An Efficient Implementation of a Static Move Descriptor-based Local Search Heuristic

Beek, Onne; Raa, Birger; Dullaert, Wout; Vigo, Daniele

published in
Computers and Operations Research
2018

DOI (link to publisher)
10.1016/j.cor.2018.01.006

document version
Publisher's PDF, also known as Version of record

Computers and Operations Research 94 (2018) 1–10

Contents lists available at ScienceDirect

Computers and Operations Research

journal homepage: www.elsevier.com/locate/cor

An Efficient Implementation of a Static Move Descriptor-based Local Search Heuristic

Onne Beek[a], Birger Raa[a,*], Wout Dullaert[b], Daniele Vigo[c,b]

[a] Department of Industrial Systems Engineering and Product Design, Ghent University, Belgium
[b] Department of Information, Logistics and Innovation, Vrije Universiteit Amsterdam, Belgium
[c] Department of Electrical, Electronic, and Information Engineering, University of Bologna,Belgium

ARTICLE INFO

Article history:
Received 15 September 2017
Revised 11 December 2017
Accepted 6 January 2018
Available online 13 February 2018

Keywords:
Efficient Local Search
Vehicle Routing
Static Move Descriptors
Heuristic Priority Queue

ABSTRACT

This paper proposes several strategies for a more efficient implementation of the concept of Static Move Descriptors (SMDs), a recently developed technique that significantly speeds up Local Search-based algorithms. SMDs exploit the fact that each local search step affects only a small part of the solution and allow for efficient tracking of changes at each iteration, such that unnecessary reevaluations can be avoided. The concept is highly effective at reducing computation times and is sufficiently generic to be applied in any Local Search-based algorithm. Despite its significant advantages, the design proposed in the literature suffers from high overhead and high implementational complexity. Our proposals lead to a much leaner and simpler implementation that offers better extendibility and significant further speedups of local search algorithms. We compare implementations for the Capacitated Vehicle Routing Problem (CVRP) - a well-studied, complex problem that serves as a benchmark for a wide variety of optimization techniques.

1. Introduction

Local Search is one of the main optimization techniques used to tackle NP-hard problems. Its popularity comes from its simplicity: by iteratively applying small changes to a solution, it thoroughly explores the solution space surrounding the current (or 'incumbent') solution and gradually improves towards a local optimum. The nature of these small changes makes Local Search a very flexible technique. Local Search operators can vary from very basic operators that affect only a few solution characteristics to complex subroutines that combine multiple changes to perform a significant restructuring of the solution. These operators embody a trade-off: complex operators can perform a more extensive search of the solution space and thus can reach higher quality solutions, but the number of possible changes and thus the effort required to find improving changes increases with their complexity. As a result, most Local Search procedures usually only adopt operators that affect a very small number of characteristics.

Local Search operators are also used as building blocks in metaheuristic solution approaches. Pure Local Search metaheuristics generally make use of a larger set of simple operators, combined with a guiding strategy to steer the search out of local optima. The potential of these pure Local Search metaheuristics has been demonstrated by multiple powerful algorithms, such as Tabu Search (Glover, 1989), which allows worsening moves and prevents cycling by blocking moves with specific ('tabu') characteristics; Guided Local Search (Voudouris and Tsang, 1999), which guides the search away from local optima by penalizing undesirable, or overly frequent, characteristics; and Variable Neighborhood Search (Mladenović and Hansen, 1997), which escapes from local optima of one operator by invoking another. All these metaheuristics are capable of diversifying the search using only Local Search operators.

In contrast to pure Local Search-based metaheuristics, there is also widespread use of Local Search in so-called hybrid metaheuristics. These algorithms use the power of Local Search as a tool for intensification - a strong, localized search to improve a solution without drastically altering its structure. This Local Search phase is then alternated with a diversification method. Popular examples of such hybrid metaheuristics are Memetic Algorithms (Norman and Moscato, 1989), where the Local Search is wrapped into an evolutionary, population-based optimization framework, and Iterated Local Search (Martin et al., 1992), which applies a strongly disruptive *shake* method to a solution stuck in a local optimum, and then restarts the Local Search on this diversified solution.

* Corresponding author at: Technologiepark 903, 9052 Zwijnaarde, Ghent, Belgium.
E-mail address: Birger.Raa@UGent.be (B. Raa).

https://doi.org/10.1016/j.cor.2018.01.006
0305-0548/

The underlying principle of these hybrid algorithms is clear: use Local Search to improve the incumbent solution, then apply a different method to the solution to escape from the local optimum. Although many authors focus on novel ideas for the diversification, it is still the Local Search phase that transforms candidate solutions into high-quality solutions. E.g., Nagata and Bräysy (2008) note that their powerful memetic algorithm spends 80-90% of its time on the Local Search phase. Johnson and McGeoch (1997) note that in case of the hybrid GA, spending more time on the Local Search phase is more valuable than increasing the population size, as shown by the 'population of one' genetic algorithm that lead to Iterated Local Search.

Finding efficient ways of performing Local Search thus has a critical effect on the performance of any heuristic or metaheuristic algorithm. In this paper, we therefore focus on this important and often overlooked aspect. We do this by applying Local Search to the well-known Capacitated Vehicle Routing Problem, a fundamental model for transportation problems with multiple vehicles. Given a set of locations each requiring the delivery of a given volume, the goal is to design a set of routes so that each location is visited exactly once in the minimal total distance or time travelled. Every route is limited to a maximum capacity (storage space, service time, etc.). The VRP is conceptually simple, but computationally challenging. Its practical relevance and computational challenges make the VRP an excellent problem for benchmarking experiments.

Local Search is a popular choice of optimization technique for many variants of the VRP. The problem structure of the VRP can be exploited in relatively simple Local Search operators that yet prove to be very powerful. E.g., moving a customer to a different position in the visiting sequence, or swapping two customers in the sequence, are simple operators that are easy to evaluate both in terms of how they change the solution quality and in terms of solution feasibility. Additionally, the scope of such Local Search operators can easily be limited (e.g., to nearby customers only) in order to speed up the search (see Section 2.2).

In the following section, we will explain the strengths and challenges of efficient Local Search strategies applied to the VRP. In Section 3 we discuss the concept of Static Move Descriptors, a technique recently introduced by Zachariadis and Kiranoudis (2010) to speed up Local Search. We analyze the performance and identify the drawbacks of their implementation in Section 4. In Section 5, we outline the changes we made in order to create a leaner, simpler implementation that outperforms the original. In Section 6 we empirically show the benefits of our version over the original by performing extensive experiments on VRP benchmark instances, followed by concluding remarks in Section 7.

2. Efficient Local Search

The application of Local Search is straightforward for most problems. As soon as a suitable solution representation is determined, operators such as moving or swapping elements in the solution can quickly be implemented. A common, yet naive implementation of Local Search simply loops over all possible moves for a set of operators in order, identifies the best feasible move and applies the corresponding change to the solution; this cycle repeats until no more improving moves can be found.

As an example, consider the Swap operator for the CVRP that swaps the position of two locations in the sequence. This operator has a cardinality of $O(n^2)$ since any combination of two locations unambiguously defines a specific move. Enumerating all Swap moves is as simple as having two nested loops go over all locations. Similarly, the Relocate operator moves a location to a different position in the sequence. This operator also has a cardinality of $O(n^2)$ since any location could be moved to any position in the sequence. A naive Local Search implementation would evaluate the effect on the solution and feasibility of all Swap and Relocate moves (possibly along with other Local Search moves that have been identified).

The goal of Efficient Local Search, and also the main contribution this paper wants to make, is to come up with a less naive, more powerful implementation of Local Search that reduces the computational effort required to achieve high-quality solutions. In previous research, various techniques to accomplish this goal have already been suggested, each with their own advantages and disadvantages. These techniques can be categorized into three groups: Acceptance and Search strategies, Candidate Set strategies and Locality Tracking strategies.

2.1. Acceptance and Search strategies

As explained above, a naive implementation evaluates all possible moves at each iteration, after which the best feasible move is selected and applied to the solution. This known as the 'best-accept' strategy. An alternative, known as the 'first-accept' strategy, is to immediately apply an improving feasible move as soon as it is encountered without first evaluating all other possible moves. This can drastically reduce the search time per iteration. However, the downside is that smaller improvements are applied and thus more iterations are usually required to reach a similar solution quality. Therefore, an actual speedup using the first-accept strategy can only be achieved if the order in which moves are evaluated somehow corresponds to the moves' improvement potential, i.e., if the best improving moves are encountered first. This can be achieved with a clever Search strategy. A good example of this is the Sequential Search of Irnich et al. (2006), in which edges are sorted by their cost in a pre-processing step. This allows the Local Search to look at moves involving nearby nodes first and effectively discards (partial) non-improving moves (involving nodes that are far apart). The one-time pre-processing step requires $O(n^2 \log n)$ time, but leads to significant speedups: commonly used $O(n^2)$ operators reach speedups of a factor 100, whereas the $O(n^3)$ operator 3-Opt obtains a speedup factor up to 14,000 under ideal circumstances.

Another common search strategy is to only consider one operator at a time and apply best-accept per operator, which became known as the Variable Neighborhood Descent (VND) strategy (Mladenović and Hansen, 1997). Only a single operator is evaluated until no more improving moves are found. Then, the search switches to a different operator. The underlying insight is that a local optimum for one operator is not necessarily locally optimal for another operator. This strategy works particularly well when operators of different cardinality are used: first, the simple operators are exhausted and only then the higher-order operators are used.

2.2. Candidate Set strategies

The search strategy determines the order in which moves are evaluated, and whether a single or multiple operators are considered at once. However, even with first-accept, the entire search space has to be evaluated eventually. This can be very expensive, especially for large scale instances. Candidate Set strategies therefore limit the search to a subset of moves that seem more promising, or conversely, skip the evaluation of moves that are deemed less likely to result in improvements.

For the Traveling Salesman Problem, Lin and Kernighan (1973) introduced the K-nearest neighbors strategy, in which the search only considers the K nearest neighbors of a vertex, i.e., the K cheapest incident edges (with $K < n$). Johnson and McGeoch (1997) extended this strategy by selecting the K-nearest neighbors of each quadrant around a vertex, ensuring the existence of

a Hamiltonian Tour in the reduced edge set. The Granularity principle of Toth and Vigo (2003) also builds candidate sets based on edge costs, by only considering edges with a cost below a pre-set threshold. This leads to variable-size candidate sets per vertex, allowing for more flexibility for vertices in dense areas.

Reinelt (1994) does not directly consider edge costs, but uses the Delaunay Graph as the basis for candidate sets, since this graph accurately captures the structure of the vertex distribution and ensures connectivity in all directions. Similarly, Fang et al. (2013) use Voronoi K-rings (a dual representation of the Delaunay Graph) to determine the candidate sets.

Search limitation and candidate sets can also be used to achieve diversification. An example of this is found in Nagata and Bräysy (2008), where the authors combine a genetic algorithm with Local Search. After each crossover, a Local Search phase is applied to the child; however, the Local Search is limited to specific edges, such as common edges of the parents or new edges introduced during mutation. As such, the workload of the Local Search is reduced, while at the same time the convergence between similar solutions, typically caused by Local Search, is reduced.

All of these search limitation methodologies suffer the same weakness, namely they all run the risk of ignoring important edges. This is especially true for the simpler, edge cost-based criteria.

2.3. Locality Tracking

A third group of performance enhancing strategies for Local Search are those that exploit its basic premise: a Local Search operator only causes a minor, 'local' change to a solution. This local effect means that, after applying a move, there are many possible moves that are not affected by this local update and therefore need not be reevaluated. This insight lead to the introduction of 'Don't-Look Bits' (Bentley, 1990): during the evaluation of moves, vertices that were not affected by previous moves are tagged (using a single bit); during the move execution phase, all vertices involved are untagged as their surroundings have changed.

Whereas Don't-Look Bits avoid unnecessary reevaluation of certain moves, the locality tracking principle can be extended further by storing information about every possible move in memory. This is what Zachariadis and Kiranoudis (2010) did by introducing what they called Static Move Descriptors (SMDs). Since SMDs and their implementation are the subject of this paper, they will be discussed in detail in the next sections.

3. Static Move Descriptors

The idea of Static Move Descriptors (Zachariadis and Kiranoudis, 2010) is to build a memory structure that stores moves and efficiently accesses and updates those moves that are affected by executing another move. Therefore, for each possible move, an SMD is created that contains the necessary information to unambiguously define it (e.g., a Swap move is defined by the two vertices being swapped). For each SMD, there is also a dynamic tag. This tag holds the effect on the objective function if the move defined by the SMD were to be executed on the current solution. Note that the feasibility of a move is not included in the tag. During the Local search, these dynamic tags need to be updated. However, since a move only has a local effect, it is unnecessary to update all tags. Tags require updating only if their SMD contains a vertex that was affected by the move last executed.

For a further analysis of the performance of Local Search using SMD, we first explain the underlying algorithm. We can discern three separate phases: a one-time *Initialization* phase, followed by iterating between *Search* and *Update* phases.

Initialization phase. Given a starting solution, an SMD for each move is generated and the corresponding cost tags are calculated. The SMDs are then inserted into a *priority queue* (PQ), which keeps them organized according to their 'key' (current cost tag). The overhead of creating this PQ makes this step computationally more expensive than the first iteration of an implementation with no or more basic locality tracking (e.g., Don't-Look Bits). However, this initialization only occurs once and its overhead is amortized over all further iterations of the Local Search.

Search phase. The SMD with the best tag is extracted from the PQ and the feasibility of the corresponding move is checked. If infeasible, the SMD is added to a temporary container and the next best SMD is extracted from the PQ. Once a feasible SMD is found, the search ends and all SMDs in the temporary container are reinserted into the PQ. The final extracted SMD contains the best-improving (or least-worsening) feasible move. This move is then executed. This phase of the Local Search is much faster than in an implementation with no or more basic locality tracking, since the next move to be evaluated is readily available in front of the PQ. The Search phase therefore does not require looping through the incumbent solution again (possibly skipping some of the vertices because of their Don't-Look Bits).

Update phase. Executing the move changes the solution. Based on the move definition in its SMD and *update rules* that describe how each operator affects the solution, a list can be assembled of SMDs that are affected by executing the move. These SMDs are removed from the PQ, after which their cost tags are updated with regards to the new solution and they are reinserted into the PQ.

Compared to a more naive implementation, an SMD-based implementation requires less evaluations per iteration. However, this comes at the cost of operations to create the PQ and to keep its elements organized. This trade-off becomes more favorable as instance size increases, making the technique effective for tackling (very) large problem instances.

Zachariadis and Kiranoudis (2010) use only Swap, Relocate and the inter-route and intra-route versions of 2-Opt as their operators. The first three operators affect at most 6, 5 and 4 vertices respectively, while the intra-route 2-Opt affects at most one entire route. If we denote the number of affected vertices by c, then the number of tag updates per iteration (and thus the number of removals from and insertions into the PQ) is only a fraction c/n of the total number of tags. Since c is constant for most operators, this provides a clear scaling benefit as this ratio decreases when the instance size n increases. The actual cost of updating depends on the implementation of the priority queue but can be done in $O(\log n)$. As a result, the computational effort of the aforementioned operators grows as $O(n \log n)$, compared to $O(n^2)$ in the naive implementation described at the start of Section 2.

The results shown by Zachariadis and Kiranoudis (2010) are very promising, reaching a speedup factor of nearly 10 for instance sizes of a thousand vertices. However, no other researchers have adopted SMDs so far. We believe that this is mainly due to the implementational complexity. In the following section, we provide suggestions that reduce the complexity of the implementation and that nevertheless further increase speedup factors. Thus, we hope to help increase the adoption of SMD-based Local Search by other researchers.

4. Implementation challenges

Cordeau et al. (2002) describes four attributes of good (VRP) heuristics: *speed, accuracy, simplicity* and *flexibility*. Our main goal is to suggest an alternative implementation of SMD-based Local Search. Since its performance was already shown by

Zachariadis and Kiranoudis (2010), this means we are not targeting accuracy. Instead, we do target the three other attributes described.

The crucial element in an SMD implementation is the choice of data structure for the priority queue. Because we always need to extract the best tag from this PQ, a Heap data structure is most appropriate. Heaps can be seen as trees that satisfy the *Heap property*: a parent node always has a better key than its child nodes. As a result, the best element in a Heap is at the root of the tree. The computational effort required for the other PQ operations (inserting and removing elements) depends on the Heap implementation. Different versions exist with different complexity bounds on these operations. The most common Heap types are the Binary Heap (Williams, 1964), the Binomial Heap (Vuillemin, 1978) and the Fibonacci Heap (Fredman and Tarjan, 1987).

In the remainder of this section, we will explain some implementational details of the SMD-based algorithm and how this relates to the three attributes we are targeting.

Fibonacci Heap. Zachariadis and Kiranoudis (2010) implement the PQ using a Fibonacci Heap, introduced by Fredman and Tarjan (1987). This complex data structure is known for having excellent theoretical performance. However, Fredman and Tarjan (1987) warn for its practical performance. The Fibonacci Heap relies entirely on pointer links for all its operations, which leads to high constant factors hidden in its complexity bounds, as well as high memory overhead.

Choosing the Fibonacci heap for its great theoretical performance is clearly motivated by the speed objective. However, due to the complexity of the Fibonacci Heap, this may not be a good choice with regards to simplicity.

Cross-Operator Effects. The original implementation uses all Local Search operators simultaneously. This means that multiple PQs need to be maintained and that executing a move affects all PQs simultaneously. Thus, a series of *update rules* is required that define how executing a move of one operator affects the SMDs of all other operators. Additionally, any time an operator is added to the set, two series of update rules need to be defined: one series that defines how the new operator affects the other operators and one series for the reverse. In fact, adding a higher-order operator even degrades the performance of the lower-order operators: an $O(n^3)$ operator generally affects more nodes (higher c) and requires $O(c\,n^2)$ moves to be updated; even executing a move of one of the $O(n^2)$ operators would carry the full $O(c\,n^2)$ update cost.

As a result, simultaneously considering all Local Search operators strongly decreases the simplicity, flexibility and even the speed of the solution framework.

Initialization Overhead. During the initialization phase, the PQ data structures have to be constructed. This includes requesting memory, calculating the inital SMD tags and building the Heaps for every operator. This is an extremely costly phase: it includes a single iteration of a naive implementation, plus the significant overhead required to build the data structures. However, this phase is a one-time investment and its cost can be amortized over all the iterations in the Local Search phase.

Only a limited number of metaheuristics can take advantage of the SMD implementation proposed by Zachariadis and Kiranoudis (2010), as it requires sufficiently long Local Search runs to make up for the initial investment for building the Fibonacci Heaps for all of the operators. This is often not the case in popular hybrid metaheuristics, where restarts, disruptive diversification and partial searches are commonly used, requiring heaps to be more frequently rebuilt.

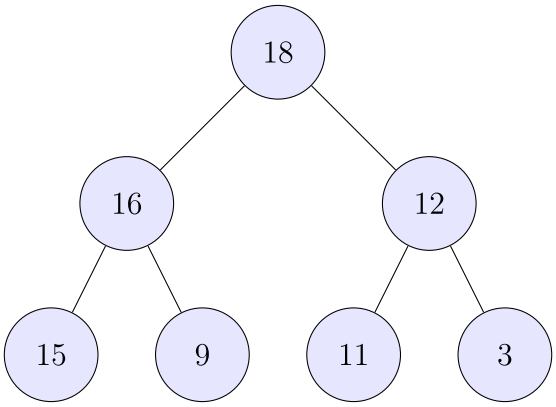

Fig. 1. Tree representation of a Binary Heap.

Table 1
Array representation of a Binary Heap.

Index	0	1	2	3	4	5	6
Value	18	16	12	15	9	11	3

Memory Usage. In the implementation of Zachariadis and Kiranoudis (2010), every SMD element is stored during the entire runtime of the algorithm. The memory requirements increase with the number of operators and the complexity of each operator (e.g.: $O(n^2)$ for 2-Opt vs. $O(n^3)$ for 3-Opt). Apart from the SMD data, the Fibonacci Heap requires four pointers and three extra data fields (to keep the Heap organized) per element. Further, the volatile structure of the Fibonacci Heap results in poor usage of cache memory, which also has strong implications for the performance of the algorithm on modern computer architectures.

Now that we have identified the drawbacks of the original SMD-based Local Search algorithm, we will next discuss our proposed changes and its implementation to obtain a version that is simpler, more flexible and much faster.

5. Efficient SMD Implementation

This section will explain the changes we propose for implementing local search with SMDs. Our main goal is to provide a simpler implementation that is more flexible in its use. This way, we hope to enable more widespread adoption of the SMD framework into state-of-the-art algorithms.

5.1. Binary Heap

Our first suggestion is to use a Binary Heap instead of a Fibonacci Heap. The Binary Heap (Williams, 1964) is the oldest Heap type. Its implementation is both easy and elegant: it can be encoded as an array structure where child-parent relations can be determined with simple index manipulation. This makes it very memory-efficient, since no additional variables or pointers are required. E.g., a node located at index k in the array will find its parent at $k/2$ and its left and right children at $2k$ and $2k+1$, respectively. Fig. 1 shows an example of a Binary Heap. It is organized according to the Heap property: every parent has a tag that is larger than its children's tags. Table 1 shows the array representation of the same Heap.

Apart from the array, we also make use of a supporting data structure to store the SMD elements. Rather than directly storing SMDs in the PQ, we build a matrix that holds all elements. The PQ then holds pointers to the matrix elements and the matrix elements also track its position in the PQ. This way, we can efficiently

Table 2
Update decision table

Old tag	New tag	Update
$\geq T$	$\leq T$	*delete-element*
$\geq T$	$\geq T$	*update-key*
$\leq T$	$\geq T$	*insert-element*

find specific elements in the Heap. Additionally, all SMDs exist permanently in the matrix data structure, meaning we are not forced to store them in the PQ at all times.

With these data structures, only two pointers and the tag value need to be stored in memory (since the SMD is uniquely defined by its position in the matrix). For a quadratic Local Search operator, this results in a memory requirement that is only half of what is required with a Fibonacci Heap.

Building a Binary Heap is very efficient and the strongly reduced memory cost results in a much faster *Initialization phase*. This has only a minor impact on the algorithm proposed by Zachariadis and Kiranoudis (2010) as the Initialization phase only takes place once, but it is something we will use to our advantage as explained in Section 5.3.

Element Pruning. The size of a Binary Heap, denoted N, determines the speed of the implementation, since all its operations are $O(\log N)$. As our implementation uses the matrix support structure, we can prune SMDs from the PQ, such as those with a cost tag above a chosen threshold value T (e.g., $T = 0$, meaning only improving moves are considered in the PQ). When the current solution of the algorithm is of decent quality, this simple pruning strategy drastically reduces the Heap size, improving the speed of all operations.

Heap Update. Any Heap data structure comes with functionality to manipulate its elements. Most Heap implementations only support the basic functionality of efficient *insert-element, delete-root* and *improve-key* operations. Some Heaps can be augmented with additional, non-standard operations, but they may be costly. As an example, the Fibonacci Heap's general *delete-element* requires an *improve-key* to make it the new root and a *delete-root* to remove it from the Heap, before its key is updated to the new value and the element is reinserted using *insert-element*. In contrast, for the Binary Heap it is trivial to augment it with efficient generalized *update-key* and *delete-element* operations. This enables us to update the cost tag of SMD elements with only minimal movements inside the Heap. Instead of removing and reinserting all affected SMDs, our Update procedure now handles SMDs differently depending on the old tag, the new tag and the pruning threshold T, as shown in Table 2.

With the pruning and update rules, the Binary Heap outperforms the Fibonacci Heap in the *Initialization* and *Update* phases, but not in the *Search phase*. The structure of the Binary Heap does enable us, however, to adopt a change in the Acceptance strategy.

5.2. Acceptance strategy

In Section 2.1 we stated that accepting the first improving move can reduce the required search time, but only in combination with an effective Search strategy that can guarantee good moves will be found first.

Due to the Heap property ($parent(k) \geq k \geq children(k)$), the elements with good tags in a Binary Heap are likely to be located at the beginning of the array. Therefore, instead of extracting the best improving SMD from the Heap until a feasible one is found, we read linearly through our array-mapped Heap, avoiding Heap operations entirely during the Search phase. As such, we adopt a first-accept strategy in which good moves are indeed likely to be encountered first. The only downside of this approach could be that the search could encounter feasible worsening moves while there are still improving ones available. However, pruning the Heap with $T = 0$ prevents this from occurring.

To illustrate this first-accept strategy, consider the example in Fig. 1 and Table 1. The values in the array are not exactly sorted in descending order, but only roughly. Suppose that the two first tags, 18 and 16, correspond to infeasible moves. Then the move with tag 12 will be evaluated next and executed if it is feasible, while the best possible feasible move could be the one with tag 15.

Table 3
Comparison of implementations

Heap type	ZK10	Fibonacci
	BRDV	Binary
Heap size	ZK10	Entire move set
	BRDV	Improving moves only
Acceptance strategy	ZK10	Best-accept
	BRDV	First-accept (among roughly sorted moves)
Search strategy	ZK10	All operators simultaneously
	BRDV	VND with operator cycling

5.3. Variable Neighborhood Descent

One major drawback of the original SMD algorithm are the cross-operator effects. For each operator in the operator set, a Heap has to be maintained. Increasing the operator set requires also increases the set of update rules as well as the actual update effort at each iteration.

To overcome this downside, we propose a third change and adopt a different Search strategy by using Variable Neighborhood Descent (cfr. section 2.1). With this choice, we reach a different trade-off: Search and Update phases become easier since only one operator is considered at a time, but every operator switch incurs the cost of an initialization. However, as mentioned above, initializations are relatively cheap for a Binary Heap.

Furthermore, we adopt an operator cycling strategy, which completely exhausts an operator before moving to the next and cyclically iterates over the operators in a given sequence, until none of them still find an improvement. This maximizes the number of iterations per operator, which helps amortizing the initialization cost over a larger number of iterations.

By iteratively optimizing with a single operator, no cross-operator updating is needed, drastically reducing the cost of the Update phase at every iteration. An additional benefit is that this SMD implementation is both simpler and more modular; plugging in a new operator only requires defining the operator logic and a single update rule for its own SMDs, rather than two update rules for every other operator in the set. This has the added benefit that adding higher-order operators no longer degrades the performance of lower-order operators.

Finally, since only one Heap is kept at all times, adopting VND further reduces the memory requirements. We already saw that memory requirements per SMD element for Binary Heaps are only half of that for Fibonacci Heaps. The overall memory reduction factor is therefore 2 times the number of operators (i.e., 6 when using Swap, Relocate and 2-Opt).

The proposed changes to the SMD implementation are summarized in Table 3, where ZK10 refers to the original implementation and BRDV refers to out suggested implementation. The result is a significantly simplified implementation that nevertheless outperforms the original, as we will show in Section 6.

Because of the differences in acceptance and search strategies, BRDV will lead to a different solution at the end of the Local Search

than ZK10. In the computational tests of Section 6.3, we will therefore evaluate runtimes as well as solution quality.

6. Results

This section examines the impact of the different implementation changes that we suggest. We start from the original SMD implementation, denoted 'ZK10', and consider the effect on computation times of adopting a Binary Heap instead of a Fibonacci Heap and pruning (with threshold $T = 0$).

After that, we implement changes that not only affect computation times but also lead to differences in the results. The first of these is the switch to the Variable Neighborhood Descent, where only one Local Search operator is considered at a time until it is exhausted and the algorithm cycles along the different operators until all are exhausted. The final change, which leads to our suggested version, denoted 'BRDV', adopts the first-accept strategy instead of the best-accept strategy.

For these experiments, the same construction heuristic and Local Search operators as in Zachariadis and Kiranoudis (2010) are used. In particular, the well-known and widely adopted Clarke-Wright algorithm (Clarke and Wright, 1964) is used to obtain initial solutions and the three Local Search operators are Relocate, Swap and 2-Opt (both intra- and inter-route).

Additionally, two more experiments are reported. In the first, the Local Search is started from a low-quality initial solution, which leads to a much longer Local Search phase with many more iterations until it ends up in a local minimum. The second additional experiment evaluates the impact of adding a fourth Local Search operator to the operator set.

6.1. Benchmark Instances

Although the classic 'ABEFMP' instance sets (Augerat et al., 1995; Christofides and Eilon, 1969; Fisher, 1994) and the Taillard instances (Taillard, 1993) are commonly used for CVRP benchmarking, they are less suited to evaluate the benefits of efficient local search since most of the instances in those sets are relatively small. We have therefore not considered these sets in our experiments.

The large instances proposed by Golden et al. (1998) (with $n =$ 240 to 420) and Li et al. (2005) (with $n =$ 560 to 1200) were not considered either. As pointed out by Uchoa et al. (2017), these instances are not suited for a good benchmark, because of their artificiality (concentric geometric figures and demands following very symmetric patterns) and their relative homogeneity (e.g., no clustered customers).

Because Efficient Local Search is most relevant for (very) large problem instances, we selected the 100 large 'X' instances proposed by Uchoa et al. (2017) and the 4 very large 'ZK' instances provided by Zachariadis and Kiranoudis (2010) for our benchmarking experiments. For our analyses, the X instances will be divided into two subsets, namely those with a number of nodes between 100 and 400 (58 instances), denoted X-S, and those with a number of nodes between 400 and 1000 (42 instances), denoted X-L. The 4 very large ZK instances have 3000 nodes.

6.2. Heap type and pruning

In a first experiment, we consider the original algorithm and look at the effects of (i) implementing its PQ data structure with a Fibonacci Heap or with a Binary Heap, and (ii) only adding improving moves or adding all possible moves to the PQ, i.e., pruning with threshold value $T = 0$ or not.

The solutions that are obtained with the four resulting versions are exactly the same across all 104 instances. Since only improving moves are accepted, the pruning does not lead to a different outcome. Table 4 shows the average CPU times of these four versions and the speedups. For the X-L instances, the CPU times are also plotted in Fig. 2.

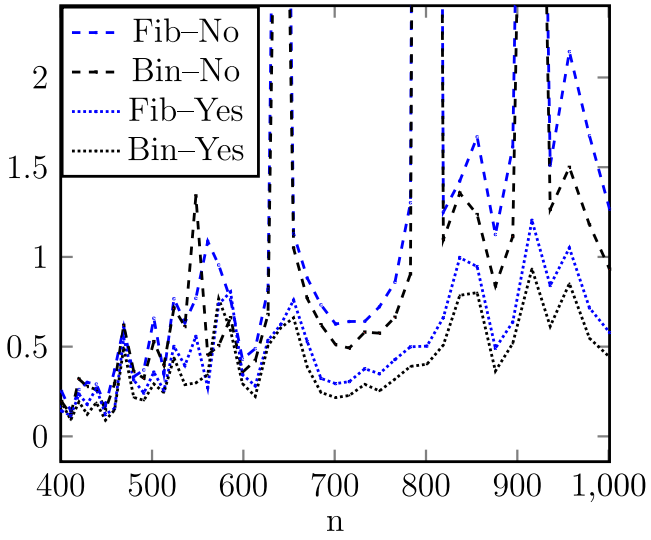

Fig. 2. CPU times (s) on the X-L instances for Fibonacci vs Binary Heap, with and without pruning ($T = 0$)

From Table 4, we can observe that merely changing the Heap type already has an important impact on the CPU times. Whereas the CPU time is slightly higher for the smaller X-S instances, it shows a significant decrease for the larger X-L instances, and a strong decrease for the ZK instances.

In the plot of Fig. 2, the Y-axis is cut off at 2.5 seconds. There are three instances for which the versions without pruning lead to even higher CPU times. These three instances have a particularly large fraction of non-improving moves, i.e., they are tightly constrained instances. Table 4 also shows the results when these three 'outliers' are not included. For the X-S and ZK instances, no such high volatility of CPU times was observed.

Next, it can be seen from Table 4 that the effect of pruning is very strong indeed. When the PQ no longer includes non-improving moves (that also need to be moved around during the Update phase to maintain the Heap property), the CPU times are dramatically reduced. Since the number of possible moves grows with instance sizes, it is no surprise that the effect of pruning becomes stronger with larger instance sizes.

For the outlier K-L instances, the CPU peaks are eliminated when pruning is adopted. This shows that with pruning, CPU times are much less affected by how tightly constrained a specific instance is, and are therefore much less volatile.

Overall, we can conclude from this first experiment that switching to a Binary Heap and adopting pruning reduces CPU times by a factor of two to three for the larger instances.

6.3. Local Search strategy

Although adopting a Binary Heap and pruning already offers a major speedup, the main motivation for doing this was only to enable further speedups by adjusting the SMD-based Local Search algorithm. These additional adjustments are (i) considering one Local Search operator at a time in a Variable Neighborhood Descent (VND) search strategy, and (ii) adopting a first-accept acceptance strategy (in the 'roughly sorted' Binary Heap). The three resulting implementations, 'Best-accept All', 'Best-accept VND', and 'First-accept VND' are evaluated here (all of them with binary heaps and pruning).

Table 4
CPU times: Fibonacci vs Binary Heap, with and without pruning ($T = 0$)

	X-S ($n < 400$)		X-L ($n \geq 400$)		X-L (no outliers)		ZK ($n = 3000$)	
Heap–Prune	CPU(ms)	Rel.	CPU(ms)	Rel.	CPU(ms)	Rel.	CPU(ms)	Rel.
Fib–No	110.2	100 %	1650.3	100%	821.3	100%	14481	100%
Bin–No	119.5	108.4%	1429.6	86.6%	673.5	86.2%	9255	63.9%
Fib–Yes	88.5	80.3%	487.4	29.5%	465.1	61.5%	6538	45.1%
Bin–Yes	78.1	70.8%	394.7	23.9%	375.3	49.8%	4895	33.8%

Table 5
Solution quality of different LS strategies

	X-S		X-L		ZK		Overall
Search strategy (s)	Cost	Δ_s	Cost	Δ_s	Cost	Δ_s	Δ_s
Best-accept All	41342.0	–	99947.9	–	4894.2	–	–
Best-accept VND	41338.2	-0.015%	99975.4	+0.03%	4893.9	-0.014%	+0.003%
First-accept VND	41344.5	+0.003%	99982.8	+0.04%	4894.2	-0.003%	+0.018%

Table 6
Average number of LS moves made by different search strategies

Search strategy	X-S	X-L	ZK
Best-accept All	19.8	56.7	209.5
Best-accept VND	21.8	59.9	217.5
First-accept VND	23.0	63.1	234.5

Solution quality

Before checking the CPU times and speedups, we first need to evaluate the impact of making these adjustments on the solution quality, since this leads the Local Search heuristic towards different solutions. Table 5 shows the average cost and the cost gaps across the X-S, X-L and ZK instances for the three different search strategies.

The reported cost gaps are the average of the individual relative gaps, calculated as follows: $\Delta_s = \frac{1}{|S|} \sum_{i \in S} \frac{C_i^s - C_i^0}{C_i^0}$, where s denotes the search strategy (0 being Best-accept All) and S denotes the set of instances being considered (i.e., X-S, X-L or ZK).

It can be concluded that the impact of the search strategy on solution quality is negligible, with an average overall gap of less than 0.02%. Compared to the (orginal) Best-accept All, the First-accept VND strategy has better solutions for 28 out of 104 instances, 56 that are worse, and a tie for 20 instances. The worst individual relative gap is only 0.38%.

CPU times

Now that the effect of the search strategy on solution quality has proven to be insignificant, we can consider the CPU times of the three different search strategies. In doing that, we will first consider the number of iterations (or moves) the various versions of the algorithm make before converging to their local optima. Table 6 shows these numbers. As can be expected, both the VND and the first-accept strategy lead to an increase in the (average) number of moves the SMD-based Local Search heuristic makes before converging. However, this increase is relatively limited.

The effect on the CPU times is very significant, however, as can be observed in Table 7. Switching to the VND (and only considering one operator at a time) reduces the CPU time overall with a factor 11.9. A further reduction to a factor 15.2 is obtained by adopting the first-accept strategy. Note that these implementations have a Binary Heap and pruning, which was already 2 to 3 times faster than the original ZK10 implementation.

It can be seen that the speedup obtained by adjusting the search strategy is stronger for the smaller instances where the speedup factors exceed 10. However, even for the largest ZK instances, an important speedup factor of more than 3 is achieved, without affecting solution quality.

Original implementation (ZK10) versus our suggested implementation (BRDV)

We can now make a summarizing comparison between the original ZK10 version (Fib–No Best-accept All) and our suggested BRDV implementation (Bin–Yes First-accept VND). Table 8 shows that ZK10 and BRDV achieve solutions of the same quality (average individual relative gap 0.018%), but BRDV does this in CPU times that are 29 times less on average.

6.4. Longer Local Search phases

In a next experiment, an alternative option is considered for the construction of initial solutions. Whereas the Clarke-Wright algorithm, denoted 'CW', produces medium to high-quality solutions, the alternative, denoted 'INS', produces low-quality solutions which are obtained by sequentially inserting nodes as they appear in the dataset.

Using Local Search on medium to high-quality initial solutions (CW) corresponds to what occurs in metaheuristics that perform quick Local Search runs, such as a Hybrid Genetic algorithm, or where the diversification method does not move far away from the local optimum. On the other hand, starting from low-quality initial solutions (INS) is also evaluated here because this leads to longer Local Search phases. This occurs in pure Local Search-based metaheuristics where the diversification is done by the Local Search itself, such as Guided Local Search and Tabu Search.

In the above experiments, we already observed the speedups of BRDV compared to ZK10 for relatively short Local Search phases (CW). This experiment now evaluates BRDV and ZK10 for longer Local Search phases (INS).

Table 9 shows the solution characteristics for the CW and INS versions of both ZK10 and BRDV. As can be expected, starting from the INS solutions leads to final solutions that are significantly worse than those obtained starting from the CW solutions. However, the purpose of this experiment is not to find better solutions. Instead, we want to observe the CPU time performance during longer Local Search phases. When looking at the number of moves made during the Local Search (also in Table 9), then indeed these are substantially higher for INS than for CW. In other words, the INS initial solution are so poor that many more moves need to be made before ending up in local optima (which are still relatively poor). Furthermore, since BRDV adopts a first-accept VND search strategy, the increase in the number of moves is stronger

Table 7
CPU times for the different search strategies

	X-S		X-L		ZK		Overall
Search strategy	CPU (ms)	Speedup	CPU (ms)	Speedup	CPU (ms)	Speedup	speedup
Best-accept All	78.1	1	394.7	1	4895	1	1
Best-accept VND	6.45	16.0	68.8	7.1	2151	2.3	11.9
First-accept VND	5.31	19.4	43.3	10.5	1563	3.2	15.2

Table 8
Comparison of original ZK10 and our BRDV

	X-S	X-L	ZK	Overall
ZK10 Cost	41342.1	99947.9	4894.2	63608.0
BRDV Cost	41344.5	99982.8	4894.2	63623.4
Avg. gap	+0.003%	+0.04%	-0.003%	+0.018%
ZK10 CPU (ms)	110.2	1650.3	14481	1284.9
BRDV CPU (ms)	5.3	43.3	1563	80.6
Avg. speedup	25.1	36.9	9.3	29.2

Table 9
Solution characteristics of different versions

	Cost			Moves		
Version	X-S	X-L	ZK	X-S	X-L	ZK
ZK10-CW	41342.0	99947.9	4894.2	19.8	56.7	209.5
ZK10-INS	42707.2	102938.1	5239.2	293.9	973.3	5558.3
BRDV-CW	41344.5	99982.8	4894.2	23.0	63.1	234.5
BRDV-INS	43101.9	103805.5	5282.5	427.6	1468.7	7905.3

Table 10
CPU time and CPU time per iteration (TPI) for the different versions

	CPU (ms)			TPI (ms)		
Version	X-S	X-L	ZK	X-S	X-L	ZK
ZK10-CW	110.2	1650.3	14481	5.13	25.1	69.5
ZK10-INS	2739.4	42958.4	653452	8.51	36.4	117.5
BRDV-CW	5.3	43.3	1563	0.24	0.66	6.91
BRDV-INS	23.6	362.8	21841	0.05	0.21	2.76

than for ZK10, which adopts the best-of-all-operators search strategy.

Table 10 displays the CPU times of these experiments. Of course, since the Local Search phase has more moves, the CPU times are also longer for both ZK10 and BRDV. To be able to make a comparison, we therefore also report the CPU time *per iteration* (denoted TPI), which is the CPU time divided by the number of moves being made during the Local Search.

When we compare these TPI, another advantage of BRDV over ZK10 can be observed. While the CPU time per iteration increases for ZK10 when the Local Search phase takes longer, the TPI for BRDV decreases with the length of the Local Search. This is illustrated in Fig. 3 for the X-L instances, where we can also observe that the TPI behavior for ZK10 (left) is much more erratic than for BRDV (right).

This experiment illustrates that the drawbacks of the original ZK10 implementation that we highlighted in Section 4 are aggravated while at the same time the advantages of the adjustments we suggest in Section 5 become more pronounced in this situation. With CW starting solutions, we already saw that BRDV was about 29 times faster than ZK10 on average, but for the INS solutions, the average speedup increases to no less than 120.

Initialization Overhead

One of the advantages of BRDV over ZK10 that is worthwhile pointing out again, is that BRDV can efficiently discard and reinitialize its PQ thanks to the simple Binary Heap structure. To prove the reduced initialization overhead, we ran both ZK10 and BRDV on the very large ZK instances, from both the CW and the INS initial solution, but stopped the algorithm as soon as the first feasible SMD is found. Starting from the poor INS solution obviously leads to larger initial heap sizes and thus longer initialization times.

Table 11
Heap initialization times for ZK instances (ms)

	ZK10	BRDV	Speedup
INS	2294	118	19.6
CW	2120	69	31.2

Table 12
Solution characteristics of different versions

	Cost			Moves		
Version	X-S	X-L	ZK	X-S	X-L	ZK
ZK10-3Ops	41342.0	99947.9	4894.2	19.8	56.7	209.5
ZK10-4Ops	41340.8	99925.8	4893.8	19.9	57.0	212.5
BRDV-3Ops	41344.5	99982.8	4894.2	23.0	63.1	234.5
BRDV-4Ops	41343.6	99982.7	4894.2	23.0	63.1	236.5

For ZK10, all three neighborhoods are initialized together, so three heaps are constructed. For BRDV, only a single operator is used at a time. Since the 2-Opt neighborhood proved to be the most expensive in terms of initialization, its initialization time is reported here in Table 11. If both versions would be equally efficient in terms of initialization, the initialization time for ZK10 would therefore be about 3 times longer.

Instead, Table 11 shows that BRDV initializes about 20 times faster than ZK10 (INS). When heap sizes are smaller (CW), the initialization cost of BRDV is also much smaller, whereas the ZK10 initialization cost is barely affected. As a result, the speedup factor of BRDV over ZK10 increases to over 30 for the initialization. This insight is highly relevant for modern metaheuristics that often intensify the search near high-quality solutions and thus have short search phases and more frequent initializations.

If we compare the BRDV initialization overhead of 69 and 118 ms for the ZK instances with the total CPU times of 1563 and 21841 ms (see Table 10), then we can conclude that the initialization overhead is only a small penalty to be paid for being able to perform all consecutive Local Search iterations much more quickly.

6.5. Flexibility

In all previous experiments, three Local Search operators are used. To test the extendibility of the algorithms, we also implemented a fourth operator, the 2-Relocate or (2,0)-exchange. This operator takes a sequence of two nodes and moves it to a new location. It also has a cardinality of $O(n^2)$, but it is slightly more expensive to evaluate and to check for feasibility than regular Relocate. Further, because the operator performs a one-way transfer of two nodes, the odds of finding feasible moves are smaller.

Table 12 shows how this impacts the results. For both ZK10 and our BRDV, the solutions obtained after adding the fourth operator

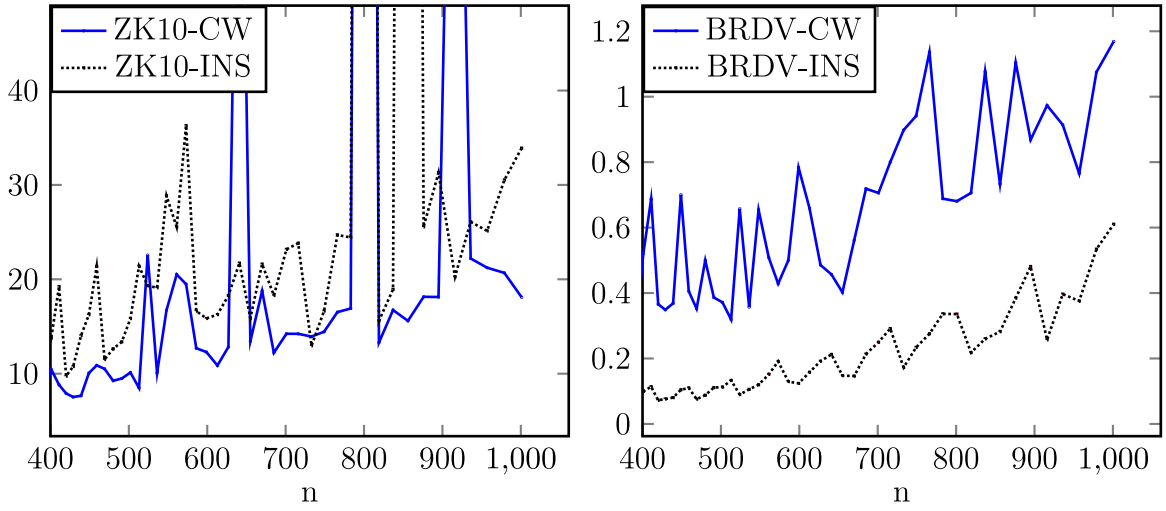

Fig. 3. CPU time per iteration (ms) on the X-L instances

Table 13
CPU times and CPU time per iteration (TPI) for the different versions

	CPU (ms)			TPI (ms)		
Version	X-S	X-L	ZK	X-S	X-L	ZK
ZK10-3Ops	110.2	1650.3	14481	5.13	25.1	69.5
ZK10-4Ops	159.0	4605.6	22147	6.93	61.1	104.6
BRDV-3Ops	5.3	43.3	1563	0.24	0.66	6.91
BRDV-4Ops	5.8	54.0	1853	0.26	0.81	8.09

are only slightly better and the number of moves is only slightly higher.

In terms of CPU times, however, the impact of adding the fourth operator is much stronger, as can be seen in Table 13. ZK10 is strongly affected by this additional operator since the CPU time per iteration increases very strongly, especially for the X-L instances where they more than double. Analysis showed that the (2,0)-Relocate operator accounts for less than 5% of the executed moves, because it has a low feasibility rate. This forces ZK10 to perform many more Heap extractions (and subsequent reinsertions) before a feasible move is found. This experiment confirms another weakness of ZK10: operators with low success rates have a large negative impact on the total performance of the algorithm. This means that carefully composing the operator set is crucial for ZK10 performance, where only operators that perform well under all circumstances should be selected.

On the other hand, BRDV shows only a 18% increase in CPU time per iteration because the overhead of a low-success-rate operator is limited to an additional initialization phase per cycle. So, despite the adoption of an expensive operator with low success rate, BRDV barely suffers any decrease in performance. Thus, BRDV is much more robust than ZK10 as it dependence on the set of operators is much less pronounced.

Fig. 4 plots the TPI for the X-L instances. It can be observed that the TPI are very volatile for ZK10, with peaks corresponding to tightly constrained instances with many non-improving moves. This corresponds to the peaks we observed in the CPU time plots of Fig. 2. As explained above, the (2,0)-Relocate operator has a low feasibility rate and thus aggravates this behavior. Indeed, when this fourth operator is added, the frequency and magnitude of the peaks in the TPI plot increases.

For BRDV, on the other hand, the right plot of Fig. 4 again illustrates that CPU times and TPI for BRDV are not only some orders of magnitude smaller, but also much less volatile (mainly because of pruning).

Apart from the peaks for ZK10, we can see in Fig. 4 that both ZK10 and BRDV scale well with instance size. Both implementations of the SMD framework are effective at improving the scaling of Local Search algorithms. Zachariadis and Kiranoudis (2010) note that the SMD-based Local Search with quadratic operators scales as $O(n \log n)$ and our results confirm this.

Overall, we can conclude that BRDV outperforms ZK10 significantly. In terms of *accuracy*, the difference in performance is negligible, while in terms of *speed, simplicity* and *flexibility*, our experimental results show that BRDV is the preferred option.

7. Conclusion

In this paper we have taken a critical look at the Static Move Descriptors (SMDs), a framework introduced by Zachariadis and Kiranoudis (2010) for improving the performance of Local Search algorithms. Although they report that the SMD framework reduces the complexity of some commonly used quadratic complexity Local Search operators to an almost linearithmic complexity, we have identified different aspects in which the implementation can be significantly improved and further speedups can be obtained.

First, we have simplified the design by eliminating the complex Fibonacci Heap, which is well-known for its implementational complexity, having high memory overhead and being very inflexible in usage. We have replaced it with a Binary Heap. The use of a supporting matrix (that indicates where specific SMDs can be found in the Heap) allows us to strongly prune the Heap. Non-improving moves are no longer added to the Heap and therefore no longer slow down the Update phase of the Local Search procedure. Pruning thus leads to a strong reduction of CPU times. Furthermore, it also leads to less volatility of the CPU times.

Second, we have adopted a VND search strategy that cycles through Local Search operators instead of simultaneously evaluating all of them. This further simplifies the implementation of the algorithm by removing the cross-operator effects in updating the Priority Queue. Our experiments show that this change does not affect solution quality significantly, whereas it does strongly improve the performance of the algorithm when additional operators are used. The original implementation by Zachariadis and Kiranoudis (2010) was shown to be vulnerable to the choice of operator set, while our suggested implementation suffers little loss in performance when additional operators, even with low feasibility rates, are added.

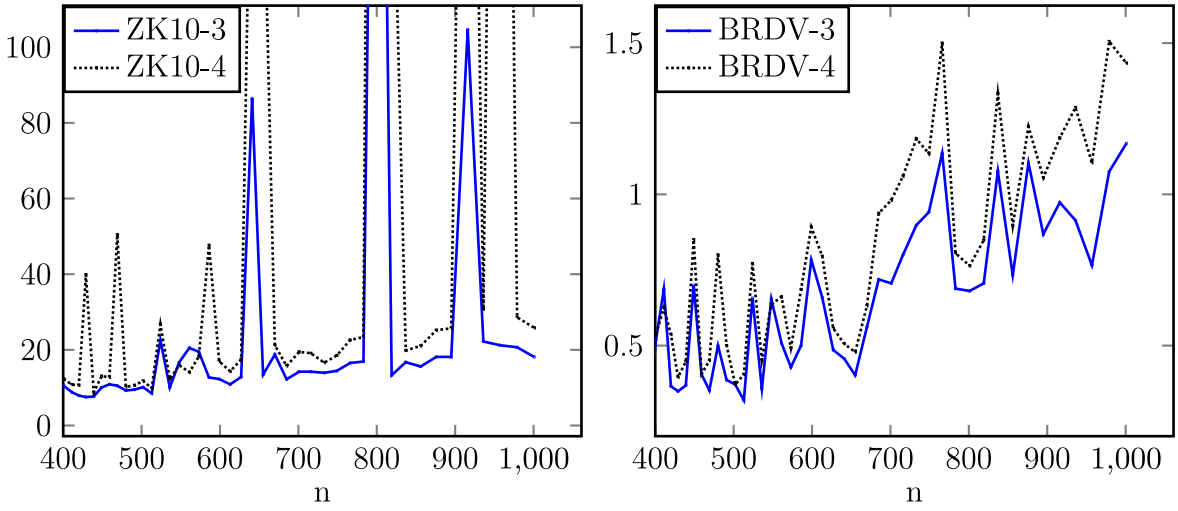

Fig. 4. CPU time per iteration (ms) on the X-L instances

Lastly, we made use of the Binary Heap's flexible design to implement a heuristic acceptance strategy. The search can find high-quality feasible moves without the use of Heap operations. However, this search no longer guarantees to return the best improving move; instead, we are using a first-accept strategy that uses the underlying design of the Binary Heap to improve the odds of finding high quality moves. We showed in the benchmark experiments that this adjustment does not significantly affect the overall solution quality either.

In summary, our newly proposed implementation is simpler to implement and greatly outperforms the original in terms of speed and flexibility, with an overall speedup factor around 30 and cost difference of less than 0.02%.

In terms of further research, two opportunities stand out. First, by implementing the SMD framework in many types of hybrid metaheuristics, the performance of state-of-the-art heuristic VRP solvers can be improved. Second, the SMD framework can be extended to other combinatorial problems for which Local Search based (meta)heuristics are being developed. We hope that our proposals for a simplified, flexible and fast SMD implementation will encourage fellow researchers to adopt it in their solution approaches.

Acknowledgement

This research is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0234.

Supplementary material

Supplementary material associated with this article can be found, in the online version, at 10.1016/j.cor.2018.01.006.

References

Augerat, P., Belenguer, J.M., Benavent, E., Corberán, A., Naddef, D., Rinaldi, G., 1995. Computational results with a branch and cut code for the capacitated vehicle routing problem. Rapport de recherche. Institut d'informatique et de mathématiques appliquées de Grenoble.

Bentley, J.L., 1990. Experiments on traveling salesman heuristics. In: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, pp. 91–99.

Christofides, N., Eilon, S., 1969. An algorithm for the vehicle-dispatching problem. Operations Research 20 (3), 309–318.

Clarke, G.U., Wright, J.W., 1964. Scheduling of vehicles from a central depot to a number of delivery points. Operations Research 12 (4), 568–581.

Cordeau, J.-F., Gendreau, M., Laporte, G., Potvin, J.-Y., Semet, F., 2002. A guide to vehicle routing heuristics. Journal of the Operational Research Society 53 (5), 512–522.

Fang, Z., Tu, W., Li, Q., Shaw, S.-L., Chen, S., Chen, B.Y., 2013. A voronoi neighborhood-based search heuristic for distance/capacity constrained very large vehicle routing problems. International Journal of Geographical Information Science 27 (4), 741–764.

Fisher, M.L., 1994. Optimal solution of vehicle routing problems using minimum k-trees. Operations Research 42 (4), 626–642.

Fredman, M.L., Tarjan, R.E., 1987. Fibonacci heaps and their uses in improved network optimization algorithms. Journal of the ACM 34 (3), 596–615.

Glover, F., 1989. Tabu search – part i. ORSA Journal on Computing 1 (3), 190–206.

Golden, B.L., Wasil, E.A., Kelly, J.P., Chao, I.-M., 1998. The impact of metaheuristics on solving the vehicle routing problem: algorithms, problem sets, and computational results. In: Crainic, T., Laporte, G. (Eds.), Fleet management and logistics. Springer, Boston, MA, pp. 33–56.

Irnich, S., Funke, B., Grünert, T., 2006. Sequential search and its application to vehicle-routing problems. Computers and Operations Research 33 (8), 2405–2429.

Johnson, D.S., McGeoch, L.A., 1997. The traveling salesman problem: A case study in local optimization. Local Search in Combinatorial Optimization 1, 215–310.

Li, F., Golden, B., Wasil, E., 2005. Very large-scale vehicle routing: New test problems, algorithms, and results. Computers and Operations Research 32 (5), 1165–1179.

Lin, S., Kernighan, B.W., 1973. An effective heuristic algorithm for the traveling-salesman problem. Operations Research 21 (2), 498–516.

Martin, O., Otto, S.W., Felten, E.W., 1992. Large-step markov chains for the tsp incorporating local search heuristics. Operations Research Letters 11 (4), 219–224.

Mladenović, N., Hansen, P., 1997. Variable neighborhood search. Computers and Operations Research 24 (11), 1097–1100.

Nagata, Y., Bräysy, O., 2008. Efficient local search limitation strategies for vehicle routing problems. In: van Hemert, J., Cotta, C. (Eds.), Evolutionary Computation in Combinatorial Optimization. Springer, Berlin, Heidelberg, pp. 48–60.

Norman, M.G., Moscato, P., 1989. A competitive and cooperative approach to complex combinatorial search. Technical Report Caltech Concurrent Computation Program. California Institute of Technology.

Reinelt, G., 1994. The traveling salesman: computational solutions for TSP applications. Lecture Notes in Computer Science, 840. Springer-Verlag Berlin Heidelberg.

Taillard, É., 1993. Parallel iterative search methods for vehicle routing problems. Networks 23 (8), 661–673.

Toth, P., Vigo, D., 2003. The granular tabu search and its application to the vehicle-routing problem. INFORMS Journal on Computing 15 (4), 333–346.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., Subramanian, A., 2017. New benchmark instances for the capacitated vehicle routing problem. European Journal of Operational Research 257 (3), 845–858.

Voudouris, C., Tsang, E., 1999. Guided local search and its application to the traveling salesman problem. European Journal of Operational Research 113 (2), 469–499.

Vuillemin, J., 1978. A data structure for manipulating priority queues. Communications of the ACM 21 (4), 309–315.

Williams, J.W.J., 1964. Algorithm-232-heapsort. Communications of the ACM 7 (6), 347–348.

Zachariadis, E.E., Kiranoudis, C.T., 2010. A strategy for reducing the computational complexity of local search-based methods for the vehicle routing problem. Computers and Operations Research 37 (12), 2089–2105.