Parallel Tabu Search and the Multiobjective Capacitated Vehicle Routing Problem with Soft Time Windows

Andreas Beham

Institute for Formal Models and Verification
Johannes Kepler University
Altenberger Straße 69
4040 Linz,Austria
andreas.beham@heuristiclab.com

February 13, 2018

Technical Report[1]

Abstract

In this paper the author presents three approaches to parallel Tabu Search, applied to several instances of the Capacitated Vehicle Routing Problem with soft Time Windows (CVRPsTW). The Tabu Search algorithms are of two kinds: Two of them are parallel with respect to functional decomposition and one approach is a collaborative multisearch TS. The implementation builds upon a framework called Distributed metaheuristics or DEME for short. Tests were performed on an SGI Origin 3800 supercomputer at the Johannes Kepler University of Linz, Austria.

1 Introduction

The VRP is a generalized name for a class of problems first formulated in the late 50s by Dantzig and Ramser [1]. The basic layout of the problem consists of a depot with several vehicles and a number of customers scattered or clustered around the depot. The goal then is to find a route from the depot to the customers and back to the depot. To harden this task there are several constraints. The two constraints used in this work are the capacity constraint and the time window constraint. The capacity constraint models a maximum load for each vehicle that it cannot exceed, whereas the time window constraint

[1]This technical report represents material published in *Computer Aided Systems Theory - EUROCAST 2007. EUROCAST 2007. Lecture Notes in Computer Science, vol 4739.* The original source of the publication is available at `https://link.springer.com/chapter/10.1007/978-3-540-75867-9_104`. The final publication is available at `link.springer.com`.

adds a duration to each customer during which it expects to be served. The resulting CVRPTW can then be further categorized in a formulation with *hard Time Windows* and *soft Time Windows* respectively. In the definition of hard Time Windows, a solution is feasible if and only if each customer is reached before its due date. Contrary to soft Time Windows, where the time window constraints are relaxed and the question of feasibility is left to the designer.

Tabu Search was introduced by Fred Glover [2] and has been studied in numerous scientific papers already. The algorithm can be described basically as a "best-improvement-local-search" algorithm. It uses a single solution at a time of which it generates a number of moves leading to other solutions. These are considered to be neighbors of the current solution. From this neighborhood Tabu Search will choose the best solution that satisfies several criteria and continue by creating a new set of moves from there. Because of this behavior the run of the algorithm will leave a trajectory in the search space. To avoid undoing changes made in previous moves and to prevent the algorithm from cycling in the search space, Tabu Search stores recent moves in a memory called *tabulist*. It then forbids to make moves towards one of the configurations that had already been visited before. Over the years Tabu Search became a metaheuristic well known for its good performance when applied to the Vehicle Routing Problem (VRP) [3] [4] [5].

The paper is further organized as follows: In section 2 the problem will be presented briefly, along with a number of operators that the algorithms use to transform the problem. In section 3 we will discuss the approaches and their implementation. Results are given in section 4 and conclusions are drawn in section 5.

2 Capacitated Vehicle Routing Problem with soft Time Windows

The basic concepts of the problem were already introduced in section 1. For a mathematical description of the problem the reader is advised to turn to the work of Thangiah [6].

To represent the problem in a computer a path oriented notation was used, specifically a string of permutated numbers representing the indices of the depot and its customers. Each tour of each vehicle starts and ends at the same depot which is assigned the index 0. Such a tour is read from left to right, so 0 is the first and last character of each permutation. In between the depot appear the indices of the customers that the vehicle visits on its tour. The whole solution is then built by concatenating all tours together into one string, overlapping at the points of intersection. For each vehicle for which no tour is assigned, a 0 is appended to the string, so an empty tour is marked by two consecutive 0s. The maximum number of vehicles R is a parameter specified by the instance of the problem.

To manipulate the solution representation several operators have been presented in the literature and many approaches made good use of them [5]. Of these operators five were selected, giving each operator the same chance to create a neighboring solution. The operators in brief are called: *Relocate*, *Exchange*, *2-opt*, *2-opt** and *Or-opt*. Relocate moves a customer from one route

to another, Exchange swaps two customers of different routes. 2-opt reverses a tour or a part of it, whereas 2-opt* interchanges two tours by crossing the first half of one tour with the second half of another and vice versa. Last but not least, Or-opt moves two consecutive customers to a different place in the same tour.

These operators would stochastically sample a given number of moves from the neighborhood of all moves which is typically very large if all possibilities are to be exhausted. As an example the number of all possible neighbors using the Exchange operator is given by $\frac{|\mathcal{C}|^2-|\mathcal{C}|}{2}$. The disadvantage of stochastic sampling is that it would be possible to miss a good move at a certain iteration and if the sampling is unable to include good moves the search would be drawn away from the better regions of the search space. To counter this a local feasibility criterion was added to each operator. This criterion disallowed to manipulate a solution when it would obviously violate the time window constraints on a local level. E.g. if a customer is to be inserted between two other customers it must be possible that this customer is reached in the best case, as well as the succeeding customer can be reached before its due date as well. This criterion was weak enough that solutions with time window violations occur and strong enough, that the algorithm could find back to a solution with all time windows satisfied.

Because the problem is attacked using a multiobjective definition there is more than one evaluation function that contributes to the fitness of an individual equally. The first evaluation function measures the total traveling distance of all vehicles combined, while the second evaluation function counts the amount of vehicles in use. Both objectives are to be minimized.

3 Algorithms

Parallel Tabu Search has been and continues to be a very active topic [7] and several strategies have been tried focusing on different levels of parallelization, i.e. functional decomposition, domain decomposition and multisearch. In the presented approaches two of them make use of functional decomposition and while a synchronous parallelization was applied early already [8], asynchronous algorithms are not yet common at this level of parallelization. Yet the asynchronous algorithms are interesting as they should perform well on both homogenous and heterogenous systems.

Multisearch spans from trivial implementations with shared memories to very complex approaches that include strategic knowledge. The third approach presented here works at this level and it was chosen to keep with a simple implementation. Section 3.3 will show a more detailed description. A combination of multisearch and functional decomposition could combine the best of two worlds.

3.1 Sequential Tabu Search

The sequential algorithm is outlined in Algorithm 1. This will be the base from which to develop the parallel algorithms. The algorithm uses three memories, the first memory is the *tabu list* and common to all Tabu Search algorithms. The tabu list is organized as a queue and will hold information about the moves made. When the tabu list is full it will forget about the oldest moves. The length

of the tabu list can be specified by the *tabu tenure* parameter. The second memory is the *medium-term memory* $\mathcal{M}_{nondom}$ that keeps a list of non-dominated solutions that had been found in past neighborhoods. If the algorithm does not find "better" solutions for a certain number of iterations, it will take one of the solutions from this memory instead of generating a new neighborhood from the current solution and continue with an emptied tabu list. Solutions are considered "better" when they dominate the current pareto front, which is stored in the third memory $\mathcal{M}_{archive}$, or that are non-dominated to the front and can be added to $\mathcal{M}_{archive}$. A chosen solution can be added to the archive when it is not dominated to the solutions in the archive and when the archive is not full. If the archive is full, the solution is added based on the result of a *crowding comparison*[9]. This comparison orders the solutions in the archive and the chosen solution by a *distance value*, which is computed by calculating the differences of the fitness values of a certain solution with respect to the other solutions. A solution that has a low distance value has similar fitness values compared to the rest of the solutions and will be deleted. This ensures that the solutions will be spread over the pareto front more equally instead of clustering at a certain position.

The initialization of the current solution is done by drawing several solutions from the powerful I1-heuristic [10] that is initialized randomly and choosing the "best" i.e. that dominates the others. The I1-heuristic is a route construction heuristic for the VRP and starts with either the customer with the earliest deadline or the one farthest away. It adds customers based on a savings value that computes the additional distance as well as time windows that the insertion of a customer will cost.

3.2 Synchronous (TSMO-SYNC) and Asynchronous (TSMO-ASYNC) Tabu Search

Common to both approaches is their parallelization of the GenerateNeighborhood() and Evaluate() functions using a master process that distributes the work among himself and several worker processes. These approaches do not improve the quality of the results, but the runtime only. In the synchronized algorithm the master divides the work into equal pieces and then waits until all the work has been collected. The asynchronous algorithm is different in that it does not wait in all cases. Sometimes the algorithm may continue with only half the neighborhood evaluated as there is a promising new solution candidate already or there are several workers waiting. The algorithm's decision function is shown in Algorithm 2. In the synchronous approach the majority of the processors are idle while the master process does the tabu check and memory update, while in the asynchronous algorithm in an ideal situation the slaves would not idle so often.

3.3 Multisearch Tabu Search (TSMO-COLL)

The third approach is asynchronous and is placed in the realm of multisearch parallel algorithms. It uses a multiple points different strategies according to the classification by Crainic et al. [11]. The parameters which determine the strategy of the algorithm vary for each, but the first as they are disturbed by

Algorithm 1 The sequential TSMO-Algorithm

```
1: procedure TSMO(iterations ↓)
2:     s ← GenerateInitialSolution()
3:     evaluations ← 0, iterations ← 0
4:     M ← InitializeMemories()
5:     while evaluations < MaximumEvaluations do
6:         if noImprovement = true then
7:             s ← SelectFrom(M_nondom ↓↑, M_archive ↓)
8:             noImprovement ← false
9:         else
10:            N ← GenerateNeighborhood(s ↓)
11:            Evaluate(N ↓↑, evaluations ↓↑)
12:            s ← Select(N ↓, M_tabulist ↓)
13:            if s ∉ N then
14:                s ← SelectFrom(M_nondom ↓↑, M_archive ↓)
15:            end if
16:        end if
17:        M ← UpdateMemories(s ↓, N ↓)
18:        if isUnchanged(M_archive ↓, iterations ↓) then
19:            noImprovement ← true
20:        end if
21:        iterations ← iterations + 1
22:    end while
23: end procedure
```

Algorithm 2 Decision function of the asynchronous TS

```
1: procedure DECISION(current ↓ N ↓ workers ↓)
2:     c1 ← {w ∈ workers|w = waiting}
3:     c2 ← {s ∈ N|s dominates current}
4:     c3 ← AreWeWaitingTooLong()
5:     c4 ← evaluations ≥ MaximumEvaluations
6:     return |c1| > 0 ∨ |c2| > 0 ∨ c3 ∨ c4
7: end procedure
```

a random variable. The threads then work in a similar way to the sequential algorithm, but communicate improving solutions that they found along the pareto front. An improving solution here is a solution that could be added to the pareto front which is stored in $\mathcal{M}_{archive}$. Likewise a non-improving solution is one that could not be added, because it is dominated or too crowded. The communication list is initialized randomly before the main loop and different for every thread. It is used to determine the collegial searcher that will receive the next improving solution found. On the receiving end this solution is stored in $\mathcal{M}_{nondom}$ and might be considered later by the searcher, specifically if after a number of iterations it has not found better solutions and the qualities of its solution cannot compete with the one received. After the solution has been sent the communication list is rotated in that the first process is moved to the bottom. This way the communication overhead does not become too large and good solutions find their way to other searchers who can explore this region as well, if they do not find improving solutions in the region they currently are. At the end one thread collects all the solutions of the other threads and returns the pareto front as output.

4 Results

The empirical results on a number of test problems were computed on an SGI Origin 3800 supercomputer located at the Johannes Kepler University, Linz, Austria. The Origin 3800 is a shared memory machine with 64GB RAM and consists of 128 R12000 MIPS processors, running at 400Mhz each.

The problemset used in this work is available at a website managed by Joerg Homberger[2]. The instances are grouped into 3 classes and 2 subclasses. The classes are: Clustered customer locations, Randomly distributed customer locations and a combination of both. Each class is then split into instances where the time windows are generally larger and instances where the time windows are smaller.

During the run of the algorithms solutions with constraint violations had been allowed, these solutions were excluded for the generation of the results. Only those solutions were considered that did not violate the time-window and capacity constraints.

As expected the synchronous algorithm completes faster than the sequential algorithm, its solution qualities however are not improving. This is not surprising as the behavior of the synchronous algorithm does not differ from the sequential one. As shown in Figure 4 a maximum speedup seems to be reached quickly with a few number of processors already. The formula to calculate the average speedup value is $speedup = T_s/T_p$, the mean execution time of the sequential algorithm divided by the mean execution time of the parallel algorithm.

The asynchronous algorithms like TSMO-ASYNC and TSMO-COLL are quite interesting and a combination of both could possibly provide the best of both worlds, namely faster execution time and better solution qualities. Of course the number of processors would have to increase even further and it would be even more important to examine the communication closer.

[2]http://www.fernuni-hagen.de/WINF/touren/inhalte/probinst.htm

Algorithm	dominates	is dominated
3 processors		
TSMO	22.57%	32.78%
TSMO-SYNC	22.65%	32.51%
TSMO-ASYNC	26.09%	29.56%
TSMO-COLL	38.55%	15.00%
6 processors		
TSMO	19.53%	33.95%
TSMO-SYNC	20.31%	33.63%
TSMO-ASYNC	22.66%	30.90%
TSMO-COLL	43.91%	7.92%
12 processors		
TSMO	19.47%	34.41%
TSMO-SYNC	19.89%	33.08%
TSMO-ASYNC	20.67%	32.97%
TSMO-COLL	45.32%	4.95%

Table 1: Results from the SetCoverage metric [12] show that the multisearch TS is able to outperform the other two variants in terms of solution quality. Unsurprisingly TSMO and TSMO-SYNC perform equally, TSMO-ASYNC seems to perform a little bit better. These results stem from an average performance on the 400 city problems.

To test the statistical significance a pairwise t-test was performed on the results. Using just 3 processors the results do not show a good level of significance, but by increasing the number of processors TSMO-COLL separates from the other algorithms. With 12 processors there's a notable difference between the results of TSMO-COLL and the other algorithms that is significant at 0.1. From the numbers in Table 1 it can be seen how the algorithms perform when competing against each other.

5 Conclusions & Future Work

Improving performance in both runtime and solution quality does not require a lot of effort and where parallel systems are available researchers should make use of the extra processing power. An interesting approach is the asynchronous master slave algorithm as it performed quite well, yet achieved a better speedup up to 6 processors. Another point to consider is the deployment in a heterogeneous computing environment where the asynchronous variant is having quite an advantage. Whether the collaborative TS in this simple form delivers a good tradeoff between extra solution quality and runtime performance is questionable, however the results from the metric suggests that solutions found were more robust than just the master-slave variants of the algorithm.

What remains for the future would be a comparison between the TSMO versions here and the well established multiobjective evolutionary algorithms in both runtime and solution quality and on different problems, as well as combining the multisearch TS with the asynchronous TS to get the best of both worlds.

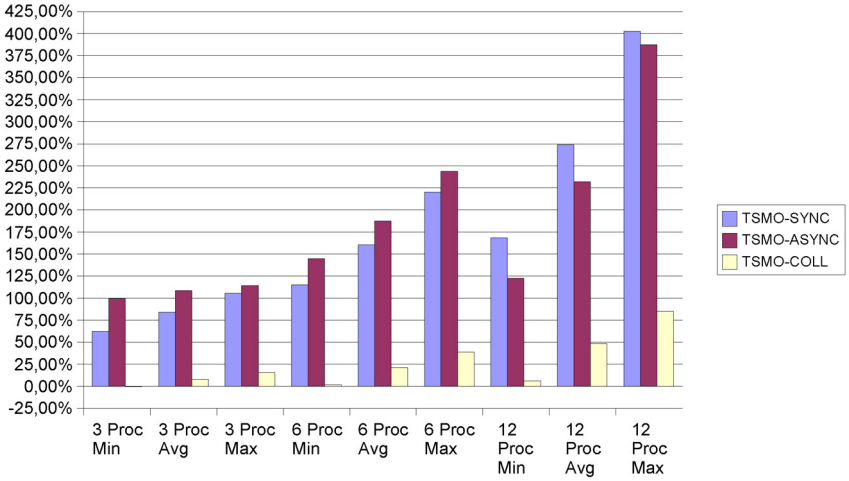

Figure 1: Graph showing the speedup values that were calculated with respect to the sequential algorithm. Note that TSMO-ASYNC can perform better with up to 6 processors and then falls behind TSMO-SYNC. This may be due to the way the master in the asynchronous variant splits the work. Using more processors it could be beneficial for the master not to be involved in doing evaluation for itself, but function purely as a control thread. TSMO-COLL can achieve a little speedup due to the parallelization of the initialization phase before the main loop of the algorithm. Each process in TSMO-COLL just generates one solution from the I1-heuristic.

References

[1] Dantzig, G. B. and Ramser, R. H. 1959. The Truck Dispatching Problem. Management Science, 6:8091

[2] Glover, F. 1986. Future paths for integer programming and links to artificial intelligence. Computers & Operations Research, 13:533–549

[3] Cordeau, J.F., and Laporte, G. 2002. Tabu Search Heuristics for the Vehicle Routing Problem. GERAD Technical report G-2002-15, University of Montreal, Canada

[4] Bräysy, O., and Gendreau, M. 2005. Vehicle Routing Problem with Time Windows, Part I: Route Construction and Local Search Algorithms. Transportation Science, 39(1):104–118

[5] Bräysy, O. and Gendreau, M. 2005. Vehicle routing problem with time windows, Part II: Metaheuristics. Transportation Science, 39(1):119–139

[6] Thangiah, S. R. 1995. Vehicle Routing with Time Windows using Genetic Algorithms. The Practical Handbook of Genetic Algorithms: New Frontiers. 253–278, CRC Press

[7] Crainic, T.G. and Gendreau, M. and Potvin, J.Y. 2005. Parallel Tabu Search. Parallel Metaheuristics: A New Class of Algorithms, 289–314. John Wiley & Sons

[8] Garcia, B.L. and Potvin, J.Y. and Rousseau, J.M. 1994. A Parallel Implementation of the Tabu Search Heuristic. Computers & Operations Research, 21(9):1025–1033

[9] Deb, K. and Agrawal, S. and Prata, A.b and Meyarivan, T. 2000. A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II. Proceedings of the Parallel Problem Solving from Nature VI Conference, 849–858

[10] Solomon, M.M. 1987. Algorithms for the Vehicle Routing and Scheduling Problem with Time Window Constraints. Operations Research, 35:254–265

[11] Crainic, T.G., Toulouse, M., and Gendreau, M. 1997. Towards a Taxonomy of Parallel Tabu Search Algorithms. INFORMS Journal on Computing, 9(1):6172. Academic Publishers

[12] Zitzler, E. 1999. Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications. Doctoral thesis, ETH No:13398, Swiss Federal Institute of Technology, Zurich, Switzerland