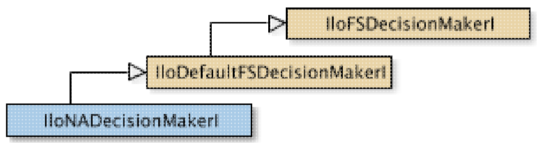
This member function attempts to link the arc associated to the decision, as returned by `getArc`.

```
public IloBool opensVehicle() const
```

This member function returns true when the decision opens a route. For example, a forward decision returns true when its visit is a first visit.

Class IloNADecisionMakerI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>



This class is a concrete subclass of `IloNADecisionMakerI`, dedicated to the nearest-addition type of first solution heuristics. The decision maker is initialized with two enumerated values which define the type of heuristics. The orientation parameter defines the orientation in which routes are built, either backward (from last to first) or forward (from first to last). The extension parameter defines whether the heuristic is a serial or a parallel one.

Constructor Summary	
`public`	`IloNADecisionMakerI(IloDispatcher dispatcher, IloNearestAdditionBehavior orientation, IloNearestAdditionExtension extension, IloBool filterArcs=IloFalse)`

Method Summary	
`public virtual IloFSDecisionI *`	`createDecision(IloVisit, IloVehicle)`
`public IloBool`	`filterArcs() const`
`public virtual IloFSDecisionI *`	`getBestDecision()`
`public IloNearestAdditionExtension`	`getExtension() const`
`public IloNearestAdditionBehavior`	`getOrientation() const`
`public void`	`init(IloVisitArray visits)`
`public virtual void`	`init()`

Inherited Methods from `IloDefaultFSDecisionMakerI`
`createDecision, init, registerVisitVehicleDecision`

Inherited Methods from `IloFSDecisionMakerI`
`commit, getBestDecision, getDispatcher, getEnv, getTracer, init, isLegal, registerGlobalDecision, registerVehicleDecision, registerVisitDecision, setTracer, storeDecision`

Constructors

`public` **`IloNADecisionMakerI`**`(IloDispatcher dispatcher, IloNearestAdditionBehavior orientation, IloNearestAdditionExtension extension, IloBool filterArcs=IloFalse)`

This constructor builds a decision maker for a Nearest-Addition heuristic. Four types of heuristics can be built using combinations of the enumerated parameters: Forward/Backward and Serial/Parallel.

The last argument `filterArcs` controls whether or not additional filtering of possible decisions is performed. This extra filtering uses the static method `IsArcFeasibleOnDimension` on all posted dimensions.

While this extra filtering can effectively reduce the number of considered decisions, it can also impact performance significantly. You can also redefine the `calcFeasibility` method to check only specific dimensions using the static method `IsArcFeasibleOnDimension` .

The default value of this flag is false.

Methods

```
public virtual IloFSDecisionI * createDecision(IloVisit, IloVehicle)
```

This virtual method is used by the `init` method to abstract the actual creation of the decision from a (visit, vehicle) couple. This method can return zero, in which case the couple is ignored.

```
public IloBool filterArcs() const
```

This member function returns true if extra filtering on dimensions is to be performed in the feasibility computations.

```
public virtual IloFSDecisionI * getBestDecision()
```

This member function is the implementation of the virtual member function defined in the abstract `IloNADecisionMakerI` class. The behavior of this member function depends on the extension parameter. If the extension mode is parallel, it returns the best possible decision, among all decisions that are yet to be considered. If the extension mode is serial, the decision maker fills vehicles one at a time, and this member function returns the best decision concerning the currently open vehicle. At the beginning, no vehicle is open, and the member function returns the best decision among all possible decisions. Afterwards, the decision's vehicle becomes the current vehicle, and only decisions registered with this vehicle are considered. When the vehicle is full, the decision maker looks again for the best global decision and chooses a new current decision, or terminates.

```
public IloNearestAdditionExtension getExtension() const
```

This member function returns the extension mode used in the decision maker.

```
public IloNearestAdditionBehavior getOrientation() const
```

This member function returns the orientation parameter used in the decision maker.

```
public void init(IloVisitArray visits)
```

This member function initializes the decision maker from an array of visits. For each visit of the array, it performs the same action as the `init` member function.

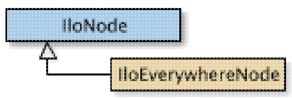
```
public virtual void init()
```

This is a redefinition of the virtual `init` member function of the `IloNADecisionMakerI` class.

This method iterates over all visits: for each visit, it iterates on all couples pairs (visit, vehicle), and calls the virtual method `createDecision`, defined at this class's level. If the `createDecision` method returns a valid decision, it is stored using the `storeDecision` method. If it returns zero, couple pair is simply ignored. The `createDecision` method can be used to create decisions only for meaningful pairs (visit, vehicle).

Class IloNode

Definition file: ildispat/ilonode.h
Include file: <ildispat/ilodispatcher.h>



Abstractly, a *node* is a part of the geographical representation of a problem. Intuitively, it represents an intersection of roads or the location of a customer site. A node is defined by coordinates that provide its location. More than one visit (instances of `IloVisit`) can be located at a single node.

For more information, see the concept Dimensions.

See Also: IloDimension2, IloDistance, IloEverywhereNode

Constructor Summary	
`public`	`IloNode(IloEnv env, IloNum x, IloNum y, IloNum z=0.0, const char * name=0)`
`public`	`IloNode(IloEnv env, const char * name=0)`
`public`	`IloNode(IloEnv env, IloBool everywhere, const char * name=0)`

Method Summary	
`public static IloBool`	`Exists(IloEnv env, const char * key)`
`public static IloNode`	`Find(IloEnv env, const char * key)`
`public IloNum`	`getDistanceTo(IloNode node, IloDimension2 dim, IloVehicle vehicle) const`
`public IloEnv`	`getEnv() const`
`public const char *`	`getKey() const`
`public const char *`	`getName() const`
`public IloAny`	`getObject() const`
`public IloNum`	`getX() const`
`public IloNum`	`getY() const`
`public IloNum`	`getZ() const`
`public IloBool`	`isEverywhere() const`
`public void`	`removeKey()`
`public void`	`setKey(const char * key)`
`public void`	`setName(const char * name) const`
`public void`	`setObject(IloAny obj) const`

Inner Class
IloNode::Iterator

Constructors

`public **IloNode**(IloEnv env, IloNum x, IloNum y, IloNum z=0.0, const char * name=0)`

This constructor creates a node whose coordinates are `x` and `y`, and allocates it upon `env`. The optional argument `z` is the third coordinate of the node. These coordinates are used in the distance functions `IloDistMax`, `IloEuclidean`, `IloGeographical`, and `IloManhattan` and in the goal `IloSweepGenerate`. The optional argument `name`, if provided, becomes the name of the node.

`public` **`IloNode`**`(IloEnv env, const char * name=0)`

This constructor creates a node and allocates it upon `env`. The optional argument `name`, if provided, becomes the name of the node.

`public` **`IloNode`**`(IloEnv env, IloBool everywhere, const char * name=0)`

This constructor creates a node and allocates it upon `env`. If everywhere is set to `IloTrue`, the node is an *everywhere* node, which is not used to compute distance. (Distances from or to everywhere nodes are assumed to be zero.) The optional argument `name`, if provided, becomes the name of the node.

Methods

`public static IloBool` **`Exists`**`(IloEnv env, const char * key)`

This static member function returns `IloTrue` if an `IloNode` object having key `key` exists and `IloFalse` if not.

`public static IloNode` **`Find`**`(IloEnv env, const char * key)`

This static member function returns the object corresponding to the key `key` set using `IloNode::setKey`. If there is no object corresponding to `key` an `IloException` is thrown.

`public IloNum` **`getDistanceTo`**`(IloNode node, IloDimension2 dim, IloVehicle vehicle) const`

This member function returns the distance from the invoking node to `node`, using extrinsic dimension `dim`, when the trip is made on vehicle `vehicle`.

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment of the invoking node.

`public const char *` **`getKey`**`() const`

The following member function returns the key set on the invoking object

`public const char *` **`getName`**`() const`

This member function returns the name of the invoking extractable object.

`public IloAny **getObject**() const`

This member function returns a pointer (may be null) to an object associated with the invoking extractable object. Such an object generally contains user-defined data.

`public IloNum **getX**() const`

This member function returns the x-coordinate of the invoking node.

`public IloNum **getY**() const`

This member function returns the y-coordinate of the invoking node.

`public IloNum **getZ**() const`

This member function returns the z-coordinate of the invoking node.

`public IloBool **isEverywhere**() const`

This member function returns `IloTrue` if the node is an everywhere node. Otherwise, it returns `IloFalse`.

`public void **removeKey**()`

The following member function allows the user to remove the key set on the invoking object.

`public void **setKey**(const char * key)`

This member function allows the user to set `key` on the invoking object. This key is unique. Each node must have a different key; otherwise, an exception is thrown.

`public void **setName**(const char * name) const`

This member function assigns `name` to the extractable object.

`public void **setObject**(IloAny obj) const`

This member function associates the object indicated by `obj` with the invoking extractable object.

Class IloOutOfRouteConstraint

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This class is a subclass of `IloConstraint`. This constraint prevents you from adding visits to the route of vehicle which are too far away from the current route.

Constructor Summary	
`public`	`IloOutOfRouteConstraint(IloVehicle vehicle, IloDimension2 dim, IloNum coef, IloOutOfRouteReference ref=IloFirstLastVisits)`

Constructors

`public` **`IloOutOfRouteConstraint`**`(IloVehicle vehicle, IloDimension2 dim, IloNum coef, IloOutOfRouteReference ref=IloFirstLastVisits)`

This constructor creates an out of route constraint.

There are three variants to this constraint.

The first one, which is typically found in long-haul trucking, is chosen when the parameter `ref` is equal to `IloFirstLastVisits`, which is the default value. It obeys the following rule:

```
vehicle.getTransitSumVar(dim) <= coef *
distance.getDistance(first.getNode(), last.getNode(), vehicle)
```

where:

- `distance` is the distance object associated with `dim`
- if the first visit `f` of the vehicle is located at an "everywhere-node", then `first` is the visit immediately after `f`; otherwise `first` is `f`
- if the last visit `l` of the vehicle is located at an "everywhere-node", then `last` is the visit immediately before `l`; otherwise `last` is `l`
- `coef` is a fixed value (at least 1) used to constrain the route

The second type of constraint is chosen when the parameter `ref` is equal to `IloNextFirstPrevLastVisits`. It obeys the following rule:

```
vehicle.getTransitSumVar(dim) <= coef *
distance.getDistance(nextFirst.getNode(), prevLast.getNode(), vehicle)
```

where:

- `distance` is the distance object associated with `dim`
- `nextFirst` is always the visit immediately after the first visit of the vehicle, whether the first visit is located at an "everywhere-node" or not
- `prevLast` is the visit immediately before the last visit of the vehicle, whether the last visit is located at an "everywhere-node" or not
- `coef` is a fixed value (at least 1) used to constrain the route

The third variant is chosen when the parameter ref is equal to `IloMaxDiameter`. The third type of constraint differs from the ones above by the way the "limiting" distance is computed. This distance depends on the maximum distance between any two visits on the route, which gives the following constraint:

```
 vehicle.getTransitSumVar(dim) <= coef *
 max(distance.getDistance(j.getNode(), k.getNode(), vehicle)
```

where `j` and `k` are visits belonging to the route of vehicle.

Class IloOutputManip

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

This class enables Dispatcher to define different display functionality for a basic type. Objects of this class can be used by the overloaded operator (`operator<<`) or returned by the functions `IloTerse` or `IloVerbose`.

See Also: IloTerse, IloVerbose

Constructor Summary	
`public`	`IloOutputManip(IloOutputManipI * impl)`

Method Summary	
`public void`	`display(ostream &) const`

Constructors

`public` **`IloOutputManip`**`(IloOutputManipI * impl)`

This constructor creates a handle object (an instance of `IloOutputManip`) from a pointer to an implementation object (an instance of `IloOutputManipI`).

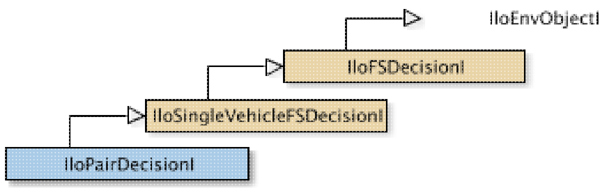
Methods

`public void` **`display`**`(ostream &) const`

This member function is called by `operator <<`.

Class IloPairDecisionI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>



This abstract class is a subclass of `IloSingleVehicleFSDecisionI`, involving one vehicle and a pair of pickup and delivery visits. This class is dedicated to PDP heuristics.

Constructor Summary	
`protected`	`IloPairDecisionI(IloVehicle, IloVisit pickup, IloVisit delivery)`

Method Summary	
`public virtual IloBool`	`calcFeasibility(IloFSDecisionMakerI * dm) const`
`public virtual void`	`display(ostream & os) const`
`public IloVisit`	`getDelivery() const`
`public IloVisit`	`getPickup() const`
`public virtual IloBool`	`isBetterThan(IloFSDecisionI * dec, const IloFSDecisionMakerI * dm) const`
`public virtual void`	`store(IloFSDecisionMakerI * dm)`

Inherited Methods from `IloSingleVehicleFSDecisionI`
`calcFeasibility, display, evaluate, getCost, getInChainStart, getOutChainEnd, getRouteCompletionGoal, getVehicle, isArcFeasible, isFeasible, isPossible`

Inherited Methods from `IloFSDecisionI`
`Compare, Compare, Compare, display, getEnv, getJustifierGoal, getRouteCompletionGoal, IsArcFeasibleOnDimension, isBetterThan, make, store`

Constructors

`protected` **`IloPairDecisionI`**`(IloVehicle, IloVisit pickup, IloVisit delivery)`

This constructor creates a pair decision from a vehicle and a (pickup, delivery) pair of visits.

Methods

`public virtual IloBool` **`calcFeasibility`**`(IloFSDecisionMakerI * dm) const`

This member function is an implementation of the pure virtual member function of the `IloSingleVehicleFSDecisionI` class. This member function returns `IloFalse` if it can prove, from a static analysis, that the insertion of the pair (pickup, delivery) on the decision's vehicle is impossible. Otherwise, it returns `IloTrue`.

This method checks next and previous variable domains, and also checks that the path constraints on all dimensions are satisfied.

Note that this member function's filtering may be incomplete when complex side constraints are present. Decisions that pass this filtering, but which are infeasible, will be rejected later by the `isLegal` method.

`public virtual void` **`display`**`(ostream & os) const`

This virtual member function displays the decision object.

`public IloVisit` **`getDelivery`**`() const`

This member function returns the delivery visit.

`public IloVisit` **`getPickup`**`() const`

This member function returns the pickup visit.

`public virtual IloBool` **`isBetterThan`**`(IloFSDecisionI * dec, const IloFSDecisionMakerI * dm) const`

This member function is an implementation of the pure virtual member function of the `IloFSDecisionI` class. It assumes that the two decisions are of the `IloPairDecisionI` type. This member function tests the two costs and if the cost of the invoking decision is lower, returns `IloTrue`.

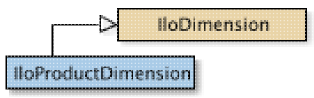
If the costs are equal, it performs a tie-breaking on the decision's vehicles if they are different. Otherwise, it performs a tie-breaking on the decision's delivery.

`public virtual void` **`store`**`(IloFSDecisionMakerI * dm)`

This member function implements the pure virtual method of class `IloFSDecisionI`. It registers the decision with its vehicle, its pickup and delivery visits, and also for global searches.

Class IloProductDimension

Definition file: ildispat/ilodim.h
Include file: <ildispat/ilodispatcher.h>



Instances of the class `IloProductDimension` represent the product of distance traveled and quantity transported along a route. These notions are represented using two external dimensions, one extrinsic dimension (distance traveled) and one intrinsic dimension (quantity transported). Relating this to variables, for a given visit `v`, an extrinsic dimension `dim2`, an intrinsic dimension `dim1` and a product dimension `prdim`, the following is true: `v.getTransitVar(prdim) == v.getTravelVar(dim2) * (v.getCumulVar(dim1) + v.getTransitVar(dim1))`.

Constructor Summary	
public	IloProductDimension(IloEnv env, IloDimension1 productDim1, IloDimension2 productDim2, const char * name=0)
public	IloProductDimension(IloEnv env, IloDimension1 productDim1, IloDimension2 productDim2, IloBool postIt, const char * name=0)

Method Summary	
public static IloBool	Exists(IloEnv env, const char * key)
public static IloProductDimension	Find(IloEnv env, const char * key)
public IloDimension1	getDimension1() const
public IloDimension2	getDimension2() const
public const char *	getKey() const
public void	removeKey()
public void	setKey(const char * key)

Inherited Methods from `IloDimension`
assumeTriangleInequality

Constructors

`public` **`IloProductDimension`**`(IloEnv env, IloDimension1 productDim1, IloDimension2 productDim2, const char * name=0)`

This constructor creates an instance of the class `IloProductDimension`, associated with the environment indicated by `env`. The intrinsic dimension `productDim1` represents the quantities transported and the extrinsic dimension `productDim2` represents the distance traveled. The optional argument `name`, if provided, becomes the name of the dimension.

`public` **`IloProductDimension`**`(IloEnv env, IloDimension1 productDim1, IloDimension2 productDim2, IloBool postIt, const char * name=0)`

This constructor creates an instance of the class `IloProductDimension`, associated with the environment indicated by `env`. The intrinsic dimension `productDim1` represents the quantities transported and the extrinsic dimension `productDim2` represents the distance traveled. The parameter `postIt` indicates whether the underlying constraint associated with the product dimension is posted or not. Setting `postIt` to `IloFalse` speeds up the search but should only be done if no constraints are posted on variables related to the invoking dimension. However, a dimension created with `postIt=IloFalse` may be safely used in the cost function. The optional argument `name`, if provided, becomes the name of the dimension.

Methods

`public static IloBool` **`Exists`**`(IloEnv env, const char * key)`

This static member function returns `IloTrue` if an `IloProductDimension` object having key `key` exists and `IloFalse` if not.

`public static IloProductDimension` **`Find`**`(IloEnv env, const char * key)`

This static member function returns the product dimension corresponding to the key `key` set using `IloProductDimension::setKey`. If there is no product dimension corresponding to `key` an exception is thrown.

`public IloDimension1` **`getDimension1`**`() const`

This member function returns the intrinsic dimension associated with the invoking product dimension. This dimension is used to access the quantity transported between each visit.

`public IloDimension2` **`getDimension2`**`() const`

This member function returns the extrinsic dimension associated with the invoking product dimension. This dimension is used to access the distance traveled between each visit.

`public const char *` **`getKey`**`() const`

This member function returns the key set on the invoking product dimension.

`public void` **`removeKey`**`()`

This member function allows the user to remove the key set on the invoking object.

`public void` **`setKey`**`(const char * key)`

This member function allows the user to set `key` on the invoking product dimension. This key is unique. Each product dimension must have a different key; otherwise, an exception is thrown.

Class IloRoutingSolution

Definition file: ildispat/ilorsol.h
Include file: <ildispat/ilodispatcher.h>

An instance of `IloRoutingSolution` stores the details of a solution to a routing problem. An instance of `IloRoutingSolution` contains an instance of `IloSolution`.

Example

The following program creates a solution to a simple vehicle routing problem and writes out a routing solution:

```
  int main(int argc, char* argv[]) {
    IloEnv env;
    IloModel mdl(env);
    IloDimension2 length(env, IloEuclidean, "Length");
    IloNode depot(env, 0.0, 0.0);
    IloVisit first1(depot, "First 1");
    IloVisit last1(depot, "Last 1");
    IloVisit first2(depot, "First 2");
    IloVisit last2(depot, "Last 2");
    IloVehicle vehicle1(first1, last1, "Vehicle 1");
    mdl.add(vehicle1);
    IloVehicle vehicle2(first2, last2, "Vehicle 2");
    mdl.add(vehicle2);
    vehicle1.setCost(length, 2.0);
    vehicle2.setCost(length, 1.0);
    IloNode n1(env, 1, 1); IloVisit v1(n1, "V1"); mdl.add(v1);
    IloNode n2(env, 1, 2); IloVisit v2(n2, "V2"); mdl.add(v2);
    IloNode n3(env, 2, 7); IloVisit v3(n3, "V3"); mdl.add(v3);
    IloNode n4(env, 3, 1); IloVisit v4(n4, "V4"); mdl.add(v4);
    IloNode n5(env, 5, 3); IloVisit v5(n5, "V5"); mdl.add(v5);
    IloNode n6(env, 0, 3); IloVisit v6(n6, "V6");
    IloConstraint initialSolution =
      (first1.getNextVar() == v1)
      && (v1.getNextVar() == v2)
      && (v2.getNextVar() == v3)
      && (v3.getNextVar() == last1)
      && (first2.getNextVar() == v4)
      && (v4.getNextVar() == v5)
      && (v5.getNextVar() == last2);
    mdl.add(initialSolution);
    IloSolver solver(mdl);
    IloRoutingSolution solution(mdl);
    IloDispatcher dispatcher(solver);
    IloGoal instantiateCost = IloDichotomize(env,
                                            dispatcher.getCostVar(),
                                            IloFalse);
    solver.solve(instantiateCost);
    solution.store(solver);
    solver.out() << solution;
    env.end();
    return 0;
  }
```

The routing solution generated by the program above is produced in the following standard format:

```
COST 41.4083
UNPERFORMED -1
ROUTE V1 V2 V3 -1
ROUTE V4 V5 -1
```

The first line of the solution contains the cost of the routing solution.

The second line lists any unperformed visits. Visits are indicated by their index number. The list is terminated with -1.

The third and following lines list the vehicle routes in vehicle index number order. The visits of each route are listed, by visit index number, in the order in which they are performed. The “first” and “last” visits are not listed. The index number -1 indicates the end of the route.

For more information, see the concept Neighborhoods and the class `IloSolution` in the *IBM ILOG Concert Technology Reference Manual*.

See Also: IloRoutingSolution::RouteIterator, IloRoutingSolution::UnperformedVisitIterator, IloRoutingSolution::VehicleIterator, IloRoutingSolution::VisitIterator

Constructor Summary	
public	IloRoutingSolution()
public	IloRoutingSolution(IloRoutingSolutionI * impl)
public	IloRoutingSolution(const IloRoutingSolution & solution)
public	IloRoutingSolution(IloEnv env, const char * name=0)
public	IloRoutingSolution(IloModel model, const char * name=0)
public	IloRoutingSolution(IloSolution solution)

Method Summary	
public void	add(IloVehicleBreakCon brk) const
public void	add(IloVisit visit) const
public void	add(IloVehicle vehicle) const
public void	add(IloModel model) const
public void	copy(IloRoutingSolution solution) const
public void	end()
public IloNum	getDurationMax(IloVehicleBreakCon brk) const
public IloNum	getDurationMin(IloVehicleBreakCon brk) const
public IloEnv	getEnv() const
public IloRoutingSolutionI *	getImpl() const
public IloVisit	getNext(IloVisit visit) const
public IloInt	getNumberOfPerformedVisits() const
public IloInt	getNumberOfUnperformedVisits() const
public IloNum	getObjectiveValue() const
public IloNumVar	getObjectiveVar() const
public IloVisit	getPosition(IloVehicleBreakCon brk) const
public IloVisit	getPrev(IloVisit visit) const
public IloInt	getRouteSize(IloVehicle vehicle) const
public IloSolution	getSolution() const
public IloNum	getStartMax(IloVehicleBreakCon brk) const
public IloNum	getStartMin(IloVehicleBreakCon brk) const
public IloVehicle	getVehicle(IloVisit visit) const
public IloBool	isBound(IloVehicleBreakCon brk) const
public IloBool	isPerformed(IloVehicleBreakCon brk) const
public IloBool	isPerformed(IloVisit visit) const

public IloRoutingSolution	makeClone(IloEnv env) const
public	operator IloSolution() const
public void	operator=(const IloRoutingSolution & h)
public void	remove(IloVehicleBreakCon brk) const
public void	remove(IloVisit visit) const
public void	remove(IloVehicle vehicle) const
public void	remove(IloModel model) const
public void	setDuration(IloVehicleBreakCon brk, IloNum val) const
public void	setDurationMax(IloVehicleBreakCon brk, IloNum max) const
public void	setDurationMin(IloVehicleBreakCon brk, IloNum min) const
public void	setNext(IloVisit visit, IloVisit next) const
public void	setPerformed(IloVehicleBreakCon brk) const
public void	setPosition(IloVehicleBreakCon brk, IloVisit visit) const
public void	setPrev(IloVisit visit, IloVisit prev) const
public void	setStart(IloVehicleBreakCon brk, IloNum val) const
public void	setStartMax(IloVehicleBreakCon brk, IloNum max) const
public void	setStartMin(IloVehicleBreakCon brk, IloNum min) const
public void	setUnperformed(IloVehicleBreakCon brk) const
public void	setUnperformed(IloVisit visit) const
public void	setVehicle(IloVisit visit, IloVehicle vehicle) const
public void	store(IloSolver solver) const

Inner Class
IloRoutingSolution::RouteIterator
IloRoutingSolution::UnperformedVisitIterator
IloRoutingSolution::VehicleIterator
IloRoutingSolution::VisitIterator

Constructors

public **IloRoutingSolution**()

This constructor creates a routing solution whose handle pointer is null. This object must be assigned before it can be used.

public **IloRoutingSolution**(IloRoutingSolutionI * impl)

This constructor creates a handle object (an instance of `IloRoutingSolution`) from a pointer to an implementation object (an instance of the class `IloRoutingSolutionI`).

```
public IloRoutingSolution(const IloRoutingSolution & solution)
```

This copy constructor creates a handle from a reference to a routing solution. That routing solution and `solution` both point to the same implementation object.

```
public IloRoutingSolution(IloEnv env, const char * name=0)
```

This constructor creates a routing solution on environment `env`. It creates an instance of `IloSolution` and sets its objective to be the cost variable of the routing problem. The optional argument `name`, if provided, becomes the name of the routing solution.

```
public IloRoutingSolution(IloModel model, const char * name=0)
```

This constructor creates a routing solution from the environment associated with `model`. It calls `IloRoutingSolution::add` and adds all the visits and vehicles in `model` to the solution. The optional argument `name`, if provided, becomes the name of the routing solution.

```
public IloRoutingSolution(IloSolution solution)
```

This constructor creates a routing solution from solution `solution`. It sets the routing solution's objective variable to be the cost variable of the routing problem.

Methods

```
public void add(IloVehicleBreakCon brk) const
```

This member function adds `brk` to the invoking routing solution.

```
public void add(IloVisit visit) const
```

This member function adds `visit` to the invoking routing solution.

```
public void add(IloVehicle vehicle) const
```

This member function adds `vehicle` to the invoking routing solution. This is done by adding the first and last visits of the vehicle to the solution and setting their saved next- and previous-variables such that the first visit is directly connected to the last. That is, the saved state of the vehicle is empty.

```
public void add(IloModel model) const
```

This member function adds `model` to the invoking routing solution. That is, the status of all visit and vehicle variables in `model` are stored in the invoking routing solution.

```
public void copy(IloRoutingSolution solution) const
```

This member function copies `solution` to the invoking routing solution. For each variable that has been added to solution, this member function copies its saved data to the invoking solution. If a particular extractable does not already exist in the invoking solution, it is automatically added first. If variables were added to the invoking solution, their restorable status is the same as in solution. Extractables that are present in the invoking solution but not in the solution being copied do not have their saved information modified.

`public void` **`end`**`()`

This member function calls `end()` on the instance of `IloSolution` contained in the invoking routing solution.

`public IloNum` **`getDurationMax`**`(IloVehicleBreakCon brk) const`

This member function returns the upper bound of the duration variable of `brk` stored in the invoking routing solution.

`public IloNum` **`getDurationMin`**`(IloVehicleBreakCon brk) const`

This member function returns the lower bound of the duration variable of `brk` stored in the invoking routing solution.

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment of the invoking routing solution.

`public IloRoutingSolutionI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking routing solution.

`public IloVisit` **`getNext`**`(IloVisit visit) const`

This member function returns the visit corresponding to the value of the next-variable associated with `visit` in the invoking routing solution. If the next-variable is unbound, an `IloException` is thrown.

`public IloInt` **`getNumberOfPerformedVisits`**`() const`

This member function returns the number of performed visits in the invoking routing solution.

`public IloInt` **`getNumberOfUnperformedVisits`**`() const`

This member function returns the number of unperformed visits in the invoking routing solution.

`public IloNum` **`getObjectiveValue`**`() const`

This member function returns the value of the objective the previous time it was stored via a call to `store`.

`public IloNumVar` **`getObjectiveVar`**`() const`

This member function returns the objective variable of the invoking routing solution.

`public IloVisit` **`getPosition`**`(IloVehicleBreakCon brk) const`

This member function returns the visit after which `brk` is performed in the invoking routing solution.

`public IloVisit` **`getPrev`**`(IloVisit visit) const`

This member function returns the visit corresponding to the value of the previous-variable associated with `visit` in the invoking routing solution. If the previous-variable is unbound, an `IloException` is thrown.

`public IloInt` **`getRouteSize`**`(IloVehicle vehicle) const`

This member function returns the number of visits, not including the first and last, or “depot,” visits, in the route of `vehicle` in the invoking routing solution.

`public IloSolution` **`getSolution`**`() const`

This member function returns the instance of `IloSolution` contained in the invoking routing solution.

`public IloNum` **`getStartMax`**`(IloVehicleBreakCon brk) const`

This member function returns the upper bound of the start variable of `brk` stored in the invoking routing solution.

`public IloNum` **`getStartMin`**`(IloVehicleBreakCon brk) const`

This member function returns the lower bound of the start variable of `brk` stored in the invoking routing solution.

`public IloVehicle` **`getVehicle`**`(IloVisit visit) const`

This member function returns the vehicle servicing `visit` in the invoking routing solution.

`public IloBool` **`isBound`**`(IloVehicleBreakCon brk) const`

This member function returns `IloTrue` if the position of `brk` is bound. Otherwise, it returns `IloFalse`.

`public IloBool` **`isPerformed`**`(IloVehicleBreakCon brk) const`

This member function returns `IloTrue` if `brk` is performed in the invoking routing solution.

`public IloBool` **`isPerformed`**`(IloVisit visit) const`

This member function returns `IloTrue` if `visit` is performed in the invoking routing solution. Otherwise, it returns `IloFalse`.

`public IloRoutingSolution` **`makeClone`**`(IloEnv env) const`

This member function makes a clone of the invoking routing solution.

`public` **`operator IloSolution`**`() const`

This cast operator casts the invoking routing solution to an `IloSolution` taking the solution returned by `IloRoutingSolution::getSolution().`

`public void` **`operator=`**`(const IloRoutingSolution & h)`

This operator assigns an address to the handle pointer of the invoking routing solution. This address is the location of the implementation object of the argument `solution`. After the execution of this operator, the invoking routing solution and `solution` both point to the same implementation object.

`public void` **`remove`**`(IloVehicleBreakCon brk) const`

This member function removes `brk` from the invoking routing solution.

`public void` **`remove`**`(IloVisit visit) const`

This member function removes `visit` from the invoking routing solution.

`public void` **`remove`**`(IloVehicle vehicle) const`

This member function removes `vehicle` from the invoking routing solution by removing both the first and last visit of the vehicle.

`public void` **`remove`**`(IloModel model) const`

This member function removes all visits and vehicles that appear in the model from the invoking routing solution.

`public void` **`setDuration`**`(IloVehicleBreakCon brk, IloNum val) const`

This member function sets the value of the duration variable of `brk` to `val` in the invoking routing solution.

`public void` **`setDurationMax`**`(IloVehicleBreakCon brk, IloNum max) const`

This member function sets the maximum of the duration variable of `brk` to `max` in the invoking routing solution.

`public void` **`setDurationMin`**`(IloVehicleBreakCon brk, IloNum min) const`

This member function sets the minimum of the duration variable of `brk` to `min` in the invoking routing solution.

`public void` **`setNext`**`(IloVisit visit, IloVisit next) const`

This member function sets `next` to be the visit just after `visit` in the invoking routing solution.

`public void` **`setPerformed`**`(IloVehicleBreakCon brk) const`

This member function sets `brk` to be performed in the invoking routing solution.

`public void` **`setPosition`**`(IloVehicleBreakCon brk, IloVisit visit) const`

This member function sets the position of `brk` to be immediately after `visit` in the invoking routing solution.

`public void` **`setPrev`**`(IloVisit visit, IloVisit prev) const`

This member function sets `prev` to be the visit just before `visit` in the invoking routing solution.

`public void` **`setStart`**`(IloVehicleBreakCon brk, IloNum val) const`

This member function sets the value of the start variable of `brk` to `val` in the invoking routing solution.

`public void` **`setStartMax`**`(IloVehicleBreakCon brk, IloNum max) const`

This member function sets the maximum of the start variable of `brk` to `max` in the invoking routing solution.

`public void` **`setStartMin`**`(IloVehicleBreakCon brk, IloNum min) const`

This member function sets the minimum of the start variable of `brk` to `min` in the invoking routing solution.

`public void` **`setUnperformed`**`(IloVehicleBreakCon brk) const`

This member function sets `brk` to be unperformed in the invoking routing solution.

`public void` **`setUnperformed`**`(IloVisit visit) const`

This member function sets `visit` to be unperformed in the invoking routing solution.

```
public void setVehicle(IloVisit visit, IloVehicle vehicle) const
```

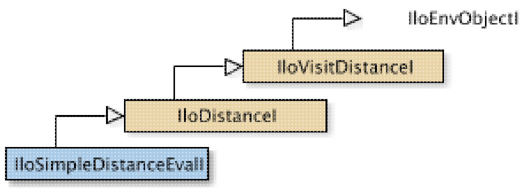
This member function sets `vehicle` to be the vehicle performing `visit` in the invoking routing solution.

```
public void store(IloSolver solver) const
```

This member function calls store on the instance of `IloSolution` contained in the invoking routing solution.

Class IloSimpleDistanceEvalI

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher lets you define the distance function for a dimension (for example, the distance, the time, or the cost necessary for going from one node to another).

This class is an implementation class, a predefined subclass of `IloDistanceI`, that you use to define a new distance function expressed by an evaluation function. This evaluation function is of type `IloSimpleDistanceFunction`. It differs from `IloDistanceEvalI` by only considering the two nodes when computing a distance and ignoring vehicles.

See Also: IloDimension, IloDimension1, IloDimension2, IloDistance, IloSimpleDistanceFunction, IloDistanceI

Constructor and Destructor Summary	
`public`	`IloSimpleDistanceEvalI(IloEnv env, IloSimpleDistanceFunction distFunction)`

Method Summary	
`public IloNum`	`computeDistance(IloNode node1, IloNode node2, IloVehicle vehicle) const`

Inherited Methods from `IloDistanceI`
`computeDistance, computeDistance, getDistance, getGroup, refresh, setCache, unsetCache, updateEquivalence`

Inherited Methods from `IloVisitDistanceI`
`computeDistance, getDistance, getGroup, refresh, setCache, unsetCache, updateEquivalence`

Constructors and Destructors

`public` **`IloSimpleDistanceEvalI`**`(IloEnv env, IloSimpleDistanceFunction distFunction)`

This constructor creates a new distance function from the evaluation function `distFunction`.

Methods

`public IloNum` **`computeDistance`**`(IloNode node1, IloNode node2, IloVehicle vehicle) const`

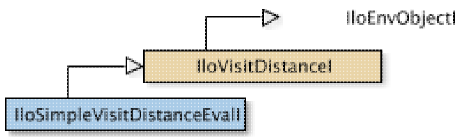
This member function returns a numeric value that represents the distance between `node1` and `node2`. This is done using a call to `distFunction`, passing `node1` and `node2` as parameters.

Note

Note that the distances are not vehicle-dependent. The parameter `vehicle` is included because this is the default interface to distance data.

Class IloSimpleVisitDistanceEvalI

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher lets you define the distance function for a dimension (for example, the distance, the time, or the cost necessary for going from one node to another).

This class is an implementation class, a predefined subclass of `IloVisitDistanceI`, which you use to define a new distance function expressed by an evaluation function. This evaluation function is of type `IloSimpleVisitDistanceFunction`. It differs from `IloVisitDistanceEvalI` by only considering the two visits when computing a distance and ignoring vehicles.

See Also: IloDimension, IloDimension1, IloDimension2, IloVisitDistance, IloSimpleVisitDistanceFunction, IloVisitDistanceI

Constructor and Destructor Summary	
`public`	`IloSimpleVisitDistanceEvalI(IloEnv env, IloSimpleVisitDistanceFunction distFunction)`

Method Summary	
`public IloNum`	`computeDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const`

Inherited Methods from `IloVisitDistanceI`
`computeDistance, getDistance, getGroup, refresh, setCache, unsetCache, updateEquivalence`

Constructors and Destructors

```
public IloSimpleVisitDistanceEvalI(IloEnv env, IloSimpleVisitDistanceFunction
distFunction)
```

This constructor creates a new distance function from the evaluation function `distFunction`.

Methods

```
public IloNum computeDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle)
const
```

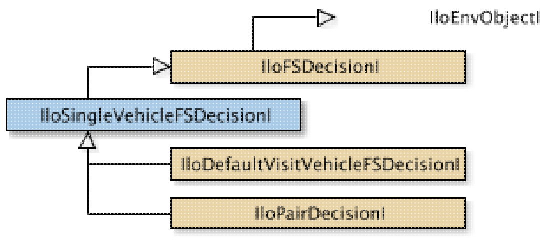
This member function returns a numeric value that represents the distance between `visit1` and `visit2`. This is done using a call to `distFunction`, passing `visit1` and `visit2` as parameters.

Note

Note that the distances are not vehicle-dependent. The parameter `vehicle` is included because this is the default interface to distance data.

Class IloSingleVehicleFSDecisionI

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>


IloEnvObjectI
IloFSDecisionI
IloSingleVehicleFSDecisionI
IloDefaultVisitVehicleFSDecisionI
IloPairDecisionI

The abstract class `IloSingleVehicleFSDecisionI` is the parent class for decisions concerning one vehicle. Note that such decisions can involve several visits. For example, placing a pickup, delivery pair on one vehicle is an example of a decision involving several visits. As the semantics of the decision are not defined at this class level, the make method is not defined.

To simplify the decision comparison process, a cost is associated to decisions, computed by a new virtual member function called `evaluate`. Subclassing from this class requires you to define an `evaluate` member function.

Constructor Summary	
`protected`	`IloSingleVehicleFSDecisionI(IloVehicle vehicle)`

Method Summary	
`public virtual IloBool`	`calcFeasibility(IloFSDecisionMakerI * dm) const`
`public virtual void`	`display(ostream &) const`
`public virtual IloNum`	`evaluate(IloFSDecisionMakerI * dm) const`
`public IloNum`	`getCost() const`
`public IloVisit`	`getInChainStart(IloDispatcher) const`
`public IloVisit`	`getOutChainEnd(IloDispatcher) const`
`public virtual IlcGoal`	`getRouteCompletionGoal(IloFSDecisionMakerI * dm)`
`public IloVehicle`	`getVehicle() const`
`public IloBool`	`isArcFeasible(IloDispatcher, IloVisit v1, IloVisit v2) const`
`public virtual IloBool`	`isFeasible(IloFSDecisionMakerI *) const`
`public IloBool`	`isPossible(IloDispatcher, IloVisit) const`

Inherited Methods from `IloFSDecisionI`
`Compare, Compare, Compare, display, getEnv, getJustifierGoal, getRouteCompletionGoal, IsArcFeasibleOnDimension, isBetterThan, make, store`

Constructors

`protected` **`IloSingleVehicleFSDecisionI`**`(IloVehicle vehicle)`

This constructor builds an `IloSingleVehicleFSDecisionI` from a vehicle.

Methods

`public virtual IloBool` **`calcFeasibility`**`(IloFSDecisionMakerI * dm) const`

This member function computes a feasibility predicate for the decision. As the precise location of the visit on the vehicle is not defined at this class level, this predicate only tests that the vehicle can be assigned to the visit. Subclasses should redefine this member function.

Decisions that are proven infeasible by the `calcFeasibility` predicate will be discarded when searching for the best decision. Feasibility status is automatically refreshed after each decision is taken.

`public virtual void` **`display`**`(ostream &) const`

This virtual member function displays the decision object.

`public virtual IloNum` **`evaluate`**`(IloFSDecisionMakerI * dm) const`

This pure virtual member function computes a cost that is stored. The cost is used to compare decisions. Decisions with the lowest cost are preferred.

`public IloNum` **`getCost`**`() const`

This member function returns the cost associated to the decision, as computed by the `evaluate` member function.

`public IloVisit` **`getInChainStart`**`(IloDispatcher) const`

This member function returns the first visit of the chain ending at the last visit of the vehicle associated with the decision. When building a solution, visits are connected together through their next variable. Connected visits form a chain.

`public IloVisit` **`getOutChainEnd`**`(IloDispatcher) const`

This member function returns the last visit of the chain starting at the first visit of the vehicle associated with the decision. When building a solution, visits are connected together through their next variable. Connected visits form a chain.

`public virtual IlcGoal` **`getRouteCompletionGoal`**`(IloFSDecisionMakerI * dm)`

This member function returns the route completion goal used for decision validation. It returns `IloGenerateRoute` on the decision vehicle.

`public IloVehicle` **`getVehicle`**`() const`

This member function returns the vehicle associated with the decision.

`public IloBool` **`isArcFeasible`**`(IloDispatcher, IloVisit v1, IloVisit v2) const`

This member function returns `IloFalse` when the arc from `v1`to `v2` on the decision vehicle can be detected as not feasible, and returns `IloTrue` otherwise.

This member function checks that `v2` is in the domain of possible next visits of `v1` (and conversely checks that `v1` is a possible previous visits of `v2`).

This member function also checks the feasibility of the arc on all dimensions by calling the `isArcFeasibleOnDimensions` method.

`public virtual IloBool` **`isFeasible`**`(IloFSDecisionMakerI *) const`

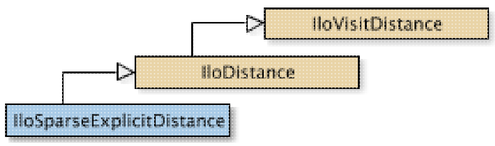
This virtual method is the defines the abstract method declared at the level of the `IloFSDecisionI` class. It returns the feasibility predicate, as computed by the `calcFeasibility` virtual method.

`public IloBool` **`isPossible`**`(IloDispatcher, IloVisit) const`

This member function returns `IloTrue` when the vehicle is in the domain of possible vehicles of `visit`. This method can be used to compute the feasibility of a decision.

Class IloSparseExplicitDistance

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>



This distance class allows the user to specify the distance matrix explicitly using the member function IloSparseExplicitDistance::setValue.

You should use this class instead of `IloExplicitDistance` when the number of nodes is quite large, but you are only interested in the distances between a small set of these nodes.

See Also: IloExplicitDistance, IloDistance, IloComposedDistance

Constructor and Destructor Summary	
public	`IloSparseExplicitDistance(IloEnv env, IloBool symmetric=IloFalse, IloNum defaultVal=IloInfinity, IloInt size=1)`
public	`IloSparseExplicitDistance(IloEnv env, IloBool symmetric, IloSimpleDistanceFunction defaultFunction, IloInt size=1)`
public	`~IloSparseExplicitDistance()`

Method Summary	
public IloNum	`getDistance(IloNode node1, IloNode node2, IloVehicle vehicle) const`
public IloBool	`isSet(IloNode node1, IloNode node2) const`
public void	`setValue(IloNode node1, IloNode node2, IloNum value)`

Inherited Methods from `IloDistance`
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setCache, setKey, unsetCache`

Inherited Methods from `IloVisitDistance`
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setKey`

Constructors and Destructors

public **IloSparseExplicitDistance**(IloEnv env, IloBool symmetric=IloFalse, IloNum defaultVal=IloInfinity, IloInt size=1)

This constructor creates a sparse explicit distance object in the environment `env`. The parameter `defaultVal` allows you to specify the value that will be returned when no actual distance between two nodes has been specified. Before, the returned value was automatically infinity. Now, infinity is returned only if you do not override it with a value of your choice.

The optional parameter `symmetric` is set to `IloTrue` if the sparse distance matrix is symmetric. The distance matrix is symmetric if, for any two nodes *a* and *b*, *dist(a,b)=dist(b,a)*. The optional parameter `size` allows you to pre-size the data structure used to store the data.

For speed considerations, it is recommended to set the `size` parameter to approximately the number of entries that will be in the sparse distance matrix. If the instance is symmetric, you only need to allocate space for half the entries. In all cases, the constructor will round `size` to the power of two that is smaller or equal to the parameter `size`.

```
public IloSparseExplicitDistance(IloEnv env, IloBool symmetric,
IloSimpleDistanceFunction defaultFunction, IloInt size=1)
```

This constructor creates a sparse explicit distance object in the environment `env`. The parameter `defaultFunction` allows you to specify a distance function which will be called when no actual distance between two nodes has been specified.

The parameter `symmetric` is set to `IloTrue` if the sparse distance matrix is symmetric. The distance matrix is symmetric if, for any two nodes *a* and *b*, *dist(a,b)=dist(b,a)*. The optional parameter `size` allows you to pre-size the data structure used to store the data.

For speed considerations, it is recommended to set the `size` parameter to approximately the number of entries that will be in the sparse distance matrix. If the instance is symmetric, you only need to allocate space for half the entries. In all cases, the constructor will round `size` to the power of two that is smaller or equal to the parameter `size`.

```
public ~IloSparseExplicitDistance()
```

This destructor returns the memory used by the matrix.

Methods

```
public IloNum getDistance(IloNode node1, IloNode node2, IloVehicle vehicle) const
```

This member function returns the distance from `node1` to `node2` in the sparse matrix.

If a value for two given nodes has not been set using `setValue`, then an exception is thrown. You can use the member function `isSet` to check if a distance has already been set.

Note

Note that the distances are not vehicle-dependent. The parameter `vehicle` is included because this is the default interface to distance data. However, since `setValue` does not set vehicle-specific data, the sparse matrix does not contain vehicle-specific data.

```
public IloBool isSet(IloNode node1, IloNode node2) const
```

This member function returns `IloTrue` if the distance between `node1` and `node2` has already been set by `setValue`.

The class `IloSparseExplicitDistance` throws an exception if you try to access data that has not been set using `setValue`.

```
public void setValue(IloNode node1, IloNode node2, IloNum value)
```

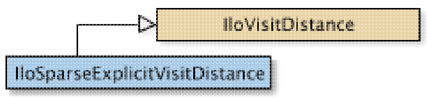
This member function sets the explicit distance `value` between two nodes, `node1` and `node2`.

Note

Note that you can change a pre-existing `value` by using `setValue` several times. The distance is set to the `value` specified by the last call to `setValue`.

Class IloSparseExplicitVisitDistance

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>



This distance class allows the user to specify the distance matrix explicitly using the member function IloSparseExplicitVisitDistance::setValue.

You should use this class instead of `IloExplicitVisitDistance` when the number of visits is quite large, but when you are only interested in the distances between a small set of these visits.

See Also: IloExplicitVisitDistance, IloVisitDistance, IloComposedVisitDistance

Constructor and Destructor Summary	
`public`	`IloSparseExplicitVisitDistance(IloEnv env, IloBool symmetric=IloFalse, IloNum defaultVal=IloInfinity, IloInt size=1)`
`public`	`IloSparseExplicitVisitDistance(IloEnv env, IloBool symmetric, IloSimpleVisitDistanceFunction defaultFunc, IloInt size=1)`
`public`	`~IloSparseExplicitVisitDistance()`

Method Summary	
`public IloNum`	`getDistance(IloVisit visit1, IloVisit visit2, IloVehicle veh) const`
`public IloBool`	`isSet(IloVisit visit1, IloVisit visit2) const`
`public void`	`setValue(IloVisit visit1, IloVisit visit2, IloNum value)`

Inherited Methods from `IloVisitDistance`
`end, Exists, Find, getDistance, getGroup, getImpl, getKey, refresh, removeKey, setKey`

Constructors and Destructors

`public` **`IloSparseExplicitVisitDistance`**`(IloEnv env, IloBool symmetric=IloFalse, IloNum defaultVal=IloInfinity, IloInt size=1)`

This constructor creates a sparse explicit distance object in the environment `env`. The parameter `defaultVal` allows you to specify the value that will be returned when no actual distance between two visits has been specified. Infinity is returned only if you do not override it with a value of your choice.

The optional parameter `symmetric` is set to `IloTrue` if the sparse distance matrix is symmetric. The distance matrix is symmetric if, for any two visits *a* and *b*, *dist(a,b)=dist(b,a)*. The optional parameter `size` allows you to pre-size the data structure used to store the data.

For speed considerations, it is recommended to set the `size` parameter to approximately the number of entries that will be in the sparse distance matrix. If the instance is symmetric, you only need to allocate space for half the entries. In all cases, the constructor will round `size` to the power of two that is smaller or equal to the parameter `size`.

`public` **`IloSparseExplicitVisitDistance`**`(IloEnv env, IloBool symmetric,` `IloSimpleVisitDistanceFunction defaultFunc, IloInt size=1)`

This constructor creates a sparse explicit distance object in the environment `env`. The parameter `defaultFunc` allows you to specify a distance function which will be called when no actual distance between two visits has been specified.

The parameter `symmetric` is set to `IloTrue` if the sparse distance matrix is symmetric. The distance matrix is symmetric if, for any two visits *a* and *b*, *dist(a,b)=dist(b,a)*. The optional parameter `size` allows you to pre-size the data structure used to store the data.

For speed considerations, it is recommended to set the `size` parameter to approximately the number of entries that will be in the sparse distance matrix. If the instance is symmetric, you only need to allocate space for half the entries. In all cases, the constructor will round `size` to the power of two that is smaller or equal to the parameter `size`.

`public` **`~IloSparseExplicitVisitDistance`**`()`

This destructor returns the memory used by the matrix.

Methods

`public IloNum` **`getDistance`**`(IloVisit visit1, IloVisit visit2, IloVehicle veh) const`

This member function returns the distance from `visit1` to `visit2` in the sparse matrix.

If a value for two given nodes has not been set using `setValue`, then an exception is thrown. You can use the member function `isSet` to check if a distance has already been set.

Note

Note that the distances are not vehicle-dependent. The parameter `veh` is included because this is the default interface to distance data. However, since `setValue` does not set vehicle-specific data, the sparse matrix does not contain vehicle-specific data.

`public IloBool` **`isSet`**`(IloVisit visit1, IloVisit visit2) const`

This member function returns `IloTrue` if the distance between `visit1` and `visit2` has already been set by `setValue`.

The class `IloSparseExplicitVisitDistance` throws an exception if you try to access data that has not been set using `setValue`.

`public void` **`setValue`**`(IloVisit visit1, IloVisit visit2, IloNum value)`

This member function sets the explicit distance `value` between two visits, `visit1` and `visit2`.

Note

Note that you can change a pre-existing `value` by using `setValue` several times. The distance is set to the `value` specified by the last call to `setValue`.

Class IloTravelSumVar

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

A travel sum variable is a constrained variable representing the sum of the travel variables of the visits belonging to the route of a vehicle for a given extrinsic dimension.

This variable can be used to limit the total distance traveled by a vehicle.

See Also: IloDimension2, IloDelaySumVar, IloVehicle, IloVisit, operator+, IloVehicle::getTravelSumVar

Constructor Summary	
`public`	`IloTravelSumVar(IloVehicle vehicle, IloDimension2 dim2)`

Constructors

`public` **`IloTravelSumVar`**`(IloVehicle vehicle, IloDimension2 dim2)`

This constructor creates a travel sum variable from a vehicle and an extrinsic dimension.

Class IloVehicle

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

Vehicles carry the goods delivered during visits. They interact with other objects through *dimensions* that are used to express *cost*s and *capacities*. They can also have variable start and end times (which are defined as *time windows* on the vehicle's first and last visits).

See Also: IloDimension, IloDimension1, IloDimension2, IloVehicleArray, IloVehicleBreakCon, IloVehicleVar, IloVehicleToNumFunction, IloVisit

Constructor Summary	
`public`	`IloVehicle(IloVisit first, IloVisit last, const char * name=0)`
`public`	`IloVehicle(IloEnv env, const char * name=0)`

Method Summary	
`public static IloBool`	`Exists(IloEnv env, const char * key)`
`public static IloVehicle`	`Find(IloEnv env, const char * key)`
`public IloNum`	`getCapacity(IloDimension1 dim) const`
`public IloNum`	`getCost(IloDimension dim) const`
`public IloNum`	`getCost() const`
`public IloNumVar`	`getCostVar() const`
`public IloDelaySumVar`	`getDelaySumVar(IloDimension2 dim) const`
`public IloVisit`	`getFirstVisit() const`
`public const char *`	`getKey() const`
`public IloVisit`	`getLastVisit() const`
`public IloNum`	`getSpeed(IloDimension2 dim) const`
`public IloNumVar`	`getTransitSumVar(IloDimension dim) const`
`public IloTravelSumVar`	`getTravelSumVar(IloDimension2 dim) const`
`public void`	`removeKey()`
`public void`	`setCapacity(IloDimension1 dim, IloNum capacity) const`
`public void`	`setCost(IloDimension dim, IloNumToNumStepFunction func)`
`public void`	`setCost(IloDimension dim, IloNumToNumSegmentFunction func)`
`public void`	`setCost(IloDimension dim, IloVisitToNumFunction func, IloVisitVar var) const`
`public void`	`setCost(IloDimension dim, IloNum unitCost) const`
`public void`	`setCost(IloNum val) const`
`public void`	`setFirstVisit(IloVisit first) const`
`public void`	`setKey(const char * key)`
`public void`	`setLastVisit(IloVisit last) const`
`public void`	`setSpeed(IloDimension2 dim, IloNum speed) const`

Constructors

```
public IloVehicle(IloVisit first, IloVisit last, const char * name=0)
```

This constructor creates a vehicle. The vehicle's starting and ending visits are `first` and `last`. The optional argument `name`, if provided, becomes the name of the vehicle.

```
public IloVehicle(IloEnv env, const char * name=0)
```

This constructor creates a vehicle associated with environment `env`. The optional argument `name`, if provided, becomes the name of the vehicle.

This vehicle does not have predetermined starting and ending visits. In fact, the start and end visits for the vehicle are located at “everywhere” nodes, which has the effect that the distance function is not called to compute any distance from the start or end point of the vehicle. Instead, all such distances are assumed to be zero.

Methods

```
public static IloBool Exists(IloEnv env, const char * key)
```

This static member function returns `IloTrue` if an `IloVehicle` object having key `key` exists and `IloFalse` if not.

```
public static IloVehicle Find(IloEnv env, const char * key)
```

This static member function returns the object corresponding to the key `key` set using `IloVehicle::setKey`. If there is no object corresponding to `key` an `IloException` is thrown.

```
public IloNum getCapacity(IloDimension1 dim) const
```

This member function returns the capacity of the invoking vehicle object in the intrinsic dimension `dim`.

```
public IloNum getCost(IloDimension dim) const
```

This member function returns the cost per unit of `dim` associated with the invoking vehicle object.

```
public IloNum getCost() const
```

This member function returns the fixed cost associated with the invoking vehicle object.

```
public IloNumVar getCostVar() const
```

This member function returns an object corresponding to the total cost (fixed and proportional) associated with the invoking vehicle object.

```
public IloDelaySumVar getDelaySumVar(IloDimension2 dim) const
```

This member function returns a numerical variable representing the sum of the delay variables of the visits belonging to the route of the invoking vehicle for the extrinsic dimension `dim`.

If the extrinsic dimension represents time, this variable can be used to limit the total service time spent by a vehicle.

See Also: IloDelaySumVar

```
public IloVisit getFirstVisit() const
```

This member function returns the first visit of the invoking vehicle object, which corresponds to the vehicle's start point.

```
public const char * getKey() const
```

This member function returns the key set on the invoking object

```
public IloVisit getLastVisit() const
```

This member function returns the last visit made by the invoking vehicle object, which corresponds to the vehicle's end point.

```
public IloNum getSpeed(IloDimension2 dim) const
```

This member function returns the speed of the invoking vehicle object in dimension `dim`.

```
public IloNumVar getTransitSumVar(IloDimension dim) const
```

This member function returns a numerical variable corresponding to the sum of the transit variables, for dimension `dim`, for all visits assigned to the invoking vehicle.

```
public IloTravelSumVar getTravelSumVar(IloDimension2 dim) const
```

This member function returns a numerical variable representing the sum of the travel variables of the visits belonging to the route of the invoking vehicle for the extrinsic dimension `dim`.

This variable can be used to limit the total distance traveled by a vehicle.

See Also: IloTravelSumVar

```
public void removeKey()
```

This member function allows the user to remove the key set on the invoking object.

```
public void setCapacity(IloDimension1 dim, IloNum capacity) const
```

This member function sets the capacity of the invoking vehicle object to `capacity` according to dimension `dim`.

```
public void setCost(IloDimension dim, IloNumToNumStepFunction func)
```

This member function associates a cost function with the invoking vehicle object. The function `func` represents the value of the cost for the dimension `dim` according to the value of the corresponding transit sum variable. If a cost function has already been specified using this member function, it will be replaced by `func`. If a cost function has been specified using cost coefficients, the function `func` is added to it.

```
public void setCost(IloDimension dim, IloNumToNumSegmentFunction func)
```

This member function associates a cost function with the invoking vehicle object. The function `func` represents the value of the cost for the dimension `dim` according to the value of the corresponding transit sum variable. If a cost function has already been specified using this member function, it will be replaced by `func`. If a cost function has been specified using cost coefficients, the function `func` is added to it.

```
public void setCost(IloDimension dim, IloVisitToNumFunction func, IloVisitVar var)
const
```

This member function associates a visit-dependent proportional cost with the invoking vehicle object. The function `func` represents the value of the cost coefficient for the dimension `dim` according to the value of `var`.

```
public void setCost(IloDimension dim, IloNum unitCost) const
```

This member function associates a proportional cost with the invoking vehicle object. This cost is equal to `unitCost` per unit of dimension `dim`. In Dispatcher, the cost function can use negative elements and thus be negative itself if so desired. This member function can be used with a negative value for `unitCost`.

```
public void setCost(IloNum val) const
```

This member function associates a fixed cost, `val`, with the invoking vehicle object. In Dispatcher, the cost function can use negative elements and thus be negative itself if so desired. This member function can be used with a negative value for `val`.

```
public void setFirstVisit(IloVisit first) const
```

This member function sets `first` as the first visit for the invoking vehicle object.

```
public void setKey(const char * key)
```

This member function allows the user to set `key` on the invoking object. This key is unique. Each vehicle must have a different key; otherwise, an exception is thrown.

```
public void setLastVisit(IloVisit last) const
```

This member function sets `last` as the last visit for the invoking vehicle object.

```
public void setSpeed(IloDimension2 dim, IloNum speed) const
```

This member function sets the speed of `dim` to be equal to `speed`. The result of the computation of distance between two nodes is divided by the factor of `speed`. For example, if the instance of `IloDistance` associated with `dim` returns *6* for a pair of nodes, and the speed of the vehicle is set to *2*, the distance taken into account by the vehicle will be *6/2 = 3*. By default, the speed of a vehicle is equal to 1.

Class IloVehicleBreakCon

Definition file: ildispat/ilobreak.h
Include file: <ildispat/ilodispatcher.h>

A *break* is a period of time during a route when a vehicle is not available to make a visit, such as during the driver's lunch period. Breaks are performed by a vehicle in a particular *dimension* (usually time). The break includes a *start time*, *duration*, and a *position* that can be variable. Breaks can interrupt customer visits or not, as desired.

Breaks operate by using up waiting time between one visit and the next. (See `IloVisit::getWaitVar`.)

Breaks are instantiated through the use of goals.

See Also: IloDimension2, IloInstantiateVehicleBreak, IloInstantiateVehicleBreakDuration, IloInstantiateVehicleBreakPosition, IloInstantiateVehicleBreaks, IloInstantiateVehicleBreakStart, IloVehicle

Constructor Summary	
public	IloVehicleBreakCon(IloVehicle vehicle, IloDimension2 dim2, IloNumVar startVar, IloNumVar durationVar, const char * name=0)
public	IloVehicleBreakCon(IloVehicle vehicle, IloDimension2 dim2, IloNumVar startVar, IloNum duration, const char * name=0)

Method Summary	
public static IloBool	Exists(IloEnv env, const char * key)
public static IloVehicleBreakCon	Find(IloEnv env, const char * key)
public IloDimension2	getDimension() const
public IloNumVar	getDurationVar() const
public const char *	getKey() const
public IloVisitVar	getPositionVar() const
public IloNumVar	getStartVar() const
public IloVehicle	getVehicle() const
public IloConstraint	justAfter(IloVisitArray visits) const
public IloConstraint	justAfter(IloVisit visit) const
public IloConstraint	performed() const
public void	removeKey()
public void	setKey(const char * key)
public IloConstraint	unperformed() const

Constructors

public **IloVehicleBreakCon**(IloVehicle vehicle, IloDimension2 dim2, IloNumVar startVar, IloNumVar durationVar, const char * name=0)
public **IloVehicleBreakCon**(IloVehicle vehicle, IloDimension2 dim2, IloNumVar startVar, IloNum duration, const char * name=0)

These constructors create a vehicle break constraint on vehicle `vehicle` in the dimension `dim` with a start time of `startVar` and duration of either `durationVar` or `duration`.

Methods

`public static IloBool` **`Exists`**`(IloEnv env, const char * key)`

This static member function returns `IloTrue` if an `IloVehicleBreakCon` object having key `key` exists and `IloFalse` if not.

`public static IloVehicleBreakCon` **`Find`**`(IloEnv env, const char * key)`

This static member function returns the object corresponding to the key `key` set using `IloVehicleBreakCon::setKey`. If there is no object corresponding to `key` an `IloException` is thrown.

`public IloDimension2` **`getDimension`**`() const`

This member function returns the dimension associated with the invoking vehicle break constraint.

`public IloNumVar` **`getDurationVar`**`() const`

This member function returns the constrained expression associated with the invoking vehicle break constraint representing the duration of the break.

`public const char *` **`getKey`**`() const`

The following member function returns the key set on the invoking object

`public IloVisitVar` **`getPositionVar`**`() const`

This member function returns the visit-variable associated with the invoking vehicle break constraint. This visit-variable, an instance of `IloVisitVar`, is a constrained variable representing the visit immediately before the break.

`public IloNumVar` **`getStartVar`**`() const`

This member function returns the numeric variable associated with the invoking vehicle break constraint representing the start time of the break.

`public IloVehicle` **`getVehicle`**`() const`

This member function returns the vehicle with which the invoking vehicle break constraint is associated.

`public IloConstraint` **`justAfter`**`(IloVisitArray visits) const`

This member function constrains the invoking break to happen just after one of the visits of `visits`.

```
public IloConstraint justAfter(IloVisit visit) const
```

This member function constrains the invoking break to happen just after `visit`.

```
public IloConstraint performed() const
```

This member function returns a constraint stating that the invoking vehicle break must be performed. This constraint is useful for creating goals to instantiate breaks when those breaks are involved in metaconstraints.

```
public void removeKey()
```

The following member function allows the user to remove the key set on the invoking object.

```
public void setKey(const char * key)
```

This member function allows the user to set `key` on the invoking object. This key is unique. Each break must have a different key; otherwise, an exception is thrown.

```
public IloConstraint unperformed() const
```

This member function returns a constraint stating that the invoking vehicle break must not be performed. This constraint is useful for creating goals to instantiate breaks when those breaks are involved in metaconstraints.

Class IloVehicleBreakConIterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

An instance of the class `IloVehicleBreakConIterator` is an iterator that traverses all instances of the class `IloVehicleBreakCon` in a model.

See Also: IloVehicleBreakCon

Constructor Summary	
`public`	`IloVehicleBreakConIterator(IloModel mdl, IloBool deep=IloTrue)`
`public`	`IloVehicleBreakConIterator(const IloVehicleBreakConIterator & iter)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVehicleBreakCon`	`operator*() const`
`public const IloVehicleBreakConIterator &`	`operator++()`
`public const IloVehicleBreakConIterator &`	`operator=(const IloVehicleBreakConIterator & iter)`

Constructors

`public` **`IloVehicleBreakConIterator`**`(IloModel mdl, IloBool deep=IloTrue)`

This constructor creates an iterator which will iterate over all instances of `IloVehicleBreakCon` in model `mdl`. If the parameter `deep` has the value `IloTrue`, all submodels of `mdl` will form part of the iteration. If `deep` has the value `IloFalse`, submodels will not be investigated by the iterator.

`public` **`IloVehicleBreakConIterator`**`(const IloVehicleBreakConIterator & iter)`

This copy constructor creates an iterator from another iterator `iter`. After execution, both the newly created iterator and `iter` will be at the same position within the model.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloFalse` if the iterator has scanned all instances of `IloVehicleBreakCon` in the model. Otherwise, it returns `IloTrue`.

`public IloVehicleBreakCon` **`operator*`**`() const`

This operator returns the instance of `IloVehicleBreakCon` at which the iterator is currently pointing.

`public const IloVehicleBreakConIterator & **operator++**()`

This operator moves the iterator on to the next instance of `IloVehicleBreakCon` within the model, providing one exists. The operator returns the invoking iterator at its new position.

`public const IloVehicleBreakConIterator & **operator=**(const IloVehicleBreakConIterator & iter)`

This assignment operator copies the state of `iter` to the iterator on the left-hand side of the operator. After execution, both iterators will be at the same position within the model.

Class IloVehicleEquiv

Definition file: ildispat/ilovehicleequiv.h
Include file: <ildispat/ilodispatcher.h>

This class is the handle class of `IloVehicleEquivI`, which is used to define equivalence of pairs of vehicles.

When distance functions are specified in Dispatcher, they can be cached, if distance computations are slow, through `IloDimension2::setCached`. (Normally, caching of distance functions is disabled.) When the distance depends not only on the starting and ending node, but the vehicle used to perform the trip, it becomes useful to introduce the notion of *vehicle equivalence*.

If two vehicles are specified as equivalent with respect to a particular distance metric and the distance between two nodes using one of the vehicles resides in the cache, the distance between the same two nodes using the other vehicle can be assumed to be the same. Thus, fewer cache slots are used. It is functionally unnecessary to specify a vehicle equivalence class in Dispatcher, but definition of such a class can lead to speed increases through better caching of distance data.

See Also: IloAllVehiclesDifferent, IloAllVehiclesEquivalent, IloVehicleEquivEvalI, IloVehicleEquivI

Constructor and Destructor Summary	
`public`	`IloVehicleEquiv(IloEnv env, IloVehicleEquivFunction equivFunction)`

Method Summary	
`public void`	`end()`
`public IloEnv`	`getEnv() const`
`public IloInt`	`getGroup(IloVehicle vehicle) const`
`public IloBool`	`isEquivalent(IloVehicle vehicle1, IloVehicle vehicle2) const`
`public void`	`update(IloEnv env) const`

Constructors and Destructors

`public` **`IloVehicleEquiv`**`(IloEnv env, IloVehicleEquivFunction equivFunction)`

This constructor creates a vehicle equivalence object for the environment `env`, using vehicle equivalence function `equivFunction`. In order to do this, the constructed handle will point to an implementation of type `IloVehicleEquivEvalI`, which was constructed using `equivFunction`.

Methods

`public void` **`end`**`()`

This member function frees all resources used by the invoking vehicle equivalence object. You cannot use the invoking vehicle equivalence object after a call to this member function.

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment of the invoking vehicle equivalence object.

```
public IloInt getGroup(IloVehicle vehicle) const
```

This member function returns the group, which is internally created and managed by Dispatcher, for `vehicle`.

Two vehicles deemed equal by `IloVehicleEquiv::isEquivalent` have the same group identifier. Those that are deemed different by `isEquivalent` have different group identifiers.

```
public IloBool isEquivalent(IloVehicle vehicle1, IloVehicle vehicle2) const
```

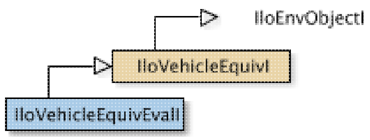
This member function makes a call to `IloVehicleEquivI::isEquivalent` and returns `IloTrue` if `vehicle1` and `vehicle2` are equivalent. It returns `IloFalse` otherwise.

```
public void update(IloEnv env) const
```

This member function is called when the equivalence of vehicles may have changed, for example when a new vehicle is created.

Class IloVehicleEquivEvalI

Definition file: ildispat/ilovehicleequiv.h
Include file: <ildispat/ilodispatcher.h>



This class is a special subclass of `IloVehicleEquivI`, which redefines the member function `IloVehicleEquivEvalI::isEquivalent` using an equivalence function passed as an argument in the constructor.

The type of this vehicle equivalence evaluation function is `IloVehicleEquivFunction`.

See Also: IloAllVehiclesDifferent, IloAllVehiclesEquivalent, IloVehicleEquiv, IloVehicleEquivI

Constructor and Destructor Summary	
`public`	`IloVehicleEquivEvalI(IloEnv env, IloVehicleEquivFunction equivFunction)`

Method Summary	
`public IloBool`	`isEquivalent(IloVehicle vehicle1, IloVehicle vehicle2) const`

Inherited Methods from **`IloVehicleEquivI`**
`getEnv, getGroup, isEquivalent, update`

Constructors and Destructors

`public` **`IloVehicleEquivEvalI`**`(IloEnv env, IloVehicleEquivFunction equivFunction)`

This constructor creates an instance of a vehicle equivalence class from a routing plan and an equivalence function. It works by redefining `IloVehicleEquivI::isEquivalent` to use the function `equivFunction`.

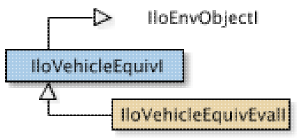
Methods

`public IloBool` **`isEquivalent`**`(IloVehicle vehicle1, IloVehicle vehicle2) const`

This function makes a call to `equivFunction`, using two vehicles as arguments. If the two vehicles are equivalent, `IloTrue` is returned. Otherwise, `IloFalse` is returned.

Class IloVehicleEquivI

Definition file: ildispat/ilovehicleequiv.h
Include file: <ildispat/ilodispatcher.h>



The equivalence of pairs of vehicles, for distance caching purposes, is specified by defining the pure virtual member function `IloVehicleEquivI::isEquivalent`, which takes two vehicles as arguments.

If the equivalence of your vehicles can be expressed by a function rather than a class, you can use `IloVehicleEquivEvalI`, or the constructor `IloVehicleEquiv` specifying a function.

When distance functions are specified in Dispatcher, they can be cached, if distance computations are slow, through `IloDimension2::setCached`. (Normally, caching of distance functions is disabled.) When the distance depends not only on the starting and ending node, but the vehicle used to perform the trip, it becomes useful to introduce the notion of *vehicle equivalence*.

If two vehicles are specified as equivalent with respect to a particular distance metric and the distance between two nodes using one of the vehicles resides in the cache, the distance between the same two nodes using the other vehicle can be assumed to be the same. Thus, fewer cache slots are used. It is functionally unnecessary to specify a vehicle equivalence class in Dispatcher, but definition of such a class can lead to speed increases through better caching of distance data.

See Also: IloAllVehiclesDifferent, IloAllVehiclesEquivalent, IloVehicleEquiv, IloVehicleEquivEvalI

Constructor and Destructor Summary	
`public`	`IloVehicleEquivI(IloEnv env)`

Method Summary	
`public IloEnv`	`getEnv() const`
`public IloInt`	`getGroup(const IloVehicle vehicle) const`
`public virtual IloBool`	`isEquivalent(IloVehicle vehicle1, IloVehicle vehicle2) const`
`public void`	`update()`

Constructors and Destructors

`public` **`IloVehicleEquivI`**`(IloEnv env)`

This constructor creates a vehicle equivalence object associated with vehicles from the environment `env`.

Methods

`public IloEnv` **`getEnv`**`() const`

This member function returns the environment of the invoking vehicle equivalence object.

```
public IloInt getGroup(const IloVehicle vehicle) const
```

This member function returns the group, which is internally created and managed by Dispatcher, for `vehicle`. Two vehicles deemed equivalent by `isEquivalent` have the same group identifier. Those that are deemed different by `isEquivalent` have different group identifiers.

```
public virtual IloBool isEquivalent(IloVehicle vehicle1, IloVehicle vehicle2) const
```

This pure virtual member function returns `IloTrue` if the two instances of `IloVehicle` are equivalent. Otherwise, it returns `IloFalse`.

```
public void update()
```

This member function is called when the equivalence of vehicles may have changed, for example when a new vehicle is created.

Note

It is not normally necessary to redefine this function, as its default behavior is to update an internal table of groups for the vehicles. However, it can be defined if additional behavior is desired.

In such a case, `IloVehicleEquivI::update` must always be called in any redefined update function. It is recommended that this function be redefined with extreme care, and only where necessary.

NEEDS TO BE REMOVED AND REPLACED BY void update(IloEnv env)

Class IloVehicleIterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

An instance of the class `IloVehicleIterator` is an iterator that traverses all instances of the class `IloVehicle` in a model.

See Also: IloVehicle

Constructor Summary	
`public`	`IloVehicleIterator(IloModel mdl, IloBool deep=IloTrue)`
`public`	`IloVehicleIterator(const IloVehicleIterator & iter)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVehicle`	`operator*() const`
`public const IloVehicleIterator &`	`operator++()`
`public const IloVehicleIterator &`	`operator=(const IloVehicleIterator & iter)`

Constructors

`public` **`IloVehicleIterator`**`(IloModel mdl, IloBool deep=IloTrue)`

This constructor creates an iterator which will iterate over all instances of `IloVehicle` in model `mdl`. If the parameter `deep` has the value `IloTrue`, all submodels of `mdl` will form part of the iteration. If `deep` has the value `IloFalse`, submodels will not be investigated by the iterator.

`public` **`IloVehicleIterator`**`(const IloVehicleIterator & iter)`

This copy constructor creates an iterator from another iterator `iter`. After execution, both the newly created iterator and `iter` will be at the same position within the model.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloFalse` if the iterator has scanned all instances of `IloVehicle` in the model. Otherwise, it returns `IloTrue`.

`public IloVehicle` **`operator*`**`() const`

This operator returns the instance of `IloVehicle` at which the iterator is currently pointing.

`public const IloVehicleIterator & **operator++**()`

This operator moves the iterator on to the next instance of `IloVehicle` within the model, providing one exists. The operator returns the invoking iterator at its new position.

`public const IloVehicleIterator & **operator=**(const IloVehicleIterator & iter)`

This assignment operator copies the state of `iter` to the iterator on the left-hand side of the operator. After execution, both iterators will be at the same position within the model.

Class IloVehicleLIFOConstraint

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This class is a subclass of `IloConstraint`. This constraint ensures that pickup and delivery visits are performed in reverse order along the route of a vehicle. This is equivalent to stating that what is loaded last on the vehicle must be unloaded first.

See Also: IloVehicle

Constructor Summary	
`public`	`IloVehicleLIFOConstraint(IloVehicle vehicle)`

Constructors

```
public IloVehicleLIFOConstraint(IloVehicle vehicle)
```

This constructor creates a last-in, first-out constraint on `vehicle`.

Class IloVehiclePair

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This class represents a pair of vehicles.

See Also: IloVehicle

Constructor Summary	
`public`	`IloVehiclePair(IloVehicle vehicle1, IloVehicle vehicle2)`

Method Summary	
`public IloVehicle`	`getVehicle1() const`
`public IloVehicle`	`getVehicle2() const`

Constructors

`public` **`IloVehiclePair`**`(IloVehicle vehicle1, IloVehicle vehicle2)`

This constructor creates a vehicle pair from the two vehicles `vehicle1` and `vehicle2`.

Methods

`public IloVehicle` **`getVehicle1`**`() const`

This member function returns the first vehicle of the pair.

`public IloVehicle` **`getVehicle2`**`() const`

This member function returns the second vehicle of the pair.

Class IloVehicleToNumFunction

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This class is the handle class of `IloVehicleToNumFunctionI`, the class that defines a function from a vehicle to an `IloNum`. The member function `IloVehicleToNumFunction::getValue` returns the value corresponding to a visit. The member function `IloVehicleToNumFunction::getUnperformedValue` returns the value corresponding to the unperformed state of a visit. This function can be used to create an `IloNumVar` whose domain is the values of the function accessed through an `IloVehicleVar`.

See Also: IloVehicle, IloVehicleVar

Constructor Summary	
`public`	`IloVehicleToNumFunction()`
`public`	`IloVehicleToNumFunction(IloVehicleToNumFunctionI * impl)`
`public`	`IloVehicleToNumFunction(const IloVehicleToNumFunction & func)`
`public`	`IloVehicleToNumFunction(IloEnv env, IloVehicleArray vehicles, IloNumArray values)`
`public`	`IloVehicleToNumFunction(IloEnv env, IloVehicleArray vehicles, IloNumArray values, IloNum unperformedValue, IloNum defaultValue)`
`public`	`IloVehicleToNumFunction(IloEnv env, IloVehicleArray vehicles, IloNumArray values, IloNum unperformedValue)`
`public`	`IloVehicleToNumFunction(IloEnv env, IloSimpleVehicleToNumFunction f)`

Method Summary	
`public IloVehicleToNumFunctionI *`	`getImpl() const`
`public IloNum`	`getUnperformedValue() const`
`public IloNum`	`getValue(IloVehicle vehicle) const`
`public IloNumVar`	`operator()(IloVehicleVar var) const`
`public void`	`operator=(const IloVehicleToNumFunction & func)`

Constructors

`public` **`IloVehicleToNumFunction`**`()`

This constructor creates a vehicle to `IloNum` function whose handle pointer is null. This object must be assigned before it can be used.

`public` **`IloVehicleToNumFunction`**`(IloVehicleToNumFunctionI * impl)`

This constructor creates a handle object (an instance of `IloVehicleToNumFunction`) from a pointer to an implementation object (an instance of the class `IloVehicleToNumFunctionI`).

`public` **`IloVehicleToNumFunction`**`(const IloVehicleToNumFunction & func)`

This copy constructor creates a handle from a reference to a vehicle to `IloNum` function. That vehicle to `IloNum` object and `func` both point to the same implementation object.

`public` **`IloVehicleToNumFunction`**`(IloEnv env, IloVehicleArray vehicles, IloNumArray values)`

This constructor takes an array of vehicles and an array of values. Both arrays should have the same size. The vehicle and the value with the same index in these arrays are associated.

`public` **`IloVehicleToNumFunction`**`(IloEnv env, IloVehicleArray vehicles, IloNumArray values, IloNum unperformedValue, IloNum defaultValue)`

This constructor takes an array of vehicles and an array of values. The vehicle and the value with the same index in these arrays are associated. The value `unperformedValue` corresponds to the visit unperformed state. The value `defaultValue` is the value returned by the function for a vehicle for which no value has been specified. The implementation object of the newly created handle is an instance of `IloArrayVehicleToNumFunctionI`.

`public` **`IloVehicleToNumFunction`**`(IloEnv env, IloVehicleArray vehicles, IloNumArray values, IloNum unperformedValue)`

This constructor takes an array of vehicles and an array of values. The vehicle and the value with the same index in these arrays are associated. The value `unperformedValue` corresponds to the visit unperformed state. The implementation object of the newly created handle is an instance of `IloArrayVehicleToNumFunctionI`.

`public` **`IloVehicleToNumFunction`**`(IloEnv env, IloSimpleVehicleToNumFunction f)`

This constructor creates a vehicle to `IloNum` function based on `f`. The implementation object of the newly created handle is an instance of `IloEvalVehicleToNumFunctionI`.

Methods

`public IloVehicleToNumFunctionI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking function.

`public IloNum` **`getUnperformedValue`**`() const`

This member function returns the value associated with the visit unperformed state.

`public IloNum` **`getValue`**`(IloVehicle vehicle) const`

This member function returns the value associated with `vehicle`.

`public IloNumVar` **`operator()`**`(IloVehicleVar var) const`

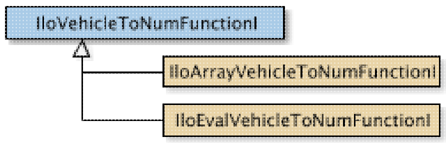
This operator returns an `IloNumVar` constrained by `var`. For clarity, let's call *f* the invoking function. When `var` is bound to the vehicle *v*, the value of the expression is *f.getValue(v)*. More generally, the domain of the `IloNumVar` is the set of values *{gi| gi = f(vi)}* where the *vi* are in the domain of `var`.

```
public void operator=(const IloVehicleToNumFunction & func)
```

This operator assigns an address to the handle pointer of the invoking vehicle to `IloNum` function. That address is the location of the implementation object of the argument `func`. After the execution of this operator, the invoking function and `func` both point to the same implementation object.

Class IloVehicleToNumFunctionI

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>



This class is the implementation class of `IloVehicleToNumFunction`, the class that defines a function from a vehicle to an `IloNum`. The virtual member function `IloVehicleToNumFunctionI::getValue` returns the value corresponding to a vehicle. The virtual member function `IloVehicleToNumFunctionI::getUnperformedValue` returns the value corresponding to the unperformed state of a visit. This function can be used to create an `IloNumVar` whose domain is the values of the function accessed through an `IloVehicleVar`.

Constructor and Destructor Summary	
`public`	`IloVehicleToNumFunctionI(IloEnv env)`

Method Summary	
`public virtual IloNum`	`getUnperformedValue()`
`public virtual IloNum`	`getValue(IloVehicle vehicle)`

Constructors and Destructors

`public` **`IloVehicleToNumFunctionI`**`(IloEnv env)`

This constructor creates an implementation vehicle to `IloNum` object in the environment `env`.

Methods

`public virtual IloNum` **`getUnperformedValue`**`()`

This virtual member function can be redefined to return a numeric value corresponding to the unperformed state of a visit. For a given function, the return value should not vary between two calls. By default, this member function returns 0.

`public virtual IloNum` **`getValue`**`(IloVehicle vehicle)`

This pure virtual member function must be redefined to return a numeric value corresponding to `vehicle`. For a given function, the return value should not vary between two calls with the same parameter.

Class IloVehicleVar

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

A visit is performed by only one vehicle. A vehicle variable is a constrained variable representing the vehicle performing the associated visit. In Solver, it is extracted to an `IlcIntVar` representing the index of the vehicle performing the associated visit.

See Also: IloVehicle, IloVehicleToNumFunction

Method Summary	
`public IloVisit`	`getVisit() const`

Methods

`public IloVisit` **`getVisit`**`() const`

This member function returns the visit associated with the invoking vehicle variable.

Class IloVisit

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

Visits represent the activities vehicles must perform. Visits are made at a node. Each visit has a *service time* depending on *dimensions*. Visits can also have time windows in various dimensions—such as their visiting hours and availability.

A visit is performed by only one vehicle.

See Also: IloDimension, IloDimension1, IloDimension2, IloVisitVar, IloVehicle, IloVisitArray, IloVisitToNumFunction

Constructor Summary	
`public`	`IloVisit(IloNode node, const char * name=0)`
`public`	`IloVisit(IloNode node1, IloNode node2, const char * name=0)`

Method Summary	
`public static IloBool`	`Exists(IloEnv env, const char * key)`
`public static IloVisit`	`Find(IloEnv env, const char * key)`
`public IloNumVar`	`getCumulVar(IloDimension dim) const`
`public IloNumVar`	`getDelayVar(IloDimension2 dim) const`
`public IloNum`	`getDistanceTo(IloVisit visit, IloDimension2 dim, IloVehicle vehicle) const`
`public IloNumVar`	`getDurationVar(IloDimension2 dim) const`
`public IloNumToNumSegmentFunction`	`getEndCost(IloDimension2 dim) const`
`public IloNumVar`	`getEndCumulVar(IloDimension2 dim) const`
`public IloNode`	`getEndNode() const`
`public const char *`	`getKey() const`
`public IloVisitVar`	`getNextVar() const`
`public IloNode`	`getNode() const`
`public IloNum`	`getPenaltyCost() const`
`public IloVisitVar`	`getPrevVar() const`
`public IloNumVar`	`getRankVar() const`
`public IloNumToNumSegmentFunction`	`getStartCost(IloDimension2 dim) const`
`public IloNode`	`getStartNode() const`
`public IloNumVar`	`getTransitVar(IloDimension dim) const`
`public IloNumVar`	`getTravelVar(IloDimension2 dim) const`
`public IloVehicleVar`	`getVehicleVar() const`
`public IloNumVar`	`getWaitVar(IloDimension2 dim) const`
`public IloConstraint`	`isAfter(IloVisit visit) const`
`public IloConstraint`	`isBefore(IloVisit visit) const`

public IloBool	isBreakable() const
public IloBool	isFirstVisit() const
public IloBool	isLastVisit() const
public IloBool	operator!=(const IloVisit visit) const
public IloBool	operator==(const IloVisit visit) const
public IloConstraint	performed() const
public void	removeKey()
public void	setBreakable(IloBool val) const
public void	setEndCost(IloDimension2 dim, IloNumToNumSegmentFunction func)
public void	setKey(const char * key)
public void	setPenaltyCost(IloNum val) const
public void	setStartCost(IloDimension2 dim, IloNumToNumSegmentFunction func)
public IloConstraint	unperformed(IloBool deep=IloTrue) const

Constructors

public **IloVisit**(IloNode node, const char * name=0)

This constructor creates a visit issued by node. The optional argument name, if provided, becomes the name of the visit.

public **IloVisit**(IloNode node1, IloNode node2, const char * name=0)

This constructor creates a visit from two nodes, which means that the visit occurs on an arc rather than on a single node. The starting node of the arc is node1; the ending node of the arc is node2. The optional argument name, if provided, becomes the name of the visit.

Dispatcher computes the distance from this visit to a node *N* as the distance from node2 to *N*. Dispatcher computes the distance to this visit from a node *N* as the distance from *N* to node1.

This type of visit can be used to model arc-routing (as opposed to node-routing) problems.

Methods

public static IloBool **Exists**(IloEnv env, const char * key)

This static member function returns IloTrue if an IloVisit object having key key exists and IloFalse if not.

public static IloVisit **Find**(IloEnv env, const char * key)

This static member function returns the object corresponding to the key key set using IloVisit::setKey. If there is no object corresponding to key an IloException is thrown.

public IloNumVar **getCumulVar**(IloDimension dim) const

This member function returns the constrained cumulative expression associated with the invoking visit object and the parameter `dim`.

The cumuls on dimensions are handled by constraints that are functionally equivalent to one constraint of the following form for dimension `d`, and each pair of visits `v1` and `v2`.

```
IloIfThen(env,
          v1.isJustBefore(v2),
          v2.getCumulVar(d) == v1.getCumulVar(d)
                            + v1.getTransitVar(d));
```

`public IloNumVar` **`getDelayVar`**`(IloDimension2 dim) const`

This member function returns the constrained delay variable associated with the invoking visit object and the dimension `dim`.

The delay-variable of `v1`, returned by `v1.getDelayVar(d)`, represents the delay in terms of dimension `d` at visit `v1`. This is useful for representing the time needed to load/unload a truck.

`public IloNum` **`getDistanceTo`**`(IloVisit visit, IloDimension2 dim, IloVehicle vehicle) const`

This member function returns the distance from the invoking visit object to the object indicated by `visit` in the dimension `dim`.

`public IloNumVar` **`getDurationVar`**`(IloDimension2 dim) const`

This member function returns the variable representing the duration of the invoking visit on dimension `dim`. In the absence of vehicle breaks, or when the invoking visit is not breakable (see `isBreakable` and `setBreakable`), this member function returns a variable that is always maintained to be equivalent to that returned by `getDelayVar(dim)`. In the presence of vehicle breaks that break the invoking visit, this variable is constrained to be equal to the delay of the invoking visit plus the durations of any breaks that interrupt the visit.

`public IloNumToNumSegmentFunction` **`getEndCost`**`(IloDimension2 dim) const`

This member function returns a function representing the cost of performing the invoking visit according to the value of its end-cumul variable for the extrinsic dimension `dim`.

`public IloNumVar` **`getEndCumulVar`**`(IloDimension2 dim) const`

This member function returns the variable representing the end of the execution of the invoking visit. It is equal to the sum of the cumul variable (see `getCumulVar(IloDimension)`) and the duration variable (see `getDurationVar(IloDimension2)`).

`public IloNode` **`getEndNode`**`() const`

This member function returns the end node of the invoking visit object. It should be called if different start and end nodes were specified when creating the visit.

`public const char *` **`getKey`**`() const`

The following member function returns the key set on the invoking object

`public IloVisitVar` **`getNextVar`**`() const`

This member function returns the next-variable associated with the invoking visit object. This constrained variable denotes the index of the visit served immediately after the invoking visit object.

`public IloNode` **`getNode`**`() const`

This member function returns the node corresponding to the location that issued the order for the invoking visit object.

`public IloNum` **`getPenaltyCost`**`() const`

This member function returns the penalty cost that was set for not performing the visit to the invoking visit object.

`public IloVisitVar` **`getPrevVar`**`() const`

This member function returns the previous-variable associated with the invoking visit object. This constrained variable denotes the index of the visit served immediately before the invoking visit object.

`public IloNumVar` **`getRankVar`**`() const`

This member function returns the rank of the invoking visit object. A visit with `rank`*r* is the *rth* visit in a route. The visit returned by `IloVehicle::getFirstVisit` has rank 0.

`public IloNumToNumSegmentFunction` **`getStartCost`**`(IloDimension2 dim) const`

This member function returns a function representing the cost of performing the invoking visit according to the value of its cumul variable for the extrinsic dimension `dim`.

`public IloNode` **`getStartNode`**`() const`

This member function returns the starting node of the invoking visit object. It should be called if different start and end nodes were specified when creating the visit.

`public IloNumVar` **`getTransitVar`**`(IloDimension dim) const`

If `dim` is an instance of `IloDimension1`, this member function returns the constrained expression associated with the invoking visit object representing the quantity of the dimension `dim`. If `dim` is an instance of `IloDimension2`, this member function returns the constrained variable associated with the invoking visit object representing the sum of the delay, travel, and wait variables.

The transit variable associated with a visit represents the change in the cumul between that visit and the following visit. The way in which the transit variables are constrained varies according to the type of dimension under consideration. For intrinsic dimensions (of type `IloDimension1`), transit variables are not constrained (unless the user explicitly constrains them).

For a visit `v` and an extrinsic dimension `dim2` (of type `IloDimension2`), the transit variables are maintained by the following rule:

```
  v.getTransitVar(dim2) == v.getDelayVar(dim2)
                         + v.getTravelVar(dim2)
                         + v.getWaitVar(dim2);
```

Thus, the difference in extrinsic cumuls between any two successive visits is equal to the sum of the delay, travel, and wait variables of the first of the pair of visits under consideration.

`public IloNumVar` **`getTravelVar`**`(IloDimension2 dim) const`

This member function returns the constrained travel expression associated with the invoking visit object and the dimension `dim`.

The travel-variable of `v1`, returned by `v1.getTravelVar(d)`, represents the quantity of dimension `d` taken by `w`, the vehicle serving `v1,` to get to `v2`. It is maintained by the following rule:

```
  IloIfThen(env,
            v1.isJustBefore(v2) && w.visits(v1),
            v1.getTravelVar(d) == v1.getDistanceTo(v2, d, w)
                                / w.getSpeed(d));
```

`public IloVehicleVar` **`getVehicleVar`**`() const`

This member function returns the constrained variable representing the index of the vehicle performing the invoking visit object.

`public IloNumVar` **`getWaitVar`**`(IloDimension2 dim) const`

This member function returns the constrained wait variable associated with the invoking visit object and the dimension `dim`.

The wait-variable of `v1`, returned by `v1.getWaitVar(d)`, represents the additional quantity of dimension `d` consumed between `v1` and its successor, over the time required to serve `v1` and travel from `v1` to its successor. In the case where `d` represents time, `v1.getWaitVar(d)` represents the waiting time.

`public IloConstraint` **`isAfter`**`(IloVisit visit) const`

This member function returns a constraint stating that the invoking visit object must be performed after `visit` if they are on the same route.

`public IloConstraint` **`isBefore`**`(IloVisit visit) const`

This member function returns a constraint stating that the invoking visit object must be performed before `visit` if they are on the same route.

`public IloBool` **`isBreakable`**`() const`

This member function returns `IloTrue` if the invoking visit object is breakable by an `IloVehicleBreakCon` object. Otherwise, it returns `IloFalse`.

`public IloBool` **`isFirstVisit`**`() const`

This member function returns `IloTrue` if the invoking visit object is the first visit of a vehicle. Otherwise, it returns `IloFalse`.

`public IloBool` **`isLastVisit`**`() const`

This member function returns `IloTrue` if the invoking visit object is the last visit of a vehicle. Otherwise, it returns `IloFalse`.

`public IloBool` **`operator!=`**`(const IloVisit visit) const`

This operator returns `IloTrue` if the invoking visit object and `visit` point to different implementation objects. Otherwise it returns `IloFalse`.

`public IloBool` **`operator==`**`(const IloVisit visit) const`

This operator returns `IloTrue` if the invoking visit object and `visit` both point to the same implementation object. Otherwise, it returns `IloFalse`.

`public IloConstraint` **`performed`**`() const`

This member function returns a constraint stating that the invoking visit object must be performed by a vehicle. A performed visit is a visit that is assigned to a vehicle.

Normally, it is not necessary to use this constraint. As long as no penalty cost has been set on a visit, this constraint is satisfied automatically.

`public void` **`removeKey`**`()`

The following member function allows the user to remove the key set on the invoking object.

`public void` **`setBreakable`**`(IloBool val) const`

This member function sets the status of the invoking visit object to `breakable`. If a visit is breakable, a vehicle break can interrupt the visit. By default, visits are not breakable.

`public void` **`setEndCost`**`(IloDimension2 dim, IloNumToNumSegmentFunction func)`

This function sets `func` as the function representing the cost of performing the invoking visit according to the value of its end-cumul variable for the extrinsic dimension `dim`. It can be used to express earliness or tardiness costs on the end time of a visit using the `IloEarlinessFunction` or `IloTardinessFunction` functions.

```
public void setKey(const char * key)
```

This member function allows the user to set `key` on the invoking object. This key is unique. Each visit must have a different key; otherwise, an exception is thrown.

```
public void setPenaltyCost(IloNum val) const
```

This member function sets the cost of not performing a visit. By default, this cost is `IloInfinity`, which means the visit must be performed. If a visit is disabled, its penalty cost is not taken into account.

In Dispatcher, the cost function can use negative elements and thus be negative itself if so desired. This member function can be used with a negative value for `val`.

```
public void setStartCost(IloDimension2 dim, IloNumToNumSegmentFunction func)
```

This function sets `func` as the function representing the cost of performing the invoking visit according to the value of its cumul variable for the extrinsic dimension `dim`. It can be used to express earliness or tardiness costs on the start time of a visit using the `IloEarlinessFunction` or `IloTardinessFunction` functions.

Note

The segmented cost function must always take positive values.

```
public IloConstraint unperformed(IloBool deep=IloTrue) const
```

This member function returns a constraint stating that the invoking visit object must not be performed in a route.

Class IloVisitAlternativeConstraint

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This class is a subclass of `IloConstraint`. This constraint constrains a single visit out of a given set to be performed. It can be used to express alternatives between visits or visit disjunctions. All unperformed visits in the alternative set will be charged their unperformed penalty.

The following code

```
IloVisitAlternativeConstraint ct(env);
ct.add(visit1);
ct.add(visit2);
model.add(ct);
```

is equivalent to writing

```
model.add(visit1.performed() + visit2.performed() == 1);
```

Solutions containing visits constrained by this constraint can be modified using `IloVisitAlternativeSwap`.

In general, using this constraint will lead to increased performance from the local search neighborhoods provided by Dispatcher.

See Also: IloVisit, IloVisitAlternativeSwap

Constructor Summary	
`public`	`IloVisitAlternativeConstraint(IloEnv env)`

Method Summary	
`public void`	`add(IloVisit visit)`
`public IloBool`	`contains(IloVisit visit) const`
`public void`	`remove(IloVisit visit)`

Constructors

`public` **`IloVisitAlternativeConstraint`**`(IloEnv env)`

This constructor creates a visit alternative constraint. Initially the set of alternative visits is empty.

Methods

`public void` **`add`**`(IloVisit visit)`

This member function adds a visit to the set of alternative visits.

`public IloBool` **`contains`**`(IloVisit visit) const`

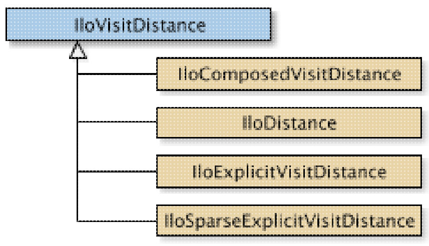
This member function returns a Boolean stating whether `visit` is part of the alternative visit set.

```
public void remove(IloVisit visit)
```

This member function removes a visit from the set of alternative visits.

Class IloVisitDistance

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher lets you define the distance function for a *dimension* (for example, the distance, the time, or the cost necessary for going from one visit to another).

This class is the handle class of the object that defines this distance function.

This handle class uses the virtual member function `IloVisitDistanceI::computeDistance` to retrieve distance values.

See Also: IloDimension, IloDimension1, IloDimension2, IloVisitDistanceEvalI, IloVisitDistanceFunction, IloVisitDistanceI, IloSimpleVisitDistanceEvalI, IloSimpleVisitDistanceFunction, IloComposedVisitDistance, IloExplicitVisitDistance, IloSparseExplicitVisitDistance

Constructor and Destructor Summary	
`public`	`IloVisitDistance(IloVisitDistanceI * dist=0)`
`public`	`IloVisitDistance(const IloVisitDistance & dist)`
`public`	`IloVisitDistance(IloEnv env, IloVisitDistanceFunction distFunction)`
`public`	`IloVisitDistance(IloEnv env, IloSimpleVisitDistanceFunction distFunction)`
`public`	`IloVisitDistance(IloVisitDistanceFunction distFunction, IloVehicleEquiv equiv)`

Method Summary	
`public void`	`end()`
`public static IloBool`	`Exists(IloEnv env, const char * key)`
`public static IloVisitDistance`	`Find(IloEnv env, const char * key)`
`public IloNum`	`getDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const`
`public IloInt`	`getGroup(IloVehicle vehicle) const`
`public IloVisitDistanceI *`	`getImpl() const`
`public const char *`	`getKey()`
`public void`	`refresh() const`
`public void`	`removeKey()`
`public void`	`setKey(const char * key)`

Constructors and Destructors

`public **IloVisitDistance**(IloVisitDistanceI * dist=0)`

This constructor creates a handle object (an instance of `IloVisitDistance`) from a pointer to an object (an instance of the implementation class `IloVisitDistanceI`).

`public **IloVisitDistance**(const IloVisitDistance & dist)`

This copy constructor creates a handle from a reference to a distance object. That distance object and `dist` both point to the same implementation object.

When distance functions are specified in Dispatcher, they can be cached, if distance computations are slow, through `IloDimension2::setCached`. (Normally, caching of distance functions is disabled.) When the distance depends not only on the starting and ending visit, but also the vehicle used to perform the trip, it becomes useful to introduce the notion of *vehicle equivalence*.

If two vehicles are specified as equivalent with respect to a particular distance metric and the distance between two visits using one of the vehicles resides in the cache, the distance between the same two visits using the other vehicle can be assumed to be the same. Thus, fewer cache slots are used. It is functionally unnecessary to specify a vehicle equivalence class in Dispatcher, but definition of such a class can lead to speed increases through better caching of distance data.

`public **IloVisitDistance**(IloEnv env, IloVisitDistanceFunction distFunction)`

This constructor creates a distance object in the environment `env`. The implementation object of the newly created handle is an instance of the class `IloVisitDistanceEvalI` constructed with the distance function `distFunction`.

`public **IloVisitDistance**(IloEnv env, IloSimpleVisitDistanceFunction distFunction)`

This constructor creates a distance object in the environment `env`. The implementation object of the newly created handle is an instance of the class `IloSimpleVisitDistanceEvalI` constructed with the distance function `distFunction`.

`public **IloVisitDistance**(IloVisitDistanceFunction distFunction, IloVehicleEquiv equiv)`

This constructor creates a handle to a distance object. The implementation object of this newly created handle is an instance of the class `IloVisitDistanceEvalI` constructed with the distance function `distFunction` for the vehicle equivalence group `equiv`.

Methods

`public void **end**()`

This member function frees all resources used by the invoking distance object. You cannot use the invoking distance object after a call to this member function.

```
public static IloBool Exists(IloEnv env, const char * key)
```

This static member function returns `IloTrue` if an `IloVisitDistance` object having key `key` exists and `IloFalse` if not.

```
public static IloVisitDistance Find(IloEnv env, const char * key)
```

This static member function returns the object corresponding to the key `key` set using `IloVisitDistance::setKey`. If there is no object corresponding to `key` an `IloException` is thrown.

```
public IloNum getDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle)
const
```

This member function returns the distance from `visit1` to `visit2` using vehicle `vehicle`.

```
public IloInt getGroup(IloVehicle vehicle) const
```

This member function returns the group as specified by the vehicle equivalence object associated with `vehicle`.

```
public IloVisitDistanceI * getImpl() const
```

This member function returns a pointer to the implementation object corresponding to the invoking distance.

```
public const char * getKey()
```

This member function returns the key set on the invoking object

```
public void refresh() const
```

This member function flushes any internal caches on the distance function, and uses any vehicle equivalence class specified in the constructor to update the group of each vehicle.

This member function thus allows the distance function to be changed. This means that after a call to `refresh`, `IloVisitDistanceI::computeDistance` can return a different value for the same three parameters than before the call to `refresh`. However, the new distance function must be as consistent as the old one in that successive calls using the same parameters must produce the same distance value.

```
public void removeKey()
```

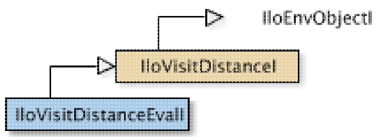
This member function allows the user to remove the key set on the invoking object.

```
public void setKey(const char * key)
```

This member function allows the user to set `key` on the invoking object. This key is unique. Each distance must have a different key; otherwise, an exception is thrown.

Class IloVisitDistanceEvalI

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher lets you define the distance function for a dimension (for example, the distance, the time, or the cost necessary for going from one node to another).

This class is an implementation class, a predefined subclass of `IloVisitDistanceI`, which you use to define a new distance function expressed by an evaluation function. This evaluation function is of type `IloVisitDistanceFunction`.

See Also: IloDimension, IloDimension1, IloDimension2, IloVisitDistance, IloVisitDistanceFunction, IloVisitDistanceI

Constructor and Destructor Summary	
`public`	`IloVisitDistanceEvalI(IloEnv env, IloVisitDistanceFunction distFunction)`
`public`	`IloVisitDistanceEvalI(IloVisitDistanceFunction distFunction, IloVehicleEquiv equiv)`

Method Summary	
`public IloNum`	`computeDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const`

Inherited Methods from `IloVisitDistanceI`
`computeDistance, getDistance, getGroup, refresh, setCache, unsetCache, updateEquivalence`

Constructors and Destructors

`public` **`IloVisitDistanceEvalI`**`(IloEnv env, IloVisitDistanceFunction distFunction)`

This constructor creates a new distance function from the evaluation function `distFunction`.

When distance functions are specified in Dispatcher, they can be cached, if distance computations are slow, through `IloDimension2::setCached`. (Normally, caching of distance functions is disabled.) When the distance depends not only on the starting and ending visit, but also the vehicle used to perform the trip, it becomes useful to introduce the notion of *vehicle equivalence*.

If two vehicles are specified as equivalent with respect to a particular distance metric and the distance between two visits using one of the vehicles resides in the cache, the distance between the same two visits using the other vehicle can be assumed to be the same. Thus, fewer cache slots are used. It is functionally unnecessary to specify a vehicle equivalence class in Dispatcher, but definition of such a class can lead to speed increases using better caching of distance data.

`public` **`IloVisitDistanceEvalI`**`(IloVisitDistanceFunction distFunction, IloVehicleEquiv equiv)`

This constructor creates a new distance function for the vehicle equivalence group `equiv` from the evaluation function `distFunction`.

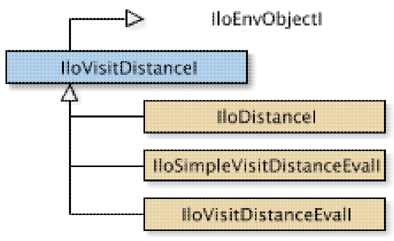
Methods

```
public IloNum computeDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle)
const
```

This member function returns a numeric value that represents the distance between `visit1` and `visit2` for the given `vehicle`. This is done using a call to `distFunction` passing `visit1`, `visit2`, and `vehicle` as parameters.

Class IloVisitDistanceI

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>



Dispatcher lets you define the distance function for a dimension (for example, the distance, the time, or the cost necessary for going from one visit to another).

This class is the implementation class for `IloVisitDistance`, the class of object that defines a distance function for a dimension. The virtual member function `IloVisitDistanceI::computeDistance` returns the distance between two visits.

To express new distance functions, you can define a subclass of `IloVisitDistanceI`. If this distance can be expressed by an evaluation function, you can use the predefined subclasses `IloVisitDistanceEvalI` or `IloSimpleVisitDistanceEvalI` for that purpose.

See Also: IloDimension, IloDimension1, IloDimension2, IloVisitDistance

Constructor and Destructor Summary	
public	IloVisitDistanceI(IloEnv env, IloBool symmetric=IloFalse)
public	IloVisitDistanceI(IloVehicleEquiv equiv, IloBool symmetric=IloFalse)

Method Summary	
public virtual IloNum	computeDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const
public virtual IloNum	getDistance(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const
public virtual IloInt	getGroup(IloVehicle vehicle) const
public virtual void	refresh()
public virtual void	setCache(IloEnv env, IloInt log2Rows, IloInt log2Cols)
public virtual void	unsetCache()
public void	updateEquivalence(IloEnv env)

Constructors and Destructors

`public` **`IloVisitDistanceI`**`(IloEnv env, IloBool symmetric=IloFalse)`

This constructor creates an implementation distance object in the environment `env`.

When distance functions are specified in Dispatcher, they can be cached, if distance computations are slow, through `IloDimension2::setCached`. (Normally, caching of distance functions is disabled.) When the distance depends not only on the starting and ending visit, but also the vehicle used to perform the trip, it becomes useful to introduce the notion of *vehicle equivalence*.

If two vehicles are specified as equivalent with respect to a particular distance metric and the distance between two visits using one of the vehicles resides in the cache, the distance between the same two visits using the other vehicle can be assumed to be the same. Thus, fewer cache slots are used. It is functionally unnecessary to specify a vehicle equivalence class in Dispatcher, but definition of such a class can lead to speed increases through better caching of distance data.

`public` **`IloVisitDistanceI`**`(IloVehicleEquiv equiv, IloBool symmetric=IloFalse)`

This constructor creates an implementation distance object for the vehicle equivalence group `equiv`.

Methods

`public virtual IloNum` **`computeDistance`**`(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const`

You redefine this pure virtual member function to return a floating-point value that represents the distance from `visit1` to `visit2`. The return value of the function must depend only on `visit1` and `visit2`, and must produce the same value for each call with the same parameters.

`public virtual IloNum` **`getDistance`**`(IloVisit visit1, IloVisit visit2, IloVehicle vehicle) const`

This member function returns the distance from `visit1` to `visit2`, using vehicle `vehicle`. If caching is enabled, this member function first searches the cache for the distance value. If the value is not found, this function calls `IloVisitDistanceI::computeDistance`, returns the value obtained, and places that value into the cache.

`public virtual IloInt` **`getGroup`**`(IloVehicle vehicle) const`

This member function returns the vehicle equivalence group for `vehicle`. In the case where the invoking object was constructed with only an `IloEnv`, this function returns zero. Otherwise, it returns the group as specified by the vehicle equivalence object associated with the implementation.

`public virtual void` **`refresh`**`()`

This member function flushes any internal caches on the distance function, and uses any vehicle equivalence class specified in the constructor to update the group of each vehicle.

This member function thus allows the distance function to be changed. This means that after a call to `refresh`, `IloVisitDistanceI::computeDistance` can return a different value for the same three parameters than before the call to `refresh`. However, the new distance function must be as consistent as the old one in that successive calls using the same parameters must produce the same distance value.

`public virtual void` **`setCache`**`(IloEnv env, IloInt log2Rows, IloInt log2Cols)`

This member function adds a cache to the invoking distance object so that distance computations can be cached. The parameter `env` indicates the environment upon which the distance object is allocated. The cache is set-associative with *2log2Rows* rows, and a set-associative width of *2log2Cols*.

The method `IloDimension2::setCached` uses this member function to add a cache to the distance object associated with the invoking dimension. No cache is added if one already exists.

```
public virtual void unsetCache()
```

This member function stops caching of distance values.

```
public void updateEquivalence(IloEnv env)
```

This member function updates the vehicle equivalence group associated with the invoking distance object.

Class IloVisitIterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

An instance of the class `IloVisitIterator` is an iterator that traverses all instances of the class `IloVisit` in a model.

See Also: IloVisit

Constructor Summary	
`public`	`IloVisitIterator(IloModel mdl, IloBool deep=IloTrue)`
`public`	`IloVisitIterator(const IloVisitIterator & iter)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVisit`	`operator*() const`
`public const IloVisitIterator &`	`operator++()`
`public const IloVisitIterator &`	`operator=(const IloVisitIterator & iter)`

Constructors

`public` **`IloVisitIterator`**`(IloModel mdl, IloBool deep=IloTrue)`

This constructor creates an iterator which will iterate over all instances of `IloVisit` in model `mdl`. If the parameter `deep` has the value `IloTrue`, all submodels of `mdl` will form part of the iteration. If `deep` has the value `IloFalse`, submodels will not be investigated by the iterator.

`public` **`IloVisitIterator`**`(const IloVisitIterator & iter)`

This copy constructor creates an iterator from another iterator `iter`. After execution, both the newly created iterator and `iter` will be at the same position within the model.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloFalse` if the iterator has scanned all instances of `IloVisit` in the model. Otherwise, it returns `IloTrue`.

`public IloVisit` **`operator*`**`() const`

This operator returns the instance of `IloVisit` at which the iterator is currently pointing.

`public const IloVisitIterator & **operator++**()`

This operator moves the iterator on to the next instance of `IloVisit` within the model, providing one exists. The operator returns the invoking iterator at its new position.

`public const IloVisitIterator & **operator=**(const IloVisitIterator & iter)`

This assignment operator copies the state of `iter` to the iterator on the left-hand side of the operator. After execution, both iterators will be at the same position within the model.

Class IloVisitPair

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This class represents a pair of visits.

See Also: IloVisit

Constructor Summary	
`public`	`IloVisitPair(IloVisit visit1, IloVisit visit2)`

Method Summary	
`public IloVisit`	`getVisit1() const`
`public IloVisit`	`getVisit2() const`

Constructors

`public` **`IloVisitPair`**`(IloVisit visit1, IloVisit visit2)`

This constructor creates a visit pair from the two visits `visit1` and `visit2`.

Methods

`public IloVisit` **`getVisit1`**`() const`

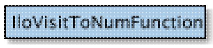
This member function returns the first visit of the pair.

`public IloVisit` **`getVisit2`**`() const`

This member function returns the second visit of the pair.

Class IloVisitToNumFunction

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>



This class is the handle class of `IloVisitToNumFunctionI`, the class that defines a function from a visit to an `IloNum`. The member function `IloVisitToNumFunction::getValue` returns the value corresponding to a visit. The member function `IloVisitToNumFunction::getUnperformedValue` returns the value corresponding to the unperformed state of a visit. This function can be used to create an `IloNumVar` whose domain is the values of the function accessed through an `IloVisitVar`.

See Also: IloVisitVar, IloVisit

Constructor Summary	
`public`	`IloVisitToNumFunction()`
`public`	`IloVisitToNumFunction(IloVisitToNumFunctionI * impl)`
`public`	`IloVisitToNumFunction(const IloVisitToNumFunction & func)`
`public`	`IloVisitToNumFunction(IloEnv env, IloVisitArray visits, IloNumArray values, IloNum unperformedValue)`
`public`	`IloVisitToNumFunction(IloEnv env, IloSimpleVisitToNumFunction f)`

Method Summary	
`public IloVisitToNumFunctionI *`	`getImpl() const`
`public IloNum`	`getUnperformedValue() const`
`public IloNum`	`getValue(IloVisit visit) const`
`public IloNumVar`	`operator()(IloVisitVar var) const`
`public void`	`operator=(const IloVisitToNumFunction & func)`

Constructors

`public` **`IloVisitToNumFunction`**`()`

This constructor creates a visit to `IloNum` function whose handle pointer is null. This object must be assigned before it can be used.

`public` **`IloVisitToNumFunction`**`(IloVisitToNumFunctionI * impl)`

This constructor creates a handle object (an instance of `IloVisitToNumFunction`) from a pointer to an implementation object (an instance of the class `IloVisitToNumFunctionI`).

`public` **`IloVisitToNumFunction`**`(const IloVisitToNumFunction & func)`

This copy constructor creates a handle from a reference to a visit to `IloNum` function. That visit to `IloNum` object and `func` both point to the same implementation object.

`public` **`IloVisitToNumFunction`**`(IloEnv env, IloVisitArray visits, IloNumArray values, IloNum unperformedValue)`

This constructor takes an array of visits and an array of values. The visit and the value with the same index in these arrays are associated. The value `unperformedValue` corresponds to the visit unperformed state. The implementation object of the newly created handle is an instance of `IloArrayVisitToNumFunctionI`.

`public` **`IloVisitToNumFunction`**`(IloEnv env, IloSimpleVisitToNumFunction f)`

This constructor creates a visit to `IloNum` function based on `f`. The implementation object of the newly created handle is an instance of `IloEvalVisitToNumFunctionI`.

Methods

`public IloVisitToNumFunctionI *` **`getImpl`**`() const`

This member function returns a pointer to the implementation object corresponding to the invoking function.

`public IloNum` **`getUnperformedValue`**`() const`

This member function returns the value associated with the visit unperformed state.

`public IloNum` **`getValue`**`(IloVisit visit) const`

This member function returns the value associated with `visit`.

`public IloNumVar` **`operator()`**`(IloVisitVar var) const`

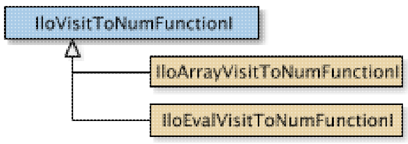
This operator returns an `IloNumVar` constrained by `var`. For clarity, let's call *f* the invoking function. When `var` is bound to the visit *v*, the value of the expression is *f.getValue(v)*. More generally, the domain of the `IloNumVar` is the set of values *{gi| gi = f(vi)}* where the *vi* are in the domain of `var`.

`public void` **`operator=`**`(const IloVisitToNumFunction & func)`

This operator assigns an address to the handle pointer of the invoking visit to `IloNum` function. That address is the location of the implementation object of the argument `func`. After the execution of this operator, the invoking function and `func` both point to the same implementation object.

Class IloVisitToNumFunctionI

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>



This class is the implementation class of `IloVisitToNumFunction`, the class that defines a function from a visit to an `IloNum`. The virtual member function `IloVisitToNumFunctionI::getValue` returns the value corresponding to a visit. The virtual member function `IloVisitToNumFunctionI::getUnperformedValue` returns the value corresponding to the unperformed state of a visit. This function can be used to create an `IloNumVar` whose domain is the values of the function accessed through an `IloVisitVar`.

Constructor and Destructor Summary	
`public`	`IloVisitToNumFunctionI(IloEnv env)`

Method Summary	
`public virtual IloNum`	`getUnperformedValue()`
`public virtual IloNum`	`getValue(IloVisit visit)`

Constructors and Destructors

`public` **`IloVisitToNumFunctionI`**`(IloEnv env)`

This constructor creates an implementation visit to `IloNum` object in the environment `env`.

Methods

`public virtual IloNum` **`getUnperformedValue`**`()`

This virtual member function can be redefined to return a numeric value corresponding to the unperformed state of a visit. For a given function, the return value should not vary between two calls. By default, this member function returns 0.

`public virtual IloNum` **`getValue`**`(IloVisit visit)`

This pure virtual member function must be redefined to return a numeric value corresponding to `visit`. For a given function, the return value should not vary between two calls with the same parameter.

Class IloVisitVar

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

A visit-variable is a constrained variable representing the visit served immediately before or after the associated visit. In Solver, it is extracted to an `IlcIntVar`, representing the index of the next visit.

See Also: IloVisit, IloVisitToNumFunction

Method Summary	
`public IloVisit`	`getVisit() const`

Methods

`public IloVisit` **`getVisit`**`() const`

This member function returns the visit associated with the invoking visit-variable.

Class IloVisitVehicleCompat

Definition file: ildispat/ilocompat.h
Include file: <ildispat/ilodispatcher.h>

Dispatcher lets you define compatibility relations between visits and vehicles. Compatibility relations are used to build compatibility constraints that are used to restrict the possible choices of vehicles for a visit. This class is the handle class for all visit/vehicle compatibility relations. This handle class uses the virtual member function `IloVisitVehicleCompatI::isCompatible` to determine the compatibility between a visit and a vehicle.

Constructor Summary	
`public`	`IloVisitVehicleCompat(IloVisitVehicleCompatI * impl=0)`
`public`	`IloVisitVehicleCompat(const IloVisitVehicleCompat & other)`
`public`	`IloVisitVehicleCompat(IloEnv env, IloVisitVehicleCompatPredicate predicate)`

Method Summary	
`public void`	`end()`
`public IloVisitVehicleCompatI *`	`getImpl() const`
`public IloBool`	`isCompatible(IloVisit visit, IloVehicle vehicle)`

Constructors

`public` **`IloVisitVehicleCompat`**`(IloVisitVehicleCompatI * impl=0)`

This constructor creates a handle from a pointer to an implementation class. Called without argument, this constructor creates an empty handle.

`public` **`IloVisitVehicleCompat`**`(const IloVisitVehicleCompat & other)`

This copy constructor creates a handle from a reference to another handle. The resulting object and `other` both point to the same object.

`public` **`IloVisitVehicleCompat`**`(IloEnv env, IloVisitVehicleCompatPredicate predicate)`

This constructor creates a handle class from an environment and a compatibility predicate. The implementation object of the newly created handle is an instance of the class `IloVisitVehiclePredicateCompatI`, constructed with the predicate `predicate`.

Methods

`public void` **`end`**`()`

This member function frees all resources used by the invoking compatibility object. You cannot use the invoking compatibility object after a call to this member function.

```
public IloVisitVehicleCompatI * getImpl() const
```

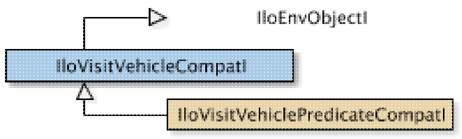
This member function returns a pointer to the implementation object corresponding to the invoking handle.

```
public IloBool isCompatible(IloVisit visit, IloVehicle vehicle)
```

This member function returns a Boolean stating whether visit and vehicle are compatible.

Class IloVisitVehicleCompatI

Definition file: ildispat/ilocompat.h
Include file: <ildispat/ilodispatcher.h>



This class is the implementation class of the handle class `IloVisitVehicleCompat`. The virtual member function `isCompatible` returns a Boolean stating whether a visit and a vehicle are compatible. By compatible, it is meant that the vehicle is a possible candidate for the visit. Compatibility relations are used to build a compatibility constraint using the function `IloCompatible`.

To define a compatibility relation, you can define a subclass of `IloVisitVehicleCompatI`. If the compatibility relation can be expressed with a predicate function, you can use the predefined subclass `IloVisitVehiclePredicateCompatI`.

Constructor and Destructor Summary	
`protected`	`IloVisitVehicleCompatI(IloEnv env)`

Method Summary	
`public virtual IloBool`	`isCompatible(IloVisit visit, IloVehicle vehicle)`

Constructors and Destructors

`protected` **`IloVisitVehicleCompatI`**`(IloEnv env)`

This constructor creates an implementation compatibility relation in the environment `env`. As this class is a pure virtual class, the constructor is declared as protected only.

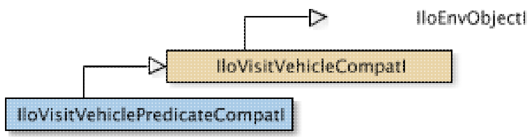
Methods

`public virtual IloBool` **`isCompatible`**`(IloVisit visit, IloVehicle vehicle)`

You redefine this pure virtual member function to return `IloTrue` when the visit and vehicle are compatible, `IloFalse` otherwise. Incompatible vehicles are removed from the possible vehicles by the compatibility constraint built with a compatibility relation.

Class IloVisitVehiclePredicateCompatI

Definition file: ildispat/ilocompat.h
Include file: <ildispat/ilodispatcher.h>



This class is an implementation class, a predefined subclass of `IloVisitVehicleCompatI`, used to define a compatibility relation best expressed by a predicate.

Constructor Summary	
`public`	`IloVisitVehiclePredicateCompatI(IloEnv env, IloVisitVehicleCompatPredicate predicate)`

Method Summary	
`public virtual IloBool`	`isCompatible(IloVisit visit, IloVehicle vehicle)`

Inherited Methods from `IloVisitVehicleCompatI`
`isCompatible`

Constructors

`public` **`IloVisitVehiclePredicateCompatI`**`(IloEnv env, IloVisitVehicleCompatPredicate predicate)`

This constructor creates a new compatibility implementation object from an environment and a compatibility predicate.

Methods

`public virtual IloBool` **`isCompatible`**`(IloVisit visit, IloVehicle vehicle)`

This member function uses the predicate to return a Boolean stating whether visit and vehicle are compatible.

Class IloDimensionWindows::Iterator

Definition file: ildispat/iloproto.h
Include file: <ildispat/ilodispatcher.h>

`Iterator` is a class nested in the class `IloDimensionWindows`. It allows you to step through the permitted intervals of a dimension windows constraint, in increasing interval lower bound order.

See Also: IloDimensionWindows, IloDimensionWindows::ForbiddenIterator

Constructor Summary	
`public`	`Iterator(IloDimensionWindows win)`

Method Summary	
`public IloNum`	`getLB() const`
`public IloNum`	`getUB() const`
`public IloBool`	`ok() const`
`public Iterator &`	`operator++()`

Constructors

`public` **`Iterator`**`(IloDimensionWindows win)`

This constructor creates an iterator to traverse the permitted intervals contained in `win`.

Methods

`public IloNum` **`getLB`**`() const`

This member function returns the lower bound of the permitted interval to which the invoking iterator points.

`public IloNum` **`getUB`**`() const`

This member function returns the upper bound of the permitted interval to which the invoking iterator points.

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the permitted intervals have been scanned.

`public Iterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next permitted interval (the first one starting after the current permitted interval).

Class IloNode::Iterator

Definition file: ildispat/ilonode.h
Include file: <ildispat/ilodispatcher.h>

`Iterator` is a class nested in the class IloNode. It allows you to step through all the nodes in a particular environment.

See Also: IloNode

Constructor Summary	
`public`	`Iterator(IloEnv env)`

Method Summary	
`public IloBool`	`ok()`
`public IloNode`	`operator*() const`
`public Iterator &`	`operator++()`

Constructors

`public` **`Iterator`**`(IloEnv env)`

This constructor creates an iterator to traverse all the nodes in the environment `env`.

Methods

`public IloBool` **`ok`**`()`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the nodes have been scanned by the iterator.

`public IloNode` **`operator*`**`() const`

This operator returns the current instance of `IloNode`, the one to which the invoking iterator points.

`public Iterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloNode` in the route.

Class IloDispatcherGraph::Node

Definition file: ildispat/ilographdist.h
Include file: <ildispat/ilodispatcher.h>

`Node` is a class nested in the class IloDispatcherGraph. Instances of the class `IloDispatcherGraph::Node` represent the nodes in the graph.

See Also: IloDispatcherGraph, IloDispatcherGraph::PathIterator, IloDispatcherGraph::Arc, IloDispatcherGraph::AdjacencyListIterator, IloGraphDistance

Constructor and Destructor Summary	
`public`	`Node(IloDispatcherGraph g, const IloInt id)`
`public`	`Node(const Node & node)`
`public`	`Node(IloDispatcherGraphI::NodeI * impl=0)`

Method Summary	
`public IloDispatcherGraphI::NodeI *`	`getImpl() const`
`public IloInt`	`getIndex() const`

Constructors and Destructors

public **Node**(IloDispatcherGraph g, const IloInt id)

This constructor creates an instance of a node in the graph `g`. The node identifier is set to the value given by `id`.

public **Node**(const Node & node)

This copy constructor creates a handle from a reference to a graph node object. That graph node object and `node` both point to the same implementation object.

public **Node**(IloDispatcherGraphI::NodeI * impl=0)

This constructor creates a handle object (an instance of `IloDispatcherGraphI::Node`) from a pointer to an implementation object (an instance of `IloDispatcherGraphI::NodeI`).

Methods

public IloDispatcherGraphI::NodeI * **getImpl**() const

This member function returns a pointer to the implementation object corresponding to the invoking graph node object.

public IloInt **getIndex**() const

This member function returns the node's identifier.

Class IloDispatcherGraph::PathIterator

Definition file: ildispat/ilographdist.h
Include file: <ildispat/ilodispatcher.h>

`PathIterator` is a class nested in the class IloDispatcherGraph. An instance of the class `IloDispatcherGraph::PathIterator` iterates over the instances of `IloDispatcherGraph::Node`, composing the cheapest path between two instances of `IloNode` using an instance of `IloVehicle`. This iterator can only be used if the graph stores cheapest paths.

See Also: IloDispatcherGraph, IloDispatcherGraph::Node, IloGraphDistance, IloVehicle

Constructor Summary	
public	PathIterator(IloDispatcherGraph g, IloNode node1, IloNode node2, IloVehicle v)

Method Summary	
public IloBool	ok() const
public IloDispatcherGraph::Node	operator*() const
public PathIterator &	operator++()

Constructors

public **PathIterator**(IloDispatcherGraph g, IloNode node1, IloNode node2, IloVehicle v)

This constructor creates an iterator to traverse the nodes in the graph `g`, composing the cheapest path between `node1` and `node2` using vehicle `v`.

Methods

public IloBool **ok**() const

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the nodes have been scanned by the iterator.

public IloDispatcherGraph::Node **operator***() const

This operator returns the current instance of `Node`, the one to which the invoking iterator points.

public PathIterator & **operator++**()

This left-increment operator shifts the current position of the iterator to the next instance of `Node` in the route.

Class IloDispatcher::RouteIterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

`RouteIterator` is a class nested in the class `IloDispatcher`. It allows you to step through all the visits performed by a given vehicle.

In fact, a *route* is the ordered set of visits made by a vehicle. A route is associated with each vehicle. For example, if a truck leaves the depot for three days, then the route will consist of all the visits performed (for deliveries, for pick-ups, for service calls, etc.) during this three-day period. The route of a vehicle is identified by its first and last visit.

`RouteIterator` can only be used on a route which is in an instantiated state.

For more information, see the concept Iterators.

See Also: IloDispatcher, IloVehicle, IloVisit

Constructor Summary	
`public`	`RouteIterator(IloDispatcher dispatcher, IloVehicle vehicle)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVisit`	`operator*() const`
`public RouteIterator &`	`operator++()`

Constructors

`public` **`RouteIterator`**`(IloDispatcher dispatcher, IloVehicle vehicle)`

This constructor creates an iterator to traverse all the visits in the route of `vehicle` in the dispatcher object `dispatcher`. This includes the first and last visits in the route (usually the “depot”).

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the visits have been scanned by the iterator.

`public IloVisit` **`operator*`**`() const`

This operator returns the current instance of `IloVisit`, the one to which the invoking iterator points.

`public RouteIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloVisit` in the route.

Class IloRoutingSolution::RouteIterator

Definition file: ildispat/ilorsol.h
Include file: <ildispat/ilodispatcher.h>

`RouteIterator` is a class nested in the class `IloRoutingSolution`. It allows you to step through all the visits performed by a given vehicle in a particular routing solution.

In fact, a *route* is the ordered set of visits made by a vehicle. A route is associated with each vehicle. For example, if a truck leaves the depot for three days, then the route will consist of all the visits performed (for deliveries, for pick-ups, for service calls, etc.) during this three-day period. The route of a vehicle is identified by its first and last visit.

For more information, see the concept Iterators.

See Also: IloRoutingSolution, IloVehicle, IloVisit

Constructor Summary	
`public`	`RouteIterator(IloRoutingSolution solution, IloVehicle vehicle)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVisit`	`operator*() const`
`public RouteIterator &`	`operator++()`

Constructors

`public` **`RouteIterator`**`(IloRoutingSolution solution, IloVehicle vehicle)`

This constructor creates an iterator to traverse all the visits in the route of `vehicle` in the routing solution `solution`. This includes the first and last visits in the route (usually the “depot”).

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all the visits have been scanned by the iterator.

`public IloVisit` **`operator*`**`() const`

This operator returns the current instance of `IloVisit`, the one to which the invoking iterator points.

`public RouteIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloVisit` in the route.

Class IloDispatcher::UnperformedVisitIterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

`UnperformedVisitIterator` is a class nested in the class `IloDispatcher`. It allows you to step through all the unperformed visits associated with a given routing plan.

For more information, see the concept Iterators.

See Also: IloDispatcher, IloVisit

Constructor Summary	
`public`	`UnperformedVisitIterator(IloDispatcher dispatcher)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVisit`	`operator*() const`
`public UnperformedVisitIterator &`	`operator++()`

Constructors

`public` **`UnperformedVisitIterator`**`(IloDispatcher dispatcher)`

This constructor creates an iterator to traverse all the unperformed visits associated with the dispatcher object `dispatcher`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all unperformed visits have been scanned by the iterator.

`public IloVisit` **`operator*`**`() const`

This operator returns the current instance of `IloVisit`, the one to which the invoking iterator points.

`public UnperformedVisitIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloVisit` in the routing plan.

Class IloRoutingSolution::UnperformedVisitIterator

Definition file: ildispat/ilorsol.h
Include file: <ildispat/ilodispatcher.h>

`UnperformedVisitIterator` is a class nested in the class `IloRoutingSolution`. It allows you to step through all the unperformed visits associated with a given routing solution.

For more information, see the concept Iterators.

See Also: IloRoutingSolution, IloVisit

Constructor Summary	
`public`	`UnperformedVisitIterator(IloRoutingSolution solution)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVisit`	`operator*() const`
`public UnperformedVisitIterator &`	`operator++()`

Constructors

`public` **`UnperformedVisitIterator`**`(IloRoutingSolution solution)`

This constructor creates an iterator to traverse all the unperformed visits associated with the routing solution `solution`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all unperformed visits have been scanned by the iterator.

`public IloVisit` **`operator*`**`() const`

This operator returns the current instance of `IloVisit`, the one to which the invoking iterator points.

`public UnperformedVisitIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next unperformed instance of `IloVisit` in the routing plan.

Class IloDispatcher::VehicleBreakConIterator

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

This class creates an iterator that iterates over all vehicle- extracted vehicle breaks.

Constructor Summary	
`public`	`VehicleBreakConIterator(IloDispatcher dispatcher, IloVehicle vehicle, IloDimension2 dim)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVehicleBreakCon`	`operator*() const`
`public VehicleBreakConIterator &`	`operator++()`

Constructors

`public` **`VehicleBreakConIterator`**`(IloDispatcher dispatcher, IloVehicle vehicle, IloDimension2 dim)`

This constructor creates an iterator that iterates over all vehicle- extracted vehicle breaks on vehicle `vehicle` and dimension `dim`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all unperformed visits have been scanned by the iterator.

`public IloVehicleBreakCon` **`operator*`**`() const`

This operator returns the current instance of `IloVisit`, the one to which the invoking iterator points.

`public VehicleBreakConIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloVisit` in the routing plan.

Class IloRoutingSolution::VehicleIterator

Definition file: ildispat/ilorsol.h
Include file: <ildispat/ilodispatcher.h>

`VehicleIterator` is a class nested in the class `IloRoutingSolution`. It allows you to step through all the vehicles associated with a given routing solution.

For more information, see the concept Iterators.

See Also: IloRoutingSolution, IloVehicle

Constructor Summary	
`public`	`VehicleIterator(IloRoutingSolution solution)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVehicle`	`operator*() const`
`public VehicleIterator &`	`operator++()`

Constructors

`public` **`VehicleIterator`**`(IloRoutingSolution solution)`

This constructor creates an iterator to traverse all the vehicles associated with the routing solution `solution`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all vehicles have been scanned by the iterator.

`public IloVehicle` **`operator*`**`() const`

This operator returns the current instance of `IloVisit`, the one to which the invoking iterator points.

`public VehicleIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloVisit` in the routing plan.

Class IloRoutingSolution::VisitIterator

Definition file: ildispat/ilorsol.h
Include file: <ildispat/ilodispatcher.h>

`VisitIterator` is a class nested in the class `IloRoutingSolution`. It allows you to step through all the visits associated with a given routing solution.

For more information, see the concept Iterators.

See Also: IloRoutingSolution, IloVisit

Constructor Summary	
`public`	`VisitIterator(IloRoutingSolution solution)`

Method Summary	
`public IloBool`	`ok() const`
`public IloVisit`	`operator*() const`
`public VisitIterator &`	`operator++()`

Constructors

`public` **`VisitIterator`**`(IloRoutingSolution solution)`

This constructor creates an iterator to traverse all the visits associated with the routing solution `solution`.

Methods

`public IloBool` **`ok`**`() const`

This member function returns `IloTrue` if the current position of the iterator is a valid one. It returns `IloFalse` if all visits have been scanned by the iterator.

`public IloVisit` **`operator*`**`() const`

This operator returns the current instance of `IloVisit`, the one to which the invoking iterator points.

`public VisitIterator &` **`operator++`**`()`

This left-increment operator shifts the current position of the iterator to the next instance of `IloVisit` in the routing plan.

Enumeration IloFSDecisionRejectCause

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>

This enumerated type is used to characterize the reason why a decision has been rejected in the `isLegal` test.

`IloRejectCauseMake` denotes a failure in the decision's `make` method.

`IloRejectCauseCompletion` denotes a rejection due to a failure in the decision's route completion goal, as defined by the `getRouteCompletionGoal` method. If you are using predefined decision classes, these goals perform a standard Dispatcher `IloGenerateRoute` goal.

`IloRejectCauseJustifier` denotes a rejection due to the justifier goal associated with the decision, as returned by the `getJustifierGoal` method. Dispatcher predefined decision classes have empty justifier goals.

Fields:

```
IloRejectCauseUnknown = 0L

IloRejectCauseMake

IloRejectCauseCompletion

IloRejectCauseJustifier
```

Enumeration IloNearestAdditionBehavior

Definition file: ildispat/1stsol.h
Include file: <ildispat/ilodispatcher.h>

This enumerated type controls the behavior of the `IloNearestAdditionGenerate` goal during route construction.

The mode `IloNearestAdditionForward` extends the route forward from the first visit to the nearest unrouted visit.

The mode `IloNearestAdditionBackward` extends the route backward from the last visit to the nearest unrouted visit.

The mode `IloNearestAdditionBoth` extends the route simultaneously in both directions; the nearest visit to either the start or the end of the route is connected to that portion of the route. In the case of a tie, the route is extended forward.

See Also: IloNearestAdditionGenerate

Fields:

```
IloNearestAdditionForward

IloNearestAdditionBackward

IloNearestAdditionBoth

IloNearestAdditionForward

IloNearestAdditionBackward

IloNearestAdditionBoth

IloNearestAdditionPDPFILO
```

Enumeration IloNearestAdditionExtension

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>

This enumerated type is used to characterize the two different ways to build a first solution in nearest-addition types of heuristics. Serial mode denotes heuristics where at most one vehicle is open at a given time, while parallel mode denotes a mode where several vehicles can be open simultaneously. This enumerated type is used to build the specialized decision maker class dedicated to nearest addition heuristics.

The mode `IloNearestAdditionSerial` builds vehicle routes one at a time. This heuristic will build routes as densely as possible, and leave part of the vehicles unused.

The mode `IloNearestAdditionParallel` denotes a nearest addition heuristic where several vehicles can be open at the same time. The heuristic always looks for the best decision to test, regardless of the open vehicles at that point, possibly opening many vehicles and building routes that are not close to capacity.

Fields:

`IloNearestAdditionSerial`

`IloNearestAdditionParallel`

Enumeration IloOutOfRouteReference

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This enumerated type controls the behavior of the `IloOutOfRouteConstraint` constraint.

Fields:

`IloFirstLastVisits`

`IloNextFirstPrevLastVisits`

`IloMaxDiameter`

`IloFirstLastVisits`

`IloNextFirstPrevLastVisits`

`IloMaxDiameter`

Global function IloManhattan

```
public IloNum IloManhattan(IloNode node1, IloNode node2)
```

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

This function is a pre-defined distance function that returns the Manhattan distance between two nodes for the specified vehicle. This distance derives its name from the fact that it respects the distance imposed by a grid, as does a vehicle moving through the grid-like city-block pattern of Manhattan.

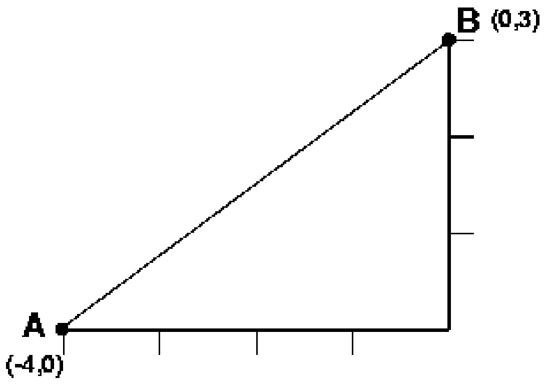
Implementation

This function may be implemented like this:

```
IloNum IloManhattan(IloNode node1, IloNode node2){
    IloNum x = node1.getX()-node2.getX();
    IloNum y = node1.getY()-node2.getY();
    IloNum z = node1.getZ()-node2.getZ();
    return IloAbs(x) + IloAbs(y) + IloAbs(z);
}
```

Example

In the example below, the Manhattan distance between A and B is equal to $|-4-0| + |0-3| = 4 + 3 = 7$. (The z-coordinates of A and B are assumed to be equivalent.)



See Also: IloDistance, IloDistMax, IloEuclidean, IloGeographical, IloNode, IloSimpleDistanceFunction

Global function IloOrderedVisitPair

```
public IloConstraint IloOrderedVisitPair(IloEnv env, IloVisit visit1, IloVisit
visit2, const char * name=0)
public IlcConstraint IloOrderedVisitPair(IloSolver solver, IloVisit visit1,
IloVisit visit2, const char * name=0)
```

Definition file: ildispat/ilocon.h
Include file: <ildispat/ilodispatcher.h>

These functions create constraints which state that `visit1` and `visit2` must be performed by the same vehicle, and that `visit1` must precede `visit2`, but not necessarily directly. The parameter `name`, if specified, becomes the name of the newly created constraint. These constraints also allow both visits to be unperformed without violating the constraint.

To specify that the pair must be performed by a vehicle, use the member function `IloVisit::performed()`.

The code

```
mdl.add(IloOrderedVisitPair(env, visit1, visit2));
```

is semantically equivalent to

```
mdl.add(visit1.getVehicleVar() == visit2.getVehicleVar());

mdl.add(visit1.isBefore(visit2));
```

However, the code using `IloOrderedVisitPair` is more efficient.

See Also: IloVisit, operator==

Global function IloInstantiateVehicleBreaks

```
public IloGoal IloInstantiateVehicleBreaks(IloEnv env, IloNum precision=0.0,
IloBool independent=IloFalse)
public IloGoal IloInstantiateVehicleBreaks(IloEnv env, IloVehicle vehicle, IloNum
precision=0.0)
public IlcGoal IloInstantiateVehicleBreaks(IloSolver solver, IloNum precision=0.0,
IloBool independent=IloFalse)
public IlcGoal IloInstantiateVehicleBreaks(IloSolver solver, IloVehicle vehicle,
IloNum precision=0.0)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These goals instantiate all breaks for the vehicle `vehicle` or all vehicle breaks contained in the environment `env` or extracted by `solver`. Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal`.

The parameter `precision` is used to specify the required accuracy of the start times and durations. When the difference between the minimum and maximum of the start time or duration is less than or equal to `precision`, the domain of the corresponding variable is not reduced further. This can be used to increase solving speed when absolute accuracy is not required. The default is 0.0 (absolute accuracy).

The parameter `independent` is used to indicate whether vehicles in the plan should be treated independently. If it is set to `IloTrue`, the goal instantiates all vehicle breaks for the routing plan, but treats all vehicles independently. Therefore, if the break instantiation is completely explored for a single vehicle and the break(s) could not be placed, the whole goal fails rather than backtracking to a previous vehicle to explore another break position on that vehicle before moving forward again. The default is `IloFalse`.

See Also: IloInstantiateVehicleBreak, IloInstantiateVehicleBreakDuration, IloInstantiateVehicleBreakPosition, IloInstantiateVehicleBreakStart, IloVehicleBreakCon

Global function IloVisitAlternativeSwap

```
public IloNHood IloVisitAlternativeSwap(IloEnv env, IloDispatcherNHoodParameters
params)
public IloNHood IloVisitAlternativeSwap(IloEnv env)
```

Definition file: ildispat/perfnhood.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by exchanging a performed visit constrained by an `IloVisitAlternativeConstraint` constraint with another unperformed visit constrained by the same constraint.

The optional parameter `params` can be used to customize the behavior of the neighborhood. In particular, the vehicle array specified using `setVehicles` on the `IloDispatcherNHoodParameters` class will make the neighborhood unperform visits only if they belong to the routes of these vehicles.

For more information, see the concept Neighborhoods.

See Also: IloCross, IloExchange, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloRelocate, IloSwapPerform, IloTwoOpt, IloDispatcherNHoodParameters

Global function IloExchange

```
public IloNHood IloExchange(IloEnv env, IloDispatcherNHoodParameters params)
public IloNHood IloExchange(IloEnv)
```

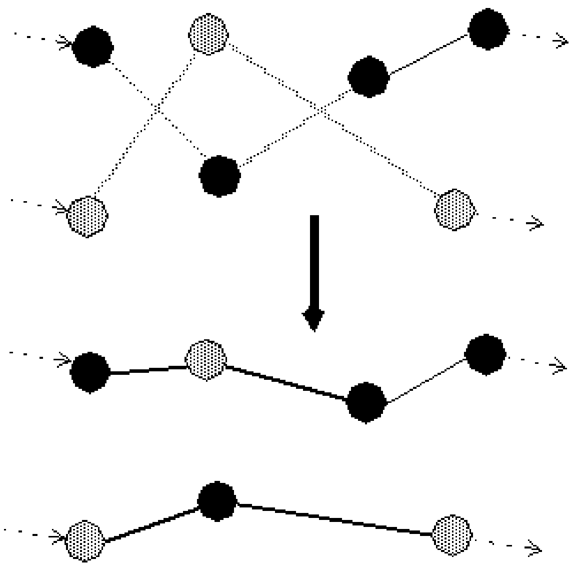
Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by exchanging two visits on different routes. These exchanges can result in cheaper routes.

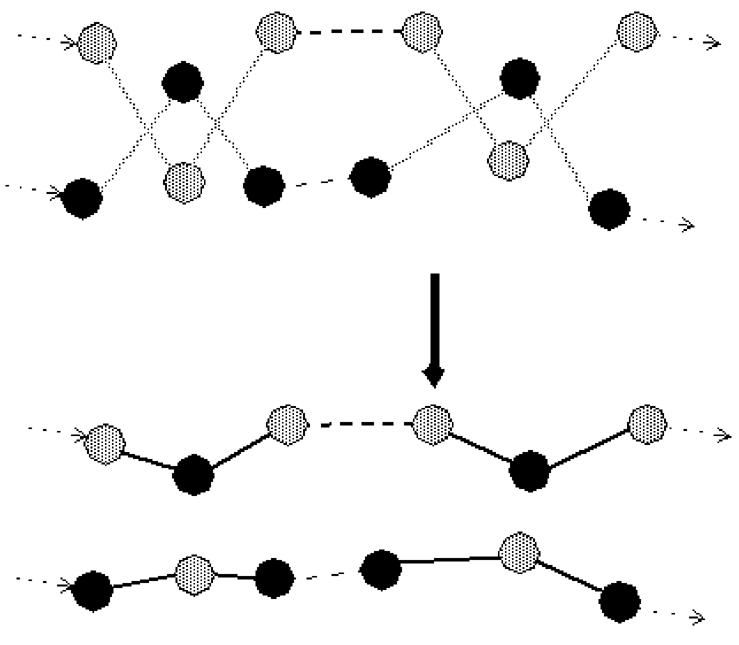
The optional parameter `params` can be used to customize the behavior of the neighborhood. In particular, the vehicle array specified using `IloDispatcherNHoodParameters::setVehicles` will make the exchange neighborhood operate only on the routes of these vehicles.

Examples:

The following figure shows the process of exchanging two visits. Here, we assume that the cost is proportional to the length of the route. The neighborhood destroys four arcs and creates four new arcs. As a result total travel distance, and thus cost, is less.



The following figure shows the process of exchanging two pairs of visits. Here, we assume that the cost is proportional to the length of the route. The neighborhood destroys eight arcs and creates eight new arcs. As a result total travel distance, and thus cost, is less.



For more information, see the concept Neighborhoods.

See Also: IloCross, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloRelocate, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloInstantiateVehicleBreakPosition

```
public IloGoal IloInstantiateVehicleBreakPosition(IloEnv env, IloVehicleBreakCon
brk)
public IlcGoal IloInstantiateVehicleBreakPosition(IloSolver solver,
IloVehicleBreakCon brk)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to instantiate the position of the vehicle break `brk`. The "position" in the context of a break refers to a visit occurring immediately before the break. This goal tries first to place the break in the arc with the most available idle spare time.

See Also: IloInstantiateVehicleBreak, IloInstantiateVehicleBreakDuration, IloInstantiateVehicleBreaks, IloInstantiateVehicleBreakStart, IloVehicleBreakCon

Global function IloMakeUnperformed

```
public IloNHood IloMakeUnperformed(IloEnv env, IloDispatcherNHoodParameters params)
public IloNHood IloMakeUnperformed(IloEnv env)
```

Definition file: ildispat/perfnhood.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by causing a performed visit to be unperformed.

The optional parameter `params` can be used to customize the behavior of the neighborhood. In particular, the vehicle array specified using `setVehicles` on the `IloDispatcherNHoodParameters` class will make the neighborhood unperform visits only if they belong to the routes of these vehicles.

For more information, see the concept Neighborhoods.

See Also: IloCross, IloExchange, IloFPRelocate, IloOrOpt, IloMakePerformed, IloMakePerformedPair, IloRelocate, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloTardinessFunction

```
public IloNumToNumSegmentFunction IloTardinessFunction(IloEnv env, IloNum
lateThresh, IloNum lateCost)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This function creates a piecewise linear function `f` such that:

- *if 0 <= x < tardyThresh, f(x) = 0,*
- *if tardyThresh <= x, f(x) = tardyCost * (x - tardyThresh).*

It can be used to express tardiness costs on visits, in conjunction with `IloVisit::setStartCost()` and `IloVisit::setEndCost()`, where `tardyCost` is the cost per unit of dimension for performing a visit after `tardyThresh`.

Global function IloVerbose

```
public IloOutputManip IloVerbose(IloDispatcher dispatcher)
```

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

This function returns an output manipulator that expands the output data displayed by the overloaded operator.

Example

The following is an example of the output produced by the `IloVerbose` operator:

```
Solution     :
Unperformed visits : None

Vehicle 0
      Route : Depot -> 14 -> 15 -> 2 -> 4 -> 12 -> 3 -> Depot
       Time : Depot [0..39.5968], delay [0..39.5968] -> travel [32.0156], wait [0..39.5968]
              -> 14 [32.0156..71.6125], delay [10] -> travel [15.8114], wait [0..39.5968]
              -> 15 [61..97.4238], delay [10] -> travel [13], wait [0..36.4238]
              -> 2 [84..120.424], delay [10] -> travel [20.2237], wait [0..36.4238]
              -> 4 [149..150.648], delay [10] -> travel [15.8114], wait [0..1.64759]
              -> 12 [174.811..176.459], delay [10] -> travel [11.1803], wait [0..1.64759]
              -> 3 [195.992..197.639], delay [10] -> travel [22.3607], wait [0..1.64759]
              -> Depot [228.352..230], delay [0..Inf) -> travel [0], wait [0..Inf)
     Weight : Depot [0..114], quantity [0] -> 14 [0..114], quantity [20]
              -> 15 [20..134], quantity [8] -> 2 [28..142], quantity [7]
              -> 4 [35..149], quantity [19] -> 12 [54..168], quantity [19]
              -> 3 [73..187], quantity [13] -> Depot [86..200], quantity [0]
  Tardiness : Depot [0..Inf), quantity [0] -> 14 [0..Inf), quantity [0..230]
              -> 15 [0..Inf), quantity [0..230] -> 2 [0..Inf), quantity [0..230]
              -> 4 [0..Inf), quantity [0..230] -> 12 [0..Inf), quantity [0..230]
              -> 3 [0..Inf), quantity [0..230] -> Depot [0..Inf), quantity [0]
     Length : Depot [0..Inf), delay [0..Inf) -> travel [32.0156], wait [0..Inf)
              -> 14 [0..Inf), delay [0..Inf) -> travel [15.8114], wait [0..Inf)
              -> 15 [0..Inf), delay [0..Inf) -> travel [13], wait [0..Inf)
              -> 2 [0..Inf), delay [0..Inf) -> travel [20.2237], wait [0..Inf)
              -> 4 [0..Inf), delay [0..Inf) -> travel [15.8114], wait [0..Inf)
              -> 12 [0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf)
              -> 3 [0..Inf), delay [0..Inf) -> travel [22.3607], wait [0..Inf)
              -> Depot [0..Inf), delay [0..Inf) -> travel [0], wait [0..Inf)

Vehicle 1
      Route : Depot -> 5 -> 16 -> 17 -> 8 -> 18 -> 6 -> 13 -> Depot
       Time : Depot [0..63.2233], delay [0..63.2233] -> travel [20.6155], wait [0..63.2233]
              -> 5 [20.6155..83.8389], delay [10] -> travel [11.1803], wait [0..63.2233]
              -> 16 [41.7959..105.019], delay [10] -> travel [11.1803], wait [0..63.2233]
              -> 17 [62.9762..126.2], delay [10] -> travel [13.9284], wait [0..63.2233]
              -> 8 [95..150.128], delay [10] -> travel [10.4403], wait [0..55.1279]
              -> 18 [115.44..170.568], delay [10] -> travel [11.1803], wait [0..55.1279]
              -> 6 [136.621..191.749], delay [10] -> travel [7.07107], wait [0..55.1279]
              -> 13 [159..208.82], delay [10] -> travel [11.1803], wait [0..49.8197]
              -> Depot [180.18..230], delay [0..Inf) -> travel [0], wait [0..Inf)
     Weight : Depot [0..106], quantity [0] -> 5 [0..106], quantity [26]
              -> 16 [26..132], quantity [19] -> 17 [45..151], quantity [2]
              -> 8 [47..153], quantity [9] -> 18 [56..162], quantity [12]
              -> 6 [68..174], quantity [3] -> 13 [71..177], quantity [23]
              -> Depot [94..200], quantity [0]
  Tardiness : Depot [0..Inf), quantity [0] -> 5 [0..Inf), quantity [0..230]
              -> 16 [0..Inf), quantity [0..230] -> 17 [0..Inf), quantity [0..230]
              -> 8 [0..Inf), quantity [0..230] -> 18 [0..Inf), quantity [0..230]
              -> 6 [0..Inf), quantity [0..230] -> 13 [0..Inf), quantity [0..230]
              -> Depot [0..Inf), quantity [0]
     Length : Depot [0..Inf), delay [0..Inf) -> travel [20.6155], wait [0..Inf)
              -> 5 [0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf)
              -> 16 [0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf)
              -> 17 [0..Inf), delay [0..Inf) -> travel [13.9284], wait [0..Inf)
              -> 8 [0..Inf), delay [0..Inf) -> travel [10.4403], wait [0..Inf)
              -> 18 [0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf)
              -> 6 [0..Inf), delay [0..Inf) -> travel [7.07107], wait [0..Inf)
```

```
                -> 13 [0..Inf), delay [0..Inf) -> travel [11.1803], wait [0..Inf)
                -> Depot [0..Inf), delay [0..Inf) -> travel [0], wait [0..Inf)
```

See Also: IloOutputManip, IloTerse

Global function IloGeographical

```
public IloNum IloGeographical(IloNode node1, IloNode node2)
```

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

This function is a pre-defined distance function that returns the geographical distance between two nodes for a specified vehicle. The x-coordinate of each node represents the longitude in degrees, and the y-coordinate represents the latitude. The distance is computed on the unit sphere (that is, a sphere with a radius of 1).

The distance on the earth can be computed with the approximate radius of 6378.137 kilometers. For example, using this function to compute the distance between Mountain View, California, (-122.067, 37.388) and Paris, France, (2.333, 48.867) gives a distance of around 8981 kilometers.

Implementation

The function `IloGeographical` may be implemented like this:

```
IloNum IloGeographical(IloNode node1, IloNode node2) {
    IloNum long1 = node1.getX() * IloPi / 180.0;
    IloNum lat1  = node1.getY() * IloPi / 180.0;
    IloNum long2 = node2.getX() * IloPi / 180.0;
    IloNum lat2  = node2.getY() * IloPi / 180.0;
    return acos( sin(lat1)*sin(lat2)
                + cos(lat1)*cos(lat2)
                *(cos(long1)*cos(long2) + sin(long1)*sin(long2))
                );
}
```

See Also: IloDistance, IloDistMax, IloEuclidean, IloManhattan, IloNode, IloSimpleDistanceFunction

Global function IloCompatible

```
public IloConstraint IloCompatible(IloVisitVehicleCompat compat, const char *
name=0)
```

Definition file: ildispat/ilocompat.h
Include file: <ildispat/ilodispatcher.h>

This function creates a compatibility constraint from a compatibility relation. This constraint ensures that only compatible vehicles can be assigned to a visit.

As with all Solver constraints, this constraint should be added to an instance of `IloModel`. The parameter `name`, if present, is used as the name of the constraint.

Global function IloEarlinessFunction

```
public IloNumToNumSegmentFunction IloEarlinessFunction(IloEnv env, IloNum
earlyThresh, IloNum earlyCost)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This function creates a piecewise linear function `f` such that:

- *if 0 <= x < earlyThresh, f(x) = earlyCost * (earlyThresh - x),*
- *if earlyThresh <= x, f(x) = 0.*

It can be used to express earliness costs on visits, in conjunction with `IloVisit::setStartCost()` and `IloVisit::setEndCost()`, where `earlyCost` is the cost per unit of dimension for performing a visit before `earlyThresh`.

Global function operator<<

```
public ostream & operator<<(ostream & stream, const IloDispatcher & plan)
```

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

This operator has been overloaded to treat Dispatcher objects appropriately as output. It directs its output to an output stream (normally, standard output) and displays information about its second argument `plan`.

See Also: IloDispatcher, IloOutputManip, IloRoutingSolution

Global function operator<<

```
public ostream & operator<<(ostream & stream, const IloOutputManip & manip)
```

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

This operator has been overloaded to treat Dispatcher objects appropriately as output. It directs its output to an output stream (normally, standard output) and displays information about its second argument `manip`.

See Also: IloDispatcher, IloOutputManip, IloRoutingSolution

Global function operator<<

```
public ostream & operator<<(ostream & stream, const IloRoutingSolution & solution)
```

Definition file: ildispat/ilorsol.h
Include file: <ildispat/ilodispatcher.h>

This operator has been overloaded to treat Dispatcher objects appropriately as output. It directs its output to an output stream (normally, standard output) and displays information about its second argument `solution`.

See Also: IloDispatcher, IloOutputManip, IloRoutingSolution

Global function IloSortedNHood

```
public IloNHood IloSortedNHood(IloEnv env, IloNHood nhood, IloNumVar objVar)
```

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that behaves as `nhood`, except that the neighborhood order is modified to return deltas sorted by increasing the lower bound of the variable `objVar`. Deltas which do not contain variable `objVar` are returned last.

In Dispatcher, all predefined neighborhoods return deltas containing the cost variable (the variable returned by `IloDispatcher::getCostVar`). Its lower bound is an evaluation of the cost of the corresponding solution computed by the neighborhood. Therefore `IloSortedNHood` can be used to (approximately) sort Dispatcher neighborhoods according to cost.

For more information, see the concept Neighborhoods.

Global function IloSortedNHood

`public IloNHood` **`IloSortedNHood`**`(IloEnv env, IloNHood nhood, IloComparator< IloSolution > comparator)`

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that behaves as `nhood`, except that the neighborhood order is modified to return deltas sorted using the solution comparator `comparator`.

Note

The only information the comparator uses is what is contained in the delta, which is nothing more than a partial routing solution created by the neighborhoods.

Example

The following code creates a comparator for which the best delta is the one involving the vehicle with the highest capacity:

```
ILOCOMPARATOR1(MyComparator,
               IloSolution, delta1, delta2,
               IloDimension1, dim1) {
  IloRoutingSolution rdelta1(delta1);
  IloNum maxCapacity1 = 0;
  for (IloRoutingSolution::VisitIterator iter1(rdelta1); iter1.ok(); ++iter1) {
    IloVisit visit = *iter1;
    if (rdelta1.isPerformed(visit)) {
      IloNum capacity = rdelta1.getVehicle(visit).getCapacity(dim1);
      maxCapacity1 = IloMax(capacity, maxCapacity1);
    }
  }
  IloRoutingSolution rdelta2(delta2);
  IloNum maxCapacity2 = 0;
  for (IloRoutingSolution::VisitIterator iter2(rdelta2); iter2.ok(); ++iter2) {
    IloVisit visit = *iter2;
    if (rdelta2.isPerformed(visit)) {
      IloNum capacity = rdelta2.getVehicle(visit).getCapacity(dim1);
      maxCapacity2 = IloMax(capacity, maxCapacity2);
    }
  }
  return maxCapacity1 > maxCapacity2;
}
```

It can be combined with a comparator created by `IloSolutionValueComparator` to only call `MyComparator` when both deltas have the same cost:

```
  IloComparator<IloSolution> myComparator = MyComparator(env, pallets);
  IloLexicographicComparator<IloSolution> comparator(env);
  comparator.add(IloSolutionValueComparator(env, dispatcher.getCostVar()));
  comparator.add(myComparator);
```

The resulting comparator can be passed to `IloSortedNHood` to select the appropriate neighbor.

For more information, see the concept Neighborhoods.

For more information, see the class `IloComparator` documented in the *IBM ILOG Solver Reference Manual*.

Global function IloInstantiateVehicleBreak

```
public IloGoal IloInstantiateVehicleBreak(IloEnv env, IloVehicleBreakCon brk,
IloNum precision=0.0)
public IlcGoal IloInstantiateVehicleBreak(IloSolver solver, IloVehicleBreakCon brk,
IloNum precision=0.0)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

A wide selection of vehicle break instantiation goals is supplied with Dispatcher so that you can build your own break instantiation goals. This is useful if you want to create breaks that have preferences, such as “take coffee breaks as late as possible and lunch breaks as early as possible,” that cannot be encoded as constraints. As in Solver, creating your own goals allows you to use variable or value ordering heuristics specific to your problem.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to instantiate the vehicle break constraint `brk`.

The instantiation proceeds as follows:

1. If it is possible that the break may not be performed, a choice point is created, and the break is NOT performed. On backtracking, the break is performed, and steps 2 - 4 are carried out.
2. The break's position is instantiated using `IloInstantiateVehicleBreakPosition`.
3. The break's duration is instantiated using `IloInstantiateVehicleBreakDuration`.
4. The break's start time is instantiated using `IloInstantiateVehicleBreakStart`.

The parameter `precision` is used to specify the required accuracy of start time and duration. When the difference between the minimum and maximum of the start time or duration is less than or equal to `precision`, the domain of the corresponding variable is not reduced further. This can be used to increase solving speed when absolute accuracy is not required. The default is 0.0 (absolute accuracy).

Note

To increase performance when instantiating breaks on incomplete routing plans, we recommend that you instantiate breaks only on vehicles with complete routes. The `IloInstantiateVehicleBreaks` goals do this automatically.

See Also: IloInstantiateVehicleBreakDuration, IloInstantiateVehicleBreakPosition, IloInstantiateVehicleBreaks, IloInstantiateVehicleBreakStart, IloVehicleBreakCon

Global function IloMakePerformed

```
public IloNHood IloMakePerformed(IloEnv env, IloDispatcherNHoodParameters params)
public IloNHood IloMakePerformed(IloEnv env)
```

Definition file: ildispat/perfnhood.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by inserting an unperformed visit after a performed one.

The optional parameter `params` can be used to customize the behavior of the neighborhood. In particular, the vehicle array specified using `setVehicles` on the `IloDispatcherNHoodParameters` class will make the neighborhood insert unperformed visits only on the routes of these vehicles.

For more information, see the concept Neighborhoods.

See Also: IloCross, IloExchange, IloFPRelocate, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloRelocate, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloTerse

```
public IloOutputManip IloTerse(IloDispatcher dispatcher)
```

Definition file: ildispat/ilodispat.h
Include file: <ildispat/ilodispatcher.h>

This function returns an output manipulator that limits the output data displayed by the overloaded operator. When this function is used, only the order in which visits are made is displayed.

Example

The following is an example of the output produced by the `IloTerse` operator:

```
Solution      :
 Unperformed visits : None
 Vehicle 0 Route : Depot -> 52 -> 7 -> 19 -> 11 -> 64 -> 49 -> 36 -> 47
                   -> 46 -> 8 -> 45 -> 17 -> 84 -> 5 -> 89 -> Depot
 Vehicle 1 Route : Depot -> 50 -> 33 -> 81 -> 34 -> 78 -> 79 -> 3 -> 77
                   -> 76 -> 28 -> depot -> 53 -> 58 -> 2 -> 57 -> 15 -> 43
                   -> 42 -> 14 -> 38 -> 44 -> 16 -> 61 -> 59 -> 95 -> Depot
 Vehicle 2 Route : Depot -> 18 -> 82 -> 48 -> 62 -> 20 -> 66 -> 65 -> 71
                   -> 35 -> 9 -> 51 -> 1 -> depot -> 26 -> 4 -> 25 -> 55
                   -> 54 -> 24 -> 29 -> 68 -> 80 -> 12 -> Depot
 Vehicle 3 Route : Depot -> 94 -> 96 -> 99 -> 60 -> 83 -> 88 -> 31 -> 10
                   -> 63 -> 90 -> 32 -> 30 -> 70 -> 69 -> 27 -> depot
                   -> 6 -> 93 -> 85 -> 86 -> 91 -> 100 -> 98 -> 37 -> 92
                   -> 97 -> 87 -> 13 -> depot -> 40 -> 73 -> 74 -> 22
                   -> 41 -> 23 -> 67 -> 39 -> 56 -> 75 -> 72 -> 21 -> Depot
```

See Also: IloOutputManip, IloVerbose

Global function IloSetVehicleVisitCumuls

```
public IlcGoal IloSetVehicleVisitCumuls(IloSolver solver, IloVehicle vehicle,
IloDimension2 dim, IloNum precision=1e-6)
public IloGoal IloSetVehicleVisitCumuls(IloEnv env, IloVehicle vehicle,
IloDimension2 dim, IloNum precision=1e-6)
```

Definition file: ildispat/setcumul.h
Include file: <ildispat/ilodispatcher.h>

The following goals greedily instantiate the cumul variables of some or all visits of the routing problem for a given dimension. The instantiation tries to minimize the cumul and end-cumul costs attached to the visits taking into account execution intervals and vehicle breaks. The parameter precision is used to specify the required accuracy of the cumul values. When the difference between the minimum and maximum of the cumul variable is less than or equal to `precision`, the domain of the variable is not reduced further. This can be used to increase solving speed when absolute accuracy is not required. The default is 1e-6.

Depending on whether `env` or `solver` is specified, the functions return an `IloGoal` or `IlcGoal` to instantiate the cumul variables of the visits performed by vehicle `vehicle` for dimension `dim`.

Global function IloInstantiateVehicleBreakStart

```
public IloGoal IloInstantiateVehicleBreakStart(IloEnv env, IloVehicleBreakCon brk,
IloNum precision=0.0, IloNum target=0.5)
public IlcGoal IloInstantiateVehicleBreakStart(IloSolver solver, IloVehicleBreakCon
brk, IloNum precision=0.0, IloNum target=0.5)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to instantiate the start time of the break `brk`. It is usually advisable to instantiate the position of the break first. (See `IloInstantiateVehicleBreakPosition`.)

The parameter `precision` is used to specify the required accuracy of the start time. When the difference between the minimum and maximum of the start time is less than or equal to `precision`, the domain of the corresponding variable is not reduced further. This can be used to increase solving speed when absolute accuracy is not required. The default is 0.0 (absolute accuracy).

A "middle" start time, indicated by the parameter `target`, is the default. The value of `target` must be between 0.0 and 1.0. The parameter is used as follows:

1. Choose a "mid" value of `startTime.getMin() + startTime.getSize() * target`.
2. Create a choice point, and try to instantiate the start time `<= mid` to its maximum value or the start time `>= mid` to its minimum value.
3. Explore the smallest portion first.

This is very useful for instantiating breaks in the middle of their window as instantiating them at the extremities can leave zero slack in parts of the vehicle route. This may be undesirable.

See Also: IloInstantiateVehicleBreak, IloInstantiateVehicleBreakDuration, IloInstantiateVehicleBreakPosition, IloInstantiateVehicleBreaks, IloVehicleBreakCon

Global function IloAllVehiclesDifferent

```
public IloBool IloAllVehiclesDifferent(IloVehicle vehicle1, IloVehicle vehicle2)
```

Definition file: ildispat/ilovehicleequiv.h
Include file: <ildispat/ilodispatcher.h>

This function returns `IloFalse` if the two vehicles passed as arguments are different with respect to a distance function to be specified. This function, which is of type `IloVehicleEquivFunction`, can be used in the definition of distance functions.

See Also: IloAllVehiclesEquivalent, IloVehicleEquiv, IloVehicleEquivI

Global function IloBoxVehiclePairPredicate

```
public IloVehiclePairPredicate IloBoxVehiclePairPredicate(IloEnv env, IloNum ratio)
```

Definition file: ildispat/ilovehiclepredicate.h
Include file: <ildispat/ilodispatcher.h>

This function returns a pre-defined vehicle pair predicate which is true for a pair of vehicles if the bounding boxes of their respective routes have a big enough intersection:

- let `v1` and `v2` be two vehicles,
- let `A1` be the area of the bounding box of the route of `v1`,
- let `A2` be the area of the bounding box of the route of `v2`,
- let `AI` be the area of the intersection of the two bounding boxes,
- the predicate accepts the pair `(v1, v2)` if `ratio * IloMin(A1, A2) <= AI`

See Also: IloVehiclePairPredicate

Global function IloEuclidean

```
public IloNum IloEuclidean(IloNode node1, IloNode node2)
```

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

This function is a pre-defined distance function that returns the Euclidean distance between two nodes.

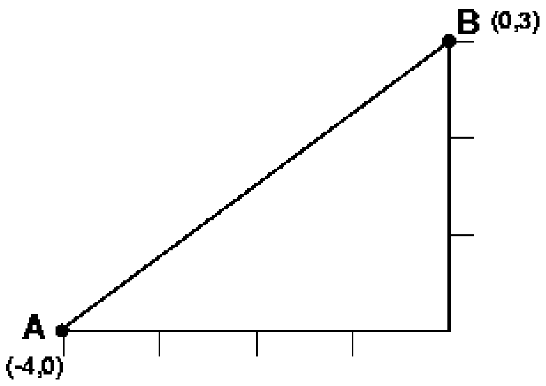
Implementation

This function may be implemented like this:

```
IloNum IloEuclidean(IloNode node1, IloNode node2){
    IloNum x = node1.getX()-node2.getX();
    IloNum y = node1.getY()-node2.getY();
    IloNum z = node1.getZ()-node2.getZ();
    return sqrt(x*x + y*y + z*z);
}
```

Example

In the figure below, the Euclidean distance between A and B is equal to $\sqrt{-4 \times -4 + -3 \times -3} = \sqrt{25} = 5$. (The z-coordinates of A and B are assumed to be equivalent.)



See Also: IloDistance, IloDistMax, IloGeographical, IloManhattan, IloNode, IloSimpleDistanceFunction

Global function IloCouple

```
public void IloCouple(IloNHood nh, IloMetaHeuristic mh)
```

Definition file: ildispat/ilometa.h
Include file: <ildispat/ilodispatcher.h>

This function couples the specified neighborhood and metaheuristic. This is done by a sharing of data structures between the neighborhood and metaheuristic.

When `mh` is of type `IloDispatcherGLS`, or is a composed metaheuristic (created by `operator +` between metaheuristics) containing an instance of `IloDispatcherGLS`, the neighborhood must be coupled to the metaheuristic. Otherwise, an `IloException` is thrown when you try to use the instance of `IloDispatcherGLS` in the search.

See Also: IloDispatcherGLS, IloDecouple

Global function IloTwoOpt

```
public IloNHood IloTwoOpt(IloEnv env, IloDispatcherNHoodParameters params, IloBool
norestarts=IloTrue)
public IloNHood IloTwoOpt(IloEnv env, IloBool norestarts=IloTrue)
```

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by breaking two arcs.

The optional argument `params` can be specified to customize the behavior of the neighborhood.

In particular, if a vehicle array has been passed to `IloDispatcherNHoodParameters::setVehicles`, the function returns a neighborhood that uses the two-opt heuristic on the routes of the specified array of vehicles.

The neighborhood `IloTwoOpt` starts looking for new neighbors at the place where the last modification took place. This behavior has changed from that of Dispatcher versions 3.2 and earlier. In earlier versions, `IloTwoOpt` would look for new neighbors starting from the first visit of the last modified vehicle. When the flag `norestarts` is set to `IloFalse`, the behavior of `IloTwoOpt` is that of Dispatcher versions 3.2 and earlier.

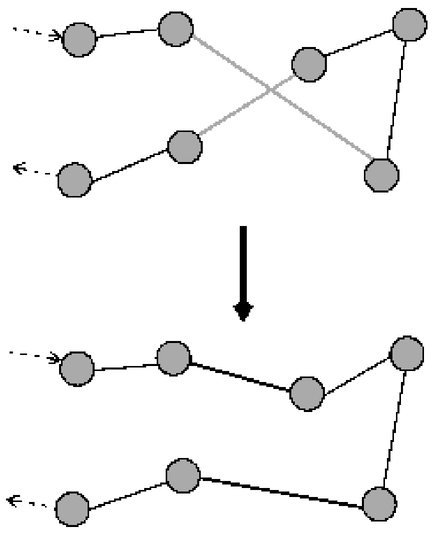
Two-Opt Heuristic

1. Take an initial route.
2. Remove two arcs from the route, and try the other possible reconnection of the remaining parts of the route.
3. If the cost has been reduced and if all constraints are satisfied, go back to Step 2.
4. End.

With this heuristic, directional flows between visits may be reversed. The presence of tight time constraints can therefore decrease its effectiveness.

Example

The following figure illustrates this process. Here, we assume that the cost is proportional to the length of the tour. The neighborhood eliminates the crossing by destroying two arcs and creating two new arcs. The resulting route is shorter, and thus less costly.



For more information, see the concept Neighborhoods.

See Also: IloCross, IloExchange, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloRelocate, IloSwapPerform, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloVehicleDependentDelayConstraint

```
public IloConstraint IloVehicleDependentDelayConstraint(IloDimension2 dim, IloVisit
visit, IloVehicleToNumFunction func)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This constraint allows you to model a vehicle dependent delay on the dimension specified by `dim` for the visit specified by `visit`.

For example, if the service time for `visit` is 10 if it is performed by *vehicle1* and 8 if it is performed by *vehicle2* you must create a function `func` to model this situation. This function is an instance of `IloVehicleToNumFunction` created with *vehicle1* and *vehicle2* and the corresponding values 10 and 8. For more information on how to create functions, see the available constructors for the class `IloVehicleToNumFunction`. Then you use the function `func` to create the vehicle dependent delay constraint for `visit` on time dimension `dim`. Finally, you add this constraint to the model.

See Also: IloDimension2, IloVisit, IloVehicleToNumFunction

Global function IloMax

```
public IloNumToNumSegmentFunction IloMax(IloNum value, IloNumToNumSegmentFunction
f)
public IloNumToNumSegmentFunction IloMax(IloNumToNumSegmentFunction f, IloNum
value)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This function returns a piecewise linear function *g* representing the maximum of value and function `f`: for all *x, g(x) = max(value, f(x))*.

Global function operator==

```
public IloConstraint operator==(IloVisit visit, IloVisitVar var)
public IloConstraint operator==(IloVisitVar var, IloVisit visit)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an equality constraint between its arguments.

These constraints state that the visit associated with `var` either must come directly before `visit` or must come directly after `visit`, depending on the meaning of `var` (a variable representing either a next or a previous visit).

See Also: IloVisitVar, IloVisit

Global function operator==

```
public IloConstraint operator==(IloVehicleVar vehicleVar1, IloVehicleVar
vehicleVar2)
```

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an equality constraint between its arguments.

This constraint states that the visit associated with `vehicleVar1` and the visit associated with `vehicleVar2` must be performed by the same vehicle.

See Also: IloVehicle, IloVehicleVar

Global function operator==

```
public IloConstraint operator==(IloVehicleArray vehicles, IloVehicleVar vehicleVar)
public IloConstraint operator==(IloVehicleVar vehicleVar, IloVehicleArray vehicles)
```

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an equality constraint between its arguments. When one of the arguments is an array, the constraint specifies an equality with one element of the array.

These constraints state that the visit associated with `vehicleVar` must be performed by one element of `vehicles`.

See Also: IloVehicleArray, IloVehicleVar

Global function operator==

```
public IloConstraint operator==(IloVisitArray visits, IloVisitVar var)
public IloConstraint operator==(IloVisitVar var, IloVisitArray visits)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an equality constraint between its arguments. When one of the arguments is an array, the constraint specifies an equality with one element of the array.

These constraints state that the visit associated with `var` either must come directly before one element of `visits` or must come directly after one element of `visits`, depending on the meaning of `var` (a variable representing either a next or a previous visit).

See Also: IloVisitVar, IloVisit

Global function operator==

```
public IloConstraint operator==(IloVehicle vehicle, IloVehicleVar vehicleVar)
public IloConstraint operator==(IloVehicleVar vehicleVar, IloVehicle vehicle)
```

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an equality constraint between its arguments.

These constraints state that the visit associated with `vehicleVar` must be performed by `vehicle`.

See Also: IloVehicle, IloVehicleVar

Global function IloDecouple

```
public void IloDecouple(IloNHood nh, IloMetaHeuristic mh)
```

Definition file: ildispat/ilometa.h
Include file: <ildispat/ilodispatcher.h>

This function decouples a previously coupled neighborhood/ metaheuristic pair.

If `nh` and `mh` are not coupled, an `IloException` is thrown.

See Also: IloDispatcherGLS, IloCouple

Global function IloGraphDistance

`public IloDistance` **`IloGraphDistance`**`(IloDispatcherGraph graph)`

Definition file: ildispat/ilographdist.h
Include file: <ildispat/ilodispatcher.h>

This function returns an instance of `IloDistance` for which the distance between two nodes for a specified vehicle is the value of the cheapest path between the two nodes using the specified vehicle. The value returned will depend on the instance of `IloDimension2` with which the distance instance is associated.

Implementation

The function `IloGraphDistance` may be implemented like this:

```
IloDispatcherGraph graph(env);
IloDistance dist = IloGraphDistance(graph);
IloDimension2 dim(env, dist);
```

If we suppose that `node1` and `node2` are instances of `IloNode` and that `vehicle` is an instance of `IloVehicle`, then `dist.getDistance(node1, node2, vehicle)` returns `graph.getDistance(node1, node2, vehicle, dim).`

See Also: IloDispatcherGraph, IloNode, IloVehicle

Global function IloMin

```
public IloNumToNumSegmentFunction IloMin(IloNum value, IloNumToNumSegmentFunction
f)
public IloNumToNumSegmentFunction IloMin(IloNumToNumSegmentFunction f, IloNum
value)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This function returns a piecewise linear function *g* representing the minimum of value and function `f`: for all *x, g(x) = min(value, f(x))*.

Global function IloSolutionValueComparator

```
public IloComparator< IloSolution > IloSolutionValueComparator(IloEnv env,
IloNumVar objVar)
```

Definition file: ildispat/lsearch.h

This function returns a solution comparator for which a solution `S1` is better than a solution `S2` if `objVar` has a lower value in `S1` than in `S2`.

For more information, see the class `IloComparator`, documented in the *IBM ILOG Solver Reference Manual*.

See Also: IloSortedNHood

Global function IloOrOpt

```
public IloNHood IloOrOpt(IloEnv env, IloDispatcherNHoodParameters params, IloBool
norestarts=IloTrue)
public IloNHood IloOrOpt(IloEnv env, IloBool norestarts=IloTrue)
```

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by using the *or-opt heuristic*. This consists of relocating segments of visits in the same route. These changes can result in lower cost.

The optional argument `params` can be specified to customize the behavior of the neighborhood.

In particular, if a vehicle array has been passed to `IloDispatcherNHoodParameters::setVehicles`, the function returns a neighborhood that uses the or-opt heuristic only on the routes of the specified array of vehicles.

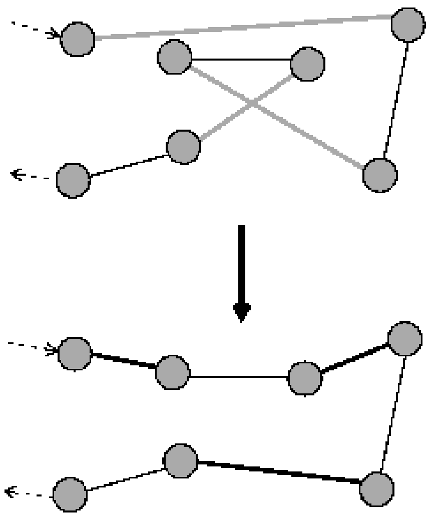
The neighborhood `IloOrOpt` starts looking for new neighbors at the place where the last modification took place. This behavior has changed from that of Dispatcher versions 3.2 and earlier. In earlier versions, `IloOrOpt` would look for new neighbors starting from the first visit of the last modified vehicle. When the flag `norestarts` is set to `IloFalse`, the behavior of `IloTwoOpt` is that of Dispatcher versions 3.2 and earlier.

Or-Opt Heuristic

1. Start with an initial route.
2. Move parts composed of one visit elsewhere in the route.
3. If the cost has been reduced and if all constraints are satisfied, go back to Step 2.
4. When all such moves have been tested, try moving parts of the route composed of two consecutive visits.
5. After testing all moves of parts composed of two consecutive visits, try moving parts of the route composed of three consecutive visits.

Example

The following figure illustrates the process. Here, the cost is assumed to be proportional to the length of the route. The operator eliminates the crossing by destroying three arcs and creating three new arcs. The resulting route is shorter, and thus less costly.



For more information, see the concept Neighborhoods.

See Also: IloCross, IloExchange, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloRelocate, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap,

Global function IloAllVehiclesEquivalent

```
public IloBool IloAllVehiclesEquivalent(IloVehicle vehicle1, IloVehicle vehicle2)
```

Definition file: ildispat/ilovehicleequiv.h
Include file: <ildispat/ilodispatcher.h>

This function returns `IloTrue` if the two vehicles passed as arguments are equivalent with respect to a distance function to be specified. This function, which is of type `IloVehicleEquivFunction`, can be used in the definition of distance functions.

See Also: IloAllVehiclesDifferent, IloVehicleEquiv, IloVehicleEquivI

Global function IloIntraRelocate

```
public IloNHood IloIntraRelocate(IloEnv env, IloDispatcherNHoodParameters params)
```

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by relocating individual visits to a new position in the same route. These relocations can result in cheaper routes.

The optional argument `params` can be specified to customize the behavior of the neighborhood.

This function is similar to `IloRelocate`, except that it relocates visits to a new position in the same route. Since it explores fewer options for the relocated visit, this neighborhood is potentially smaller than one created by `IloRelocate`.

For more information, see the concept Neighborhoods.

See Also: IloRelocate, IloCross, IloExchange, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloRelocate

```
public IloNHood IloRelocate(IloEnv env, IloDispatcherNHoodParameters params)
public IloNHood IloRelocate(IloEnv env)
```

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by relocating individual visits to a new position in another route. These relocations can result in cheaper routes.

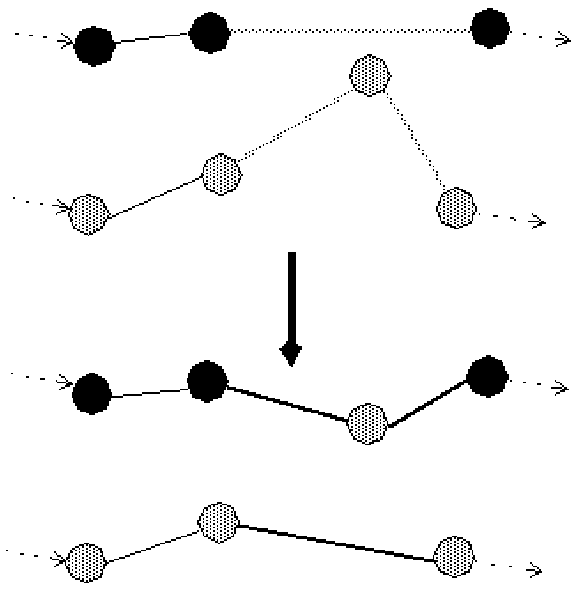
The optional argument `params` can be specified to customize the behavior of the neighborhood.

In particular, if a vehicle array has been passed to `IloDispatcherNHoodParameters::setVehicles`, the function returns a relocate neighborhood that operates on the routes of these vehicles.

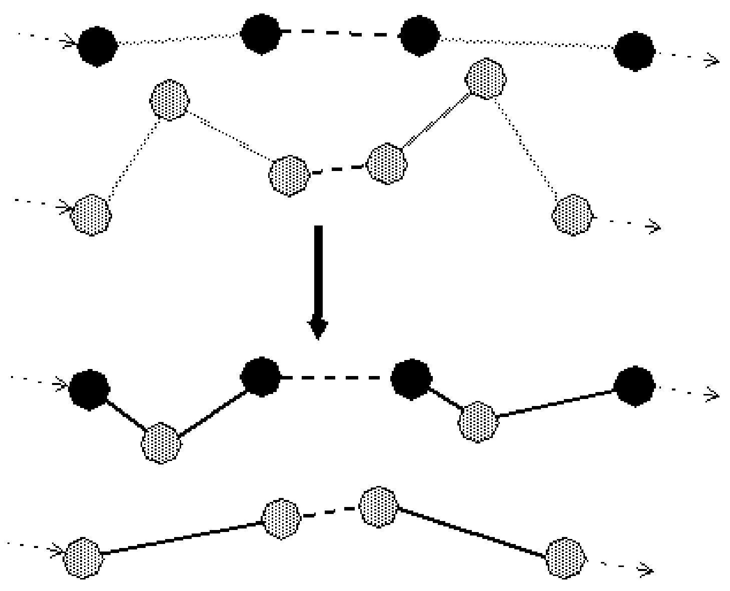
If `IloDispatcherNHoodParameters::setIgnorePairs` has been called with `IloTrue` as argument the neighborhood will perform a move only involving single visits. Otherwise pairs of visits can be moved together. This is the case when pairs of visits must be performed by the same vehicle. This move operator is useful for optimizing problems such as the Pickup-and-Delivery Problem (PDP).

Examples:

The following figure shows the process of relocating a visit. Here, we assume that the cost is proportional to the length of the route. The neighborhood destroys three arcs and creates three new arcs. As a result total travel distance, and thus cost, is less.



The following figure shows the process of relocating a pair of visits. Here, we assume that the cost is proportional to the length of the route. The neighborhood destroys six arcs and creates six new arcs. As a result total travel distance, and thus cost, is less.



For more information, see the concept Neighborhoods.

See Also: IloCross, IloExchange, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloAllUnperformedGenerate

```
public IloGoal IloAllUnperformedGenerate(IloEnv env)
public IlcGoal IloAllUnperformedGenerate(IloSolver solver)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` that creates a solution in which all visits are unperformed. For this goal to succeed, all visits must have a finite penalty cost, set through `IloVisit::setPenaltyCost`.

This goal quickly creates an initial solution that can be optimized with Dispatcher's improvement methods. Optimization in this case corresponds to reducing penalty cost by performing visits, and, as usual, reducing the cost of the vehicle routes.

See Also: IloDispatcherGenerate, IloInsertionGenerate, IloNearestAdditionGenerate, IloNearestDepotGenerate, IloSavingsGenerate, IloSweepGenerate, IloVisit

Global function operator!=

```
public IloConstraint operator!=(IloVisit visit, IloVisitVar var)
public IloConstraint operator!=(IloVisitVar var, IloVisit visit)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an inequality constraint between its arguments.

These constraints state that the visit associated with `var` either cannot come directly before `visit` or cannot come directly after `visit`, depending on the meaning of `var` (a variable representing a next or a previous visit).

See Also: IloVisitVar, IloVisit, IloVisitArray

Global function operator!=

```
public IloConstraint operator!=(IloVehicle vehicle, IloVehicleVar vehicleVar)
public IloConstraint operator!=(IloVehicleVar vehicleVar, IloVehicle vehicle)
```

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an inequality constraint between its arguments.

These constraints state that the visit associated with `vehicleVar` cannot be performed by `vehicle`.

See Also: IloVehicle, IloVehicleVar

Global function operator!=

```
public IloConstraint operator!=(IloVehicleVar vehicleVar1, IloVehicleVar
vehicleVar2)
```

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an inequality constraint between its arguments.

This constraint states that the visit associated with `vehicleVar1` and the visit associated with `vehicleVar2` cannot be performed by the same vehicle.

See Also: IloVehicle, IloVehicleVar

Global function operator!=

```
public IloConstraint operator!=(IloVehicleArray vehicles, IloVehicleVar vehicleVar)
public IloConstraint operator!=(IloVehicleVar vehicleVar, IloVehicleArray vehicles)
```

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an inequality constraint between its arguments. When one of the arguments is an array, the constraint specifies an inequality with all elements of the array.

These constraints state that the visit associated with `vehicleVar` cannot be performed by any element of `vehicles`.

See Also: IloVehicleArray, IloVehicleVar

Global function operator!=

```
public IloConstraint operator!=(IloVisitArray visits, IloVisitVar var)
public IloConstraint operator!=(IloVisitVar var, IloVisitArray visits)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This operator creates and returns an inequality constraint between its arguments. When one of the arguments is an array, the constraint specifies an inequality with all elements of the array.

These constraints state that the visit associated with `var` either cannot come directly before any element of `visits` or cannot come directly after any element of `visits`, depending on the meaning of `var` (a variable representing a next or a previous visit).

See Also: IloVisitVar, IloVisit, IloVisitArray

Global function IloDistMax

`public IloNum **IloDistMax**(IloNode node1, IloNode node2)`

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

This function is a pre-defined distance function that returns the “maximum-distance” between two nodes. Here, maximum is defined as the maximum of the three absolute values representing the differences between the x-, y-, and z- coordinates of the two nodes.

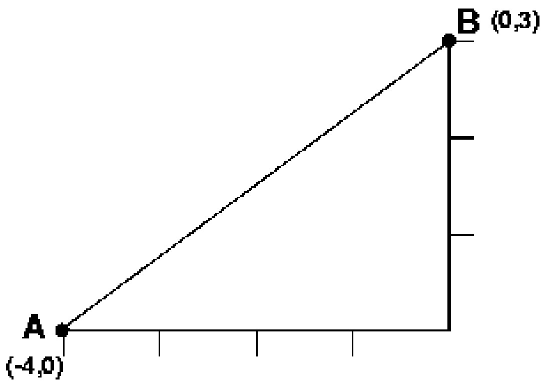
Implementation

This function may be implemented like this:

```
IloNum IloDistMax(IloNode node1, IloNode node2) {
    IloNum x = IloAbs(node1.getX()-node2.getX());
    IloNum y = IloAbs(node1.getY()-node2.getY());
    IloNum z = IloAbs(node1.getZ()-node2.getZ());
    return IloMax(x, IloMax(y, z));
}
```

Example

In the figure below, the “maximum-distance” between A and B is equal to $max(|-4-0|,|0-3|) = max(4,3) = 4$. (The z-coordinates of A and B are assumed to be equivalent.)



See Also: IloDistance, IloEuclidean, IloGeographical, IloManhattan, IloNode

Global function IloInstantiateTransits

```
public IloGoal IloInstantiateTransits(IloEnv env, IloVehicle vehicle, IloDimension
dim)
public IlcGoal IloInstantiateTransits(IloSolver solver, IloVehicle vehicle,
IloDimension dim)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These goals instantiate the transit variables for different aspects of the routing problem. A transit variable exists for each visit and dimension pair and corresponds to the usage of the dimension from the visit to the next visit in the route. Each of these goals tries to instantiate the transit to its smallest value.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to instantiate the transit variables for all visits on the specified vehicle, but only for the specified dimension.

See Also: IloDimension, IloVehicle

Global function IloInstantiateTransits

```
public IloGoal IloInstantiateTransits(IloEnv env, IloDimension dim)
public IlcGoal IloInstantiateTransits(IloSolver solver, IloDimension dim)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These goals instantiate the transit variables for different aspects of the routing problem. A transit variable exists for each visit and dimension pair and corresponds to the usage of the dimension from the visit to the next visit in the route. Each of these goals tries to instantiate the transit to its smallest value.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to instantiate the transit variables for all visits, but only for the specified dimension.

See Also: IloDimension, IloVehicle

Global function IloInstantiateTransits

```
public IloGoal IloInstantiateTransits(IloEnv env, IloVehicle vehicle)
public IlcGoal IloInstantiateTransits(IloSolver solver, IloVehicle vehicle)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These goals instantiate the transit variables for different aspects of the routing problem. A transit variable exists for each visit and dimension pair and corresponds to the usage of the dimension from the visit to the next visit in the route. Each of these goals tries to instantiate the transit to its smallest value.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to instantiate the transit variables for all visits on the specified vehicle and in all dimensions.

See Also: IloDimension, IloVehicle

Global function IloInstantiateTransits

```
public IloGoal IloInstantiateTransits(IloEnv env)
public IlcGoal IloInstantiateTransits(IloSolver solver)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These goals instantiate the transit variables for different aspects of the routing problem. A transit variable exists for each visit and dimension pair and corresponds to the usage of the dimension from the visit to the next visit in the route. Each of these goals tries to instantiate the transit to its smallest value.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to instantiate the transit variables for all visits in all dimensions.

See Also: IloDimension, IloVehicle

Global function IloInsertVisit

```
public IloGoal IloInsertVisit(IloEnv env, IloVisit visit, IloRoutingSolution
solution, IloDispatcherInsertionParameters param=0)
public IloGoal IloInsertVisit(IloEnv env, IloVisit visit, IloRoutingSolution
solution, IloDispatcherGoalFactory goalFactory, IloDispatcherInsertionParameters
param=0)
public IlcGoal IloInsertVisit(IloSolver solver, IloVisit visit, IloRoutingSolution
solution, IloDispatcherInsertionParameters param=0)
public IlcGoal IloInsertVisit(IloSolver solver, IloVisit visit, IloRoutingSolution
solution, IloDispatcherGoalFactory goalFactory, IloDispatcherInsertionParameters
param=0)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These functions return goals that insert `visit` into a routing solution using a best insertion algorithm.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to insert `visit` at the cheapest place in `solution`. If `visit` is already in `solution`, a re-insertion is performed. These functions also allow you to specify a secondary goal to be satisfied using the `IloDispatcherGoalFactory` parameter. If a subgoal is specified, it executes after each insertion attempt.

See Also: IloVisit

Global function IloInsertVisit

```
public IloGoal IloInsertVisit(IloEnv env, IloVisit visit, IloVehicle vehicle,
IloRoutingSolution solution, IloDispatcherInsertionParameters param=0)
public IloGoal IloInsertVisit(IloEnv env, IloVisit visit, IloVehicle vehicle,
IloRoutingSolution solution, IloDispatcherGoalFactory goalFactory,
IloDispatcherInsertionParameters param=0)
public IlcGoal IloInsertVisit(IloSolver solver, IloVisit visit, IloVehicle vehicle,
IloRoutingSolution solution, IloDispatcherInsertionParameters param=0)
public IlcGoal IloInsertVisit(IloSolver solver, IloVisit visit, IloVehicle vehicle,
IloRoutingSolution solution, IloDispatcherGoalFactory goalFactory,
IloDispatcherInsertionParameters param=0)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These functions return goals that insert `visit` into a routing solution using a best insertion algorithm.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to insert `visit` at the cheapest place in `vehicle` in `solution`. If `visit` is already in `vehicle`, a re-insertion is performed. These functions also allow you to specify a secondary goal to be satisfied using the `IloDispatcherGoalFactory` parameter. If a subgoal is specified, it executes after each insertion attempt.

See Also: IloVisit

Global function IloInsertVisit

```
public IloGoal IloInsertVisit(IloEnv env, IloVisit visit, IloVisit visitBefore,
IloRoutingSolution solution)
public IloGoal IloInsertVisit(IloEnv env, IloVisit visit, IloVisit visitBefore,
IloRoutingSolution solution, IloGoal subGoal)
public IlcGoal IloInsertVisit(IloSolver solver, IloVisit visit, IloVisit
visitBefore, IloRoutingSolution solution)
public IlcGoal IloInsertVisit(IloSolver solver, IloVisit visit, IloVisit
visitBefore, IloRoutingSolution solution, IlcGoal subGoal)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These functions return goals that insert `visit` into a routing solution using a best insertion algorithm.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to insert `visit` after `visitBefore` in `solution`. If `visit` is already in `solution`, a re-insertion is performed. These functions also allow you to specify a secondary goal to be satisfied. If a subgoal is specified, it executes after each insertion attempt.

See Also: IloVisit

Global function IloInsertVisit

```
public IloGoal IloInsertVisit(IloEnv env, IloVisit visit, IloVehicleArray vehicles,
IloRoutingSolution solution, IloDispatcherInsertionParameters param=0)
public IloGoal IloInsertVisit(IloEnv env, IloVisit visit, IloVehicleArray vehicles,
IloRoutingSolution solution, IloDispatcherGoalFactory goalFactory,
IloDispatcherInsertionParameters param=0)
public IlcGoal IloInsertVisit(IloSolver solver, IloVisit visit, IloVehicleArray
vehicles, IloRoutingSolution solution, IloDispatcherInsertionParameters param=0)
public IlcGoal IloInsertVisit(IloSolver solver, IloVisit visit, IloVehicleArray
vehicles, IloRoutingSolution solution, IloDispatcherGoalFactory goalFactory,
IloDispatcherInsertionParameters param=0)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

These functions return goals that insert `visit` into a routing solution using a best insertion algorithm.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to insert `visit` at the cheapest place in `vehicles` in `solution`. If `visit` is already in `vehicles`, a re-insertion is performed. These functions also allow you to specify a secondary goal to be satisfied using the `IloDispatcherGoalFactory` parameter. If a subgoal is specified, it executes after each insertion attempt.

See Also: IloVisit

Global function IloFunctionDistance

```
public IloVisitDistance IloFunctionDistance(IloNumToNumSegmentFunction f,
IloVisitDistance d)
```

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>

This function returns a distance object for which distances between visits are the distances returned by `f(dval)`, where `dval` is the distance returned by `d`.

Global function IloAffineFunction

```
public IloNumToNumSegmentFunction IloAffineFunction(IloEnv env, IloNum slope,
IloNum a, IloNum fa)
```

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

This function creates an affine function `f` of slope `slope` such that *f(a) = fa*. In other words, for all *x, f(x) = slope * (x - a) + fa*.

Global function IloInsertionGenerate

```
public IloGoal IloInsertionGenerate(IloEnv env, IloDispatcherInsertionParameters
iparam=0)
public IloGoal IloInsertionGenerate(IloEnv, IloDispatcherGoalFactory goalFactory,
IloDispatcherInsertionParameters iparam=0)
public IlcGoal IloInsertionGenerate(IloSolver solver,
IloDispatcherInsertionParameters iparam=0)
public IlcGoal IloInsertionGenerate(IloSolver solver, IloDispatcherGoalFactory
goalFactory, IloDispatcherInsertionParameters iparam=0)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Dispatcher provides various heuristic algorithms to build a first solution for a routing plan, among them, an *insertion heuristic*.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` that builds a solution to the routing model using an algorithm which consists of inserting visits at the lowest cost position *at the time of insertion*. Note that this insertion point will not necessarily be the visit's lowest cost position when the entire routing plan has been constructed.

These functions also allow you to specify a secondary goal to be satisfied using the `IloDispatcherGoalFactory` parameter. If a subgoal is specified, it executes after each insertion attempt.

This function uses `IloInsertVisit` in the insertion process.

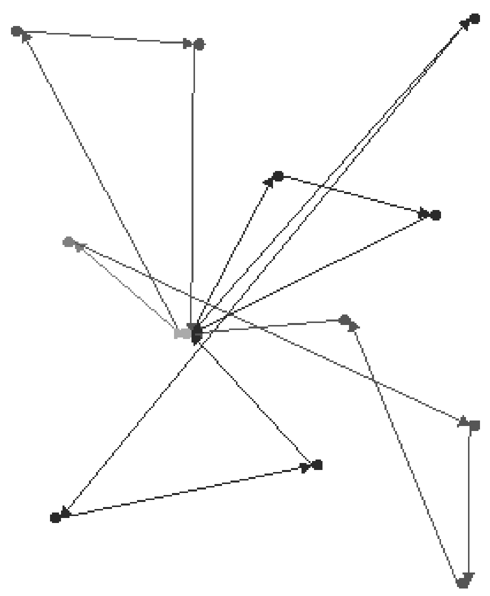
If not all visits can be inserted, the goal fails. In this case it may be better to start with an empty routing plan and insert visits using `IloInsertVisit`. Another option is to relax the problem, for example by adding more vehicles or allowing visits to be unperformed by setting a penalty cost on them.

Insertion Heuristic

1. Let all vehicles have empty routes.
2. Let *L* be the list of unassigned visits.
3. Take a visit *v* in *L*.
4. Insert *v* in a route at a feasible position where there will be the least increase in cost. If there is no feasible position, then the goal fails.
5. Remove *v* from *L*.
6. If *L* is not empty, go to 3.

Example

The following figure provides an example of the use of `IloInsertionGenerate` on a VRP with capacity constraints. The coordinates of the depot are (30,40). The capacity of the truck is 50 and there are 11 visits to perform. The tuples (x,y,quantity) of the visits are (17,63,19), (31,62,23), (52,64,16), (21,47,15), (37,52,7), (49,49,30), (42,41,19), (20,26,9), (40,30,21), (52,33,11) and (51,21,5).



If the cost of the routing plan is equal to the Euclidean distance traveled, the cost in this example is 320.83.

See Also: IloAllUnperformedGenerate, IloDispatcherGenerate, IloNearestAdditionGenerate, IloNearestDepotGenerate, IloSavingsGenerate, IloSweepGenerate

Global function IloSweepGenerate

```
public IloGoal IloSweepGenerate(IloEnv env)
public IloGoal IloSweepGenerate(IloEnv env, IloDispatcherGoalFactory goalFactory)
public IlcGoal IloSweepGenerate(IloSolver solver)
public IlcGoal IloSweepGenerate(IloSolver solver, IloDispatcherGoalFactory
goalFactory)
public IloGoal IloSweepGenerate(IloEnv env, IloSearchLimit limit)
public IloGoal IloSweepGenerate(IloEnv env, IloDispatcherGoalFactory goalFactory,
IloSearchLimit limit)
public IlcGoal IloSweepGenerate(IloSolver solver, IlcSearchLimit limit)
public IlcGoal IloSweepGenerate(IloSolver solver, IloDispatcherGoalFactory
goalFactory, IlcSearchLimit limit)
public IloGoal IloSweepGenerate(IloEnv, IloDispatcherFSParameters param)
public IlcGoal IloSweepGenerate(IloSolver, IloDispatcherFSParameters param)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Dispatcher provides various heuristic algorithms to build a first solution for a routing plan, among them, the *sweep heuristic*.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` that builds a solution to the problem at hand using the sweep heuristic. These functions also allow you to specify a secondary goal to be satisfied using the `IloDispatcherGoalFactory` parameter.

This goal relies on the x- and y-coordinates of nodes. Thus, each node in the routing plan must have its x- and y-coordinates defined before this method can be executed.

In the heuristic `IloSweepGenerate`, the goal `IloGenerateRoute` is used each time Dispatcher extends a route to ensure that the proposed extension is feasible. When the extension is not feasible, the search tree created by `IloGenerateRoute` is often too large to completely explore.

The parameter `limit`, if specified, places a limit on the search carried out in the route generation goal. This limit does not apply to any additional subgoals that you specify. If you also wish search to be limited in any additional goal, then a search limit should be applied to it individually.

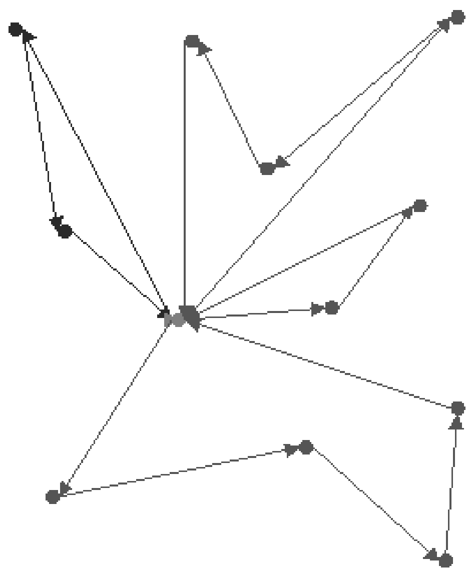
The parameter `IloDispatcherFSParameters` can be used to parameterize `IloSweepGenerate` in a variety of ways. Owing to the various different parameters that can be passed, passing these directly in the constructor of the first solution method can be cumbersome. The handle class `IloDispatcherFSParameters` encapsulates the different types of parameters that can be passed. For more information, see `IloDispatcherFSParameters`.

Sweep Heuristic

1. Let *O* be a site from which vehicles leave (usually a depot), and let *A* (different from *O*) be another site which serves as a reference.
2. Sort all the sites *S* in the routing plan by increasing angle *AOS*. Put the result in a list *L*.
3. The visits corresponding to the sites in *L* will be allocated to the vehicles in that order as long as constraints are respected.
4. If all vehicles have been used, the remaining visits are constrained to be unperformed. If one or more of these visits must be performed, the goal fails.

Example

The following figure provides an example of the use of `IloSweepGenerate` on a VRP with capacity constraints. The coordinates of the depot are (30,40). The capacity of the truck is 50 and there are 11 visits to perform. The tuples (x,y,quantity) of the visits are (17,63,19), (31,62,23), (52,64,16), (21,47,15), (37,52,7), (49,49,30), (42,41,19), (20,26,9), (40,30,21), (52,33,11) and (51,21,5).



`IloSweepGenerate` calculates 4 routes with total quantities collected of 34, 46, 49 and 46. If the cost of the routing plan is equal to the Euclidean distance traveled, the cost in this example is 270.40.

See Also: IloAllUnperformedGenerate, IloDispatcherGenerate, IloInsertionGenerate, IloNearestAdditionGenerate, IloNearestDepotGenerate, IloSavingsGenerate

Global function IloCross

```
public IloNHood IloCross(IloEnv env, IloDispatcherNHoodParameters params)
public IloNHood IloCross(IloEnv env)
```

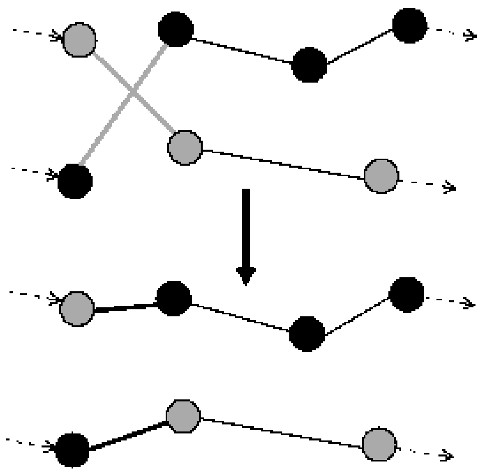
Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by exchanging the end parts of two routes. Chains of any length can be exchanged, and such exchanges can result in cheaper routes. The cross neighborhood is also capable (by an exchange of segments beginning just after the first visit) of swapping all visits of one vehicle with those of another.

The optional parameter `params` can be used to customize the behavior of the neighborhood. In particular, the vehicle array specified using `IloDispatcherNHoodParameters::setVehicles` will make the cross neighborhood operate only on the routes of these vehicles.

Example

The following figure illustrates the process. Here, we assume that the cost is proportional to the length of the route. The neighborhood eliminates the crossing by destroying two arcs and creating two new arcs. The resulting routes are shorter, and thus less costly.



For more information, see the concept Neighborhoods.

See Also: IloExchange, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloRelocate, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloNearestDepotGenerate

```
public IloGoal IloNearestDepotGenerate(IloEnv env)
public IlcGoal IloNearestDepotGenerate(IloSolver solver)
public IloGoal IloNearestDepotGenerate(IloEnv env, IloDispatcherGoalFactory
goalFactory)
public IlcGoal IloNearestDepotGenerate(IloSolver solver, IloDispatcherGoalFactory
goalFactory)
public IloGoal IloNearestDepotGenerate(IloEnv env, IloSearchLimit limit)
public IlcGoal IloNearestDepotGenerate(IloSolver solver, IlcSearchLimit limit)
public IloGoal IloNearestDepotGenerate(IloEnv env, IloDispatcherGoalFactory
goalFactory, IloSearchLimit limit)
public IlcGoal IloNearestDepotGenerate(IloSolver solver, IloDispatcherGoalFactory
goalFactory, IlcSearchLimit limit)
public IloGoal IloNearestDepotGenerate(IloEnv, IloDispatcherFSParameters param)
public IlcGoal IloNearestDepotGenerate(IloSolver, IloDispatcherFSParameters param)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Dispatcher provides various heuristic algorithms to build a first solution for a routing plan, among them, the *nearest to depot heuristic*.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` that builds a solution to the problem at hand using an algorithm that adds the visit that is the nearest to the “depot.” These functions also allow you to specify a secondary goal to be satisfied using the `IloDispatcherGoalFactory` parameter.

In the heuristic `IloNearestDepotGenerate`, the goal `IloGenerateRoute` is used each time Dispatcher extends a route to ensure that the proposed extension is feasible. When the extension is not feasible, the search tree created by `IloGenerateRoute` is often too large to completely explore.

The parameter `limit`, if specified, places a limit on the search carried out in the route generation goal. This limit does not apply to any additional subgoals that you specify. If you also wish search to be limited in any additional goal, then a search limit should be applied to it individually.

The parameter `IloDispatcherFSParameters` can be used to parameterize `IloNearestDepotGenerate` in a variety of ways. Owing to the various different parameters that can be passed, passing these directly in the constructor of the first solution method can be cumbersome. The handle class `IloDispatcherFSParameters` encapsulates the different types of parameters that can be passed. For more information, see `IloDispatcherFSParameters`.

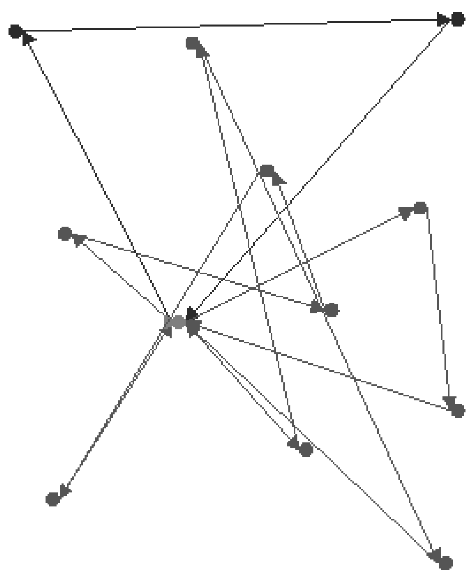
Nearest to Depot Heuristic

For all vehicles:

1. Denote the vehicle to be considered by *w*.
2. Start with a partial route consisting of the departure from the depot.
3. Find the visit *v* which is closest to the starting point of the current partial route of *w*. If it is not possible to find such a visit without violating constraints, close the current partial route of *w*, choose another empty vehicle and go to step 2. If no empty vehicles remain, the goal fails.
4. Add *v* to the end of the partial route.
5. If there are more visits to schedule, go to step 3.
6. If all vehicles have been used, the remaining visits are constrained to be unperformed. If one or more of these visits must be performed, the goal fails.

Example

The following figure provides an example of the use of `IloNearestDepotGenerate` on a VRP with capacity constraints. The coordinates of the depot are (30,40). The capacity of the truck is 50 and there are 11 visits to perform. The tuples (x,y,quantity) of the visits are (17,63,19), (31,62,23), (52,64,16), (21,47,15), (37,52,7), (49,49,30), (42,41,19), (20,26,9), (40,30,21), (52,33,11) and (51,21,5).



`IloNearestDepotGenerate` calculates 4 routes with total quantities collected of 50, 35, 41 and 49. If the cost of the routing plan is equal to the Euclidean distance traveled, the cost in this example is 369.30.

See Also: IloAllUnperformedGenerate, IloDispatcherGenerate, IloInsertionGenerate, IloNearestAdditionGenerate, IloSavingsGenerate, IloSweepGenerate

Global function IloSetVisitCumuls

```
public IlcGoal IloSetVisitCumuls(IloSolver solver, IloDimension2 dim, IloNum
precision=1e-6)
public IloGoal IloSetVisitCumuls(IloEnv env, IloDimension2 dim, IloNum
precision=1e-6)
```

Definition file: ildispat/setcumul.h
Include file: <ildispat/ilodispatcher.h>

The following goals greedily instantiate the cumul variables of some or all visits of the routing problem for a given dimension. The instantiation tries to minimize the cumul and end-cumul costs attached to the visits taking into account execution intervals and vehicle breaks. The parameter precision is used to specify the required accuracy of the cumul values. When the difference between the minimum and maximum of the cumul variable is less than or equal to `precision`, the domain of the variable is not reduced further. This can be used to increase solving speed when absolute accuracy is not required. The default is 1e-6.

Depending on whether `env` or `solver` is specified, the functions return an `IloGoal` or `IlcGoal` to instantiate the cumul variables of all visits for dimension `dim`.

Global function IloInstantiateVehicleBreakDuration

```
public IloGoal IloInstantiateVehicleBreakDuration(IloEnv env, IloVehicleBreakCon
brk, IloNum precision=0.0, IloNum target=0.0)
public IlcGoal IloInstantiateVehicleBreakDuration(IloSolver solver,
IloVehicleBreakCon brk, IloNum precision=0.0, IloNum target=0.0)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` to instantiate the duration of the vehicle break `brk`.

The parameter `precision` is used to specify the required accuracy of the duration. When the difference between the minimum and maximum of the duration is less than or equal to `precision`, the domain of the corresponding variable is not reduced further. This can be used to increase solving speed when absolute accuracy is not required. The default is 0.0 (absolute accuracy).

A minimum duration, indicated by the parameter `target`, is the default. The parameter is used as follows:

1. Choose a “mid” value of `duration.getMin() + duration.getSize() * target`.
2. Create a choice point, and try to instantiate the duration `<= mid` to its maximum value or the duration `>= mid` to its minimum value.
3. Explore the smallest portion first.

See Also: IloInstantiateVehicleBreak, IloInstantiateVehicleBreakPosition, IloInstantiateVehicleBreaks, IloInstantiateVehicleBreakStart, IloVehicleBreakCon

Global function IloFinalizePlan

```
public IloGoal IloFinalizePlan(IloEnv env)
public IlcGoal IloFinalizePlan(IloSolver solver)
```

Definition file: ildispat/fsdecision.h
Include file: <ildispat/ilodispatcher.h>

When building a Dispatcher routing plan, the resulting plan may not be complete. This means that some visits are left unassigned, and that some routes are not complete. By complete, we mean that route visits must have their next variables instantiated from first to last. The routing plan may not be complete if, for example, only a subset of visits had been passed to the decision maker, or if a time-out occurred. An incomplete plan cannot be stored in an `IloRoutingSolution`, as some decision variables are left unbound.

The `IloFinalizePlan` function returns a goal that fixes an incomplete plan, if possible.

The goal performs the two functions:

1. Sets all visits that have no route assigned to unperformed.
2. Closes all incomplete routes.

Note that this goal may still fail. If, for example, a mandatory visit, having infinite penalty cost, has been left unassigned, then the `IloFinalizePlan` goal will try to set it to be unperformed, and fail.

You should run this goal after a decision maker attempts to finalize the plan and reach a state where the plan can be saved into an `IloRoutingSolution`.

Global function IloDispatcherGenerate

```
public IloGoal IloDispatcherGenerate(IloEnv env)
public IlcGoal IloDispatcherGenerate(IloSolver solver)
```

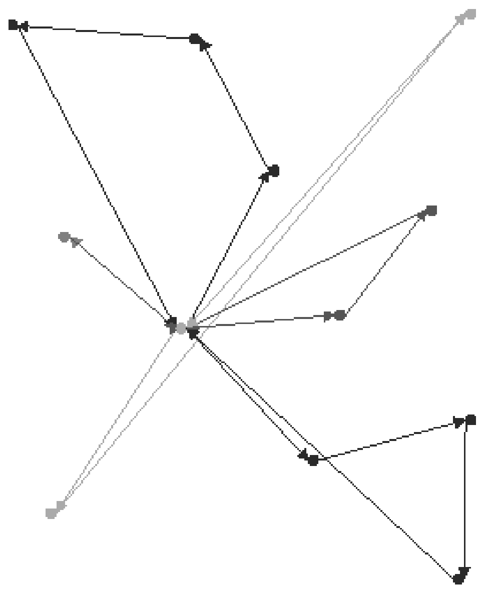
Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Dispatcher provides various algorithms to build a first solution for a routing problem, among them, a basic, complete *enumeration method*.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` that builds a solution to the problem at hand using an algorithm that completely explores the search space using backtracking. Of course, this method should only be used for small problems.

Example

The figure below provides an example of the use of `IloDispatcherGenerate` on a Vehicle Routing Problem (VRP) with capacity constraints. The coordinates of the depot are (30,40). The capacity of the truck is 50 and there are 11 visits to perform. The tuples (x,y,quantity) of the visits are (17,63,19), (31,62,23), (52,64,16), (21,47,15), (37,52,7), (49,49,30), (42,41,19), (20,26,9), (40,30,21), (52,33,11) and (51,21,5).



`IloDispatcherGenerate` calculates 4 routes with total quantities collected of 39, 42, 46 and 48. If the cost of the routing plan is equal to the Euclidean distance traveled, the cost in this example is 352.57.

See Also: IloAllUnperformedGenerate, IloInsertionGenerate, IloNearestAdditionGenerate, IloNearestDepotGenerate, IloSavingsGenerate, IloSweepGenerate

Global function operator+

```
public IloDistance operator+(IloDistance d1, IloDistance d2)
```

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

This operator returns a distance object for which the distance between two nodes is the sum of the distances between these two nodes returned by `d1` and `d2`.

Global function operator+

```
public IloVisitDistance operator+(IloVisitDistance d1, IloVisitDistance d2)
```

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>

This operator returns a distance object for which the distance between two visits is the sum of the distances between these two visits returned by `d1` and `d2`.

Global function operator+

```
public IloNumExprArg operator+(IloTravelSumVar travelSumVar, IloDelaySumVar
delaySumVar)
public IloNumExprArg operator+(IloDelaySumVar delaySumVar, IloTravelSumVar
travelSumVar)
```

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

This function creates a constrained expression representing the sum of a delay sum variable and of a travel sum variable.

This expression can be used to limit the amount of work performed by a vehicle.

Global function IloGenerateRoute

```
public IloGoal IloGenerateRoute(IloEnv env, IloVehicle vehicle)
public IlcGoal IloGenerateRoute(IloSolver solver, IloVehicle vehicle)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

This function returns a goal that generates a route for `vehicle` using a complete search method. The route generated is composed of visits that must be placed on the vehicle, that is, those which have constraints specifying that the only legal vehicle is `vehicle`.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal`.

To generate a route for a vehicle containing a specific set of visits, you can add constraints to limit the legal vehicles for the visits to the `vehicle`, use the goal, and then remove those constraints afterwards.

Global function IloFPRelocate

```
public IloNHood IloFPRelocate(IloEnv env, IloDispatcherNHoodParameters params)
public IloNHood IloFPRelocate(IloEnv env)
```

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by relocating individual visits to a new position in another route. These relocations can result in cheaper routes.

The optional argument `params` can be specified to customize the behavior of the neighborhood.

This function is similar to `IloRelocate` for PDP problems, except that it explores more options for the delivery component of a pickup-delivery pair that is being relocated. For example, consider one pair of pickup-delivery visits: *p1-d1*. `IloRelocate` would try to move *p1* after a performed pickup visit *p* and *d1* immediately after the corresponding delivery visit *d*. `IloFPRelocate` would try to move *p1* after a performed visit *v*, and then try to locate *d1* at every position on the route of the vehicle performing *v*. Thus, this neighborhood is potentially larger than that created by `IloRelocate`.

For more information, see the concept Neighborhoods.

See Also: IloRelocate, IloCross, IloExchange, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloGetDispatcherDefaultVehicleEquivalence

```
public IloVehicleEquiv IloGetDispatcherDefaultVehicleEquivalence(IloEnv env)
```

Definition file: ildispat/iloroutingplan.h
Include file: <ildispat/ilodispatcher.h>

This function returns Dispatcher's default vehicle equivalence class. Two vehicles are equivalent according to this class if:

- they share the same cost function
- the vehicles are equivalent according to the distance of the extrinsic dimensions appearing in their cost function
- the start visits of the vehicles are located at the same node
- the end visits of the vehicles are located at the same node

See Also: IloDispatcherNHoodParameters, IloVehicle, IloVehicleEquiv

Global function IloSameNodeArcPredicate

```
public IloArcPredicate IloSameNodeArcPredicate(IloEnv env)
```

Definition file: ildispat/iloarcpredicate.h
Include file: <ildispat/ilodispatcher.h>

This function returns an arc predicate which is satisfied if the two visits of an arc are located at the same node (an instance of `IloNode`). An arc is created when two visits are consecutive on a vehicle.

See Also: IloArcPredicate

Global function IloMergeAndRelocateTours

```
public IloNHood IloMergeAndRelocateTours(IloEnv env, IloDispatcherNHoodParameters
params, IloInt nbTours=2)
public IloNHood IloMergeAndRelocateTours(IloEnv env, IloInt nbTours=2)
```

Definition file: ildispat/reltours.h
Include file: <ildispat/ilodispatcher.h>

A tour is a sequence of pickup and delivery visits (visits constrained by an `IloOrderedVisitPair` constraint) such that:

- if the tour contains a pickup, it contains the corresponding delivery,
- if the tour contains a delivery, it contains the corresponding pickup,
- if a pickup is before the tour in the route, then its corresponding delivery is before the tour.

This function returns a neighborhood that modifies a solution by merging consecutive tours into one tour, relocating the new tour to a new position in the route of another vehicle and repairing the new tour if it violates the capacity constraints of the vehicle to which it was moved. The neighborhood will also define neighbors which only perform the merging and repairing and not relocating.

Merging tours is done by building a new tour which starts with the pickups of the tours being merged and ends with the corresponding deliveries. The relative order is kept of, respectively, the pickups and the deliveries.

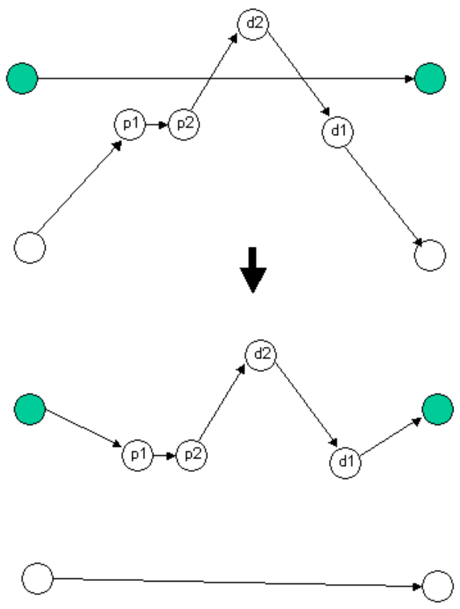
Repairing a tour is done by splitting it into smaller tours. When the capacity of the vehicle is exceeded by a pickup, the current tour is ended (using the necessary deliveries) and a new tour is started.

The optional argument `params` can be specified to customize the behavior of the neighborhood.

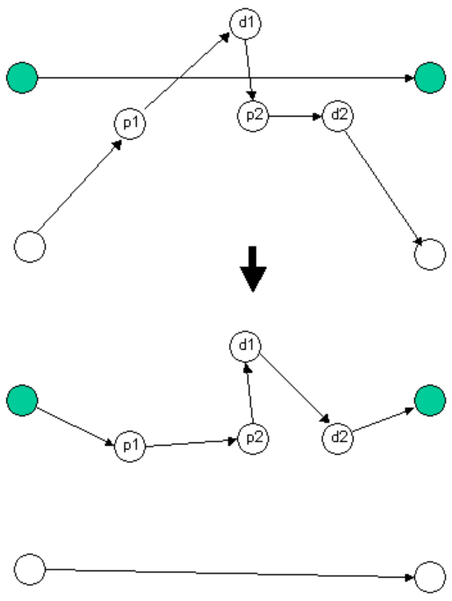
In particular, if a vehicle array has been passed to `IloDispatcherNHoodParameters::setVehicles`, the function returns a neighborhood that operates on the routes of these vehicles.

The parameter `nbTours` specifies the maximum number of consecutive tours the neighborhood is going to merge. By default, up to two tours will be merged.

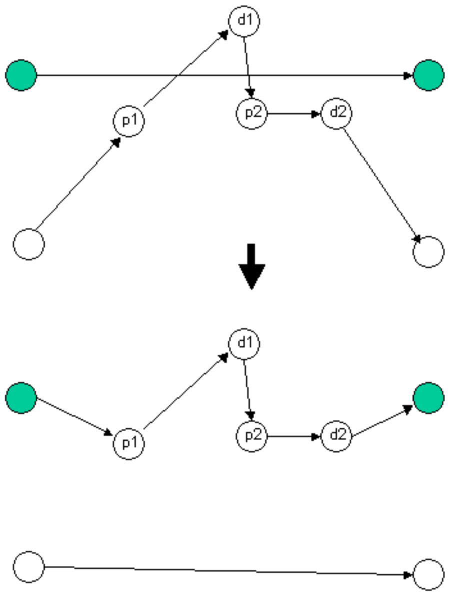
The following figure shows one tour relocated:



The following figure shows two tours merged and relocated:



The following figure shows two tours merged, relocated, and repaired due to capacity of destination vehicle:



Global function IloSavingsGenerate

```
public IloGoal IloSavingsGenerate(IloEnv env)
public IloGoal IloSavingsGenerate(IloEnv env, IloSearchLimit limit)
public IloGoal IloSavingsGenerate(IloEnv env, IloInt size)
public IloGoal IloSavingsGenerate(IloEnv env, IloInt size, IloSearchLimit limit)
public IloGoal IloSavingsGenerate(IloEnv env, IloDispatcherGoalFactory goalFactory)
public IloGoal IloSavingsGenerate(IloEnv env, IloDispatcherGoalFactory goalFactory,
IloSearchLimit limit)
public IloGoal IloSavingsGenerate(IloEnv env, IloInt size, IloDispatcherGoalFactory
goalFactory)
public IloGoal IloSavingsGenerate(IloEnv env, IloInt size, IloDispatcherGoalFactory
goalFactory, IloSearchLimit limit)
public IlcGoal IloSavingsGenerate(IloSolver solver)
public IlcGoal IloSavingsGenerate(IloSolver solver, IlcSearchLimit limit)
public IlcGoal IloSavingsGenerate(IloSolver solver, IloInt size)
public IlcGoal IloSavingsGenerate(IloSolver solver, IloInt size, IlcSearchLimit
limit)
public IlcGoal IloSavingsGenerate(IloSolver solver, IloDispatcherGoalFactory
goalFactory)
public IlcGoal IloSavingsGenerate(IloSolver solver, IloDispatcherGoalFactory
goalFactory, IlcSearchLimit limit)
public IlcGoal IloSavingsGenerate(IloSolver solver, IloInt size,
IloDispatcherGoalFactory goalFactory)
public IlcGoal IloSavingsGenerate(IloSolver solver, IloInt size,
IloDispatcherGoalFactory goalFactory, IlcSearchLimit limit)
public IloGoal IloSavingsGenerate(IloEnv, IloDispatcherFSParameters param)
public IlcGoal IloSavingsGenerate(IloSolver, IloDispatcherFSParameters param)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Dispatcher provides various heuristic algorithms to build a first solution for a routing plan, among them, the *savings heuristic*. The *savings heuristic* builds first solutions for both single depot and multiple depot vehicle routing problems.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` that builds a solution to the problem at hand using the *savings heuristic*. These functions also allow you to specify a secondary goal to be satisfied using the `IloDispatcherGoalFactory` parameter.

The argument `size` forces the *savings heuristic* to consider only the `n` closest visits (in terms of cost). While this may reduce the quality of the solution generated, reducing the number of neighbors can dramatically reduce the amount of memory needed to store the array of savings. If `size` is not specified, all visits are considered.

In the heuristic `IloSavingsGenerate`, the goal `IloGenerateRoute` is used each time Dispatcher extends a route to ensure that the proposed extension is feasible. When the extension is not feasible, the search tree created by `IloGenerateRoute` is often too large to completely explore.

The parameter `limit`, if specified, places a limit on the search carried out in the route generation goal. This limit does not apply to any additional subgoals that you specify. If you also wish search to be limited in any additional goal, then a search limit should be applied to it individually.

The parameter `IloDispatcherFSParameters` can be used to parameterize `IloSavingsGenerate` in a variety of ways. Owing to the various different parameters that can be passed, passing these directly in the constructor of the first solution method can be cumbersome. The handle class `IloDispatcherFSParameters` encapsulates the different types of parameters that can be passed. For more information, see `IloDispatcherFSParameters`.

Savings Heuristic

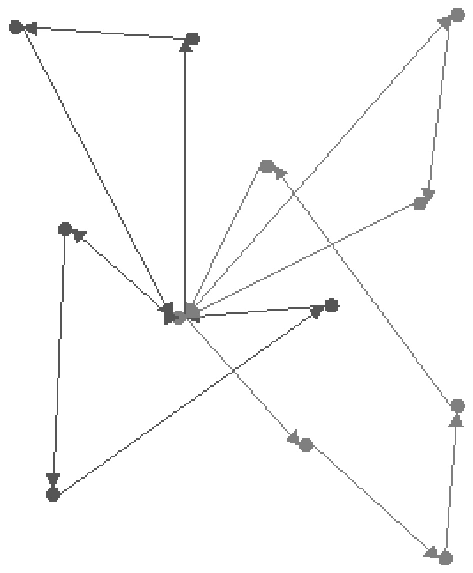
1. For all vehicles *v* having a starting visit *S* and an ending visit *E*, and for all pairs of visits *(i, j)*, compute

the savings function: *savings(i, j, v) = Mincost(i) + Mincost(j) - cost(S, i, v) - cost(i, j, v) - cost(j, E, v)* where *Mincost(i)* is the minimum value of *cost(S', i, v') + cost(i, E', v')* using vehicle *v'* (starting at *S'* and ending at *E'*).

2. Sort the arcs *(i, j, v)* according to *savings(i, j, v)* in descending order, and put them in a list *L*.
3. Scan *L* to find a feasible arc *(i, j, v)* that can be used to create an initial partial route for *v*. If no such legal arc can be found, go to step 5, otherwise remove the chosen arc from *L*.
4. Scan *L* to find an arc *(i', j', v)* that can be added to the start or end of the current partial route of vehicle *v*. If no such arc can be found, close vehicle *v* and go to step 3, otherwise remove the arc from *L*, and repeat step 4.
5. If all visits are scheduled, the goal succeeds. If there are unscheduled visits, they are constrained to be unperformed.

Example

The following figure provides an example of the use of `IloSavingsGenerate` on a VRP with capacity constraints. The coordinates of the depot are (30,40). The capacity of the truck is 50 and there are 11 visits to perform. The tuples (x,y,quantity) of the visits are (17,63,19), (31,62,23), (52,64,16), (21,47,15), (37,52,7), (49,49,30), (42,41,19), (20,26,9), (40,30,21), (52,33,11) and (51,21,5).



`IloSavingsGenerate` calculates 4 routes with total quantities collected of 42, 46, 44 and 43. If the cost of the routing plan is equal to the Euclidean distance traveled, the cost in this example is 280.95.

See Also: IloAllUnperformedGenerate, IloDispatcherGenerate, IloInsertionGenerate, IloNearestAdditionGenerate, IloNearestDepotGenerate, IloSweepGenerate

Global function operator*

```
public IloDistance operator*(IloNum c, IloDistance d)
```

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

This operator returns a distance object for which distances between nodes are the distances returned by `d` multiplied by a constant `c`.

Global function operator*

```
public IloVisitDistance operator*(IloNum c, IloVisitDistance d)
```

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>

This operator returns a distance object for which distances between visits are the distances returned by `d` multiplied by a constant `c`.

Global function operator*

```
public IloDistance operator*(IloDistance d, IloNum c)
```

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

This operator returns a distance object for which distances between nodes are the distances returned by `d` multiplied by a constant `c`.

Global function operator*

```
public IloVisitDistance operator*(IloVisitDistance d, IloNum c)
```

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>

This operator returns a distance object for which distances between visits are the distances returned by `d` multiplied by a constant `c`.

Global function IloNearestAdditionGenerate

```
public IloGoal IloNearestAdditionGenerate(IloEnv env)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloInt size)
public IloGoal IloNearestAdditionGenerate(IloEnv, IloDispatcherGoalFactory
goalFactory)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloInt size,
IloDispatcherGoalFactory goalFactory)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloSearchLimit limit)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloInt size, IloSearchLimit
limit)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloDispatcherGoalFactory
goalFactory, IloSearchLimit limit)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloInt size,
IloDispatcherGoalFactory goalFactory, IloSearchLimit limit)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IloInt size)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver,
IloDispatcherGoalFactory goalFactory)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IloInt size,
IloDispatcherGoalFactory goalFactory)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IlcSearchLimit limit)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IloInt size,
IlcSearchLimit limit)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver,
IloDispatcherGoalFactory goalFactory, IlcSearchLimit limit)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IloInt size,
IloDispatcherGoalFactory goalFactory, IlcSearchLimit limit)
public IloGoal IloNearestAdditionGenerate(IloEnv, IloNearestAdditionBehavior,
IloDispatcherFSParameters param)
public IlcGoal IloNearestAdditionGenerate(IloSolver, IloNearestAdditionBehavior,
IloDispatcherFSParameters param)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloNearestAdditionBehavior
mode)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloInt size,
IloNearestAdditionBehavior mode)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloDispatcherGoalFactory
goalFactory, IloNearestAdditionBehavior mode)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloInt size,
IloDispatcherGoalFactory goalFactory, IloNearestAdditionBehavior mode)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloSearchLimit limit,
IloNearestAdditionBehavior mode)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloInt size, IloSearchLimit
limit, IloNearestAdditionBehavior mode)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloDispatcherGoalFactory
goalFactory, IloSearchLimit limit, IloNearestAdditionBehavior mode)
public IloGoal IloNearestAdditionGenerate(IloEnv env, IloInt size,
IloDispatcherGoalFactory goalFactory, IloSearchLimit limit,
IloNearestAdditionBehavior mode)
public IloGoal IloNearestAdditionGenerate(IloEnv, IloDispatcherFSParameters param)
public IlcGoal IloNearestAdditionGenerate(IloSolver, IloDispatcherFSParameters
param)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver,
IloNearestAdditionBehavior mode)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IloInt size,
IloNearestAdditionBehavior mode)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver,
IloDispatcherGoalFactory goalFactory, IloNearestAdditionBehavior mode)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IloInt size,
IloDispatcherGoalFactory goalFactory, IloNearestAdditionBehavior mode)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IlcSearchLimit limit,
IloNearestAdditionBehavior mode)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IloInt size,
IlcSearchLimit limit, IloNearestAdditionBehavior mode)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver,
IloDispatcherGoalFactory goalFactory, IlcSearchLimit limit,
IloNearestAdditionBehavior mode)
public IlcGoal IloNearestAdditionGenerate(IloSolver solver, IloInt size,
IloDispatcherGoalFactory goalFactory, IlcSearchLimit limit,
IloNearestAdditionBehavior mode)
```

Definition file: ildispat/ilogoals.h
Include file: <ildispat/ilodispatcher.h>

Dispatcher provides various heuristic algorithms to build a first solution for a routing plan, among them, the *nearest addition heuristic*. The *nearest addition heuristic* builds first solutions for both single depot and multiple depot vehicle routing problems.

Depending on whether `env` or `solver` is specified, these functions return an `IloGoal` or `IlcGoal` that builds a solution to the problem at hand using the nearest addition heuristic. These functions also allow you to specify a secondary goal to be satisfied using the `IloDispatcherGoalFactory` parameter.

The argument `size` forces the *nearest addition heuristic* to consider only the `n` closest visits (in terms of cost). While this may reduce the quality of the solution generated, reducing the number of neighbors can dramatically reduce the amount of memory needed to store the array of nearest additions. If `size` is not specified, all visits are considered.

In the heuristic `IloNearestAdditionGenerate`, the goal `IloGenerateRoute` is used each time Dispatcher extends a route to ensure that the proposed extension is feasible. When the extension is not feasible, the search tree created by `IloGenerateRoute` is often too large to completely explore.

The parameter `limit`, if specified, places a limit on the search carried out in the route generation goal. This limit does not apply to any additional subgoals that you specify. If you also wish search to be limited in any additional goal, then a search limit should be applied to it individually.

The parameter `mode`, if specified, indicates the behavioral mode of the heuristic `IloNearestAdditionGenerate` during execution. The behavioral mode determines how the heuristic extends routes. The mode `IloNearestAdditionForward` extends the route forward from the first visit. The mode `IloNearestAdditionBackward` extends the route backward from the last visit. The mode `IloNearestAdditionBoth` extends the route simultaneously in both directions. In the case of `IloNearestAdditionBoth`, the nearest visit to either the start or the end of the route is connected to that portion of the route. In the case of a tie, the route is extended forward. If `mode` is unspecified, `IloNearestAdditionForward` is assumed.

The parameter `IloDispatcherFSParameters` can be used to parameterize `IloNearestAdditionGenerate` in a variety of ways. Owing to the various different parameters that can be passed, passing these directly in the constructor of the first solution method can be cumbersome. The handle class `IloDispatcherFSParameters` encapsulates the different types of parameters that can be passed. For more information, see `IloDispatcherFSParameters`.

Nearest Addition Heuristic

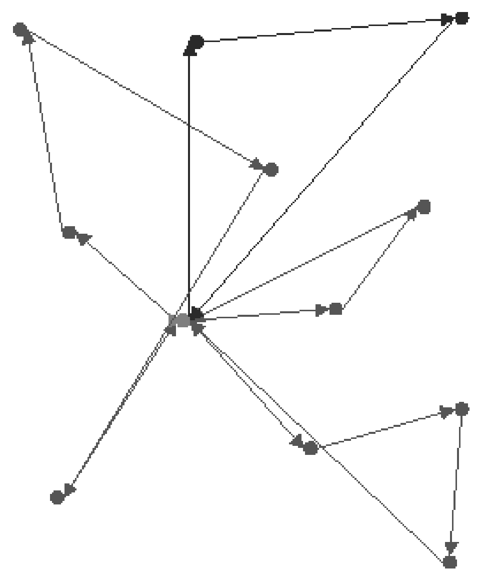
1. For all vehicles *v* starting at visit *S*, and for all visits *i* find the couple *(V, I)* minimizing *cost(S, i, v)* such that *(S, i)* is a feasible partial route for *v*. If none is found, go to step 5.
2. Start with a partial route consisting of the *(S, I)* for vehicle *V*.
3. Find the visit *J* that minimizes *cost(i, j, v)* and that can extend the current partial route of *V*. If it is not possible to find such a visit without violating constraints, close the current partial route of *V* and go to step 1.
4. Add *J* to the end of the partial route of *V* and let *I = J*. Go to step 3.
5. If all visits are scheduled, the goal succeeds. If there are unscheduled visits, they are constrained to be unperformed.

Note

This explanation describes the heuristic `IloNearestAdditionGenerate` operating in the behavioral mode `IloNearestAdditionForward`.

Example

The following figure provides an example of the use of `IloNearestAdditionGenerate` on a VRP with capacity constraints. The coordinates of the depot are (30,40). The capacity of the truck is 50 and there are 11 visits to perform. The tuples (x,y,quantity) of the visits are (17,63,19), (31,62,23), (52,64,16), (21,47,15), (37,52,7), (49,49,30), (42,41,19), (20,26,9), (40,30,21), (52,33,11) and (51,21,5).



`IloNearestAdditionGenerate` calculates 4 routes with total quantities collected of 50, 39, 49 and 37. If the cost of the routing plan is equal to the Euclidean distance traveled, the cost in this example is 285.23.

See Also: IloAllUnperformedGenerate, IloDispatcherGenerate, IloInsertionGenerate, IloNearestDepotGenerate, IloSavingsGenerate, IloSweepGenerate

Global function IloMakePerformedPair

```
public IloNHood IloMakePerformedPair(IloEnv env, IloDispatcherNHoodParameters
params)
public IloNHood IloMakePerformedPair(IloEnv env)
```

Definition file: ildispat/perfnhood.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by making an unperformed visit pair performed. For each vehicle route in which the pair is inserted, every combination of positions for the two visits will be tried. This behavior is different from the one of `IloMakePerformed` which will only try to move the pair immediately after a performed pair of visits.

The optional parameter `params` can be used to customize the behavior of the neighborhood. In particular, the vehicle array specified using `setVehicles` on the `IloDispatcherNHoodParameters` class will make the neighborhood insert only the visits on the routes of these vehicles.

For more information, see the concept Neighborhoods.

See Also: IloCross, IloExchange, IloFPRelocate, IloMakePerformed, IloMakeUnperformed, IloOrOpt, IloRelocate, IloSwapPerform, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloSwapPerform

```
public IloNHood IloSwapPerform(IloEnv env, IloDispatcherNHoodParameters params)
public IloNHood IloSwapPerform(IloEnv env)
```

Definition file: ildispat/perfnhood.h
Include file: <ildispat/ilodispatcher.h>

This function returns a neighborhood that modifies a solution by exchanging a performed visit with an unperformed one.

The optional parameter `params` can be used to customize the behavior of the neighborhood. In particular, the vehicle array specified using `setVehicles` on the `IloDispatcherNHoodParameters` class will make the neighborhood unperform visits only if they belong to the routes of these vehicles.

For more information, see the concept Neighborhoods.

See Also: IloCross, IloExchange, IloFPRelocate, IloMakePerformed, IloMakePerformedPair, IloMakeUnperformed, IloOrOpt, IloRelocate, IloTwoOpt, IloVisitAlternativeSwap, IloDispatcherNHoodParameters

Global function IloDistanceThresholdArcPredicate

```
public IloArcPredicate IloDistanceThresholdArcPredicate(IloEnv env, IloDimension2
dim, IloNum threshold)
```

Definition file: ildispat/iloarcpredicate.h
Include file: <ildispat/ilodispatcher.h>

This function returns an arc predicate which is satisfied if the distance between the two visits of an arc is less than `threshold`. The distance is computed using the distance object corresponding to the extrinsic dimension `dim`. An arc is created when two visits are consecutive on a vehicle.

See Also: IloArcPredicate

Global function IloDistanceThresholdArcPredicate

```
public IloArcPredicate IloDistanceThresholdArcPredicate(IloEnv env, IloDistance
distance, IloNum threshold)
```

Definition file: ildispat/iloarcpredicate.h
Include file: <ildispat/ilodispatcher.h>

This function returns an arc predicate which is satisfied if the distance between the two visits of an arc is less than `threshold`. The distance is computed using `distance`. An arc is created when two visits are consecutive on a vehicle.

See Also: IloArcPredicate

Global function IloRejectNeighbor

```
public void IloRejectNeighbor(IloSolver solver, IloNHood nhood,
IloNeighborIdentifier nid)
public void IloRejectNeighbor(IloSolver solver, IloNHood nhood,
IlcNeighborIdentifier nid)
```

Definition file: ildispat/lsearch.h
Include file: <ildispat/ilodispatcher.h>

This function notifies a neighborhood `nhood` that the neighbor corresponding to `nid` has been rejected.

This function can be used to reject neighbors when a fail occurs in a subgoal called after the deltas from the neighborhood are applied to the current solution.

Implementation

Here is an implementation of such a usage:

```
ILCGOAL2(RejectNHood,
IloNHood, nh,
IlcNeighborIdentifier, nid) {
IloSolver solver = getSolver();
IloRejectNeighbor(solver, nh, nid);
return IloGoalFail(solver);
}

ILCGOAL5(SingleMove,
IloSolution, solution,
IloNHood, nh,
IloMetaHeuristic, mh,
IlcSearchSelector, sel,
IlcGoal, subgoal) {
IloSolver solver = getSolver();
IloSolutionDeltaCheck check;
if (mh.getImpl()) check = mh.getDeltaCheck();

IlcNeighborIdentifier nid(solver);
IlcGoal scan = IloScanNHood(solver,
nh,
nid,
solution,
check);
IlcGoal testGoal = IloTest(solver, mh, nid);
IlcGoal exploreNHood = IlcAnd(
IloStart(solver, mh, solution),
scan,
testGoal,
IlcOr(subgoal, RejectNHood(solver, nh, nid)),
testGoal
);

IlcGoal saveGoal = IlcAnd(IloNotify(solver, nh, nid),
IloNotify(solver, mh, nid),
IloStoreSolution(solver, solution)
);

return IlcAnd(IlcSelectSearch(exploreNHood, sel), saveGoal);
}
```

The predefined neighborhoods provided in Dispatcher take this information into account to reduce the actual number of neighbors to examine. This results in a potential speedup of the search.

For more information, see the concept Neighborhoods.

Typedef IloArcPredicate

Definition file: ildispat/iloarcpredicate.h
Include file: <ildispat/ilodispatcher.h>

```
IloPredicate< IloVisitPair > IloArcPredicate
```

This C++ type represents a predicate on a pair of visits. This typedef allows you to write less code when you heavily use Dispatcher predicate classes.

An arc is created when two visits are consecutive on a vehicle. Arc predicates can be set using Dispatcher neighborhood parameters. These predicates can limit the scope of neighborhoods, which usually improves the performance of the search and potentially the quality of the resulting solutions. Functions are available to return arc predicates based on distance thresholds and whether the two visits of an arc are located at the same node. Operators are available to return the conjunction, disjunction, and negation of arc predicates. For more information, see `IloPredicate`, `operator &&`, `operator ||`, `operator !=`, and `IloIfThenElse`, documented in the *IBM ILOG Solver Reference Manual*.

The member function `IloDispatcherNHoodParameters::setArcFocusPredicate` can be used to forbid neighborhoods from accepting new arcs which do not satisfy the predicate. The member function `IloDispatcherNHoodParameters::setArcKeeperPredicate` can be used to forbid neighborhoods from removing arcs which satisfy the predicate.

See Also: IloVisit, IloVisitPair, IloExplicitArcPredicate, IloDistanceThresholdArcPredicate, IloSameNodeArcPredicate, IloDispatcherNHoodParameters::setArcFocusPredicate, IloDispatcherNHoodParameters::setArcKeeperPredicate

Typedef IloDistanceFunction

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

```
IloNum(* IloDistanceFunction)(IloNode, IloNode, IloVehicle)
```

This C++ type represents a pointer to a function that takes three arguments, two nodes and a vehicle, and returns a numeric value which is the distance for traveling between the two specified nodes using the specified vehicle. This “distance” can used to represent a variety of other values, such as costs or times.

See Also: IloDimension, IloDimension1, IloDimension2, IloDistance

Typedef IloSimpleDistanceFunction

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>

```
IloNum(* IloSimpleDistanceFunction)(IloNode, IloNode)
```

This C++ type represents a pointer to a function that takes two nodes as arguments and returns a numeric value which is the distance for traveling between the two specified nodes. This “distance” can used to represent a variety of other values, such as costs or times.

See Also: IloDimension, IloDimension1, IloDimension2, IloDistance, IloEuclidean, IloGeographical, IloManhattan

Typedef IloSimpleVehicleToNumFunction

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

```
IloNum(* IloSimpleVehicleToNumFunction)(IloVehicle)
```

This C++ type represents a pointer to a function that takes a vehicle and returns a numeric value. Functions of this type can be used to create an instance of the `IloEvalVehicleToNumFunctionI` class.

Typedef IloSimpleVisitDistanceFunction

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>

```
IloNum(* IloSimpleVisitDistanceFunction)(IloVisit, IloVisit)
```

This C++ type represents a pointer to a function that takes two visits as arguments and returns a numeric value which is the distance for traveling between the two specified visits. This “distance” can be used to represent a variety of other values, such as costs or times.

See Also: IloDimension, IloDimension1, IloDimension2, IloVisitDistance

Typedef IloSimpleVisitToNumFunction

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

```
IloNum(* IloSimpleVisitToNumFunction)(IloVisit)
```

This C++ type represents a pointer to a function that takes a visit and returns a numeric value. Functions of this type can be used to create an instance of the `IloEvalVisitToNumFunctionI` class.

Typedef IloVehicleArray

Definition file: ildispat/ilovehicle.h
Include file: <ildispat/ilodispatcher.h>

```
IloSimpleArray< IloVehicle > IloVehicleArray
```

This C++ type represents an array of instances of `IloVehicle`.

See Also: IloVehicle

Typedef IloVehicleEquivFunction

Definition file: ildispat/ilovehicleequiv.h
Include file: <ildispat/ilodispatcher.h>

```
IloBool(* IloVehicleEquivFunction)(IloVehicle, IloVehicle)
```

This C++ type represents a pointer to a function that takes two arguments and returns a Boolean value. The two arguments are instances of the class `IloVehicle`. The function determines whether or not the two vehicles specified are equivalent with respect to a distance function to be specified.

See Also: IloAllVehiclesDifferent, IloAllVehiclesEquivalent, IloVehicleEquiv, IloVehicleEquivI

Typedef IloVehiclePairPredicate

Definition file: ildispat/ilovehiclepredicate.h
Include file: <ildispat/ilodispatcher.h>

```
IloPredicate< IloVehiclePair > IloVehiclePairPredicate
```

This C++ type represents a predicate on a pair of vehicles. This typedef allows you to write less code when you heavily use Dispatcher predicate classes.

Vehicle pair predicates can be set using Dispatcher neighborhood parameters. These predicates can limit the scope of neighborhoods, which usually improves the performance of the search and potentially the quality of the resulting solutions. Operators are available to return the conjunction, disjunction, and negation of vehicle pair predicates. For more information, see `IloPredicate`, `operator &&`, `operator ||`, `operator !=`, and `IloIfThenElse`, documented in the *IBM ILOG Solver Reference Manual*.

The member function `IloDispatcherNHoodParameters::setVehiclePairFocusPredicate` can be used to forbid neighborhoods from performing moves between pairs of vehicle which do not satisfy the predicate.

See Also: IloVehicle, IloVehiclePair, IloBoxVehiclePairPredicate, IloDispatcherNHoodParameters::setVehiclePairFocusPredicate

Typedef IloVisitArray

Definition file: ildispat/ilovisit.h
Include file: <ildispat/ilodispatcher.h>

```
IloSimpleArray< IloVisit > IloVisitArray
```

This C++ type represents an array of instances of `IloVisit`.

See Also: IloVisit

Typedef IloVisitDistanceFunction

Definition file: ildispat/ilovisitdist.h
Include file: <ildispat/ilodispatcher.h>

```
IloNum(* IloVisitDistanceFunction)(IloVisit, IloVisit, IloVehicle)
```

This C++ type represents a pointer to a function that takes three arguments, two visits and a vehicle, and returns a numeric value which is the distance for traveling between the two specified visits using the specified vehicle. This “distance” can be used to represent a variety of other values, such as costs or times.

See Also: IloDimension, IloDimension1, IloDimension2, IloVisitDistance

Typedef IloVisitVehicleCompatPredicate

Definition file: ildispat/ilocompat.h
Include file: <ildispat/ilodispatcher.h>

```
IloBool(* IloVisitVehicleCompatPredicate)(IloVisit, IloVehicle)
```

This C++ type represents a pointer to a function that takes as arguments a visit and a vehicle and returns a Boolean. The Boolean is true if the visit and vehicle are compatible with each other, false otherwise.

Variable IloEarthRadiusInKm

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>
This constant can be used with `IloGeographical` to compute actual distances in kilometers and is set to 6378.137 kilometers.

Variable IloEarthRadiusInMiles

Definition file: ildispat/ilodist.h
Include file: <ildispat/ilodispatcher.h>
This constant can be used with `IloGeographical` to compute actual distances in miles and is set to 3963.19 miles.