Downloaded from orbit.dtu.dk on: Dec 17, 2017

Technical University of Denmark

PALNS - A software framework for parallel large neighborhood search

Røpke, Stefan

Published in:
8th Metaheuristic International Conference CDROM

Publication date:
2009

Document Version
Early version, also known as pre-print

Link back to DTU Orbit

Citation (APA):
Røpke, S. (2009). PALNS - A software framework for parallel large neighborhood search. In 8th Metaheuristic International Conference CDROM

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Parallel large neighborhood search - a software framework

Stefan Ropke*

*Department of Transport, Technical University of Denmark
Bygningstorvet 115, 2800 Kgs. Lyngby, Denmark
sr@transport.dtu.dk

1 Introduction

This paper propose a simple, parallel, portable software framework for the metaheuristic named *large neighborhood search* (LNS). The aim is to provide a framework where the user has to set up a few data structures and implement a few functions and then the framework provides a metaheuristic where parallelization "comes for free". LNS was proposed in [14] and has been gaining popularity in the recent years, but is not as widely used e.g. tabu search, simulated annealing or genetic algorithms. We apply the parallel LNS heuristic to two different problems: *the traveling salesman problem with pickup and delivery* (TSPPD) and the *capacitated vehicle routing problem* (CVRP). We explain the TSPPD in Section 2 while we, because of the page-limitation, refer the reader to [17] for a description of the more well-known CVRP. Section 3 describes the large neighborhood search metaheuristic, Section 4 describes how the metaheuristic has been parallelized, 5 and 6 explains how the framework has been applied to the TSPPD and CVRP, respectively. Section 7 presents computational results.

2 The traveling salesman problem with pickup and delivery

In the TSPPD a salesman has to visit a set of vertices V. The set of vertices is split into three groups: A set of pickup vertices P, a set of delivery vertices D and the depot vertex. The salesman starts and his route in the depot vertex. The sets P and D has the same cardinality and we will write the sets as $P = \{1, \ldots, n\}$ and $D = \{n+1, \ldots, 2n\}$. A traveling cost c_{ij} is given between each pair of vertices (i, j), $i, j \in V$, $i \neq j$. The TSPPD ask for finding a tour, starting and ending at the depot and visiting all vertices such that vertex i is visited before vertex $n+i$ for all $i \in P$. The tour we are looking for is the one minimizing the sum of the costs of the arcs used in the tour. We say that the pairs $(i, n+i)$ form *request* for all $i \in P$ (goods are picked up at i and delivered at $n+i$). Notice that the problem is uncapacitated. It is perfectly fine to first visit all pickup vertices and afterwards all delivery vertices. The TSPPD turns out to be very difficult to solve to optimality. Instances with 51 vertices are still unsolved. The most recent exact algorithm for the TSPPD is described in [4]. Heuristics for the TSPPD have been proposed in [12].

MIC 2009

3 Large neighborhood search

The *large neighborhood search* (LNS) metaheuristic that forms the basis of the method described in this paper was proposed in [14]. In an LNS metaheuristic the neighborhood is defined implicitly by a *destroy* and a *repair* method. A destroy method destructs part of the current solution while a repair method rebuilds the destroyed solution. The destroy method typically contains an element of stochasticity such that different parts of the solution are destroyed in every invocation of the method. The neighborhood of a solution is then defined as the set of solutions that can be reached by first applying the destroy method and then the repair method. For the TSPPD a destroy method could remove a number of request pairs from the solution while the repair method could reinsert the requests while respecting the precedence constraints. The destroy method will typically destroy a large part of the solution (say remove 20 requests for the TSPPD). This together with the many ways of repairing the solution mean that the neighborhood of a single solution contains a large number of solutions. This explains the name of the heuristic. A concept closely related to LNS is that of *Very Large Scale Neighborhood* Search (VLSN) defined in [1]. While the LNS is a heuristic framework, VLSN is the family of heuristics that searches neighborhoods whose sizes grow exponentially as a function of the problem size, or neighborhoods that simply are too large to be searched explicitly in practice, according to [1]. The LNS is one example of a VLSN heuristic. In [13] it was suggested to use several kinds of removal and destroy methods in the LNS method and a method for selecting between the different destroy and repair methods based on their performance was proposed. This version of LNS was coined *adaptive large neighborhood search* (ALNS). The software framework described in this paper follows the ALNS principles.

We now present the LNS heuristic in more detail. Pseudo-code for the heuristic is shown in Algorithm 1. Three variables are maintained by the algorithm. The variable x^b is the best solution observed during the search, x is the current solution and x^t is a temporary solution that can be discarded or promoted to the status of current solution. In line 2 the global best solution is initialized. In line 4 a destroy and a repair method is chosen, this is described in Section 3.2. In line 5 the heuristic first applies the chosen destroy heuristic and then the chosen repair heuristic to obtain a new solution x^t. More specifically, $d(x)$ returns a copy of x that is partly destroyed. Applying $r(\cdot)$ to the partly destroyed solution repairs it, that is, it returns a feasible solution built from the destroyed one. In line 6 the new solution is evaluated, and the heuristic determines whether this solution should become the new current solution (line 7) or whether it should be rejected. The *accept* function can be implemented in different ways. The simplest choice is to only accept improving solutions. Line 9 checks whether the new solution is better than the best known solution, here $c(x)$ denotes the objective of solution x. The best solution is updated in line 10 if necessary. In line 12 the termination condition is checked, it is up to the implementer to choose the termination criterion, but a limit on the number of iterations or a time limit would be typical choices. In line 13 the best solution found is returned.

3.1 Acceptance criteria

In our implementation we use a simulated annealing criterion as in [13] in the accept function in line 6 of Algorithm 1. The temporary solution x^t is always accepted if $c(x^t) \leq c(x)$, and accepted with probability $e^{-(c(x^t)-c(x))/T}$ if $c(x) < c(x^t)$. Here $T > 0$ is the current *temperature*. The temperature is initialized at $T_0 > 0$ and is decreased at each iteration by performing the update $T_{new} = \alpha T_{old}$,

Algorithm 1 Large neighborhood search

```
1: input: a feasible solution x
2: x^b = x;
3: repeat
4:    select a destroy method d and a repair method r;
5:    x^t = r(d(x));
6:    if accept(x^t, x) then
7:       x = x^t;
8:    end if
9:    if c(x^t) < c(x^b) then
10:      x^b = x^t;
11:   end if
12: until stop criterion is met
13: return x^b
```

where $0 < \alpha < 1$ is a parameter. The idea is that T initially is relatively high and allow deteriorating solutions to be accepted. As the search progresses T decreases and towards the end of the search only a few or no deteriorating solutions will be accepted. Because of the acceptance criterion, one can view the LNS proposed in this paper as a standard simulated annealing algorithm with a complex neighborhood definition. Details about simulated annealing algorithms can be found in [6].

3.2 Selecting destroy and repair methods

This section describes how removal and repair heuristics are selected in line 4 of Algorithm 1. We use the approach suggested by [13] with a minor simplification from [3]. Let Ω^- and Ω^+ be the sets of destroy and repair methods, respectively. Let $\rho^- \in \mathbb{R}^{|\Omega^-|}$ ($\rho^+ \in \mathbb{R}^{|\Omega^+|}$) be a vector with a component for each destroy (repair) method. Each component indicates the weight of the corresponding method, the weight is used to calculate the probability of choosing the method in a *roulette wheel selection mechanism*. The probability ϕ_j^- of choosing the j^{th} destroy method is

$$\phi_j^- = \frac{\rho_j^-}{\sum_{k=1}^{|\Omega^-|} \rho_k^-} ,$$

and the probabilities for choosing the repair methods are determined in the same way. The weights are adjusted automatically, based on the recorded performance of the destroy and repair methods. The idea is that the ALNS heuristic should *adapt* to the instance at hand. When an iteration of the ALNS heuristic is completed a score ψ for the destroy and repair method used in the iteration is computed as

$$\psi = \begin{cases} \omega_1 & \text{if the new solution is a new global best,} \\ \omega_2 & \text{if the new solution is better than the current one,} \\ \omega_3 & \text{if the new solution is accepted,} \\ \omega_4 & \text{if the new solution is rejected,} \end{cases} \tag{1}$$

where $\omega_1, \omega_2, \omega_3$ and ω_4 are parameters. Let a and b be the indices of the selected destroy and repair methods, respectively. The components corresponding to the selected destroy and repair methods

MIC 2009

in the ρ^d and ρ^r vectors are updated using equations (2) in each iteration of Algorithm 1:

$$\rho_a^- = \lambda\rho_a^- + (1-\lambda)\psi, \quad \rho_b^+ = \lambda\rho_b^+ + (1-\lambda)\psi, \tag{2}$$

where $\lambda \in [0;1]$ is the *decay* parameter that controls how sensitive the weights are to changes in the performance of the destroy and repair methods. Note that the weights that are not used at the current iteration remain unchanged.

4 Parallel large neighborhood search

This section describes how the (A)LNS heuristic has been parallelized. We denote the resulting algorithm and software framework PALNS for *parallel adaptive large neighborhood search*. A shared memory model is used. In the proposed parallelization one current solution and one global best solution is shared among the worker threads. Each worker thread obtains a copy of the current solution and performs destroy and repair operations on its local copy. The shared current and global best solutions are updated as necessary. This is shown in Algorithm 2. In lines 2 to 4 variables are declared. Notice that the weight of the destroy and repair methods also are shared among the worker threads. Three locks are declared in line 3 to provide means of synchronization among the worker threads when accessing shared data. L_ρ, L_c and L_b controls synchronization of the weight vectors ρ^- and ρ^+, the current solution and the best solution, respectively. In line 5 the shared variables are initialized before the parallel part of the algorithm is started in lines 7 to 23. The weights of all destroy and repair methods are all set to 1 initially. In line 8 destroy and repair methods are selected, this requires locking ρ^- and ρ^+. The operation `lock(`L`)` acquires the lock L if it is vacant or blocks until it can acquire L. The operation `unlock(`L`)` frees the lock L. In lines 9 and 10 a copy of the current solution is obtained and the temporary solution is destroyed and repaired as in the original algorithm. In lines 11 to 20 the solution is compared to the current and the global best solutions as in algorithm 1. The only difference is that locks have to be obtained to avoid simultaneous updates. At the end of each iteration the ρ^- and ρ^+ vectors are updated. In line 23 the algorithm wait for all worker threads to end. When all have ended, the global best solution can be returned in line 24.

Two more shared variables are maintained by the algorithm that are not mentioned in the pseudo-code: the temperature T and an iteration counter. The iteration counter is used for the stopping criterion in line 22: when all worker threads have performed a certain number of iterations in total, the algorithm terminates.

The parallelization should be able to achieve good speedups relative to the number of processing units available in current desktop/workstation computers (i.e up to 4 or 8), assuming that the destroy and repair operations are time consuming compared to the rest of the operations in one iteration of the LNS algorithm. It is difficult to predict the solution quality of the PALNS compared to the LNS — the two algorithms follow different search trajectories. The major difference is that PALNS attempts several moves in parallel and thus, once a destroy and repair operation has finished the current solution might no longer be the same as the one that triggered the two operations.

The implemented software framework has been implemented in C++ and uses *Intel Threading Building Blocks*, which is a cross-platform software library for writing multi-threaded applications (see [7]). To use the software framework the user needs to supply two data structures: one representing an instance of the problem in question and one representing a solution to the instance.

Algorithm 2 Parallel large neighborhood search

1: input: a feasible solution x;
2: shared data: x, x^b, ρ^-, ρ^+;
3: shared locks: L_ρ, L_c, L_b;
4: private data: x^t;
5: sequential initialization: $x^b = x$, $\rho^- = \mathbf{1}, \rho^+ = \mathbf{1}$;
6: perform in parallel:
7: **repeat**
8: lock(L_ρ); select destroy and repair methods d and r using ρ^- and ρ^+; unlock(L_ρ);
9: lock(L_c); $x^t = x$; unlock(L_c);
10: $x^t = r(d(x^t))$;
11: lock(L_c);
12: **if** accept(x^t, x) **then**
13: $x = x^t$;
14: **end if**
15: unlock(L_c);
16: lock(L_b);
17: **if** $c(x^t) < c(x^b)$ **then**
18: $x^b = x^t$;
19: **end if**
20: unlock(L_b);
21: lock(L_ρ); update ρ^- and ρ^+; unlock(L_ρ);
22: **until** stop criterion is met
23: wait for all threads;
24: **return** x^b

Furthermore the user has to provide at least one destroy and one repair method that both operate on the solution data structure. Linking this with the software framework provides a working, parallel LNS heuristic. More destroy and repair method can be added as needed, the software framework will take care of determining which method to use, based on the mechanism described in section 3.2 The user has the opportunity to set parameters controlling the temperature trail, stopping criterion and weight adjustment policy or he can use a set of standard parameters to get started. At the time of writing the framework is not publicly available yet, but we plan to release it as open source software.

5 PALNS for the TSPPD

The implemented LNS for the TSPPD is simple. It consists of one destroy method and one repair method. The destroy method selects κ requests at random and removes them from the solution. The tour is shortcut where vertices are removed. κ is chosen as a random number in the interval $[\min\{4, 0.1n\}; \max\{20, 0.5n\}]$ where n is the number of requests (In the case that $0.1n > 20$ then 20 request are removed).

The repair method is a simple greedy heuristic. It takes a partial TSPPD solution as input. We say that the requests that are not in the tour are *free*. The free requests are ordered in a random

MIC 2009

fashion and inserted one by one according to the ordering. The insertion of a request is done by inserting the pickup and delivery of the request at the positions that increases the cost of the tour the least. After all requests have been inserted the tour is post-optimized by attempting to *relocate* each request. In other words, a steepest descent local search is performed with a neighborhood consisting of the solutions that can be reached by relocating one request. The destroy and repair operations were already used in a simple, sequential LNS for the TSPPD presented in [4].

6 PALNS for the CVRP

The implemented LNS for the CVRP use three destroy methods and one repair method. The methods have proved to be successful for earlier LNS heuristics for VRP variants (see [13, 9]).

Destroy methods Common to all destroy methods is the number of customers to remove, κ, which is selected as a random number in the interval $[\min\{10, 0.1n\}; \max\{50, 0.4n\}]$ where n is the number of customers. The simplest of the three destroy methods selects κ customers at random and removes them from the solution, shortcutting the routes where customers have been removed. The second destroy method follows the *relatedness* principle suggested in [14]. We use distance between customers to define relatednes. The last destroy method uses information gathered during the search to remove customers. For each arc (i, j), in the graph on which the CVRP is defined, a number f_{ij} is stored. The number represents the cost of the best solution that used arc (i, j), observed during the search. For a customer i let $\pi(i)$ and $\sigma(i)$ be the nodes preceding and succeeding customer i in the current solution, respectively. The *score* s_i of customer i is then $s_i = f_{\pi(i),i} + f_{i,\sigma(i)}$. The destroy method removes customers with high s_i scores as they seem to be misplaced. Some randomness is introduced such that the customer with highest s_i score has the highest probability of being removed, but customers with smaller scores also has a (lower) chance. In a parallel setting each thread has its own copy of the f_{ij} matrix, storing information that has been available to the particular thread. Each thread will therefore have a slightly different view of the historic cost of the arcs.

Repair methods The repair method employed for the CVRP is a randomized regret heuristic which is a construction heuristic. We refer to [9] for a description (there it is denoted regret-2). After exectuting the repair method the individual routes are attempted improved by a 2-opt local search algorithm. We use two different variants of the construction heuristic: one randomized and one deterministic. The PALNS framework choses between the two variants using the method described in Section 3.2.

7 Computational results

This section presents computational results. Due to the page limitation we do not provide details on how parameters were selected. Actually we did very little paramter tuning and relied on parameter values determined in earlier projects. All test were performed on a computer with two Xeon E5430 2.66GHz quad core processors, providing 8 processing units. In all tests we ran 50,000 iterations of the PALNS algorithm.

7.1 Computational results for the TSPPD

We consider a set of instances that was proposed in [12] and was constructed from TSP instances from the TSPLIB. This set of instances contains instances with up to 493 nodes. However, in this test we only use the ones with up 200 nodes.

The computational tests serve two purposes: the main purpose is to investigate the effect of the parallelization, both on running time and on solution quality. The secondary purpose is to compare the LNS algorithm to the heuristic from [12]. Table 1 contain a summary of the computational experiments. The algorithm was run with 1,2,4 and 8 parallel threads and applied 10 times to each instances for each thread configuration. Each row in the table corresponds to experiments with a particular number of threads. The columns should be interpreted as follows: *#Threads*: number of threads used in the experiment, *Avg. time (s)*: average time for one run on one instance, in seconds, *Avg. Gap (%)*: the average over the gap between average solution and best solution. That is, for each instance we calculate the gap $100(\bar{z} - z^*)/z^*$where $\bar{z}$ is the average solution over the 10 runs and z^* is the best known solution (best solution observed in the experiments performed here and in [4]). The result reported is the average over all instances, *Speedup*: reports the speedup relative to the one-thread algorithm.

The conclusions from the table are clear: In general solution quality decreases when more threads are employed. After presenting the results for the CVRP we will get back to possible explanations for this behavior. Regarding the running time we are observing a superlinear speedup. Such performance is possible because the serial and parallel algorithms do not follow the same search trajectory. Still it is surprising to see that the superlinear speedup is consistent over both datasets. Further investigations are needed to fully explain this behavior. Table 2 compares the LNS heuristic

#Threads	Avg. time (s)	Avg. Gap (%)	Speedup
1	49.6	1.44	1.0
2	23.2	1.65	2.1
4	11.9	1.87	4.2
8	5.9	1.87	8.4

Table 1: TSPPD results. Instances contain 51 to 199 vertices. 75 instances in total.

with the best heuristic from [12] (best out of 7 proposed heuristics). The comparison is made on data set used in the previous experiment. The two first major columns report results from the LNS heuristic using 1 and 8 threads, respectively. The sub columns should be interpretted as explained above. The last major column report results from the [12] heuristic denoted SP3. The results are split into two different problem sizes (50–99 nodes and 100 to 199 nodes) and into three problem types A,B and C (see [12] for details). The results for the SP3 heuristic were obtained on a Pentium II 200 MHz computer which is much slower than the one used in this study. It is safe to say that the SP3 heuristic is faster than the LNS heuristic when the computer speed is taken into account. Comparing solution quality we see that the sequential LNS heuristic is doing better than the SP3 heuristic while the parallel version using 8 threads is about on par with the SP3 heuristic. It should be noted that a precise comparison using the reported gaps is not possible: the set of best known solutions that form the basis of the comparision are not identical. The best known solutions from the [12] paper were not available to us. We do believe that the best known solutions available to us are of high quality: the LNS heuristic is able to recreate the optimal solution for all instances solved to optimality in [4].

		1 thread		8 threads		RBL02 - SP3	
		Avg. gap (%)	Avg. time (s)	Avg. gap (%)	Avg. time (s)	Avg. gap (%)	Avg. time (s)
50-99	A	0.5	30.9	0.7	3.5	2.6	37
	B	0.5	30.0	1.0	3.5	2.3	35
	C	1.8	29.5	2.3	3.4	2.2	27
100-199	A	1.4	65.3	1.8	7.8	2.6	217
	B	1.9	64.5	2.4	7.8	1.7	206
	C	1.8	65.5	2.3	7.8	1.9	168

Table 2: Comparison to Renaud, Boctor & Laporte heuristic.

7.2 Computational results for the CVRP

The computational tests for the CVRP is carried out using two well known datasets: The CMT dataset from [2] and the GWKC dataset from [5]. We follow the same test strategy as for the TSPPD: first we evaluate solution quality and speedup as a function of number of threads. Second we compare the LNS heuristic to the best from the literature. Table 3 shows results obtained on the CMT and GWKC datasets. The table should be interpretted as Table 1. We again see that solution quality is decreasing when the number of threads go up. We believe that the parallelization strategy to some extend is working against the simulated annealing principle underlying the LNS heuristic: consider a current solution x at some point during the search. In a sequential LNS we might move away from this solution during e.g. 8 iterations. In a parallel LNS we may move away from this solution for 7 iterations, but when the destroy and repair operation that was initiated with x is finished we may move back to a solution close to x and thereby cancel the work done in the intermediate iterations. Further experiments are necessary to fully understand this effect. The speedups achieved by the parallel LNS are quite respectable, almost linear. The superlinear performance observed for the TSPPD is not repeated. Regarding the quality of the solution obtained

	CMT			GWKC		
#Threads	Avg. time (s)	Avg. gap (%)	Speedup	Avg. time (s)	Avg. gap (%)	Speedup
1	14.9	0.19	1	52.4	0.85	1.0
2	7.6	0.20	2	26.2	0.95	2.0
4	3.8	0.24	3.9	13.2	1.14	4.0
8	1.9	0.30	7.8	6.6	1.37	8.0

Table 3: Summary of experiments on CMT and GWKC datasets. CMT: 14 instances containing 50 to 200 customers, GWKC: 20 instances containing 200 to 483 customers.

by the LNS compared to state of the art heuristics, tables 4 and 5 provides a comparison. Dozens, if not hundreds of metaheuristics for the CVRP have been proposed in the scientific literature so it can be difficult to select the top heuristics. We have chosen to select the same set as in [11] and added the ALNS heuristic from [9] as it forms the basis of the heuristic proposed in this paper. The names in the tables refer to the heuristics from the following papers: P09 — [11], MB07 — [8], TK02 — [16], T05 — [15], P04 — [10], PR07 — [9]. PALNS-1 and PALNS-8 refer to the heuristic from this paper using 1 and 8 threads respectively. The gaps and running times reported are averages over a single run or over the average of several runs so they are comparable when taking the utilized CPU into account. The best known solutions used in the calculation of the average gap were taken

MIC2009

from the solution table at `www.diku.dk/~sropke`. We see from the table that the PALNS based heuristic is able to stand its ground against the top CVRP heuristics and that it is quite fast.

Heuristic	Avg. gap (%)	t(s)	CPU
MB07 Std.	0.03	162.7	P4 2.8 GHz
P09	0.07	15.9	P4 2.8 GHz
MB07 Fast	0.08	3.0	P4 2.8 GHz
TK02	0.18	313.0	P2 0.4 GHz
PALNS-1	0.19	14.9	Xeon 2.66 GHz
T05	0.20	337.5	P2 0.4 GHz
P04	0.24	311.2	P3 1 GHz
PALNS-8	0.30	1.9	Xeon 2.66 GHz
PR07	0.31	104.6	P4 3.0 GHz

Table 4: Comparison to state-of-the art heuristics on the CMT instances.

Heuristic	Avg. gap (%)	t(s)	CPU
MB07 Std.	0.16	1461.2	P4 2.8 GHz
P09	0.46	436.2	P4 2.8 GHz
T05	0.76	2728.8	P2 0.4 GHz
PALNS-1	0.85	52.4	Xeon 2.66 GHz
MB07 Fast	1.06	13.5	P4 2.8 GHz
PR07	1.18	645.7	P4 3.0 GHz
PALNS-8	1.37	6.6	Xeon 2.66 GHz

Table 5: Comparison to state-of-the art heuristics on the GWKC instances.

8 Conclusion

We have presented a framework for implementing parallel metaheuristics based on the large neighborhood search paradigm. The framework makes it easy to implement an LNS metaheuristic and provides parallelization "for free". The framework has produced promising results in tests on the traveling salesman problem with pickup and delivery and the capacitated vehicle routing problem. Future work should aim at improving solution quality when the degree of parallelism increases.

References

[1] R.K. Ahuja, Ö. Ergun, J.B. Ergun, and A.P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123:75–102, 2002.

[2] N. Christofides, A. Mingozzi, and P. Toth. The vehicle routing problem. In N. Christofides, A. Mingozzi, P. Toth, and C. Sandi, editors, *Combinatorial Optimization*, pages 315–338. Wiley, Chichester, UK, 1979.

MIC 2009

[3] J.-F. Cordeau, G. Laporte, F. Pasin, and S. Ropke. Scheduling technicians and tasks in a telecommunications company. Technical Report G-2008-45, GERAD, 2008.

[4] I. Dumitrescu, S. Ropke, J.-F. Cordeau, and G. Laporte. The traveling salesman problem with pickup and delivery: Polyhedral results and a branch-and-cut algorithm. *Mathematical Programming, Ser. A*, 2009. Forthcoming.

[5] B.L. Golden, E.A. Wasil, J.P. Kelly, and I.-M. Chao. The impact of metaheuristics on solving the vehicle routing problem: algorithms, problem sets and computational results. In T.G. Crainic and G. Laporte, editors, *Fleet Management and Logistics*, pages 33–56. Kluwer, Boston (MA), 1998.

[6] D. Henderson, S.H. Jacobson, and A.W. Johnson. The theory and practice of simulated annealing. In F. Glover and G.A. Kochenberger, editors, *Handbook of metaheuristics*, chapter 10, pages 287–319. Kluwer, 2003.

[7] *Intel Threading Building Blocks, Reference Manual.* Intel Corporation, 2009. Revision 1.13. Document Number 215415-001US.

[8] D. Mester and O. Bräysy. Active-guided evolution strategies for large-scale capacitated vehicle routing problems. *Computers & Operations Research*, 34(10):2964–2975, 2007.

[9] D. Pisinger and S. Ropke. A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34(8):2403–2435, 2007.

[10] C. Prins. A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31(12):1985–2002, 2004.

[11] C. Prins. A grasp $\times$ evolutionary local search hybrid for the vehicle routing problem. In F.B. Pereira and J. Tavares, editors, *Bio-inspired Algorithms for the Vehicle Routing Problem*, volume 161 of *Studies in Computational Intelligence*, pages 35–53. Springer, 2009.

[12] J. Renaud, F.F. Boctor, and G. Laporte. Pertubation heuristics for the pickup and delivery traveling salesman problem. *Computers & Operations Research*, 29:1129–1141, 2002.

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

[14] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *CP-98 (Fourth International Conference on Principles and Practice of Constraint Programming)*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431, 1998.

[15] C.D. Tarantilis. Solving the vehicle routing problem with adaptive memory programming methodology. *Computers & Operations Research*, 32:2309–2327, 2005.

[16] C.D. Tarantilis and C.T. Kiranoudis. Bone route: an adaptive memory-based method for effective fleet management. *Annals of Operations Research*, 115:227–241, 2002.

[17] P. Toth and D. Vigo. An overview of vehicle routing problems. In P. Toth and D. Vigo, editors, *The Vehicle Routing Problem*, volume 9 of *SIAM Monographs on Discrete Mathematics and Applications*, chapter 1, pages 1–26. SIAM, Philadelphia, 2002.

MIC 2009