Flexible routing combing Constraint Programming, Large Neighbourhood Search, and Feature-based Insertion

Philip Kilby[1,2] **and Andrew Verden** [1]
[1] NICTA, [2] Australian National University
Philip.Kilby@nicta.com.au, Andrew.Verden@nicta.com.au

Abstract

Vehicle Routing Problems in the Operations Research and Artificial Intelligence literature often allow only standard constraints plus (optionally) one "side" constraint. However, in practice, every problem has a number of side constraints, some of which have never been seen in the literature. This paper describes an architecture for handling such arbitrary side constraints, based on Constraint Programming, Large Neighbourhood Search, and sophisticated insertion methods. This architecture allows many problems that arise in fleet logistics to be solved efficiently.

1 Introduction

In the classical Vehicle Routing Problem (VRP), a set of customers must be visited by a fleet of vehicles at minimum cost. Typically, a constraint on the capacity of each vehicle is observed. In the VRPTW, an additional constraint forcing the visit time to fall within a given time window is also enforced. For surveys on methods for these classical problems, see [Toth and Vigo, 2002; Marinakis and Migdalas, 2007].

However, in many problems in logistics, while the basic structure of delivery by vehicles to locations is the same, problems exhibit a wide variety of additional constraints. These constraints can be fairly general - for example a constraint on vehicle re-use that allows a vehicle to perform subsequent routes. Some constraints, however, are very specific to a particular workplace, and not likely to be see again (for example overtime allowed only if no more than 5 days worked in the last seven, and also 6 hours break between the previous overtime shift and the last shift where a B-double vehicle was driven).

In this paper we examine methods which allow a wide variety of side constraints to be handled simultaneously. The system uses a combination of Constraint Programming (CP) and Operations Research (OR) techniques to achieve an efficient and flexible solution technique.

We begin with a description of the types of problem we wish to solve. We then describe an architecture that can handle these types of problem efficiently. We then suggest a solution technique based on Large Neighbourhood Search ([Shaw, 1997]) and insertion methods [Solomon and Desrosiers, 1988]. Insertion methods build a solution by inserting one visit at a time into an emerging route set. Although some insertion methods can give good results over a wide variety of problems, we show how bespoke insert methods offer a modular way of tailoring the solution method to speed up solution of problems with particular side constraints.

2 Problem Description

We wish to supply goods to n customers. Each customer order i specifies the location for service (l_i), and the quantity of goods required (q_i). A fleet of m vehicles is available to perform deliveries. Vehicle k has capacity Q_k. We are given the cost of travel between each pair of customers (c_{ij}).

In the most basic form of the problem, we wish to find a set of routes, one for each vehicle, such that customer requests are satisfied at minimum total cost, subject to the constraint that the total quantity of deliveries assigned to each truck does not exceed the truck capacity.

In this paper we use the term *route* to mean the sequence of customer *visits* performed by a truck.

This sort of formulation can represent problems where goods are delivered, or all goods are picked up, or problems where a service is provided.

Some additional problem variants and constraints have been studied in the Operations Research and Artificial Intelligence literature

- Time Windows, where the allowed times for the visit (earliest start time, latest start time) are specified (See [Bräysy and Gendreau, 2005] for a recent survey).
- Request/Request compatibility constraints specify that some pairs of visits cannot be assigned to the same route, or that they *must* be assigned to the same route. For example, it may not be permitted to carry particular chemical together on the same vehicle.
- Request/Vehicle compatibility constraints are similar, but they specify that some requests *must* be carried by a particular vehicle or *must not* be carried by a particular vehicle. [Nakari *et al.*, 2007] discusses problems with compatibility constraints.
- Pickup and Deliver Problem (PDP) constraints. Problems where goods are picked up at one location, and delivered to another. The PDP constraints ensure the

goods are picked up before they are delivered, and that the same route picks up and delivers. See [Savelsbergh and Sol, 1995] for a survey of PDP type problem and solution methods.

These basic constraints together also define the *General Vehicle Routing Problem* [Goel and Gruhn, 2008]. However, we wish to be able to solve problems with even more general constraints. Constraints such as the following have been investigated individually in the literature

- "Blood bank" constraint – a pickup and deliver where the pickup can occur any time in the morning, but delivery must be within 20 minutes of pickup. Because the delivery time window cannot be specified a-priori, this type of constraint cannot be expressed by the usual time window constraint.
- Multi-delivery (split-delivery) routing. Here, customer quantities may exceed the size of the largest truck, and so must be visited more than once. This type of problem is common in line-haul routing, where deliveries are made from a production facility to distribution centres (e.g. [Archetti *et al.*, 2006]).
- Loading dock constraint – Vehicle dispatch is limited by the number of docks available for loading. Alternatively, in PDP problems where there are multiple deliveries to a single customer, the number of deliveries that can be performed simultaneously is limited by the number of docks.
- Movable partition constraint. In this problem, multiple commodities are carried simultaneously. While the total capacity of the vehicle is fixed, each time it is loaded a decision can be made as to how much capacity is given to each commodity.
- Prize collecting problems, where visits have different values, and we wish to maximise the value of visits assigned less travel cost. This may mean that some visits are left unassigned ([Feillet *et al.*, 2005] discusses the case where there is a single vehicle).
- Route rendezvous constraints – that ensure different routes rendezvous to transfer goods. The time of the rendezvous may depend on the duration of individual routes.

These constraints are usually examined individually – that is, there is the usual vehicle routing problem plus one set of extra constraints that are known *a-priori*.

Our aim is to solve problems with a variety of these extra constraints applying simultaneously, while at the same time to avoid making the solution method dependent on which constraints are actually present.

The NICTA Intelligent Fleet Logistics project has developed a system called *Indigo* that is able to handle a variety of problems in logistics. Its architecture and methods are described in the following sections.

3 Architecture

Constraint Programming is an obvious technique to express the side constraints seen in practical Vehicle Routing problems, and to support solving instances of these more general problems.

However, constraint programming can introduce an expensive overhead to handle some constraints. For instance, if a capacity constraint is tied to the variable which indicates the route to which the visit is assigned, then each time that variable is altered, or any time the load changes on any route which is within the domain of that variable, then the capacity constraint will be re-checked. This will ensure correct operation, but can lead to much redundant checking.

For this reason, the *Indigo* system has two "classes" of constraint.

The first class of constraints (called "native" in this context) include all the constraints of the General Vehicle Routing Problem [Goel and Gruhn, 2008]:

- Capacity constraints (across multiple commodities but with fixed capacity for each commodity)
- Usage constraints, that limit the total usage of resources such as time and distance accumulated during each run.
- Time window constraints, specifying earliest and latest start time.
- Pickup-and-Deliver constraints, enforcing precedence and same-route constraints between a pair of visits.
- Request/Request and Request/Vehicle compatibility constraints

Native constraints are handled very efficiently by the system with minimal interaction with the Constraint Programming system. Any decision made by the solver is guaranteed to observe all of the native constraints.

However, additional side constraints can be specified and handled using the CP system. In the long-term, we wish to be able to use the Zinc language [Marriott *et al.*, 2008] to specify these constraints, and then solve the problem within the G12 system [Wallace and the G12 Team, 2009]. In the short term, however, propagators for each side-constraint are hand coded, using a simple bespoke constraint programming system.

Even with some hand-coding still required, the advantage of using a CP paradigm is clear. The propagator for each constraint is a self-contained piece of code that is only "known" to the CP system. The alternative in traditional OR systems would require the main body of VRP solving code to be modified for each new constraint, which makes maintenance difficult, and unexpected interactions almost inevitable.

Special, separate modules also convert different variants of the problem into standard form. For multi-delivery routing, for instance, a separate module breaks each order quantity into smaller pieces that can be assigned efficiently to trucks of various sizes.

The advantage of approach handling *native* constraints internally was demonstrated in [Kilby *et al.*, 2010]. It was shown that handling these constraints internally, rather than as constraints in the CP system, decreases the CPU time by a factor of about 2. It also slightly improves solution quality in some cases.

4 Interaction with the CP System

The *Indigo* system is integrated with a CP system. The CP system has a number of variables for each visit and route, including: A *successor variable* indicating which visit should follow the given visit. A *predecessor variable* indicating which visit should precede the given visit. A *route variable* indicating which route the visit is assigned to. If time is used, then *arrive time* and *service start time* variables are used. For each assigned visit, and for each commodity, there are variables specifying the cumulative load.

Constraints can be posted in the CP system to limit the values these variables can take. For instance, in the case of the blood donor constraint, as soon as the pickup visit is assigned, then the delivery visit will have its service start time variable constrained.

In operation, the *Indigo* system acts as a variable/value choice heuristic for the underlying CP system. *Indigo* maintains an internal representation of an emerging route set, including the list of partially built routes, plus the list of yet-to-be assigned visits.

Each time *Indigo* is called, it chooses a visit to insert, and a position in which to insert it. It can then propagate the effects of this choice to the CP system.

As discussed in [Kilby and Shaw, 2006], chronological backtracking in CP imposes limits on the amount of information that can be propagated to the CP system. For instance if the *Indigo* system decides to place visit 10 after visit 2, we cannot bind the *successor* variable of visit 2 just yet. If we were to make the assignment *succ[2] = 10*, then we would not be able to insert any other visit after 2 for the rest of the execution of the procedure. So instead, we make all propagations that can be inferred from the assignment. For instance, we can remove 10 from the successor variable of every visit except 2. We can also update the route variable for visit 10. A number of other propagations are possible. For example, let us say that visit v will follow visit p in route r

- All other assigned visits (except p) can be removed from the predecessor variable of v. Similarly the successor variable of v can be updated.
- If we assume the triangle inequality for time ($\forall a, b, c, t_{ac} \leq t_{ab} + t_{bc}$) then we know that we cannot arrive at v any earlier than we currently do. We can therefore update the bound on the arrival time to be at most the current arrival time.
- The latest arrival time cannot be any later than the current value (again assuming the triangle inequality). Hence we can also update the upper bound on the arrive time to be the current latest feasible arrival time.
- In pickup-up only problems, the load on any commodity cannot be less than the current value. We may therefor update the appropriate bound on the load variable.

If, during execution of the propagations following an assignment, a failure occurs (i.e. CP has identified an inconsistency) then the internal data structure within *Indigo* must be updated as part of the backtracking of the CP system. The CP system will ensure that the same assignment is not attempted again in the future.

When the *Indigo* system is subsequently called, changes to the successor, predecessor or time variables must be noted, and the next choices must be consistent with these values.

5 Solution method

Like many VRP solution methods, *Indigo* relies on local search methods to improve an initial solution. Many local search techniques for routing problems have been developed (for example 2-opt, 3-opt, Or-opt). However, these methods that move directly from one solution to another do not make use of the full power of CP.

Again, local search methods are limited by the chronological backtracking restrictions imposed by the CP architecture. Hence, methods that build up a solution one piece at a time, using CP search procedures, are preferable to local search type methods such as those above that move directly from solution to solution.

Large Neighbourhood Search (LNS) [Shaw, 1997] is a local search procedure where part of a solution is destroyed, and a then a new solution created by finding new values for the freed variables. This method uses exactly the sort of incremental solution building method that can exploit propagation in CP to guide the solution towards good, feasible solutions.

The *Indigo* system draws on the work of Ropke and Pisinger [Ropke and Pisinger, 2006] (R&Phere). Like that work, it uses insertion methods to create an initial solution, and then again to repair the solution in each iteration of LNS. The LNS algorithm can be given as follows:

1 Create initial solution S

2 Choose a "destroy" method d

3 Create S' by removing customers from S according to method d

4 Choose an insert method i

5 Create solution S'' from S' by inserting customers according to method i

6 If the acceptance method accepts solution S''

7 Replace S with S''

8 If iterations remain, return to line 2

This method is characterised by

- The destroy methods available at line 2
- The insert methods available at line 4
- The acceptance methods available at line 6
- The number of iterations available at line 8.

In this paper, we look only at enhancements to the insert methods available at line 4 that allow for some of the more general instances to be solved effectively. These methods are described in the next section.

6 Insert methods

Insertion methods proceed by repeating two stages:

First, amongst all unassigned visits, the best position to insert each is selected. Then, the visit which is to be inserted is chosen. The visit is then inserted in its best position. Solomon

[Solomon, 1987] seems to be the first to suggest this two-score system.

The best insert position for each still-unassigned visit is then updated. The method can then iterate until all visits have been assigned a position.

In much previous work, only a limited number of features are considered when making these two choices. In some work, only *minimum cost insertion* is considered – i.e. visits are always inserted in the position which gives rise to the smallest increase in cost, and the visit with the smallest increase is inserted first.

R&Pshow that using a combination of insertion methods gives better results than a single method, as it allows different methods to be used at different times. Running on benchmark problems with limited constraints (PDP, time window and capacity constraints), R&Pidentified a set of insert methods that gave good performance.

We wish to extend this idea, and use a variety of insertion criteria when making these choices. We will show below how this can be advantageous in real-world problems.

We will describe several criteria, or "features" which can be used in either choosing where to insert a visit, or choosing the visit to insert. Each criteria is described below. The degree to which a particular feature is present is rated on a score of 0 to 1, with 0 meaning "not present", and 1, "present". Along with each feature, the "base" value which is used to normalise the value (as described in Section 6.1) is also given. If *reversed* is specified, then (1 - val) is returned, rather than val. Two types of normalisation are also possible, as discussed in Section 6.2

The following symbols are used in the description below. The visit v is to be inserted between p and s on vehicle k. The cost of insertion is $c' = c(p, v) + c(v, s) - c(p, s)$.

Route domain Favour visits with few feasible routes. Val is number of routes v can be feasibly inserted into. Base is total number of routes. Reversed.

Num ins pos Favour visits with few feasible insert positions. Val is the number of feasible insert positions. Base is number of assigned visits. Reversed

Distance to depot Favour visits far from a depot. Val is distance to the closest route start or end. Base is max dist to route start or end.

Value For use in prize-collecting problems, favours inserting high-value visits first. Value is prize-value of the visit. Base is max prize-value over all visits.

Load Favour largest load first. Value is load. Base is max vehicle capacity

Nearest neighbour Encourages v to be inserted near its neighbours. Val is min $(c(p, v), c(v, s)$. Base is distance to v's 10^{th}-nearest neighbour. Reversed. Normalised with method 2.

Min insert cost Cheapest insert first. Value is c'. Base is twice the average insert cost. Ave insert cost is (Total cost of inserted visits) / (number of inserted visits). Reversed. Normalised by method 2.

Max insert cost Reverse of above. Calculated same way, but not reversed.

Regret, 3-Regret, 4-Regret See below. Base is same as *Minimum insert cost*. Normalise by method 2.

Rand Randomise slightly. Val is a uniform-random number in [0,1).

Time Window width Encourages smallest time window to be inserted first. Val is width of v's time window. Base is max of time window widths. Reversed.

Time Window end Encourages visit with latest time window to be inserted first. Val is the end time of the last time window. Base is max time window end.

Wait time Encourages vehicles not to arrive at a location before the start time window (as the vehicle must then wait for the time window to open). Val is the time the vehicle must wait at v before service starts. Base is (Last time window) / 10. Reversed.

Pickup Late, Deliver early Encourages vehicle to do deliveries at a location before doing pickups.

Lost slack Encourages spare time to be preserved. Val is how much "spare time" (difference between arrival time and time window end) is lost at s. Base is max time window width. Reversed. Normalised using method 2

Fill vehicle Used when problem has a bin-packing flavour, and favours inserts that fill the vehicle. Value is spare capacity after insert. Base is max capacity. Reversed.

Balance routes Encourages routes to have similar length, as measured by difference between shortest and longest route. Penalise adding to longest route, and reward adding to shortest.

6.1 Base values

Base values are used to normalise the feature values into the range [0,1] by diving *val* by *base*. Since we wish the values to be comparable, we must be careful in the base value chosen. For example, for nearest neighbour, the "safe" base is the length of the longest arc in the problem. However, this is likely to be very large, and not ever used in a solution. We therefor use a base value which is the distance to the 10th-nearest neighbour, as the neighbour of most visits is likely to come from this set.

6.2 Normalisation

The basic method of normalisation is to simply divide by the base. This is done whenever the base is a guaranteed maximum for the feature value (e.g. maximum time window width as base for the time window width feature)

However, for reasons outlined above, some base values are "optimistic" or heuristic values, and can be exceeded. Since we still wish to rank values that are greater than the base value, an alternative, non-linear normalisation is used. To normalise a value v with a base b, the normalised value is calculated as follows

$$\text{tmp} = \max(v/b, 0); \text{return tmp}/(0.5 + \text{tmp});$$

This normalisation always falls in the range [0,1]. The values 0 to 1 map to normalised values 0 to 0.6667, with 0.5 mapping to 0.5. The value of 0.6667 makes values normalised using this method approximately comparable to values normalised with the first method.

6.3 Regret

Regret is based on the difference between the best and next-best insert positions for a visit. If there is a big difference (a large regret), then if the visit does not get its favoured position, the effect on the objective is high. The method therefore chooses the visit with maximum regret to insert first.

More formally, if the minimum cost to insert visit i in route k is c_{ik}, and the permutation $o(k)$ permutes the routes into increasing order; i.e. $c_{i,o(1)} \leq c_{i,o(2)} \leq ... \leq c_{i,o(m)}$. Then regret($i$) is

$$c_{i,o(2)} - c_{i,o(1)}$$

3-Regret allows slightly more look-ahead, taking the first three positions into account: Then 3regret(i) is

$$((c_{i,o(2)} - c_{i,o(1)}) + (c_{i,o(3)} - c_{i,o(1)}))/2$$

4-regret is defined analogously, over the cost of the four cheapest routes.

6.4 Implementation

All of these methods (except the regret methods) can be calculated using just the visit to be inserted and the predecessor in the route. In the *Indigo* system, features are defined using a base class. New features can be incorporated easily by specialising the base class to calculate the required value.

7 Use of features

Features are combined using weights. Two separate weight sets are used – weight set 1 is used to decide which visit to insert; weight set 2 is used to decide where to insert it. Each score is simply the scalar product of the weights and the feature values.

For example, selecting the visit to insert using 3regret with a small amount of randomness; and position to insert using min-insert-cost, the following could be used
Feature set 1: { 3regret, rand }. Weight set 1: { 0.95, 0.05 }.
Feature set 2: { min-insert-cost }. Weight set 2: { 1.0 }.

For efficiency, only those features with a non-zero weight need to be evaluated. Weight sets can be general-purpose, or specific for a a portfolio of instances.

The new insert features allow flexibility in non-standard problems. For instance, in a multi-delivery pickup-and-delivery problem there may be multiple pick-ups and deliveries at a single location. While this type of problem is seen relatively often in practice (it is one way of modelling a vehicle leaving and returning to the depot multiple times) it does not appear in benchmarks. There is nothing in a standard VRP heuristic which makes us deliver before we pick up at the same location. The "pickup-early, deliver-late" ensures this sequence. However, in standard benchmarks, there is no call to use such a feature. It is only in the more flexible routing that it is useful – but there it is indispensable.

8 Computational testing

In order to test the effectiveness of the architecture, the system was tested on some standard benchmarks. These do not exhibit the flexibility of the system, but indicate the effectiveness on standard problems.

The system was tested on the VRPTW benchmark problems of Solomon [Solomon and Desrosiers, 1988] with 100 customers, and the extended Solomon benchmarks of Gehring and Homberger [Gehring and Homberger, 1999] with between 200 and 1000 customers.

The experimental setup used "standard" parameters similar to those used in R&P

- Accept function is Simulated with a temperature gradient of 0.99975, and an initial probability chosen so there is a 50% chance of accepting an increase of 5%.
- Removal selection functions and Insertion functions as per R&P.
- Adaptive learning of which selection and which insertion method to use, using rewards similar to those used by R&P
- 30,000 iterations of LNS
- 50 customers removed for size 100 and 200 problems. 100 customers removed for larger problems.
- 5 runs of each problem, best solution reported

Because the current *Indigo* system does not have a feature to reduce the number of vehicles, the problems were modified so that only the number of vehicles in the best-known solution were available to the system. This makes the comparison a little less fair, as most systems initially try to reduce the number of vehicles. However, it is consistent with many real-world problems where the vehicle fleet is fixed *a-priori*.

Because of limited space to report we give just the basic results. We express performance as a ratio of the increase as a ratio of best-known solution. E.g. 1.02 means the results was 2% higher than the best-known solution.

All problems were solved with the best-known number of vehicles. We produced new, best-known solutions to 83 of the 300 benchmarks. Table 1 shows the results. *Size* gives the number of customers in the problems; *Best* gives the number of problems where a "new best" solution was found; *Mean* is the mean increase; *80%* gives the 80th percentile of increase (i,e, 80% of values were less than this value); and *Max* gives the maximum increase over best-known solution. *CPU* gives mean time per run in CPU seconds. The tests used one CPU of an 8-core 32 bit Intel Xeon running at 2GHz.

Source	Size	Best	Mean	80%	Max	CPU
Solomon	100	0	1.01	1.01	1.05	53
G & H	200	11	1.01	1.02	1.05	120
G & H	400	13	1.01	1.03	1.06	487
G & H	600	19	1.02	1.04	1.10	766
G & H	800	18	1.02	1.05	1.11	1108
G & H	1000	22	1.03	1.06	1.14	1450

Table 1: Results on benchmark problems

For the smaller problems (100-200 customers) these results are very good for relatively small CPU times. Since other systems often run for much longer than the maximum 30 minutes allowed this system, the results for larger problems are reasonable, although some work is required to ensure the system performs as well on the larger problems as it does on the small.

9 Future work

With a large number of features, the space of possible weights is very large. The task of finding effective feature weights can be very difficult. We are currently looking a two methods for choosing weights: *static* and *dynamic*. Static methods will calculate a weight set *a-priori*, using a portfolio of similar problems from a given user. Dynamic method we make use of the fact that in LNS we are essentially solving the same problem many thousands of times. We can dynamically adapt the weight set, and test the effectiveness of the new set in subsequent runs.

We are also looking at ways of increasing the search diversity in larger problems, to improve the performance on some of the larger problems reported in section 8.

10 Conclusions

We have described an architecture for solving a variety of logistics problems, including, for instance, line-haul problems that are not well suited to traditional VRP methods. This architecture has the advantage of handling many of the most common constraints very efficiently, while allowing additional side constraints to be specified and handled in a modular and flexible way by an underlying Constraint Programming system.

We have argued that sophisticated insertion methods make an ideal partner for Large Neighbourhood Search and Constraint Programming for solving real-world vehicle routing problems. We have given a method of calculating and combining a number of feature scores, that allows insertion methods to be tailored more easily to the characteristics of the problem at hand.

Acknowledgements

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[Archetti *et al.*, 2006] C. Archetti, M. G. Speranza, and A. Hertz. A tabu search algorithm for the split delivery vehicle routing problem. *Transportation Science*, 40(1):64–73, 2006.

[Bräysy and Gendreau, 2005] Olli Bräysy and Michel Gendreau. Vehicle routing problem with time windows, part I: Route construction and local search algorithms. *Transportation Science*, 39(1):104–118, 2005.

[Feillet *et al.*, 2005] Dominique Feillet, Pierre Dejax, and Michel Gendreau. Traveling salesman problems with profits. *Transportation Science*, 39(2):188, 2005.

[Gehring and Homberger, 1999] H. Gehring and J. Homberger. A parallel hybrid evolutionary metaheuristic for the vehicle routing problem with time windows. In K. Miettinen, M. Makela, and J. Toivanen, editors, *Proceeding of EUROGEN99 - Short Course on Evolutionary Algorithms in Engineering and Computer Science,*, pages 57–64. University of Jyväskylä, 1999.

[Goel and Gruhn, 2008] Asvin Goel and Volker Gruhn. A general vehicle routing problem. *European Journal Of Operational Research*, 191(3):650–660, 2008.

[Harvey and Ginsberg, 1995] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In Chris S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 607–615, Montréal, Québec, Canada, 1995. Morgan Kaufmann.

[Kilby and Shaw, 2006] Philip Kilby and Paul Shaw. Vehicle routing. In F. Rossi, P. Van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 23, pages 801–836. Elsevier, 2006.

[Kilby *et al.*, 2010] Philip Kilby, Andrew Verden, and Lanbo Zheng. The cost of flexible routing. In *Proceedings of the Triennial Symposium on Transportation Analysis (TRISTAN) 2010*, 2010. To appear.

[Marinakis and Migdalas, 2007] Yannis Marinakis and Athanasios Migdalas. Annotated bibliography in vehicle routing. *Operational Research*, 7(1):27–46, 2007.

[Marriott *et al.*, 2008] Kim Marriott, Nicholas Nethercote, Reza Rafeh, Peter J. Stuckey, María García de la Banda, and Mark Wallace. The design of the Zinc modelling language. *Constraints*, 13(3):229–267, September 2008.

[Nakari *et al.*, 2007] Pentti Nakari, Olli Bräysy, and Wout Dullaert. Communal transportation: Challenges for large-scale routing heuristics. Reports of the Department of Mathematical Information Technology Series B. Scientific Computing B6/2007, University of Jyväskylä, 2007.

[Ropke and Pisinger, 2006] Stefan Ropke and David Pisinger. An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, 40(4):455–472, 2006.

[Savelsbergh and Sol, 1995] M.W.P. Savelsbergh and M. Sol. The general pickup and delivery problem. *Transportation Science*, 29(1):17–39, 1995.

[Shaw, 1997] Paul Shaw. A new local search algorithm providing high quality solutions to vehicle routing problems. Working paper, University of Strathclyde, Glasgow, Scotland, 1997.

[Solomon and Desrosiers, 1988] Marius M. Solomon and Jacques Desrosiers. Time window constrained routing and scheduling problems. *Transportation Science*, 22(1):1–12, February 1988.

[Solomon, 1987] M. Solomon. Algorithms for the vehicle routing and scheduling problem with time window constraints. *Operations Research*, 35:254–265, 1987.

[Toth and Vigo, 2002] Paolo Toth and Daniele Vigo, editors. *The Vehicle Routing Problem*, volume 9 of *SIAM Monographs on Discrete Mathematics and Applications*. SIAM, Philadelphia, PA, 2002.

[Wallace and the G12 Team, 2009] Mark Wallace and the G12 Team. G12 – towards the separation of problem modelling and problem solving. In *Proc. CP-AI-OR'09*, volume 5547 of *Lecture Notes in Computer Science*, pages 8–10. Springer, 2009.