Route Finder: Efficiently Finding *k* Shortest Paths Using Constraint Programming

Michel P. Lefebvre[1], Jean-François Puget[2], Petr Vilím[3]

[1] IBM. 21 chemin de la Sauvegarde, 69130 Ecully, France
[2] IBM, 350 Avenue de Boulouris, 83700 Saint Raphael, France
[3] IBM, V Parku 2294/4, 148 00 Praha 4 - Chodov, Czech Republic
`mlefebvre@fr.ibm.com`, `j-f.puget@fr.ibm.com`, `petr_vilim@cz.ibm.com`

Abstract. In this paper, we describe a Constraint Programming (CP) route finding application for a container transportation company. Mathematically, this amounts to finding the k shortest paths in a directed graph. However the nature of the business constraints rule out known algorithms such as Dijkstra's. Indeed, one cannot unfold all constraints into a directed graph as the resulting graph would be too large. Given an origin and destination (two places), the problem is to decide which ships should be used (routes), and when and where the containers should be loaded from one ship to another (connections), while satisfying many business rules specified by the transportation company. The CP model described in this paper is quite simple, it doesn't use any specialized constraints, but it is surprisingly effective. Queries for the best route are answered in a matter of a second or fraction of a second, although the problem is very large: around 900 places, 2,300 routes, 22,000 connections and 4,200 business rules. The system gracefully handles 100,000 requests a day on a single server.

Keywords: Transportation, Shortest Path, Routing

1 Problem Description

One of the major container carrier companies has contacted IBM about a route finding (RF) application they needed to rewrite. The RF application is part of the system that supports the commercial operations of that company. Basically, the problem is about rapidly finding feasible ways to ship goods from one place to another place, worldwide. The system response must be fast enough to be used during a phone call where a company representative negotiates with a potential customer. In order to support the commercial negotiation the RF system must propose several alternative ways for shipping goods. The fastest way does not have to be necessarily the cheapest, the company needs a system able to propose several different paths so that the customer can make a choice. The RF system is also used by various IT applications as a subroutine. All in all, RF system must answer about 100,000 requests a day.

In the following we concentrate on transportation by container ships but the definitions can be extended to other transportation means such as trains or barges. Therefore, in data description, we will use generic terms such as place, connection and so on. This section gives a quick overview of the problem, terminology and notation.

Container ships usually operate on fairly regular schedules which are known in advance and do not change a lot. Typically, a ship follows a circular path called *route* (see Figure 1). A route doesn't have any particular start or end, containers can be loaded or unloaded at each stop. Transition times between places on the route are known in advance. A route is therefore a closed loop that visits a series of *places* (harbors usually) in a predefined order. A ship following a given route visits all places on the route according to their (route specific) sequence number. The same place can appear several times on a route with different sequence numbers.

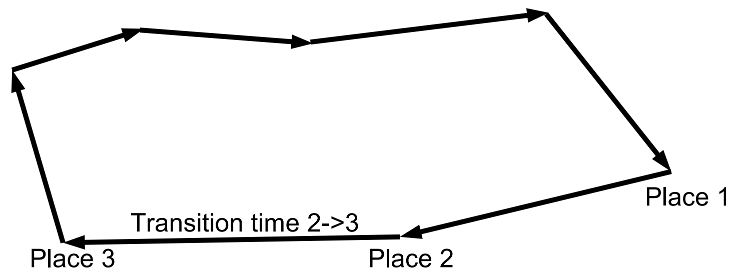

Fig. 1. Example of a route.

An operation of unloading a container from one ship and loading it on another ship is called *transhipment*. In order to make a transhipment, both ships must make a stop at the same port, but not necessarily at the same time: the container can be stored at the port for a short period of time. However, even if the two ships make a stop at the same port, it still may be impossible to make the transhipment: some ports are very large and transportation from one part of the port to another may be costly or not allowed at all. Therefore, possible connections are specified by a set of tuples of the form [place, from-route, to-route]. Note that connections are not symmetrical – from-route and to-route cannot be exchanged. See Figure 2.

The task is to find k shortest paths (in terms of duration) from Place Of Load (POL) to Place Of Discharge (POD), see Figure 3.

The path must fulfill the following conditions:

– Maximum number of transhipments is 5.
– All business rules are satisfied (will be explained later).

To simplify the problem, we solve it in two phases. In the first phase, we concentrate only on finding the best routes, without assigning the schedule. In another words, we ignore limitations such as:

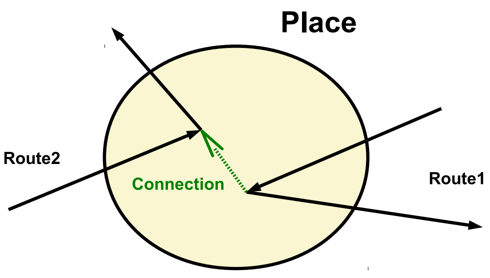

Fig. 2. Example of a connection at Place1 from Route1 to Route2.

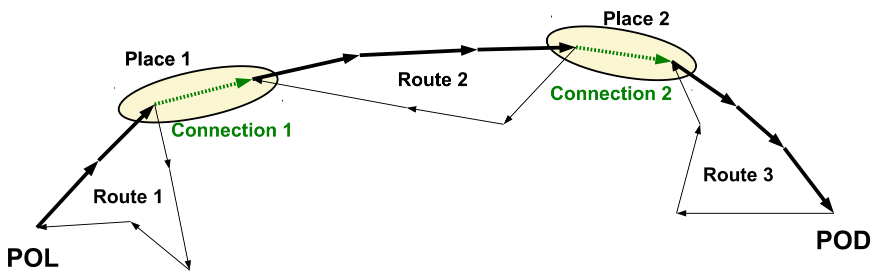

Fig. 3. Example of a path from POL to POD. The path starts by Route1, at Place1 it changes to Route2 using Connection1 etc. until it reaches POD.

- **Exact ship schedules:** We take into account the time it usually takes for a ship to travel from place A to place B, but we ignore the fact that the particular trip takes 1 more day (for example, due to planned repairs).
- **Exact transhipment times:** We precompute the usual time necessary for each allowed connection. Then we use these values instead of real connection times which depend on real ship schedules.
- **Ship capacity etc.:** We assume that the ship can always transport the cargo. For example, there are no capacity constraints or type constraints.

In the second phase (post processing) we use real ship schedules to compute the real length of the path. The second phase is pretty straightforward, it doesn't use CP, and therefore we do not describe it in this paper.

Note that post processing can change duration of routes found by CP, in extreme case some solutions found by CP can be even found infeasible during post processing. This is one of the reasons why it is necessary to find k shortest paths instead of only the best one during the first phase. There is no guarantee that (after the post processing) one of the k shortest paths is the optimal one, however we didn't see such a case in practice. Note also that the current solution developed by the customer also split the problem in this way and it was a requirement of the customer to keep it this way.

1.1 Business Rules

Aside of the path constraints, the solution must satisfy a set of approximately 4,200 business rules. These rules have the following form:

```
IF
  applicabilityPart and
  (IF1 or IF2 or ... or IFn)
THEN [NOT]
  (TH1 or TH2 or ... or THn)
```

Where

- `applicabilityPart` specifies when the rule is active in terms of POL and POD. For example, it is possible to specify that a rule is active only for paths from Europe to China. But it is also possible to specify a rule which is always active.
- `IF`*i* and `TH`*i* are if/then literals. They could be, for example, one of the following conditions:
  - Place P is on the path.
  - Route R is on the path.
  - Place P1 is directly followed by place P2.
  - Route R1 is directly followed by route R2.
- If `NOT` is present then the THEN part may be empty (no `TH`*i*). In this case all of the `IF`*i* conditions must be false in order to satisfy the rule.

Here are two examples of the business rules:

1. If POL is New York and POD is in France (`applicabilityPart`) and route R1 or route R2 is used (two `IF` literals) then place P must be on the path (one `TH` literal).
2. If POL is in Brazil and POD in Germany (`applicabilityPart`) then route R cannot be used (one `IF` literal for route R, `NOT` is present, no `TH` literal).

Note that the possible connections at a place may depend on the origin (POL) and destination (POD) of the request, the rules are not local. This is what makes the problem not solvable by classical methods as we shall see in the next section.

2 Why CP?

Shortest path problems are *very* easy to solve, and one could wonder why we did not consider using classical algorithms such as Dijkstra's. The issue is that a fundamental property required by Dijkstra and all dynamic programming approaches is that any sub path of an optimal path is also optimal. Indeed, there are business rules that make a perfectly optimal sub path not extensible into an optimal path.

For instance, while looking for a path from A to C, reaching the intermediate point B from A within 10 days does not mean we can ignore ways to reach B in more than 10 days. Indeed, the allowed path from B to C may depend on how (which route) B is reached. One way to solve this could be to construct a derived graph where the unconstrained shortest paths are the shortest constrained path in the original graph. For this one needs to duplicate place nodes so that business rules are replaced by possible arcs. Business rules depend on the origin, destination, place of transhipment, and also on incoming and outgoing routes to the place of transhipment. Therefore we need to create one node per origin/destination/place/route tuple. In our case this means about 10^{12} nodes, which is not manageable with the time and space constraints we have for finding routes. Refinement of this brute force approach is certainly possible, but we decided to keep the graph implicit and treat the business rules as constraints.

The shipping company had developed a system that was searching for constrained paths as outlined above. That system was poorly designed as it grew over time with ad hoc coding of business rules into the control flow. It was difficult to maintain as it was made of:

– several heterogeneous and interdependent modules (500,000 lines of hand made code, written in Forte)
– with additional technical features to compensate for the low performance (frequent request caching, graph reduction)
– and many similar and redundant concepts coexisting.

It was therefore decided to:

– keep the business-specific part of the system i.e., the routing rules and the authorized connections,
– to replace the complex procedural code by a declarative CP model, based on those constraints and solved by a generic solver (IBM ILOG CP Optimizer [5]),
– to isolate the static graph set-up, in order to avoid useless data reloading and graph set-up for each routing request.

2.1 Related Work

Problem of finding shortest paths using CP was already studied by several authors, see for example [11, 3, 8]. The recommended approach is to use a dedicated constraint for propagation.

In our case, the maximum number of transhipments is 5 what simplifies the problem a lot. Therefore we tried first to model the problem using only standard constraints available our CP solver. In the end, performance of this model is so good that it is not necessary to implement a dedicated constraint. Moreover, the customer already experienced how hard it is to maintain existing RF system. Therefore it was very appreciated that implementation of a new constraint is not necessary and CP could be used as a kind of black-box solver.

3 CP Model

3.1 Development

The RF application has been developed with IBM CPLEX Optimization Studio [1]. This product contains several components among which the modeling language OPL[1], and the Constraint Programming system CP Optimizer (CPO) [5]. At the beginning, we started the development in OPL. Thanks to the OPL we were able to connect to the customer database, read the data, and quickly experiment with different models until we found the best one.

In case of the routing application, every fraction of a second matters as the target is to solve thousands of requests per hour. Therefore once the model was stabilized, we converted it from OPL to the C++ API of CPO in order to eliminate overhead of model building (although the overhead was only a few percents in speed).

3.2 General Framework

CP is a great tool for solving optimization problems. However, it should be used to solve only the core part of the problem and leave remaining work to preprocessing and post processing. Therefore we use the following framework:

```
GlobalPreprocessing();
while (WaitForQuery()) do begin
  LocalPreprocessing();
  SolveCPModel();
  Postprocessing();
end;
```

Where:

- `GlobalPreprocessing` does preprocessing independent of POL and POD. For example, it computes usual transhipment times for all allowed connections. At the end of this phase, all necessary data are loaded into memory in order to avoid disc access during following phases.
- `LocalPreprocessing` does preprocessing dependent on POL and POD. For example, it filters applicable rules and adds StopPlace into the graph (will be described later).
- `PostProcessing` assigns schedules to found paths, resorts the solutions according to real duration or according to any other business criteria.

In the following, we will concentrate on the `SolveCPModel` part.

[1] The OPL language has significantly evolved since its original design by Pascal Van Hentenryck, Irv Lustig, Laurent Michel, and Jean-François Puget [4]. The current version documentation can be found at `http://publib.boulder.ibm.com/infocenter/cosinfoc/v12r2/index.jsp`

3.3 Decision Variables

As usual with real problems, there are many ways to represent them as a constrained optimization problem. For instance one could create one decision variable per place whose value would be the next place. One issue with this model is to deal with the places that won't be visited for a given request. One would need to both introduce dummy values for such nodes and extend constraints to ignore those dummy values. A seemingly better model would be to create an array of decision variables, one decision variable per visited place. For instance, the fourth variable would denote the fourth visited place. This model still requires dummy values as we do not know the number of places that the shortest path will visit. The dummy values would be used for the variables whose indices are larger than the index of the one valued with the destination.

A much better model leverages two facts. First of all, if no transhipment takes place at a place, then the container stays on the same ship before and after visiting that place. The only *real* decisions to be made are where transhipment occurs, that is, which connections are used. Therefore, a solution to the problem is a chain of routes, places and connections. For each of them we create a decision variable (see Figure 4):

- `route[i]`: i-th route on the path.
- `duration[i]`: duration of the transportation using `route[i]`.
- `place[i]`: i-th place on the path.
- `connex[i]`: connection used at `place[i]`.
- `trTime[i]`: usual time spent by `connex[i]`.

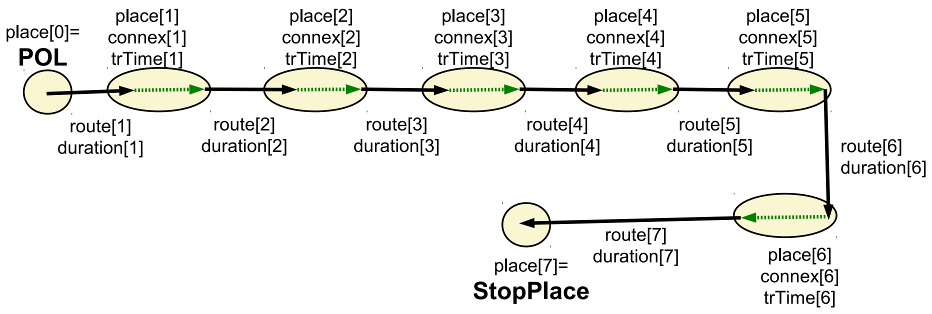

Fig. 4. Decision variables of the problem.

The second fact is that the number of transhipments is limited to 5, which limits the number of decision variables we need to consider.

The number of transhipments is not known in advance, it can range from 0 to 5. In order to deal with it we add a new place in the graph – StopPlace – which is reachable only from POD using ToStop connection and ToStop route, see Figure 5. Once StopPlace is reached the only way to continue the path is to use an artificial StopRoute back to StopPlace. This way, instead of looking for a path from POL to POD with maximum 5 transhipments, we are looking for a path from POL to StopPlace with exactly 6 transhipments (including artificial transhipments at StopPlace). That's why there are 6 connections in Figure 4 and `place[0]` is set to POL and `place[7]` is set to StopPlace.

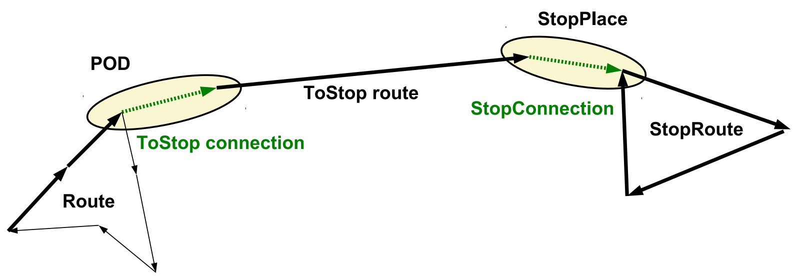

Fig. 5. Addition of StopPlace into the graph.

3.4 Constraints for Path

The variables described above are connected using the `allowedAssignment` constraint of CPO (also known as table constraint [6]). Using OPL syntax:

```
forall(i in 1..6)
  allowedAssignments(allowedConnections,
                     place[i], connex[i],
                     route[i], route[i+1], trTime[i]);
forall(i in 1..7)
  allowedAssignments(distances,
                     route[i], place[i-1], place[i], duration[i]);
```

Where `allowedConnections` is a set of tuples [place, connection, from-route, to-route, transhipment-time] and `distances` is a set of tuples [route, from-place, to-place, duration].

Of course, no place (aside from StopPlace) can appear on the path more than once. This could be modeled using the `count` expression in CPO (note that these expressions are automatically aggregated into a global cardinality constraint [10]):

```
forall(p in places)
  count(place, p) <= 1;
```

3.5 Business Rules

The constraints for applicable business rules are built on the fly. First we create integer expressions for all `IF`i and `TH`i literals: the expression has value 1 if the literal is true, 0 otherwise. Then, if `NOT` is not present in the rule then we add the following constraint:

$$\max_i \left\{\text{IfExpr}_i\right\} \leq \max_i \left\{\text{ThenExpr}_i\right\}$$

Otherwise (if `NOT` is present) we add the following constraint:

$$\max_i \left\{\text{IfExpr}_i\right\} \leq 1 - \max_i \left\{\text{ThenExpr}_i\right\}$$

If there is no `TH`i literal then we define:

$$\max_i \left\{\text{ThenExpr}_i\right\} = 1$$

In Section 1.1 we gave two examples of business rules. Assuming that they are applicable (considering the current POL and POD), they will generate the following constraints:

1. If POL is New York and POD is in France (`applicabilityPart`) and route R1 or route R2 is used (two `IF` literals) then place P must be on the path (one `TH` literal):
   ```
   max( count(route, R1), count(route, R2) ) <= count(place, P)
   ```
2. If POL is in Brazil and POD in Germany (`applicabilityPart`) then route R cannot be used (one `IF` literal for route R, `NOT` is present, no `TH` literal):
   ```
   count(route, R) <= 1 - 1
   ```

3.6 The Search

We started the development by looking for a single shortest path using default search of CPO. The results were very good, however for this particular model the default search was "too clever". Thanks to strong propagation, CPO was able to find solutions almost without any backtrack. However, the impact measurement used in its default search [9] slowed it down. Therefore we switched to simple DepthFirst search focusing on the place variables first as these variables have the biggest impact. This could be done using CPO high level search statements called *search phases*. The net result was a speedup of around 20%, even though there was more backtracking.

The customer was not interested in only one best path, the request was to find best k paths. Currently, there is no public API for this kind of output in CPO. Therefore we designed a workaround in the following way:

1. Start the search for all solutions, without supplying any objective function.
2. Iterate over solutions and remember k best solutions found so far. If there are more than 1,000 solutions then continue by 3, otherwise return best k solutions found.
3. Add constraint saying that we are interested only in solutions strictly better than the worst one from the k stored solutions. Forget all stored solutions with exception of the worst one. Restart the search and continue by 2.

Usually, one restart was enough. In rare cases it was necessary to restart the search twice. Note that without restarting we could end up by enumeration millions of solutions which would require minutes instead of a fraction of a second.

4 Added Value of CP

The CP based route finder code is quite small compared to the previous system, as it is about 2,000 lines of C++ code instead of about 50,000 lines of code for the search part of the previous system. Moreover, the declarative nature of the CP code enables consistency checking and review by humans whereas it is almost impossible to check the logic embedded in the old system. This alone has been seen as a tremendous progress by the shipping company team. A nice side effect of this is that the system is much easier to maintain and to evolve.

The CP based system prototype has been developed in less than two months in elapsed time, and four man months in total. This is much smaller than the time required to develop the original system.

Having a nice small piece of code is a desirable property, but what matters the most is the quality of the routes found, their adequateness with the business operations, and the speed at which these routes have been found, for any possible request.

The quality of the routes has been evaluated the following way. On a significant sample of 9,127 requests the two systems have been run. On 92.8% of the requests, the best route found by the old system is also found by the CP based system. Conversely, the best route found by the new system is also found by the old one in 73.2% of the requests. This shows that the quality of the new system routes is better on average. The fact that CP did not found the best stored solution in 100% of the cases can be explained by the nature of the procedural code of the old system. All business rules and constraints are coded as part of the procedural code that searches for routes, and it is probable that some mistakes have been made in these encodings. On the contrary, the declarative nature of the CP based system enables easy code review.

The adequateness of the routes found with the business needs has been evaluated the following way. A sample of routes for which the best route has been validated by a human operator has been provided to us. This is a by product of an attempt to speed up the old system. With the old system, each time a request is answered its result is cached to speed up the next query with the same (origin, destination) pair. The most frequent ones are then looked up by a human operator and stored as a routing instructions (RI). We were provided a sample of 839 routing instructions. For each of them we ran our route finder with the (origin, destination) pair and we compared with the best route stored with the one computed by CP. Our system found the stored answer in 815 of the cases, i.e. 97.1%. For the remaining few cases where CP did not find the stored solutions, CP found a better route, or some constraints were not met by the stored solution.

The speed of the new system has been evaluated against the speed of the old system using a 998 request sample. In a first experiment the requests were handled sequentially. The new system response time was about 1 second on average against 9.4 seconds for the old system. However the new system does not implement some post processing done by the old system (computing the exact schedule of the trip once a route is found). In order to make the comparison fair we take into account the time to do this post processing, and it is estimated to take 1 extra second. Therefore the new system would take about 2 seconds on average against 9.4 seconds for th old one. We then did a second experiment where requests are triggered concurrently, at the rate in which they arrive in reality (about one per second). Then the response time of the new system is unchanged, at about 1 second without post processing, whereas the old system response time goes up to 47 seconds on average. The response time of the new system is therefore about 24 times better than the old one.

The experiments above show that the new system finds better routes on average and that it finds them much faster than the old system. The new system is also easier to evolve given the declarative code used. One very interesting evolution was to use a small variant of the CP model to provide explanations using the original business rules. We introduced this as a way to debug our encoding of the business rules as follows.

Sometimes the problem didn't have any solution and we wanted to know why. Therefore, instead of applying the rules all the time we added 0/1 decision variable controlling whether the rule is turned on or not. This is straightforward to implement in CP Optimizer using conditional constraints. Then, by minimizing the sum of these additional variables we are able to identify which rule(s) make the problem infeasible (and that again in a matter of a second). This possibility became quickly very popular and due to the demand, we also added a possibility to choose which rules cannot be relaxed and to check why a particular solution is not possible.

Given the dramatic speedup and route quality improvements the shipping company has decided to integrate the CP based route finder application within its overall IT system and to deploy it. This is currently underway.

Note that the customer team did not had to understand the CP technology as we only used straightforward modeling constructs. They are able to read the C++ code that translates their business rules into CPO constraints.

Our use of CP for this project was focused on modeling. We did not write fancy search algorithms nor new constraint propagators. The resulting code is quite declarative and it can be seen as an instance of the *model and run* paradigm for CP advocated in [7]. This is one of the lessons learned from using CPO on this project. It is very important to have a code that business users can check and understand, which is key to build confidence into the system. The ability to provide explanations in terms of the original business rules was also key to build confidence.

The old system, beside its poor performance, was a gigantic black box that no one could understand at the shipping company. They had to rely on a costly third party consulting firm for any change to the system. The new one is much more maintainable and ready for evolution.

The CP approach developed for this customer is original as far as we know. There is little literature on finding constrained shortest path, with or without CP. One can cite [12] who solve a constrained shortest path problem in telecom networks. However, their problem had significant difference from the one we discussed in this paper. For instance, their graph is such that each arc (A, B) has a corresponding arc (B, A), whereas in our case arcs and connections do not necessarily have a symmetric counterpart. Another attempt at using CP for shortest path is given in [2]. This paper explores constrained variants of shortest path problems, but these variants do not cover constraints we generate from business rules.

As a summary, we have presented a quite effective use of CP for solving a complex business problem, namely how to route goods from one place to the other in the shortest possible time. A rather simple CP model requiring 2,000 lines of code yields much better performance than a system made of 500,000 lines of procedural code. The results of the application are so good that the shipping company has decided to deploy the CP based application. This is a remarkable endorsement since *all* the business of the shipping company depends on the quality of the proposed routes.

References

[1] IBM ILOG CPLEX Optimization Studio. URL `http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/`.

[2] Stefano Bistarelli and Ugo Montanari. Soft constraint logic programming and generalized shortest path problems. In *Journal of Heuristics*, volume 8:1, pages 25–41. Springer, 2002.

[3] Grégoire Dooms, Yves Deville, and Pierre Dupont. CP(graph): Introducing a graph computation domain in constraint programming. In *Principles and Practice of Constraint Programming - CP 2005*, pages 211–225, 2005.

[4] Pascal Van Hentenryck, with contributions by Irvin Lustig, Laurent Michel, and Jean-Franois Puget. The OPL optimization programming language. MIT Press, 1999.

[5] IBM. IBM ILOG CPLEX Optimization Studio documentation. volume 8:1. IBM, 2010. URL `http://publib.boulder.ibm.com/infocenter/cosinfoc/v12r2/index.jsp`.

[6] Olivier Lhomme and Jean-Charles Régin. A fast arc consistency algorithm for n-ary constraints. In *AAAI 2005*, 2005.

[7] Jean-François Puget. Constraint programming next challenge: Simplicity of use. In Mark Wallace, editor, *Principles and Practice of Constraint Programming, 10th International Conference, CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 5–8. Springer, 2004.

[8] Luis Quesada, Peter Van Roy, Yves Deville, and Raphaël Collet. Using dominators for solving constrained path problems. In *PADL*, pages 73–87, 2006.

[9] Philippe Refalo. Impact-based search strategies for constraint programming. In Mark Wallace, editor, *Principles and Practice of Constraint Programming, 10th International Conference, CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004.

[10] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on AI (AAAI/IAAI'96)*, volume 1, pages 209–215. AAAI Press / The MIT Press, 1996.

[11] Meinolf Sellmann. Cost-based filtering for shorter path constraints. In *Principles and Practice of Constraint Programming - CP 2003*, pages 694–708, 2003.

[12] Mats Petter Wallander, Radoslaw Szymanek, and Krzysztof Kuchcinski. CP-LP hybrid method for unique shortest path routing optimization. In *Proceedings of the International Network Optimization Conference*, 2007.