Neighborhood Enumeration in Local Search Metaheuristics

Michiel Van Lancker$^{1[0000-0002-2417-9928]}$, Greet Vanden Berghe$^{1[0000-0002-0275-5568]}$, and Tony Wauters$^{1[0000-0002-1014-6340]}$

KU Leuven, Department of Computer Science, CODeS, Belgium
michiel.vanlancker@cs.kuleuven.be

Abstract. Neighborhood enumeration is a fundamental concept in the design of metaheuristics. It is often the only principle of intensification present in a metaheuristic and serves as the basis for various metaheuristics. Given its importance, it is surprising that academic reporting on enumeration strategies lacks the necessary information to enable reproducible algorithms. One aspect of neighborhood enumeration in particular has been under the radar of researchers: the order in which neighbors are enumerated. In this paper, we introduce a versatile formalism for neighborhoods which makes explicit enumeration order and we analyse the impact of enumeration order on the outcome of search procedures with a small set of benchmark problems.

Keywords: Enumeration Order · Local Search · Neighborhoods · Metaheuristics

1 Introduction

Metaheuristics have gained a somewhat ambiguous reputation over the years. On the one hand they are lauded for their useful characteristics in practical applications: metaheuristics are problem-independent, general optimization algorithms. They are not only capable of being reused over a wide variety of problems, but many are also anytime algorithms which maintain a valid solution throughout the entire search process. Furthermore, they can be implemented in a highly configurable fashion, enabling automated algorithm design and parameter tuning. This results in algorithm templates that can be instantiated and automatically tailored to solve specific problems or instances. On the other hand, metaheuristics research has not yet reached the scientific rigor found in other fields, with many researchers tending to focus on algorithmic efficiency – or worse, novelty – rather than algorithmic understanding. This has led to a large variety of algorithms which differ only slightly from one another or are identical except for the terminology used [6].

While big steps have been made – especially during the last two decades – to transform the field into a more academic one with rigorous scientific discipline built on formalized concepts, many publications continue to operate in the sphere of problem-solving rather than algorithmic understanding. This resulted in many metaheuristics, but few insights. Nevertheless, efforts are underway to mature the discipline. Notable examples of this are (i) the endorsement by the Journal of Heuristics of the view that nature should no longer serve as an explicit inspiration for “novel” metaheuristics, (ii) the recognition of the need for white-box algorithm implementations, preferably described in a purely functional style[8], (iii) the call for rigorous evaluation and testing practices, and (iv) the active promotion of a view of what metaheuristics research ought to be [7].

In this paper we zoom in on one specific component of metaheuristics: the concept of local search neighborhoods. We argue that a gap exists between common theoretical neighborhood definitions and how they are implemented in practice. In other words: we argue that neighborhoods are not implemented according to the white-box principle, preventing algorithm reproducibility and standardized evaluation.

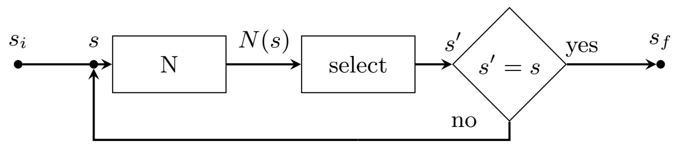


Fig. 1: Iterative improvement consists of repeatedly applying an improving operation to the solution.

Many optimization techniques can be considered instantiations of the iterative improvement-scheme (II-scheme), the distinction between which results from the interaction between their instantiating components. The II-scheme itself is straightforward: starting from an initial incumbent solution the search process consists of a series of iterations, where in each iteration a selection criterion selects an alternative solution of better quality than the incumbent solution. If a better solution is found, it is accepted as the new incumbent solution. This process is repeated until no improving solution can be found.

Most, if not all, single-solution (local search) metaheuristics can be mapped to the II-scheme shown in Fig. 1. In local search metaheuristics, a set of alternative solutions – called the neighborhood of the incumbent solution – is constructed by making a set of small modifications to the initial solution. The difference between various metaheuristics yet again results from the differing interactions between their constituent components. Which components to consider and how to combine them is the responsibility of the (human) algorithm designer. Some design choices which require some thought include how to generate a neighborhood of the incumbent solution, which solution to select from the neighborhood and how to compare solutions. It is well known that good neighborhood design is of crucial importance when it comes to the efficiency of a local search (meta)heuristic, as poor design choices can have a large negative impact on both the runtime and the behavior of the algorithm. Choosing an appropriate selection criterion is equally important, as it strongly determines the behavior of the search and can have a dramatic impact on runtime.

Given the importance of these two design questions, it is fair to assume that reporting on metaheuristic algorithms should include complete information concerning which choices were made and, ideally, why. However, at present the opposite situation is the case: many publications concerning metaheuristics do not report neighborhood specifications to the level of detail required to facilitate reproducibility. Most obvious is the lack of information concerning how operators in a neighborhood are enumerated. This information is crucial if an order-dependent selection criterion is used and, indeed, virtually all deterministic selection criteria are order-dependent. A second, more subtle issue is the lack of information concerning which operators are a priori included in a neighborhood.

Our contributions in this paper are threefold. First, we introduce a formalism for the concept of a neighborhood as used in local search, which makes explicit the enumeration order. Second, we analyze the effect of enumeration order on the outcome of a search procedure through a series of computational experiments. Third and finally, we provide several examples of the expressiveness of the proposed formalism.

The remainder of this paper is structured as follows. Beginning with the concept of iterative improvement, Section 2 introduces neighborhoods and selection methods and provides a brief overview of how neighborhood enumeration is commonly reported in metaheuristics research. In Section 3 we introduce a formalism for neighborhood enumeration. Section 4 then analyzes the effect of enumeration order on the outcome of a search procedure on a set of benchmark instances. Several examples demonstrating the flexibility of the formalism are given in Section 5. Section 6 then concludes the paper.

2 Iterative improvement, neighborhoods & selection

In this section we review the relationship between iterative improvement, neighborhoods and selection criteria. In doing so we identify a gap between the commonly used definitions for the aforementioned concepts and the components required to implement the II-scheme, resulting in an incomplete algorithm specification. The section ends with a brief analysis of how neighborhood enumeration is currently reported on in the academic literature.

To approach local search metaheuristics as instantiations of iterative improvement, strict definitions are required for the instantiating components. Consider the II-scheme shown in Fig. 1. It is clear that an instantiation of the scheme is determined by three factors, namely: a neighborhood generation function N, a neighbor selection function *select* and a condition to test whether or not the search has ended. Since we are only interested in improvement methods, the ending condition can be excluded from the analysis and thus the behavior of a deterministic II-procedure is dependent on only two functions: the neighborhood function $N : S \to \mathcal{P}(S)$ and the selection function $\mathit{select} : \mathcal{P}(S) \to S$. As is clear from its type, the neighborhood-function must map the incumbent solution to a set of alternative solutions, resulting in the common introductory definition of a neighborhood[2,9]:

Definition 1. *A neighborhood function is a mapping $N : S \to \mathcal{P}(s)$ which assigns to each solution $s \in S$ a set of solutions $N(s) \subseteq S$. The members of $N(s)$ are called neighbors of s.*

In the context of local search however, a different definition is sometimes used to more adequately capture the notion of operators and locality. A neighborhood is defined in terms of a relation – the local search operator – on S:

Definition 2. *The R-neighborhood $N_R(s)$ of solution $s \in S$ is the neighborhood defined by the relation R on S, $N_R(s) = \{s' \in S : sRs'\}$.*

The second component of the II-scheme is a selection function, which returns a single neighbor from the neighborhood it receives as input. We refrain from giving a general definition of selection criteria, but note that any selection criterion must be a function of type *select* : $\mathcal{P}(S) \to S$ and we shall examine how well two of the most popular selection criteria adhere to this definition.

The first criterion we will consider is the *argmin* selection criterion (Eq. 1), which selects the best solution from the neighborhood. Next is the $firstmin$ selection criterion (Eq. 2), which selects the first improving solution from the neighborhood. More formal definitions of both criteria are as follows:

$$\operatorname*{argmin}_{s' \in N(s)} c(s') := \{s' \mid \forall s'' \in N(s) : c(s') \leq c(s'')\} \tag{1}$$

$$\operatorname*{firstmin}_{s_i \in N(s)} c(s_i) := \{s_i \in N_{\downarrow}(s) \mid \forall s_j \in N_{\downarrow}(s) : i \leq j\} \tag{2}$$

$$\text{where } N_{\downarrow}(s) := \{s' \in N(s) \mid c(s') \leq c(s)\}$$

Note that this definition of $argmin$ does not have the required type: if multiple solutions have the best objective value, all of these solutions will be returned. As such the definition specifies a function of type $\mathcal{P}(S) \to \mathcal{P}(S)$ and a modification, a tie-breaker, is needed to acquire the required type. Common tie-breakers are to select the first, the last or a random solution from the set of most improving solutions. Only the first two of these tie-breakers are deterministic and both of these are order-dependent.

For $firstmin$, the impact of order is obvious. To be able to return the first improving neighbor an order must be imposed on neighborhood N. In the worst case all solutions in the neighborhood are improving and thus each possible ordering of N will return a different solution. It follows that the neighborhood enumeration order must be known to achieve a full specification of a single iteration in the II-scheme. While the effect of enumeration order on the outcome of a single iteration is generally fairly limited, this is less so when considering the entire II-scheme. Since every iteration starts from the outcome of the previous iteration, the effect of an enumeration order compounds throughout the whole search.

Given the effect of enumeration order on the outcome of a search procedure, it is somewhat surprising that most publications do not contain any information about it. Many publications only describe neighborhoods in terms of their local search operator. A notable exception is [5], in which the authors not only mention the use of a random enumeration order, but also published the complete source code of their implementation.

Finally, let us examine some open-source implementations of metaheuristics and see how neighborhood enumeration is implemented in them. The following two implementations serve as an example: the Java Metaheuristics Search Framework(JAMES)[4] and the suite of metaheuristic frameworks PARADISEO[3]. In JAMES it is possible for users to implement custom neighborhoods through a neighborhood- and operator-interface, but imposing orders on neighborhood sets through an interface is not possible and must be programmed from scratch by the user. When querying the full neighborhood, an eagerly constructed list of operators is returned. In PARADISEO, users can implement custom neighborhoods in a similar fashion, though here order *is* made explicit by means of an iterator-interface. Querying the full neighborhood returns a lazy iterator over the neighborhood. Furthermore, neighborhoods can be linked together into new neighborhoods.

Before continuing with the next section, we end this section with an example of what kind of issues arise when neighborhood definitions are incomplete. We will illustrate these issues by considering the *TwoOpt*-operator for the Traveling Salesperson Problem (TSP). Let $C = \{c_1, \ldots, c_n\}$ be a set of points on the Euclidean plane representing cities and let $d : C \times C \to \mathbb{N}$ be the distance between two cities. Then, the goal of the TSP is to find the shortest tour which visits each city once. Let permutation $\pi \in \Pi$ represent a tour through all cities in C and let $I_\pi = \{1, \ldots, n\}$ be the index set of π. Element $\pi_i \in \pi$, where $i \in I_\pi$, represents the i^{th} visited city in the tour. The objective value $c(\pi)$ is computed with (Eq. 3).

$$c(\pi) = \sum_{i \in J_\pi} d(\pi_i, \pi_{i+1}) + d(\pi_n, \pi_1) \tag{3}$$

Applying the *TwoOpt*-operator to a solution for the TSP equals swapping two edges in the tour, or equivalently, inverting a subsequence of the solution representation π. The operator takes as input the current tour and two indices $i, j \in I_\pi$. To implement a function to generate the *TwoOpt* neighborhood, a double for loop is typically used. A naive implementation would generate neighbors for all possible pairs $(i, j) \in I_\pi^2$. This is however redundant: *TwoOpt* is a symmetric operator, thus a more efficient implementation would only generate neighbors for the pairs (i, j) for which $i < j$, as these are sufficient to cover the whole neighborhood. Aside from redundancy, which is unwanted but not problematic, if it is unclear which moves are included in the neighborhood and which aren't, any order-dependent selection function can cause diverging search outcomes for two neighborhoods that "look" the same.

3 Neighborhood Enumeration

The previous section provided an introduction to how common definitions of neighborhoods, selection criteria and local optima are not sufficiently exact from an implementation perspective and how this in turn results in an incomplete algorithm specification. As suggested by the *TwoOpt*-example, there are two pieces of information missing from Definition 2: how many (i.e. which) solutions belong to a neighborhood and the order in which these solutions are visited. In this section we present an alternative definition of a neighborhood function, which makes concrete the aforementioned information. The purpose of the definition being introduced is to capture the structure of a local search neighborhoods in such a way that the required implementation steps become clear.

Consider the neighborhood $N_M(s) \subseteq S$. For all $s_\phi \in N_M(s)$ we know that we can move from s to s_ϕ. Let $m_\phi : s \mapsto s_\phi$ be the function representing the move from s to s_i. There are $|N_M(s)|$ such functions, one for each $s_\phi \in N_M(s)$. Thus we can define the neighborhood as $N_M(s) = \{m_i(s)\}_{i \in \Phi}$, where Φ is an index over $N_M(s)$. Note that if we provide a constructor function $M : \Phi \to (S \to S)$, we can construct function $m_\phi : S \to S$ by evaluating $M(\phi)$. Given an iterator T over Φ, the first neighbor in the neighborhood can be generated as follows: take the first element ϕ from the iterator, call constructor M to construct move m_ϕ, and apply $m_\phi(s)$. To generate subsequent neighbors, take the next element from T and repeat the process until all elements from T have been consumed. The neighborhood can then be defined as:

Definition 3. *A neighborhood $N_M(s, T)$ is the set of solutions constructed by applying each function $m_\phi : S \to S$ for each $\phi \in T$ to s, where T is an iterator over (a subset of) Φ_M, the parameter space of operator $M : \Phi_M \to (S \to S)$. As T is ordered, a neighborhood enumeration is uniquely defined by the triple (s, M, T).*

This definition results in several extra design questions concerning the parameter space used in a neighborhood. While neighborhood design typically only considers the choice of operator, now two more design choices must be made: *which operator parameters should be included in a neighborhood* and *in what order should they be generated*? In the next two sections we take a more detailed look at what options are available regarding these choices.

3.1 Parameter spaces

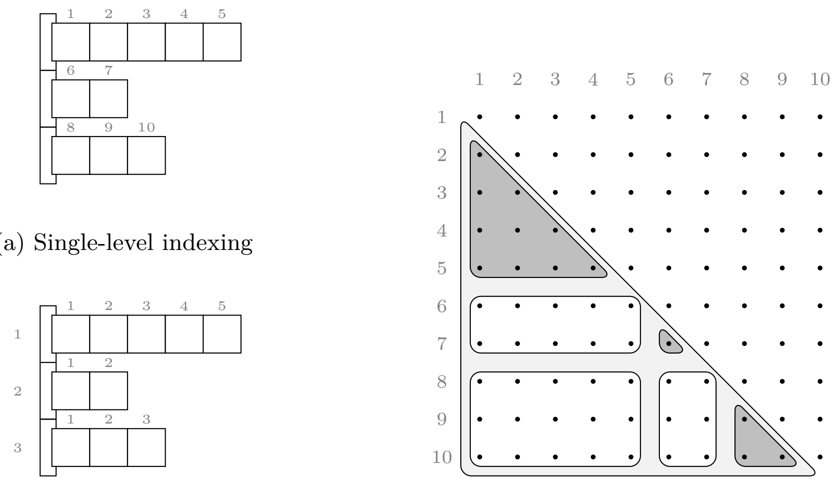
When considering operators, we make three observations: First, the parameter space Φ_M of operator M is dependent on the solution representation. Second, it is dependent on functional properties of its operator. Third, any subset of the parameter space can be used to generate a neighborhood.

Consider the TSP and three operators defined in Table 1. All three operators are quadratic and, since solution representation π is unconstrained, each operator can take any $(i, j) \in I_\pi^2$ as input, where I_π^2 is the Cartesian product of I_π.

However, depending on the operator, we can eliminate some elements from I_π^2. For example, we know that the *Swap*- and *TwoOpt*-operators are symmetric operators and thus parameter combinations (i, j) and (j, i) will construct the same moves. Furthermore, for all three operators it is the case that no matter the state of the incumbent solution, parameter (i, i) will construct the identity move.

Operator	**Parameter space**	**Neighbor Relation**
Swap	$(i, j) \in I_\pi^2 : i < j$	$\pi'_i = \pi_j \wedge \pi'_j = \pi_i$
TwoOpt	$(i, j) \in I_\pi^2 : i < j$	$\forall k \in [0, j-i] : \pi'_{i+k} = \pi_{j-k}$
Shift	$(i, j) \in I_\pi^2 : i \neq j$	$\pi'_j = \pi_i$ $\pi'_k = \begin{cases} \forall k \in [i+1, j] : \pi_{k-1}, & \text{if } i < j \\ \forall k \in [j, i-1] : \pi_{k+1} & \text{otherwise} \end{cases}$

Table 1: Definitions of the Swap, 2opt and Shift operators and their respective parameter spaces.



(a) Single-level indexing

(b) Multi-level indexing

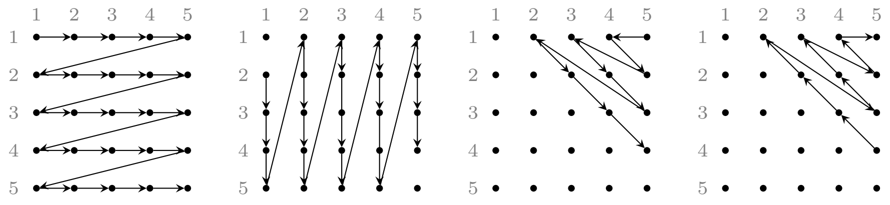
(c) Parameter space

Fig. 2: The interpretation of operators and their respective parameter spaces is dependent on the indexing system used.

The importance of the chosen solution representation and index set becomes more obvious when we consider more complex solution representations. Instead of permutation π, consider an ordered set of permutations ρ. To implement a neighborhood for this structure, we require an index set to base our parameter space on. Looking at Fig. 2 it is clear that multiple options are available. We can use a single-level, linear index – like we did for permutation π – where every position in the representation is represented by a single integer: its position in the overall element order. Alternatively, a multi-level index can be used, where every position in the representation is represented by two integers: the position of the permutation in the set and the position within the permutation. Fig. 2c illustrates the correspondence between the parameter spaces of a symmetric operator using single-level and multi-level indexing. In light grey is the parameter space based on the single-level index. In dark grey are parameters corresponding to moves that operate inside a permutation of the set of permutations, using the multi-level index. Similarly, in white are the parameters corresponding to inter-permutation moves when using the multi-level index.

3.2 Enumeration order

The final step is to impose an order on the defined parameters. Given a set of parameters of size n, there are $n!$ ways to impose an order. However, some of these orders are more interesting than others. Of special interest are those that follow particular patterns, which can usually be efficiently implemented as an iterator which generates the parameter sequence lazily. Some of these patterned sequences can be interpreted as prioritizing certain moves: consider the *TwoOpt* operator for the TSP and assume that we are using the $firstmin$ selection function. If *TwoOpt* moves are enumerated according to the scheme $(1,2),(1,3),(1,4),\ldots,$ the beginning position of the subsequence is considered more important than that of the end. Similarly scheme $(2,1),(3,1),(4,1),...$ deems the end position more important. Finally, scheme $(1,2),(2,3),(3,4),\ldots$ prioritizes moves corresponding to shorter subsequence inversions. Such semantic distinctions can help algorithm designers gain insights in the workings of their algorithms.



(a) I_π^2 (b) $I_\pi^2 : i \neq j$ (c) $I_\pi^2 : i < j$ (d) $I_\pi^2 : i < j$

Fig. 3: Various iterators over I_π^2.

Four iterators for quadratic operators are shown in Fig. 3 which differ in terms of their parameters included, order and direction. Fig. 3a illustrates an iterator over the full parameter space – the Cartesian product I_π^2 – ordered along the rows. Fig. 3b is ordered along the columns and eliminates parameters $(i, i) \in I_\pi^2$. Figures 3c and 3d are both ordered along the diagonals and eliminate parameters $(i, j) \in I_\pi^2$ for which $i \geq j$, but they differ in the direction they take.

4 Experimental Evaluation

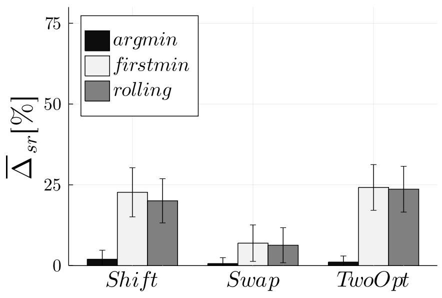
To evaluate the influence of enumeration order on search procedures we consider a search procedure to be a program of type $\mathit{solve} : S \rightarrow S$. This program takes an initial solution s_i and returns a local optimum as final solution s_f. We refer to the change induced on s_i by *solve* as $\Delta_s = |C| - |e_c| - 1$, where $|C|$ is the number of cities and $|e_c|$ is the number of edges s_i and s_f have in common. In a similar fashion, we refer to the difference between the objective value of s_i and s_f as $\Delta_v = c(s_f) - c(s_i)$ and its runtime as Δ_t.

Constructive	**Select**	**Operator**	***Order***	***Direction***
random	argmin	Swap	Column	Forward
greedy	firstmin	TwoOpt	Row	Reverse
	rolling	Shift	Diagonal	

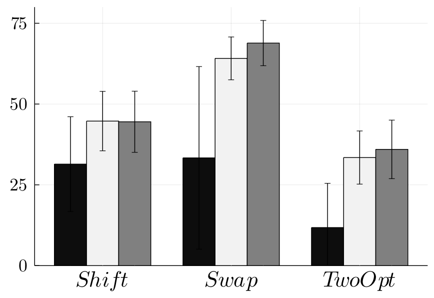
Table 2: The set of algorithm design parameters considered when experimentally evaluating enumeration order.

To study the impact of enumeration order on the search we compare Δ_s, Δ_v and Δ_t for *solve* procedures instantiated with different design parameters. Table 2 lists these design parameters. As the first three columns do no influence enumeration order, they can be considered design parameters resulting in different "contexts" in which the effect of enumeration order is evaluated. These parameters serve to broaden the scope of our analysis. All of the included design parameters have been defined in earlier sections of this paper, except for the selection function *rolling*. This selection function is an adapted version of $firstmin$. Whereas $firstmin$ begins from scratch in the next iteration after selecting the first improving neighbor $s_i = m_i(s)$, *rolling* will continue enumerating from its current position. The last two columns determine enumeration order. Three different iterators are used as parameter **Order**, each of which can be used in two **Directions**, resulting in six enumeration order. Every configuration is tested on 42 TSP instances from TSPLIB. All algorithms and experiments are implemented in the Julia programming language for technical computing[1] and run in a single-core-per-run configuration on an Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz machine with 16 cores. A complete description of the experimental setup and data is available online[1].

[1] see: `github.com/Michiel-VL/Neighborhood_Enumeration_Data`



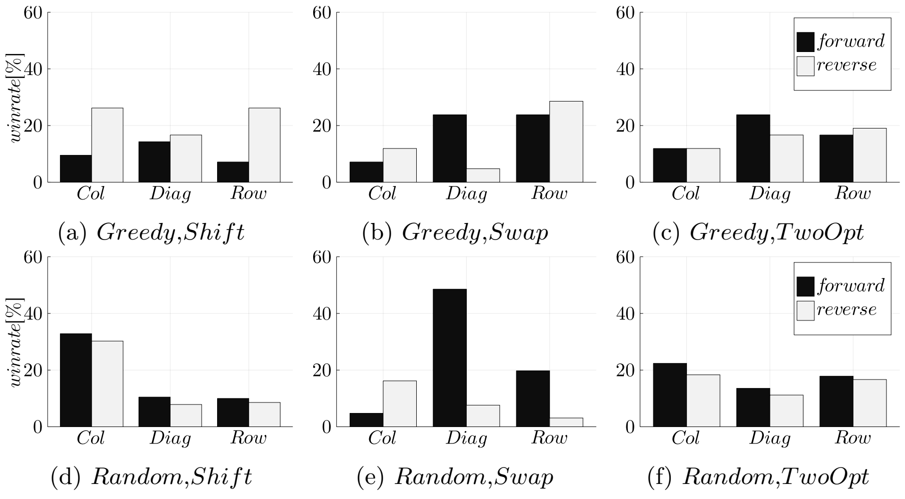
(a) Greedy



(b) Random

Fig. 4: The relative difference between the final solutions should be zero if order had no influence.

First, we examine the effect of enumeration order on the solution state. If no such effect were to exist, then the final solutions of the six runs for a given context and instance should be identical, independent of the enumeration order parameters. To measure if there is an effect of enumeration order on the solution state, we compute the mean relative pairwise distance $\overline{\Delta}_{sr}$ between the set of final solutions of a given context and instance. Fig. 4 is given for each of the 18 contexts. It is clear that the enumeration order does have an influence on the search outcome. Even for *argmin* selection, which is just barely order-dependent, $\overline{\Delta}_{sr}$ is fairly large, suggesting that the effect compounds quickly over the iterations of a search procedure.



(a) *Greedy*,*Shift* (b) *Greedy*,*Swap* (c) *Greedy*,*TwoOpt*
(d) *Random*,*Shift* (e) *Random*,*Swap* (f) *Random*,*TwoOpt*

Fig. 5: The numbers of wins for different orders and operators.

Fig. 5, shows the relative number of wins per enumeration order for different constructive heuristics and local search operators. While enumeration order does seem to affect the winrate, the results are inconclusive as to which order should be preferred for a given operator or constructive heuristic.

5 Discussion

Modeling a local search neighborhood as the combination of an operator with its own parameter space and an iterator over this parameter space has several advantages. First, it renders explicit the enumeration order used to explore the neighborhood, which we have shown has an impact on the search outcome. Furthermore it is modular, as operator, set of parameters and order are completely separable implementation-wise. This not only enables easy reuse of code but it is also expressive, offering a range of neighborhood structures at virtually no cost.

It is also possible to encode structural properties of the problem in the neighborhood. As shown in Section 3, parameter spaces based on structured index sets can be used to distinguish between different parts of a solution representation. By opening up a neighborhood's structure through its parameter space, it is possible to use a wide variety of known algorithms to construct parameter spaces and reuse these over various neighborhoods.

Given a set of neighborhood definitions, new neighborhoods can be constructed in an algorithmic manner. Using function composition, operators can be composed into new operators and through the Cartesian product and disjoint union, various enumeration structures are available. Furthermore, given that in many programming languages iterators are a data structure that can are composable in various ways – like filtering, linking or zipping – the definition as a whole is very expressive and enables concise descriptions of algorithms like Variable Neighborhood Descent and concepts such as path relinking or higher-order neighborhoods.

Note that defining a neighborhood as a triple $(s, M, T(\Phi_M))$ replaces the nested for-loops found in many neighborhood implementations with a single foreach-loop. This triple separates three different neighborhood design concerns that are typically entangled in code: local-search operators, neighborhood size and enumeration order. This enables algorithm designers not only to reuse operator, parameter space and enumeration order implementations for multiple neighborhoods, but it also leads to a more descriptive way of handling neighborhoods, enabling swift development and automated algorithm configuration.

6 Conclusion

In this paper we introduced a novel definition for neighborhoods aimed at formalizing their implementation. Defining local search neighborhoods in terms of a parametrized local search operator and an iterator over the parameter space of the operator leads to an expressive, composable definition which can be readily used during implementation. The iterator makes explicit two algorithm design considerations that are typically overlooked: in what order should neighbors be generated and which neighbors should be included in a neighborhood. Furthermore, by basing the operator parameter spaces on the indexing mechanism of a solution representation, significant parts of neighborhood design can be automatically derived from a solution representation. Finally, as many enumeration orders can be efficiently implemented as a lazy sequence, neighborhoods can be generated lazily.

While in this paper we only considered unconstrained problem representations, it would be interesting to look at constrained problems to examine how particular types of constraints affect the use of the definition, as complex constraints could prevent efficient iterator implementations. Though interesting, this primarily concerns implementation efficiency rather than formalization and thus lay outside the scope of this paper.

Acknowledgements

Research supported by Data-driven logistics (FWO-S007318N). Editorial consultation provided by Luke Connolly.

References

1. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1), 65–98 (2017). https://doi.org/10.1137/141000671
2. Blum, C., Roli, A.: Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM computing surveys (CSUR) **35**(3), 268–308 (2003)
3. Cahon, S., Melab, N., Talbi, E.G.: Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. Journal of heuristics **10**(3), 357–380 (2004)
4. De Beukelaer, H., Davenport, G.F., De Meyer, G., Fack, V.: James: An object-oriented java framework for discrete optimization using local search metaheuristics. Software: Practice and Experience **47**(6), 921–938 (2017). https://doi.org/https://doi.org/10.1002/spe.2459, `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2459`
5. Mecler, J., Subramanian, A., Vidal, T.: A simple and effective hybrid genetic search for the job sequencing and tool switching problem. Computers & Operations Research p. 105153 (2020). https://doi.org/https://doi.org/10.1016/j.cor.2020.105153, `http://www.sciencedirect.com/science/article/pii/S0305054820302707`
6. Sörensen, K.: Metaheuristics—the metaphor exposed. International Transactions in Operational Research **22**(1), 3–18 (2015)
7. Swan, J., Adraensen, S., Brownlee, A.E., Johnson, C.G., Kheiri, A., Krawiec, F., Merelo, J., Minku, L.L., Özcan, E., Pappa, G.L., et al.: Towards metaheuristics" in the large". arXiv preprint arXiv:2011.09821 (2020)
8. Swan, J., Adriaensen, S., Bishr, M., Burke, E.K., Clark, J.A., De Causmaecker, P., Durillo, J., Hammond, K., Hart, E., Johnson, C.G., et al.: A research agenda for metaheuristic standardization. In: Proceedings of the XI metaheuristics international conference. pp. 1–3 (2015)
9. Talbi, E.G.: Metaheuristics: from design to implementation, vol. 74. John Wiley & Sons (2009)