Accelerating Local Search in a Memetic Algorithm for the Capacitated Vehicle Routing Problem

Marek Kubiak and Przemysław Wesołek

Institute of Computing Science, Poznan University of Technology
Piotrowo 2, 60-965 Poznan, Poland
Marek.Kubiak@cs.put.poznan.pl

Abstract. Memetic algorithms usually employ long running times, since local search is performed every time a new solution is generated. Acceleration of a memetic algorithm requires focusing on local search, the most time-consuming component. This paper describes the application of two acceleration techniques to local search in a memetic algorithm: caching of values of objective function for neighbours and forbidding moves which could increase distance between solutions. Computational experiments indicate that in the capacitated vehicle routing problem the usage of these techniques is not really profitable, because of cache management overhead and implementation issues.

1 Introduction

Population-based algorithms are usually more time-consuming than their single-solution-based counterparts. Evolutionary algorithms, as an example of the first type, employ large computation times, as compared to e.g. simulated annealing or tabu search. Memetic algorithms (MAs) [1], a kind of evolutionary ones, are even more prone to this problem; in a memetic algorithm a local search process is conducted for every solution in a population, which makes the process of computation even longer.

On the other hand, population-based algorithms usually offer the possibility of exploration of the search space to high extent and, thus, generate better solutions than procedures based solely on local search. Therefore, algorithms managing a population of solutions are a useful tool of optimization. Nevertheless, it would be profitable if the speed of memetic algorithms could be increased without deterioration in the quality of results.

The majority of computation time of a memetic algorithm is usually spent on local search, after each recombination and mutation [1]. Consequently, each attempt to speed up the whole algorithm should be focused on local search.

The main acceleration possibility in local search concerns the computation of quality of neighbours to a current solution. If the difference of the objective function between the current solution and its neighbour may be computed faster than the objective for the neighbour from scratch, then the whole process

C. Cotta and J. van Hemert (Eds.): EvoCOP 2007, LNCS 4446, pp. 96–107, 2007.

speeds-up drastically. In [1] Merz claims that it is possible for almost every combinatorial optimization problem. Jaszkiewicz [2] mentions that local search for the TSP performed almost 300 times more function evaluations per second than a genetic procedure computing the objective from scratch. This is also the case in the capacitated vehicle routing problem (CVRP), which is considered here: neighbours of a solution (w.r.t. commonly used neighbourhood operators) may be evaluated quicker than random solutions.

However, Ishibuchi et al. [3] rightly note that there are problems for which such acceleration is not possible. They give an example of a flowshop problem with the completion time as the objective: a neighbour to a solution is not evaluated faster than a completely new solution. This results from the fact that certain objectives and/or constraints have global character and even a small change in contents of a solution require complete recomputation of the objective function and/or checking all constraints.

Another possibility of speeding-up local search requires caching of (storing in auxiliary memory) values of the difference in objective functions. This technique is not new and is also known as "don't look bits" [4]: if a neighbour of a solution has been evaluated as worse in a previous iteration of local search, then it is not evaluated at all in the current iteration. Such an approach requires that only the changing part of a neighbourhood of a current solution is evaluated.

The two mentioned techniques do not take the memetic search into account. However, there is a possibility to speed-up local search also based on information contained in the population of an MA. If the optimization problem considered exhibits the 'big valley' structure, then it means that good solutions of the problem are located near to each other, and to global optima, in the search space [5], [1], [6]. In such a case recombination operators of MAs should be respectful or distance-preserving [5], [7], [1]. Moreover, the local search process, which is always launched after a recombination, should also observe that the distance between an offspring and its parents is not inflated. This is the place where speed-up may be obtained: some moves of local search on an offspring should be forbidden and, therefore, some neighbours not checked for improvement at all, since they would lead to an increase in distance to parents. This technique was successfully applied in MAs by Merz [8] for the quadratic assignment problem and by Jaszkiewicz [9] for the TSP.

This paper is a study of the application of the two latter acceleration techniques to the capacitated vehicle routing problem. It firstly describes design of and experiments with cache in local search. Then, experiments with local search moves forbidden after distance-preserving recombination are presented. However, due to the nature of the analysed problem and implementation issues it appears that these techniques result in only small acceleration of the memetic algorithm.

2 The Capacitated Vehicle Routing Problem

The capacitated vehicle routing problem (CVRP) [10] is a very basic formulation of a problem which a transportation company might face in its everyday

operations. The goal is to find the shortest-possible set of routes for the company's vehicles in order to satisfy demands of customers for certain goods. Each of identical vehicles starts and finishes its route at the company's depot, and must not carry more goods than its capacity specifies. All customers have to be serviced, each exactly once by one vehicle. Distances between the depot and customers are given.

The version of the CVRP considered here does not fix the number of vehicles (it is a decision variable); also the distance to be travelled by a vehicle is not constrained. Compared to the multiple-TSP, the CVRP formulates one more constraint, the capacity constraint: the sum of demands of customers serviced by one vehicle (i.e. in one route) must not exceed the vehicle's capacity.

Refer to [10] for more information about the CVRP.

3 Cache for Neighbourhood Operators

3.1 The Idea of Caching Evaluations of Neighbours

When applied to a solution, neighbourhood operators in local search for the CVRP usually modify only a small fragment of its contents. Large parts of this solution stay intact. Consequently, large number of moves which modified the original solution may also be performed for the modified, new one, and the modifications of the objective function stay the same. Therefore, there is no need to recompute this change of the objective; it may be stored in cache for later use.

Nevertheless, some moves from the original solution are changed by the actually performed move. These modified moves must not be stored; they have to be removed from the cache. The set of such moves strongly depends on the performed move.

These remarks lead to the following algorithm of local search with cache:

$localSearch(s)$
    do:
        for each $s' \in N(s)$ do:
            if $\Delta f(s', s)$ is stored in the cache:
                $\Delta f = \Delta f(s', s)$ is taken from the cache
            else:
                compute $\Delta f = \Delta f(s', s) = f(s') - f(s)$
                store $\Delta f(s', s)$ in the cache for later use
            if $\Delta f < 0$ then $s_i = s'$ is an improved neighbour of s
        if s_i has been found (an improved neighbour of s):
            $s = s_i$ (move to the neighbour)
            update the cache:
                for each $s_a \in N(s)$ affected by the move, delete $\Delta f(s_a, s)$ from the cache
        else: break the main loop (a local optimum was found)
    while (true)
    return s (a local optimum)

From this description one may notice the possible source of gain in speed: instead of computing $\Delta f(s', s) = f(s') - f(s)$ (the fitness difference) for each neighbour s' of s, this value is stored in the cache for later use. However, the operation of cache update, which has to be called after a move is found in order to ensure the cache stays valid, is a possible source of computation cost. The goal of caching is to make the gain higher than the cost.

Local search is usually utilised in one of two possible ways: first improvement (greedy) or best improvement (steepest). It may be predicted [11] that the gain from caching will be greater for the steepest algorithm. It has to check the whole neighbourhood in every iteration, so the auxiliary memory will be fully up-to-date. In case of the greedy algorithm cache is initially empty and stays in this state for many iterations, until it becomes hard to find an improving neighbour. Only then it is filled with up-to-date values. However, the overhead connected with cache updates is present in every iteration.

3.2 Cache Requirements

In the CVRP not only the objective function matters. There is also the capacity constraint, which involves whole routes, not only single customers. Thus, if the capacity constraint for a neighbour is violated then this neighbour is infeasible; such moves are forbidden in local search. Therefore, not only the change in the objective function has to be stored in the cache, but also the status of feasibility of a neighbour.

Three neighbourhood operators are considered here (size of a neighbourhood is given in brackets):

- *merge*: merge of any 2 routes ($O(T^2)$; T is the number of routes in a solution)
- 2*opt*: exchange of any 2 edges ($O((n+T)^2)$; n is the number of customers)
- *swap*: exchange of any 2 customers ($O(n^2)$)

Because these operators have different semantics, cache must be designed and implemented independently for each of them (in separate data structures).

The local search considered here assumes that the neighbourhoods of these operators may be joined to form one large neighbourhood. It also means that the order of execution of operators cannot be determined in advance (it may be e.g.: *merge*, *merge*, 2*opt*, *swap*, 2*opt*,...; it may be any other order). Such a possibility makes local search potentially more powerful (there are less local optima in the search space) but also more time-consuming. In case of cache this possibility creates a requirement that when one type of move is performed, then cache of all operations has to be updated.

The neighbourhoods of the operators have different sizes; the neighbourhood of 2*opt* and *swap* is considerably larger than the one of *merge*. Moreover, the *merge* operation is very specific: the number of applications of this operator is always very limited by the minimum possible number of routes. Finally, initial experiments with MAs indicated that the number of applications of this operator amounts to 5–10% of the total number of applications of all operators; the majority of search effort is spent on 2*opt* and *swap*. Therefore, the cache was

implemented for these two operators only. The size of memory for the cache structures is the same as the size of the related neighbourhoods.

4 Speeding-Up *2opt* Feasibility Checks

In the CVRP, *2opt* may be used in two main configurations [12]:

- exchanging 2 edges inside one route (in-route *2opt*),
- exchanging 2 edges between two different routes (between-routes *2opt*).

The main computation cost of finding an improving *2opt* move is related to feasibility checks of between-routes *2opt*; it involves two routes, which may become infeasible after the move is performed, due to the capacity constraint present in the CVRP.

For the exemplary solution shown in Figure 1 (top, left) there are two ways in which a *2opt* may be executed if removing edges $(2, 3)$ and $(8, 9)$ (the marked ones):

- by connecting $(2, 8)$ and $(3, 9)$ (Figure 1, top, centre);
- by connecting $(2, 9)$ and $(3, 8)$ (Figure 1, top, right).

Both of these between-routes *2opt* configurations are prone to infeasibility; e.g. while connecting $(2, 8)$ and $(3, 9)$ if the sum of demands of customers $(1, 2, 8, 7)$ or $(6, 5, 4, 3, 9, 10, 11, 12)$ exceeds the capacity, then this move is infeasible.

All such moves require, therefore, that parts of routes (e.g. the mentioned $(1, 2)$ and $(8, 7)$) have known demands, so they could be added for the feasibility check. This is the cause of additional high computation cost in local search (pessimistically: $O(n)$), if these parts of demands are computed from scratch every time a between-routes *2opt* is checked.

In [12] a technique was described which reduces this cost to a constant. It is based on the observation that demands of parts of routes may be stored and simply updated when iterating over neighbours of a current solution in a right order. This order is called a lexicographic one.

An example of such order is given in Figure 1. The top of Figure 1 shows a *2opt* removing edges $(2, 3)$ and $(8, 9)$ (as described above); the demands of parts $(1, 2)$, $(3, 4, 5, 6)$, $(7, 8)$, $(9, 10, 11, 12)$ are required. The bottom of Figure 1 shows the immediately next *2opt* moves (in the lexicographic order), the ones removing edges $(2, 3)$ and $(9, 10)$. The required demands of parts $(1, 2)$, $(3, 4, 5, 6)$ have just been computed in the previous iteration and may be used; the demands of parts $(7, 8, 9)$ and $(10, 11, 12)$ may be computed from the previous values at the cost of two additions.

Due to the high predicted gain in computation time, this technique was used in local search in each configuration with cache.

5 Forbidden Moves of Local Search After Recombination

Based on the results of 'big valley' examination in the CVRP [7] it is known that preservation of edges is important for quality of solutions, as it is the case of the

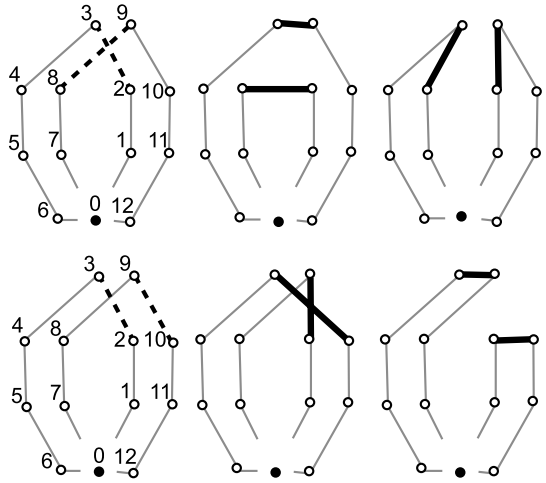

Fig. 1. Edge exchanges in lexicographic order: *2opt* for $(2, 3)$, $(8, 9)$ (*top*) and *2opt* for $(2, 3)$ and $(9, 10)$ (*bottom*)

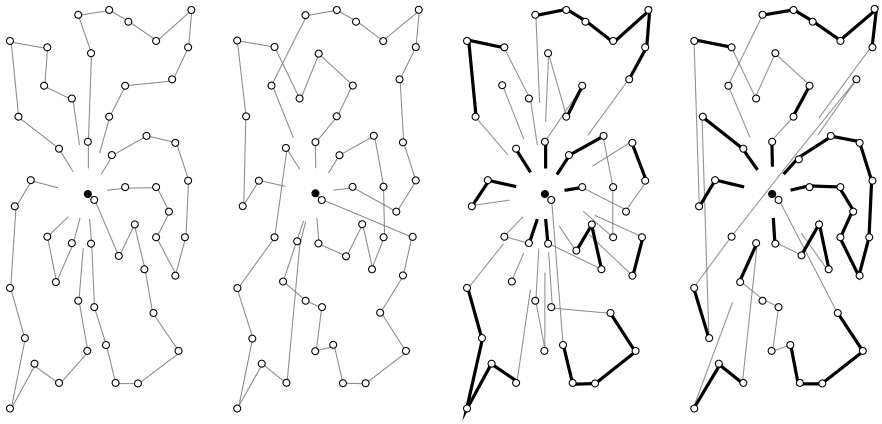

Fig. 2. Example of application of crossover operators: parent1, parent2, CECPX2 offspring (common edges emphasized), SPX offspring (edges from parent1 emphasized)

TSP [9]. Therefore, Kubiak [7] proposed a set of distance-preserving crossover operators for the problem. One of them, CECPX2, creates an offspring in such a way that it always contains all edges common to both parents (common edges), possibly including some additional ones. An example of application of CECPX2 is shown in Figure 2.

Having the idea of 'big valley' and distance preservation in mind, it makes sense after CECPX2 to forbid in local search all moves which would change any edge from the set of common edges. Therefore, an offspring of CECPX2 has the common edges marked as 'forbidden'. All neighbourhood operators check if a

move deleted one of such edges and if so, the move is forbidden. In consequence, a significant speed-up should be obtained if sets of common edges are large.

This technique might be used with other types of operators (not distance-preserving) provided that such operator explicitly computed the set of common edges. If a crossover does not determine the set, it cannot forbid moves changing common edges. As an example of such operator we use SPX (its offspring is shown in Figure 2). It is a very good and fast crossover designed by Prins [13].

6 Computational Experiments

In all experiments 7 well-known instances of the CVRP were used, taken from [14]. Their names (which also indicate the number of customers) are given in tables with results, e.g. Table 1. Instances with different sizes were selected in order to observe the effect of scale in cache and forbidden moves.

6.1 Experiments with Local Search

In order to asses the efficiency of cache in local search, an experiment with 10 different configurations of this algorithm was conducted. These configurations resulted from:

- two versions of local search: greedy and steepest;
- three versions w.r.t. cache: without cache (denoted nc); with all cache structures (c); the same as c, but without *2opt* cache (c*);
- two types of neighbourhoods: one joined neighbourhood of *merge*, 2*opt*, *swap* (described as n-3); a *merge* neighbourhood followed by a joined 2*opt* and *swap* (described as n-1-2).

Each configuration was run in a multiple start local search (MSLS) algorithm, each time starting from a new random solution. MSLS was run 10 times; each run stopped after 100 LS processes.

Average times of computation in this experiment are given in Table 1. Quality of results is not given, since all the configurations had them approximately the same.

The greedy version of local search is several times faster than the steepest one. The version n-1-2 of greedy search is slightly faster than n-3. These results were expected: greedy is usually faster; n-1-2 searches smaller neighbourhoods.

What is more important, the usage of cache in n-3 drastically deteriorates the runing times. This might be explained by the *merge* operations included in the neighbourhood. This operation has no cache on its own, but each time it is performed it results in updates to cache of other operators, making the cache almost empty and the cache management cost unacceptably high.

The usage of cache in n-1-2 gives no improvement, as well.

For the steepest version of local search, the comparison of n-3 and n-1-2 (no cache) yields the same conclusions: the latter is faster. The cache also deteriorates the situation here. Only the c* version results in slight improvements.

Table 1. Average times of computation (in seconds) for local search algorithms, greedy (*left*) and steepest (*right*)

	n-3		n-1-2		
instance	nc	c	nc	c	c*
c50	0.7	2.0	0.7	1.2	0.5
tai75d	2.5	6.9	2.2	4.1	2.5
tai100d	5.5	16.8	4.8	8.4	5.5
c120	11.5	34.6	8.7	14.3	9.2
tai150b	20.8	83.1	17.9	27.2	17.7
c199	30.4	289.4	29.1	43.3	29.5
tai385	299.9	5649.1	301.3	394.3	292.3

	n-3		n-1-2		
instance	nc	c	nc	c	c*
c50	3.5	5.0	2.0	2.7	2.0
tai75d	14.2	18.4	8.8	10.1	8.1
tai100d	34.0	44.3	20.4	22.6	18.5
c120	61.3	80.1	36.8	39.4	31.0
tai150b	122.7	178.7	77.5	81.2	68.8
c199	279.4	519.9	168.8	173.5	147.1
tai385	2543.5	7897.6	1597.5	1594.8	1385.7

To summarize, the results of this experiment show that cache structures do not really improve LS running times. Instead, they slow LS down in many configurations. The cause of this effect lies most probably in the capacity constraint. Because of this constraint the operation of cache updates is time-consuming: if a move to a neighbour changes more than one route (which happens often with 2*opt* and *swap*), then a large part of cache has to be invalidated – all moves concerning every part of the modified routes. This is not the case e.g. in TSP or other unconstrained problems (see [1]).

The possible source of this disappointing result might also lie in details of cache implementation.

6.2 Local Search Execution Profiles

We decided to make detailed profiles of local search executions in order to gain insights into the cost of search and cache operations. In this case, analytical computation of cost is difficult: it is hard to compute the actual or expected number of local search iterations, or to estimate the cache usage. That is why we analysed the issue empirically [11].

We tested LS with the following settings:

- LS version: greedy or steepest,
- cache usage: without cache (nc); with all cache structures (c); with the most promising cache settings, leaving *2opt* cache out as too costly (c*),
- neighbourhood: n-1-2.

Only two instances were tested, tai100d and c120. One run of MSLS was conducted for each setting and instance, consisting of 5 independent LS processes. The runs were limited and short because code profiling usually considerably increases the run time due to injection of timing routines into the original code.

The profiling results of greedy LS for instance c120 are presented in Table 2. They contain the times of operations (search and cache) for each profiled LS setting. Also percentages of the total run time of the base version (n-1-2-nc) are

Table 2. Times of execution of search and cache operations in greedy LS; c120

	n-1-2-nc		n-1-2-c		n-1-2-c*	
operation	time [s]	(percent)	time [s]	(percent)	time [s]	(percent)
2opt: eval. of neighbours	146.3	(50.4)	47.6	(16.4)	57.1	(19.7)
2opt: cache read/write	0.0	(0.0)	30.1	(10.4)	2.6	(0.9)
2opt: cache update	0.0	(0.0)	19.8	(6.8)	0.0	(0.0)
2opt: total search cost	146.3	(50.4)	97.5	(33.6)	59.7	(20.6)
swap: eval. of neighbours	33.2	(11.4)	13.3	(4.6)	11.5	(4.0)
swap: cache read/write	0.0	(0.0)	8.1	(2.8)	7.2	(2.5)
swap: cache update	0.0	(0.0)	5.1	(1.8)	5.1	(1.8)
swap: total search cost	33.2	(11.4)	26.5	(9.2)	23.8	(8.2)
operators: total	179.5	(61.8)	124.0	(42.7)	83.5	(28.8)
greedy LS: total	290.2	(100.0)	209.9	(72.3)	173.1	(59.6)

shown. The *merge* operator is not reported due to insignificant cost of its operations (1–2% of the total run time in all runs).

For the greedy version without cache, very high cost of search by the *2opt* operator is clearly visible (50.4% of the total run time). The cost of *swap* is lower, although it is also considerable (11.4%). Consequently, there is space for improvement in this base version.

The introduction of cache decreases the *2opt* evaluation time, to 16.4%. However, it introduces new cost components: cache reads and writes (10.4%), and cache updates after a performed move (6.8%). In total, the *2opt* search time drops from 50.5% to 33.6%; it seems that the decrease is not as high as could be: the *2opt* cache cost is considerable.

The same conclusion applies to the *swap* operator: the evaluation time drops from 11.4% to 4.6%, but cache management (read/write and update) takes another 4.6%, making the cache only slightly profitable.

The analysis of cache usage in these profiled runs demonstrated that only 28.1% of *2opt* cache is used, while for *swap* it is 58.8%. As predicted, the cached values are rarely used in the greedy version, because improving steps are usually found very quickly (the neighbourhood is not completely searched through, sparsely filling cache with valid values). Moreover, these numbers indicate that *2opt* updates invalidate large parts of cache, while for the *swap* operator most of the cache stays valid after an improving move is performed.

The last setting, n-1-2-c*, did not use *2opt* cache; it seemed that the cache management cost for this operator was too high. The results show that this approach gives the highest gain for the greedy LS: the evaluation cost for *2opt* amounts to 19.7%, but the management cost is almost none (the figure 0.9% reflects the time of calls to empty cache which is not updated at all). In conjunction with some gain from *swap*, the overall speed-up of LS equals 40.4%.

In case of instance tai100d (the detailed results are not reported) the cache management cost was generally higher, making cache usage too expensive. It indicates that cache may be beneficial for larger instances only, if ever.

The steepest version differed mainly in the cache usage: 62.0% of *2opt* neighbours were evaluated based on the cache contents; as much as 77.2% in the case of *swap*. Therefore, gains from cache were slightly higher.

To summarise this experiment, the execution profiles indicate high cache management cost which is generally compensated by gain in evaluation of neighbours, but results in no further significant improvement.

6.3 Experiments with a Memetic Algorithm

The evaluation of forbidden changes (FC) required an experiment with memetic algorithm. This algorithm was run in 12 different configurations renderred by:

- two versions of embedded local search: greedy and steepest;
- two versions w.r.t. cache usage in local search: nc (no cache) and c* (cache, but without *2opt*);
- three variants of crossover: SPX, CECPX2 and CECPX2 with forbidden changes (CECPX2-FC).

Each configuration was run 30 times. The stop criterion was the total number of generations, equal to the average number of generations required for the MA to converge. The algorithms running times are gathered in Table 3.

Table 3. Average times of computation (in seconds) for memetic algorithms, greedy (*above*) and steepest (*below*)

	SPX		CECPX2		CECPX2-FC	
instance	nc	c*	nc	c*	nc	c*
c50	1.9	2.0	1.0	1.0	1.0	1.0
tai75d	13.9	13.9	12.6	12.4	12.7	12.5
tai100d	31.2	30.6	29.4	28.7	29.7	29.1
c120	67.2	57.0	67.5	56.3	68.9	56.6
tai150b	234.2	217.9	218.6	206.9	221.1	206.6
c199	352.7	317.9	373.0	343.9	376.6	341.7
tai385	4170.3	3695.7	5033.2	4377.3	4990.6	4387.5

	SPX		CECPX2		CECPX2-FC	
instance	nc	c*	nc	c*	nc	c*
c50	2.0	2.0	2.0	2.0	2.0	2.0
tai75d	15.7	14.4	12.3	11.3	12.3	11.2
tai100d	36.1	32.2	30.0	27.8	29.7	26.2
c120	61.4	49.8	56.7	47.9	56.9	47.7
tai150b	201.5	181.0	173.9	163.7	178.4	162.3
c199	318.5	277.3	301.0	274.2	297.3	261.9
tai385	3264.5	2767.5	2897.1	2623.0	2879.8	2584.6

For greedy versions of local search embedded in the memetic algorithm the configuration with operator SPX is the fastest, especially with cache (c*), which results in some small gain in computation time (approx. 10%). For the CECPX2 operator also the versions with cache are a bit faster (also approx. 10%). The forbidden changes gain nothing, though.

The MAs with steepest local search are significantly faster than their greedy counterparts. Similarily to the latter, cache usage intoduces a small speed-up. For steepest configurations, CECPX2 is generally faster than SPX. The application of forbidden changes gives in effect a tiny gain in computation time.

In summary, however, it has to be said that both of the applied techniques, cache and forbidden changes, did not provide the expected acceleration in the memetic algorithm.

7 Conclusions

The paper presented experiments with some acceleration techniques for local search and memetic algorithm applied to the CVRP.

The obtained results indicate that the application of cache in the LS gives no real gain; compared to results reported for other problems (e.g. the TSP) there is no profitability in using cache. It seems that the main problem in the cache for the CVRP is the cache management cost, which results from the need to update the cache contents each time an improving move is performed (the capacity constraint forces large parts of cache to be invalidated). The application of forbidden moves in the MA also leads to no gain.

Comparing roughly the implementation cost it appears that cache design and implementation is very expensive (especially the tests of correctness of cache updates), which makes it an inefficient technique. The cost of implementation of forbidden changes was, in contrast, surprisingly low. Perhaps this technique requires some more attention from the authors and it may in future lead to some improvement.

However, the comparison of results from this and the initial version of this paper revealed that the running times of both LS and MA decreased 5–10 times in the final version. It was the result of changes in parts of local search code which affected all the configurations analysed in this paper. These were low-level, implementation changes (e.g. method inlining, avoiding calls to copying constructors by passing references, etc.) introduced after first code profiling. These changes caused the gains from cache and forbidden changes to become virtually invisible, although they were noticeable in first experiments. Perhaps the implementation of cache operations was not as efficient as it could be. Further implementation work should resolve this issue.

Finally, the authors are satisfied with acceleration of local search and memetic algorithm achieved during preparation of this paper.

Acknowledgements. This work was supported by Polish Ministry of Science and Higher Education through grant no 8T11F00426.

References

1. Merz, P.: Advanced fitness landscape analysis and the performance of memetic algorithms. Evolutionary Computation **12**(3) (2004) 303–325
2. Jaszkiewicz, A.: Genetic local search for multiple-objective combinatorial optimization. European Journal of Operational Research **137**(1) (2002) 50–71
3. Ishibuchi, H., Yoshida, T., Murata, T.: Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling. IEEE Transactions on Evolutionary Computation **7**(2) (2003) 204–223
4. Bentley, J.L.: Experiments on traveling salesman heuristics. In: Proceedings of the first annual ACM-SIAM symposium on discrete algorithms. (1990) 91–99
5. Jaszkiewicz, A., Kominek, P.: Genetic local search with distance preserving recombination operator for a vehicle routing problem. European Journal of Operational Research **151**(2) (2003) 352–364
6. Reeves, C.R.: Landscapes, operators and heuristic search. Annals of Operations Research **86**(1) (1999) 473–490
7. Kubiak, M.: Systematic construction of recombination operators for the vehicle routing problem. Foundations of Computing and Decision Sciences **29**(3) (2004) 205–226
8. Merz, P., Freisleben, B.: A genetic local search approach to the quadratic assignment problem. In Bäck, T., ed.: Proceedings of the Seventh International Conference on Genetic Algorithms. (1997)
9. Jaszkiewicz, A.: Improving performance of genetic local search by changing local search space topology. Foundations of Computing and Decision Sciences **24**(2) (1999) 77–84
10. Toth, P., Vigo, D.: The Vehicle Routing Problem. SIAM, Philadelphia (2002)
11. Hoos, H.H., Stutzle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kauffman (2004)
12. Kindervater, G.A.P., Savelsbergh, M.W.P.: Vehicle routing: handling edge exchanges. In Aarts, E., Lenstra, J.K., eds.: Local Search in Combinatorial Optimization. John Wiley & Sons (1997) 337–360
13. Prins, C.: A simple and effective evolutionary algorithm for the vehicle routing problem. In de Sousa, J.P., ed.: Proceedings of MIC 2001, the 4th Metaheuristics International Conference. (2001) 143–147
14. Rochat, Y., Taillard, É.D.: Probabilistic diversification and intensification in local search for vehicle routing. Journal of Heuristics **1**(1) (1995) 147–167