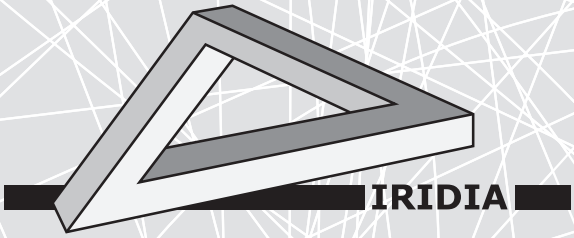

Université Libre de Bruxelles

Institut de Recherches Interdisciplinaires
et de Développements en Intelligence Artificielle

Large Neighbourhood Search Algorithms for the Founder Sequences Reconstruction Problem

Andrea Roli, Stefano Benedettini,
Thomas Stützle, and Christian Blum

IRIDIA – Technical Report Series

Technical Report No.
TR/IRIDIA/2010-012

December 2010
Last revision: December 2010

IRIDIA – Technical Report Series
ISSN 1781-3794

Published by:
IRIDIA, *Institut de Recherches Interdisciplinaires et de Développements en Intelligence Artificielle*
Université Libre de Bruxelles
Av F. D. Roosevelt 50, CP 194/6
1050 Bruxelles, Belgium

Technical report number TR/IRIDIA/2010-012

Revision history:
TR/IRIDIA/2010-012.001 December 2010

Large Neighbourhood Search Algorithms for the Founder Sequences Reconstruction Problem

Andrea Roli, Stefano Benedettini
DEIS-Cesena, *Alma Mater Studiorum* Università di Bologna, Cesena, Italy

Thomas Stützle
IRIDIA, Université libre de Bruxelles (ULB), Brussels, Belgium

Christian Blum
ALBCOM Research Group, Universitat Politècnica de Catalunya, Barcelona, Spain

December 2010

Abstract

The reconstruction of founder genetic sequences of a population is a relevant issue in evolutionary biology research. The problem consists in finding a biologically plausible set of genetic sequences (founders), which can be recombined to obtain the genetic sequences of the individuals of a given population. The reconstruction of these sequences can be modelled as a combinatorial optimisation problem in which one has to find a set of genetic sequences such that the individuals of the population under study can be obtained by recombining founder sequences minimising the number of recombinations. This problem is called the Founder Sequence Reconstruction Problem. Solving this problem can contribute to research in understanding the origins of specific genotypic traits. In this paper, we present Large Neighbourhood Search algorithms to tackle this problem. The proposed algorithms combine a stochastic local search with a Branch-and-Bound algorithm devoted to neighbourhood exploration. The developed algorithms are thoroughly evaluated on three different benchmark sets and they establish the new state of the art for realistic problem instances.

1 Introduction

In recent years, the availability of biological data has rapidly increased as a consequence of technical advances in sequencing genetic material, such as DNA and haplotyped data. Given a sample of sequences from a population of individuals, researchers may try to study the evolutionary history of the population. If the population under study has evolved from a relatively small number of founder ancestors, the evolutionary history can be studied by trying to reconstruct the sample sequences as fragments from the set of founder sequences. The genetic material of the population individuals is the result of recombination and mutation of their founders. Many findings from biological studies support the validity of this model, as, for example, the fact that the '*Ferroplasma* type II genome seems to be a composite from three ancestral strains that have undergone homologous recombination to form a large population of mosaic genomes' (quoted from [26]). The main issue is that neither the number of founder sequences, nor the founder sequences themselves, may be known. A combinatorial optimisation problem can be defined such that its solutions are biologically plausible founder sequences, provided that assumptions on the evolutionary model are formulated. This problem is called the Founder Sequence Reconstruction Problem (FSRP) [27]. Besides its origins in evolutionary biology, the problem is a hard combinatorial optimisation problem and it is of interest *per se*, as it constitutes a challenge for current optimisation techniques.

In the literature, several techniques have been proposed to tackle this problem, such as dynamic programming, tree-search and metaheuristics. In this paper, we present a state-of-the-art hybrid algorithm for the FSRP that is based on the Large Neighbourhood Search framework. The

Set of recombinants $\mathcal{C}$	Set of founders $\mathcal{F}$	Decomposition
01001000	01101110 (a)	a a|b|a a|c c c
00111000	10010011 (b)	a|c c c c c c c
10011100	10111000 (c)	b b b b|a a|c c
10111010		c c c c c c|a a
01101110		a a a a a a a a
10110011		c c c c|b b b b

Figure 1: On the left, a set of six recombinants in matrix form is shown. Assuming that the number of founders is fixed to 3, a valid solution as a matrix of three founders is shown in the centre. Denoting the first founder by a, the second founder by b, and the third one by c, on the right a decomposition of the recombinants matrix into fragments taken from the founders is shown. Breakpoints are marked by vertical lines. This is a decomposition with 8 breakpoints, which is the minimum value for this instance.

algorithm performs a local search in which neighbourhoods have exponential size in the founder sequence length; the neighbourhoods are exhaustively explored by means of an efficient tree-search technique. Several variants of this hybrid search strategy are designed and compared on three benchmark sets. The best variant is compared to the state-of-the-art solvers and shown to achieve better performance.

The paper is structured as follows. In Section 2, we formally introduce the problem. Next, we briefly survey previous and related work in Section 3. In Section 4, we describe the algorithms we designed and implemented; the experimental comparison among them is discussed in Section 5. In Section 6, the best of our solvers is compared with the state of the art. We then conclude with a summary of the contributions and an outline of future work in Section 7.

2 The Founder Sequence Reconstruction Problem

The FSRP can be defined as follows. Given is a set of n *recombinants* $\mathcal{C} = \{C_1, \ldots, C_n\}$; each recombinant C_i is a string of length m over a given alphabet Σ, that is, $C_i = c_{i1}c_{i2}\ldots c_{im}$ with $c_{ij} \in \Sigma$. In this work, we will consider a typical biological application where the recombinants are haplotyped sequences and, hence, $\Sigma = \{0, 1\}$. The symbols 0 and 1 encode the most common allele of the haplotype site (*wild-type*) and its most frequent variation in a population (*mutant*), respectively.

A *candidate solution* to the problem consists of a set of k_f *founders* $\mathcal{F} = \{F_1, \ldots, F_{k_f}\}$. Each founder F_i is a string of length m over the alphabet Σ: $F_i = f_{i1}f_{i2}\ldots f_{im}$ with $f_{ij} \in \Sigma\ \forall\, j$. A candidate solution $\mathcal{F}$ is a *valid solution* if the set of recombinants $\mathcal{C}$ can be *reconstructed* from $\mathcal{F}$. This is the case when each $C_i \in \mathcal{C}$ can be decomposed into a sequence of $p_i \leq m$ fragments (that is, strings) $Fr_{i1}Fr_{i2}\ldots Fr_{ip_i}$, such that each fragment Fr_{ij} appears at the same position in at least one of the founders. Hereby, a decomposition with respect to a valid solution is called *reduced* if two consecutive fragments do not appear in the same founder. Moreover, for each valid solution $\mathcal{F}$ we can derive in $O(n \cdot m \cdot k_f)$ time (see [29]) a so-called *minimal decomposition*. This is a decomposition where $\sum_{i=1}^{m} p_i - m$ is minimal. In the following we call this number the objective function value of $\mathcal{F}$ and denote it by $f(\mathcal{F})$. In biological terms, $f(\mathcal{F})$ is called the number of *breakpoints* of $\mathcal{C}$ with respect to $\mathcal{F}$.

The optimisation goal considered in this paper is to find a valid solution $\mathcal{F}^*$ that, given a fixed number of founders k_f, minimises the number of breakpoints. For an example, see Figure 1.

The FSRP was first proposed by Ukkonen [27] and the problem is proven to be $\mathcal{NP}$-hard [19] for $k_f > 2$.

3 Related work

This section provides an overview of the scientific literature on the FSRP. First we discuss the methods devised to tackle the FSRP. Subsequently, other problem variants will be mentioned. Particular relevance will be given to two methods, namely RecBlock and Back-and-Forth Iterated Greedy, which will be detailed in separate sections. Indeed, these techniques are strongly connected to the algorithms we present in this paper and they are state-of-the-art solvers for the FSRP.

An algorithm based on dynamic programming was first proposed by Ukkonen [27]. However, this method is not efficient when the number of founders and the number or length of recombinants is high. Another dynamic programming algorithm was introduced by Rastas and Ukkonen [19]. Lyngsø and Song [16] have proposed a Branch-and-Bound algorithm. Although promising, this method has been evaluated only on a limited test set composed of rather small instances.

Wu and Gusfield have proposed a constructive tree-search solver [29]—henceforth referred to as RecBlock—which can be run as an exact or incomplete solver. A more detailed explanation of RecBlock is provided in Section 3.1. Although the results of RecBlock are very good for instances with a rather small number of founders, the computation time of RecBlock scales exponentially with the number of founders and makes its application infeasible for large-size instances. This was the motivation for developing metaheuristic approaches [7], the first of which was a Tabu Search method, equipped with an effective constructive procedure [21]. Recently, an Iterated Greedy algorithm has been presented that achieves state-of-the-art results for large instances or in the case of many founders [4].

Many authors have provided contributions to upper and lower bound computations. Meyers and Griffiths have developed the R_h and R_s lower bounds [17], which are tighter than the one introduced by Hudson and Kaplan [15]. Nevertheless, they have worst case exponential and super-exponential complexity, respectively [3]. In the same paper, Meyers and Griffiths presented a general framework, called *composite method*. It exploits local bound information, that is, a bound on a subset of contiguous columns in a recombinant matrix, in order to obtain a possibly better overall bound estimation. Bafna and Bansal suggest a lower bound computable in $O(n \cdot m^2)$ time [3]. Song et al. further elaborate on Meyers and Griffiths bound and propose improved lower and upper bounds [25]. In another work, Wu [28] developed an analytical upper bound, that is, an estimation of the minimum number of breakpoints independent of the particular instance at hand. This estimation is a function of only n and m.

The FSRP is widely studied and it is related to a number of other problems. El-Mabrouk and Labuda [11] focus on a much simpler and tractable reconstruction problem: given a founder set and a recombinant, they want to find a minimal decomposition for the recombinant. A similar problem, called *Haplotype Colouring*, has been introduced by Schwartz et al. [23]. Given a recombinant matrix, a partition of contiguous sites, called blocks, is identified according to some rule and each recombinant is subdivided accordingly into haplotype substrings. The objective is to assign, for each block, a different colour to the distinct haplotype substrings in a block, such that the total number of colour switches between two contiguous haplotype substrings in the same recombinant is minimal. This problem of colouring each block can be encoded into an instance of the weighted bipartite matching problem. Haplotype Colouring can be solved to optimality by a $O(m \cdot n^2 \cdot \log n)$ algorithm.

The FSRP formulation we employ in this work is based on the assumption that, during the genetic evolution of the initial population of founders, mutation events are unlikely to occur. Rastas and

Ukkonen generalise the FSRP by introducing a different objective function that takes into account point mutation [19], namely $f(\mathcal{F}) + c \cdot g(\mathcal{F})$ where $c > 0$ is a constant. This new objective function is the sum of two contributions: $f(\mathcal{F})$ denotes the sum of the number of breakpoints across all recombinant sequences (i.e., the objective function we introduced in Section 2), while $g(\mathcal{F})$ is the total number of point mutations. Note that if c is large, the resulting problem is equivalent to the FSRP definition as presented in Section 2 because an optimal solution will not contain any mutation.

Zhang et al. [30] present a closely related problem to the FSRP called the *Minimum Mosaic Problem*. This problem aims at finding a minimum mosaic, that is, a block partitioning of a recombinant set such that each block is *compatible* according to the Four Gamete Test [15] and the number of blocks is minimised. Differently from the FSRP, this problem does not rely on the existence of a founder set. This mosaic structure so obtained provides a good estimation of the minimum number of recombination events (and their location) required to generate the existing haplotypes in the population, that is, a lower bound for the FSRP.

3.1 RecBlock

RecBlock is a tree-search based solver for the FSRP. It follows a constructive breadth-first search strategy starting from an empty solution. Each node ν_l at depth l in the search tree represents a feasible partial solution $\mathcal{F}_l$ up to the l-th site, that is, a $k_f \times l$ founder matrix. As a consequence, complete solutions are at a tree depth equal to m. In other words, RecBlock fills a founder matrix column by column from left to right. Each node ν_l is labelled with the number of breakpoints in the minimal decomposition of $\mathcal{C}_l$, the problem instance up to site l, with respect to the associated founder matrix $\mathcal{F}_l$. We will denote this number by $BP(\nu_l)$. In the following, ν will interchangeably refer either to a node of the search tree or to the (partial) solution associated to ν.

In order to prune the search tree that would result from a naive enumeration, RecBlock exploits symmetry breaking, bounding and dominance rules. Symmetry breaking relates on the fact that, in a solution to a FSRP instance, the founder sequence order is irrelevant. That is, any permutation of the order of the founders represents the same solution. Therefore, RecBlock constructs only solutions whose founders are sorted lexicographically. Bounding follows the usual procedure of classical Branch-and-Bound algorithms: a node is not expanded any further if its lower bound is greater than a known upper bound. The initial upper bound is obtained by running the incomplete version of RecBlock (explained below); the lower bound employed is computed by the *composite bound method* [17]. The authors of RecBlock state that this bounding strategy is not very effective because the composite lower bound is not very tight. Much more effective is to prune *dominated* nodes. Let us consider two nodes at level l, ν_l^j and ν_l^h: if $BP(\nu_l^j) - BP(\nu_l^h) \geq n$, then node ν_l^j is dominated by ν_l^h and can be pruned. The reason is that any completion of the partial solution ν_l^j can be obtained starting from ν_l^h with the introduction of at most n breakpoints. This pruning criterion can be sharpened by computing the number n' of common founders between the decompositions of $\mathcal{C}_l$ corresponding to nodes ν_l^j and ν_l^h and pruning ν_l^j in case $BP(\nu_l^j) - BP(\nu_l^h) \geq n - n'$.[1]

RecBlock can also perform an incomplete search, in which nodes are pruned by applying a heuristic criterion. This heuristic is based on a score computed as the number of breakpoints in the partial solution associated to a node. The nodes that have a score that differs from the score of the current best node more than a given threshold are discarded. In addition, also the maximum number of search paths along the tree can be limited. The resulting algorithm performs a breadth-first search limited to a part of the whole search tree and returns one solution—not proven to be optimal—only when this partial tree has been completely explored. The advantage

[1] For details about this pruning mechanism, please refer to [29].

of this version of RecBlock (henceforth called RecBlock-incomp) is that a feasible solution can be returned also to some large-size instances that are out of reach for the complete version; on the other side, it has the disadvantage that it still might require extremely high execution times to return a solution and it is not able to improve the solution returned if more time is available.

3.2 Back-and-Forth Iterated Greedy

Back-and-Forth Iterated Greedy is an algorithm for the FSRP proposed by some of the authors of this paper [4]. It is based on the Iterated Greedy metaheuristic framework [22]. The Iterated Greedy metaheuristic consists in iteratively destroying part of the current tentative solution and reconstructing it by means of a randomised constructive procedure. Back-and-Forth Iterated Greedy (B&F IG) uses an enhanced constructive lookahead heuristic that, in turn, is based on a randomised greedy heuristic procedure that takes a partial solution to the FSRP and, at each construction step, extends it by appending one column. This heuristic (henceforth referred to as RGREEDY) was used by some of the authors of this paper for constructing initial solutions in a Tabu Search algorithm [21]. This procedure provides the basis for the lookahead heuristic, briefly described in the following. Let us denote with $A|B$ the concatenation of two matrices with the same number of rows. Given a partial solution $\mathcal{F}_i$ up to site i, the lookahead heuristic constructs nt new partial solutions $\{\mathcal{F}_i|J_1, \mathcal{F}_i|J_2, \ldots, \mathcal{F}_i|J_{nt}\}$, where J_k are $k_f \times lh$ matrices obtained by consecutively applying RGREEDY a number of times equal to $min\{lh, m-i\}$, where the lookahead value lh is a parameter of the B&F IG algorithm. Each $\mathcal{F}_i|J_k$ is subsequently evaluated and the number of breakpoints $BP(\mathcal{F}_i|J_k)$ is computed. Finally, let $K = \underset{k}{\text{argmin}}\{BP(\mathcal{F}_i|J_k)\}$; the column to be appended to $\mathcal{F}_i$ is the first column of matrix J_K.

This lookahead heuristic is embedded into the construction step of B&F IG. The main idea of the algorithm is that solution construction can be performed either from left to right or from right to left (hence Back-and-Forth epithet): this is an intrinsic property (symmetry) of the FSRP. The resulting algorithm has thus the following behaviour: it takes a complete solution and considers a symmetric interval of sites $\mathcal{J} = [k, m-k]$ where k is a parameter; the orientation is initially left-to-right. For each $j \in \mathcal{J}$ it obtains a partial solution $\mathcal{F}_j$ by removing columns $j, j+1, \ldots, m$ (or columns $1, 2, \ldots, j$ if the direction is right-to-left) and it completes $\mathcal{F}_j$ by an iterative application of the lookahead heuristic described previously. The algorithm alternates between destruction and reconstruction until either it improves the incumbent solution or it has looped r times (where r is another algorithm parameter) through interval $\mathcal{J}$. At this point, the algorithm reverses direction and repeats this cycle until termination conditions are met.

4 Large neighbourhood search algorithms for the FSRP

The method that we propose is a hybrid metaheuristic [6] based on the Large Neighbourhood Search framework. Large Neighbourhood Search (LNS) is a search strategy consisting in a local search that uses a complete method for exploring a—typically very large—neighbourhood. LNS tries to combine the advantage of a large neighbourhood, that usually enhances the explorative capabilities of local search, with an exhaustive tree-search exploration that is faster than enumeration. This kind of search strategy has been successfully applied in several contexts [2, 24, 1, 18, 9]. We designed a family of solvers based on the LNS framework which employ RecBlock [29] as sub-solver. In fact, RecBlock is rather efficient for a small number of founders and our aim was to exploit this fact for a solver capable of tackling large-size instances. We first illustrate the high level search strategy we designed (Algorithm 1) and we subsequently detail the variants we implemented (Algorithm 2).

An initial solution is built by means of any constructive procedure (line 1 of Algorithm 1), then a local search with large neighbourhoods is performed. Let $\mathcal{I}$ be the set of indices of founders in $\mathcal{F}$.

Algorithm 1 High level algorithm of LNS for the FSRP

1: $\mathcal{F} \leftarrow$ BuildInitialSolution()
2: **while** termination condition not met **do**
3: $\quad \mathcal{I}_{\text{free}} \leftarrow$ ChooseFounders()
4: $\quad \mathcal{F}' \leftarrow$ RecBlock($\mathcal{F}|_{\mathcal{I}_{\text{free}}}$){Solve the instance restricted to the set of free founders}
5: $\quad$ **if** $f(\mathcal{F}') < f(\mathcal{F})$ **then**
6: $\quad\quad \mathcal{F} \leftarrow \mathcal{F}'$
7: $\quad$ **end if**
8: **end while**
9: **output:** best solution found $\mathcal{F}$

Algorithm 2 LNS-FSRP algorithm

1: Set parameters k_{min}, k_{max}, $maxCombinations$
2: $\mathcal{F} \leftarrow$ BuildInitialSolution()
3: $k \leftarrow k_{min}$
4: $i \leftarrow 1$
5: **while** $k \leq k_{max} \quad \wedge \quad$ maximum time not expired **do**
6: $\quad$ **if** not all k-combinations of k_f founders explored $\quad \wedge \quad i \leq maxCombinations$ **then**
7: $\quad\quad \mathcal{I}_{\text{free}} \leftarrow$ NextCombination(k, k_f)
8: $\quad\quad \mathcal{F}' \leftarrow$ RecBlock($\mathcal{F}|_{\mathcal{I}_{\text{free}}}$)
9: $\quad\quad i \leftarrow i + 1$
10: $\quad\quad$ **if** $f(\mathcal{F}') < f(\mathcal{F})$ **then**
11: $\quad\quad\quad \mathcal{F} \leftarrow \mathcal{F}'$
12: $\quad\quad\quad k \leftarrow k_{min}$
13: $\quad\quad\quad i \leftarrow 1$
14: $\quad\quad$ **end if**
15: $\quad$ **else**
16: $\quad\quad k \leftarrow k + 1$
17: $\quad\quad i \leftarrow 1$
18: $\quad$ **end if**
19: **end while**
20: **output:** best solution found $\mathcal{F}$

The neighbourhood is composed of all feasible assignments to a selected set of founders of indices $\mathcal{I}_{\text{free}} \subseteq \mathcal{I}$ (line 3), while the other founders of indices $\mathcal{I} \setminus \mathcal{I}_{\text{free}}$ are kept fixed; this neighbourhood is exhaustively explored by RecBlock (line 4). For this purpose, we modified RecBlock so as to have the algorithm searching only among the configurations of $k_f = |\mathcal{I}|$ founders in which $|\mathcal{I} \setminus \mathcal{I}_{\text{free}}|$ founders are fixed. If the chosen neighbouring solution has a lower number of breakpoints than the current one, it becomes the new current solution, in the spirit of a first improvement local search (lines 5–7). This strategy shares also analogies with IG [13] and Forget and Extend [8], in that it partially destroys and reconstructs the solution.

The general strategy illustrated in Algorithm 1 can be specialised in various ways and, in particular, there are several different choices for the instantiation of the function ChooseFounders(). We implemented several variants of the algorithm, which share the idea of increasing the number of unassigned founders whenever either all the combinations have been explored or no solution improvements are found for a given number of examined neighbours. We remark that the idea of enlarging the neighbourhood whenever diversification is needed is similar to that characterising Variable Neighbourhood Descent [12]. In our LNS solver, the neighbourhood size is controlled by the number k of founders to which the search is restricted. The high-level parametrised algorithm describing our family of solvers is detailed in Algorithm 2.

The algorithm has three main parameters: k_{min} and k_{max}, which denote the minimum (resp. max-

imum) number of founders to be chosen, and *maxCombinations*, which denotes the maximum number of k-combinations of founders that are considered before incrementing k. The function NextCombination(k, k_f) returns the indices of a new k-combination of founders. The neighbourhood is examined in random order.

The search is stopped when either $k > k_{max}$ or the maximum computation time allotted is expired. The time check is also performed inside the modified version of RecBlock, so as to guarantee that the solution returned is found within the time limit.

Depending on the values of the parameters, different variants of LNS-FSRP can be obtained. The ones that we tested are the following:

- `LNS-1`: $k_{min} = 1$, $k_{max} = k_f$, $maxCombinations = \infty$
- `LNS-3`: $k_{min} = 3$, $k_{max} = k_f$, $maxCombinations = \infty$
- `LNS-maxc`: $k_{min} = 1$, $k_{max} = k_f$, $maxCombinations = 10$

`LNS-1` and `LNS-3` differ in the minimum number of founders considered for defining the neighbourhood. The advantage of `LNS-3` over `LNS-1` could be to start the search with a neighbourhood that makes it possible to achieve a good balance between diversification and efficiency, because RecBlock has proven to be quite fast in solving instances with 3 founders. Note that the neighbourhoods can be explored exhaustively, because $maxCombinations = \infty$. Therefore, if no time limit is imposed, `LNS-1` and `LNS-3` both converge to a RecBlock search on the whole set of k_f founders.[2] As it will be shown in Section 6, this search strategy has the advantage of finding very good—or even optimal solutions—much faster than when running RecBlock directly in its full version; moreover, it finds good solutions to large-size instances, which are not solved be RecBlock. `LNS-maxc` differs from the previous algorithms in that it only performs a limited number of neighbourhood explorations; the rationale behind this strategy is that it could be beneficial, especially for large-size instances, to sample the k-combinations rather than enumerating all of them. *maxCombinations* is a parameter of the algorithm and it should be set after a parameter tuning process. This parameter was set to 10, because this value is the minimum number that enables the algorithm to exhaustively explore the neighbourhood of size one in the cases in which $k_f = 10$, thus achieving a good trade-off between intensification and diversification across all the instances. All the three algorithms have the property of being *anytime solvers*, because they can progressively return improved solutions.

The initial solution can be provided by a dedicated constructive heuristic or it can be a random solution. In our algorithms we feed the LNS solver with a solution generated by the Back-and-Forth IG algorithm described in [4].

A possible improvement over the algorithms described above consists in exploiting the pruning capability of RecBlock when an upper bound (UB) is provided. Indeed, each time a new best solution is found, this value can be provided to RecBlock for pruning the search tree. The effect of the use of this UB update will be discussed in Section 5.

5 Experimental analysis

In this section, we experimentally compare the LNS variants. Before this, we first introduce the test instances. The best of these algorithms is compared to the state-of-the-art solvers, which are RecBlock and B&F IG.

[2] Hence, in principle, they are complete.

5.1 Experimental setting

The algorithms have been implemented in C++, compiled with gcc 3.4.6 and run on a cluster equipped with dual core Intel Xeon 3GHz processors, 6MB of cache and 8GB of RAM. Since a short computation time is not a requirement for the FSRP, the maximal CPU time allotted for each algorithm was 3 days (72 hours). In case B&F IG was chosen for providing the initial solution, it was run for 50 seconds in the case of 30 recombinants and 100 seconds for instances with 50 recombinants. The whole set of experiments required about 4 years of CPU time.

Algorithms are compared on the basis of solution quality and, in case of ties, the computation time at which the best solution of a run was found.

The benchmark instance set used in this study comprises three sets: `random`, `evo` and `ms`. The `random` set has been used also in previous works [21, 4]. A random instance is generated by assigning value 0 or 1 to each recombinant's site with probability equal to 0.5. Only those instances that are valid and not reducible are inserted in the set. Random instances have the disadvantage of not being particularly significant as benchmark for real-world problems. To compare the solvers on instances of varying size with a structure similar to that of real-world instances, we generated two further sets based on evolutionary models. In fact, apart from the `random` instance set, in previous works only small-size real-world instances have been used. The first set is produced with the Hudson generator [14] that simulates genetic samples under neutral models. The second one is generated by simulating a simple neutral evolutionary process by iteratively applying a one-point crossover to an initial population of founders. Besides the number of sites (m), founders (k_0) and recombinants (n), the parameters of the `evo` generator are the crossover probability (cxp) and the growth factor (α). As long as the current population's size $popsize$ is less than n, a new population of size $\lfloor \alpha \cdot popsize \rfloor$ is generated by iteratively picking two individuals at random, producing two individuals by applying the crossover—with probability cxp—and inserting in the population randomly one of the two new individuals. The `evo` benchmark set was generated with $k_0 = 10, \alpha = 2, cxp = 0.8$ and is composed of valid and not reducible instances. Each set contains instances with n recombinants of length m, where $n \in \{30, 50\}$ and $m \in \{2n, 3n, 5n\}$. The instances used, along with generators and their parameter settings, are available as online additional material [20].

It is important to observe that our solvers are stochastic, therefore they might return a different solution at each run. However, in our experiments, we evaluate each algorithm only once on a benchmark instance. We do so to avoid too high computation times that would be incurred by more than a single trial per instance and because it is known that for a fixed number of N runs the minimal variance estimation is achieved by running a solver once on N instances, instead of running it more than once on a smaller subset of instances [5]. In the comparisons which follow, whenever we need to assess the statistical significance of the results, we apply the non-parametric paired Wilcoxon test for the equality of two medians [10].

5.2 Comparison among LNS-FSRP variants

We first compare algorithms `LNS-1`, `LNS-3` and `LNS-maxc`. In order to save computation time, while keeping at the same time a wide spectrum of instances, the test set chosen was composed of one instance per set (the ones generated with random seed equal to 1) and $k_f \in \{5, 7, 9\}$; therefore, the algorithms were run on 54 cases (a case is an instance with a given k_f). Since solution values and execution times can be on different scale across the test cases, we compared the algorithms on the basis of their relative difference, computed with respect to the minimum among the three values. Let us denote by $sol(\mathcal{A})$ the solution value returned by algorithm $\mathcal{A}$ and let sol_{min} be the minimum value found by the algorithms involved in the comparison. The relative solution value difference of algorithm $\mathcal{A}$ is computed as $(sol(\mathcal{A}) - sol_{min})/sol_{min}$. Relative time differences are

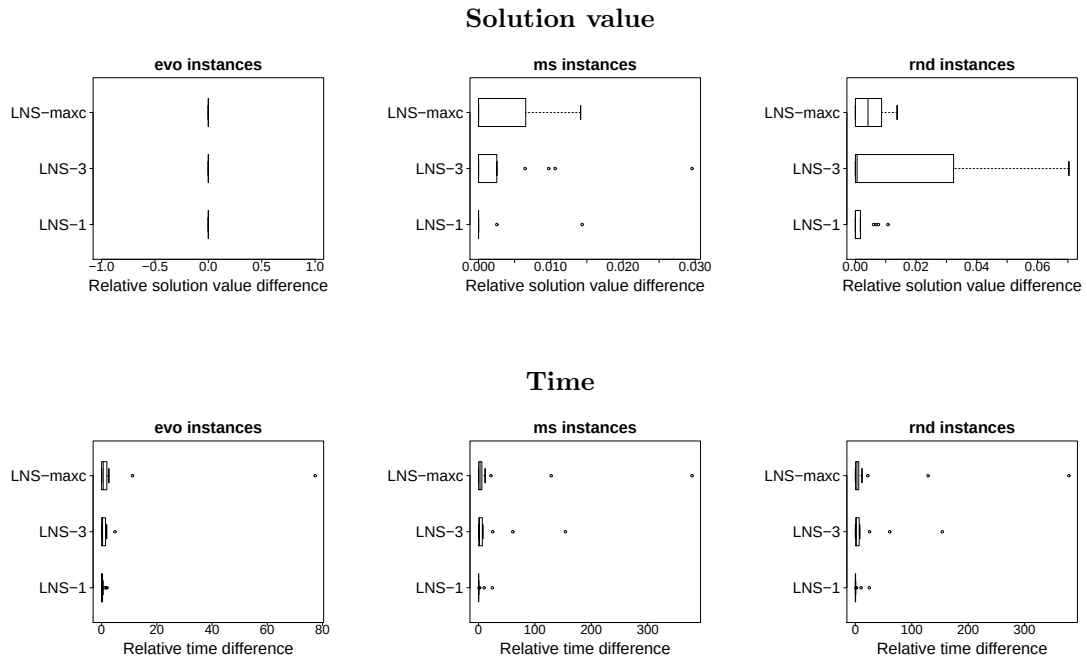

Figure 2: Boxplot of solution value (uppermost row) and execution time (lowermost row) differences between `LNS-1`, `LNS-3` and `LNS-maxc`.

computed in an analogous way to the relative solution value differences. Boxplots of the relative solution value and the relative time differences are drawn in Figure 2.

The results of this comparison show that the algorithms perform similarly, but that `LNS-1` is superior to the other two algorithms as it is statistically confirmed by the Wilcoxon test: `LNS-1` is significantly better than the other two competitors on `random` instances with respect to solution value and on `evo` and `ms` instance sets with respect to run time. No statistical difference is found in the other cases.

Before discussing the experimental results of the comparison against the state of the art, we present some further analysis on the behaviour of `LNS-1` by studying the impact of the initial solution and of a dynamic upper bound update. Furthermore, we analyse the impact of neighbourhood size on search. Finally, we present an improvement of the algorithm based on a speed-up mechanism.

5.3 Impact of initial solution

The solver we implemented starts the search from a greedy constructed solution provided by B&F IG. We also run experiments to assess the actual impact of a heuristic solution, w.r.t. a random initial one. The corresponding algorithm is named R`LNS-1`. A summary of the statistics concerning the relative improvement of greedy constructed solution over random initial solution is shown in Figure 3. Except for the case of `evo` instances, in which no statistically significant difference is found, starting from an initial solution provided by B&F IG enables the algorithm to find a better solution than starting from randomly generated founder sequences.

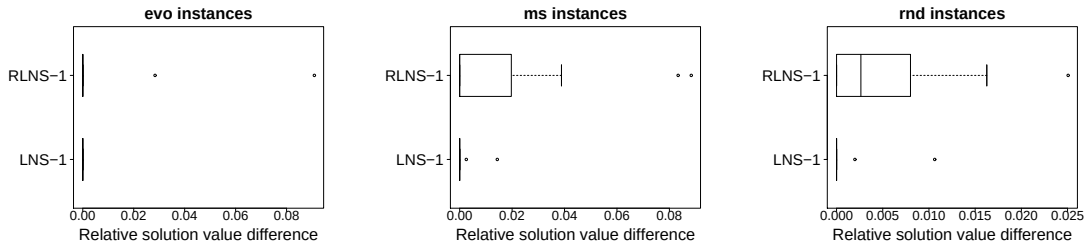

Figure 3: Boxplot of relative solution value difference between `LNS-1`, which is initialised by B&F IG and `RLNS-1`, which is initialised by a random solution.

5.4 Impact of upper bound update

The inner complete solver RecBlock accepts as input also an upper bound value on the solution. This piece of information is used to prune the search tree during the solution process. Since we use iteratively RecBlock during search, it might be beneficial to update the upper bound every time a new best solution is found. We then implemented this variant of `LNS-1` and found no evidence for an advantage of using UB information over not using it, neither in solution quality nor in execution time.

5.5 Impact of neighbourhood size

During the runs of the algorithm,[3] we recorded the neighbourhood size at which a new best solution was found, so as to estimate the impact of neighbourhood size on the search process. Histograms reporting the frequency of improvements with specific values of k for `LNS-1` are drawn in Figure 4. As it can be observed, most improvements are found for neighbourhood sizes of 1 and 2; however, a significant fraction of improvements is achieved with k equal to 3 and 4. This analysis shows that neighbourhoods larger than 2 have to be considered in order to have a solver with high performance. One may observe that for $k_f \leq 2$ the problem has a polynomial time worst case complexity and a dedicated polynomial time algorithm could be used instead of RecBlock. However, RecBlock is very fast also in this case because it exploits the constraints imposed by the fixed founder values to efficiently explore the search tree corresponding to the sub-problem restricted to assigning k founders. An *ad hoc* algorithm designed to solve efficiently the sub-problems with $k_f \leq 2$ could anyway make the overall solver more efficient.

Another insightful piece of information is given by the percentage of cases in which an improvement is achieved with a neighbourhood size smaller than that of the one for which the last improvement was achieved, that is, after resetting k to k_{min} (see line 12 in Algorithm 2). Table 1 reports the percentage of times such an event occurred for each instance class. The value p_{ij} in row i and column j means that an improvement with $k = j$ has been achieved $p_{ij}\%$ of times after an improvement attained with $k = i$. As we can note, a significant percentage of improvements has been achieved by moving to a smaller neighbourhood.

5.6 `LNS-1` speed-up

The algorithm `LNS-1` can be improved by adding a caching mechanism that makes it possible to avoid revisiting neighbourhoods included in previously explored ones. The possibility of revisiting

[3] on all instances and for $k_f \in \{5, 6, 7, 8, 9, 10\}$.

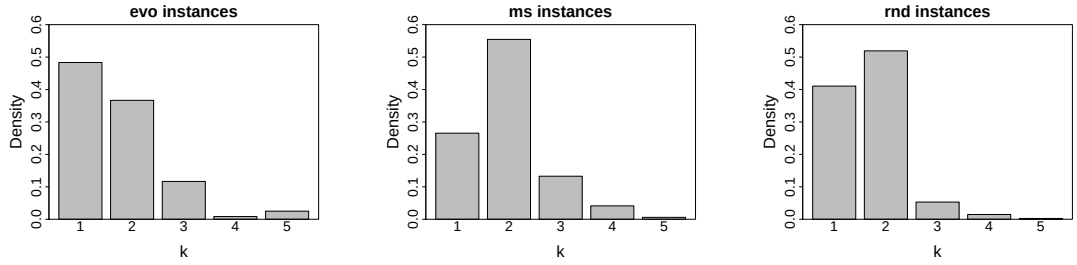

Figure 4: Histograms showing the fraction of times an improvement is found with neighbourhood of size k.

Table 1: Percentage of improvements achieved with a smaller neighbourhood w.r.t. the last improvement. The value p_{ij} in the matrix means that, after an improvement achieved with a neighbourhood i, a further improvement has been achieved with neighbourhood of size $j < i$ in p_{ij}% of the cases.

`evo` instances			
	1	2	3
2	4.6	–	–
3	7.1	7.1	–
4	0	0	0
5	0	0	33.3

`ms` instances			
	1	2	3
2	6.9	–	–
3	0	24.4	–
4	0	21.4	7.1 0

`random` instances			
	1	2	3
2	23.7	–	–
3	1.5	46.1	–
4	11.1	5.6	0

(part of) a same neighbourhood arises when an improvement is found for a given set $\mathcal{I}_\mu$ of μ founder indices: in the next iteration k is reset to 1 and the search continues by exploring neighbourhoods of size 1. All solutions that can been obtained by exploring sequences of founders whose indices are in a set $\mathcal{I}' \subseteq \mathcal{I}_\mu$ are provably not better than the current best solution, because the neighbourhood exploration is complete. Therefore, computation time can be saved by avoiding exploring subsets of already considered founders. Since the neighbourhoods are scanned in a random order, for every $k \leq \mu$ a subset of an already explored set of founders can be visited with probability $\binom{\mu}{k}/\binom{k_f}{k}$. We implemented a caching mechanism based on recording the index set $\mathcal{I}^*$ of the last visited founders that led to a solution improvement: as long as no improvement is found, before exploring a neighbourhood, we check whether it consists of founders with indices all included in $\mathcal{I}^*$. The set $\mathcal{I}^*$ is updated whenever a solution improvement is found.

We compared the performance of simple `LNS-1` against that equipped with the caching mechanism, named `LNS-1c`. The introduced variant only affects execution time, because it does not perturb the search process. Therefore, no difference whatsoever is expected—nor it has been found—with respect to solution quality. Results show that `LNS-1c` can be faster than `LNS-1` up to 15%. The average speed-up attained on `evo` instances is 2%, with a maximum of 6%. Conversely, for `ms` and `random` instances the improvement is more evident, being on average the 4% for both sets, with a maximum of 14% and 15%, respectively. Hence, `LNS-1c` is the variant we use for the comparison against the state of the art.

6 Comparison of `LNS-1c` against the state of the art

In this section, we compare `LNS-1c` with the state-of-the-art solvers RecBlock (both complete and incomplete versions) and B&F IG developed by some of the authors of this paper [4]. Tables 2,

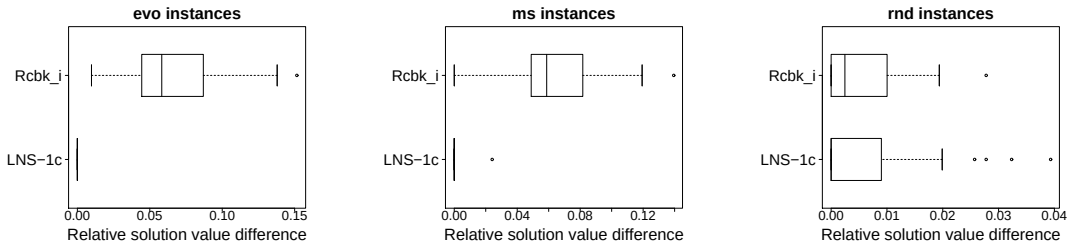

Figure 5: Boxplot of relative solution value difference between `LNS-1c` and RecBlock-incomp (labelled as Rcbk_i in figure).

3 and 4 report the results of RecBlock-comp, RecBlock-incomp and B&F IG on the three benchmark sets.

For each test set one instance was taken (the one generated with random seed equal to 1 and identified by ID number 1 in the test sets) and it was tackled with $k_f \in \{5, 6, 7, 8, 9, 10\}$. Therefore, the algorithms are compared on a set composed of 108 cases. The total time allotted to the solvers was of 3 days (initial solution's computation time included) for each run. The best solution value returned is highlighted in boldface; in case of ties, the shortest time is also marked.

6.1 Comparison against RecBlock

We compared `LNS-1c` against RecBlock both run as a complete (RecBlock-comp) and incomplete algorithm (RecBlock-incomp). In our experiments, we run the default incomplete variant of RecBlock.[4] As observed in initial experiments, this parameter setting is the one that enables RecBlock-incomp to attain a very good trade-off between solution quality and execution time. A careful parameter tuning could improve the performance of RecBlock-incomp on specific instances or restricted classes of instances; nevertheless, an automatic parameter tuning for RecBlock would require a high amount of computation time and the resulting algorithm would not be anyway guaranteed to find a solution in the allowed time limit. In fact, we would like to emphasise that the strength of RecBlock lies in its efficient tree exploration, rather than in its capabilities as heuristic.

Results from Tables 2, 3 and 4 show that many instances could not be solved by RecBlock-comp within the time limit of 3 CPU days. RecBlock-comp could solve only 33 out of 108 cases. Notably, `LNS-1c` always found the same solution value RecBlock-comp found, which is the optimal solution. Furthermore, also the performance in terms of computation time is strongly in favour of `LNS-1c`. Indeed, only in 6 cases RecBlock-comp is faster than `LNS-1c`. The Wilcoxon test confirms this hypothesis.

RecBlock-incomp returned a feasible solution in 90 out of 108 cases; we can observe that the cases in which no solution was returned are the ones with 9 or 10 founders. We compared the solution quality returned by the algorithms on the restricted set of cases in which a feasible solution was returned by RecBlock-incomp. Aggregate results are shown as boxplots of relative solution value difference in Figure 5.

`LNS-1c` returns better results than RecBlock-incomp in all `evo` and `ms` instances, except for a single case in the `ms` set. In these test sets, `LNS-1c` returns significantly better solutions than RecBlock-incomp, with differences up to 15%, and a median value around 6%. Results on `random`

[4]Indeed, each run of the plain version of RecBlock is composed of two parts: in the first, the algorithm is run in an incomplete version and its solution is used as upper bound provided to the complete version.

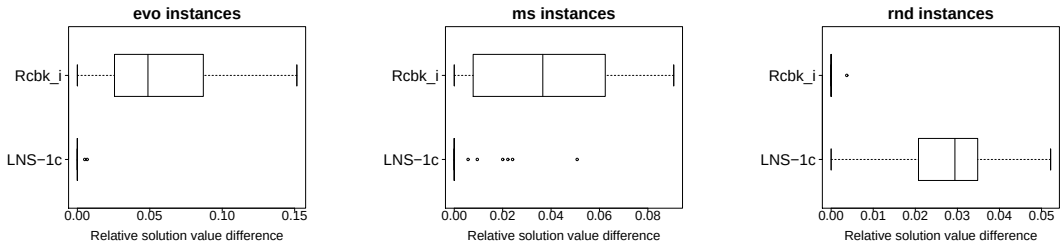

Figure 6: Boxplot of solution value relative difference between `LNS-1c` and RecBlock-incomp within the time required by RecBlock-incomp (labelled as `Rcbk_i` in figure).

instances are less clear, as `LNS-1c` returns better results in 18 out of 33 cases and has a lower median, but the Wilcoxon test does not reject the null hypothesis of equal performance.

A further comparison between `LNS-1c` and RecBlock-incomp can be done in terms of quality w.r.t. time, similarly to what is done with run time distributions [13]. Nevertheless, RecBlock-incomp does not return a series of solutions during time, but just one; in fact, it requires a finite time for returning one result at the end of the exploration of a heuristically pruned search tree and it does not exploit longer execution times. Therefore, run time distributions can not be computed. Instead, for each instance, we stored the results returned by `LNS-1c` in the time RecBlock-incomp found a feasible solution to the instance. Results are shown in Figure 6, while detailed results can be found in [20]. We can note that `LNS-1c` is superior to RecBlock-incomp on both `evo` and `ms` instances, with an average improvement of 4% and 5%, respectively. Conversely, on `random` instances RecBlock-incomp returns better solutions, with an average improvement of 3% over `LNS-1c`. The performance of `LNS-1c` on `random` instances could be improved by substituting RecBlock-comp with RecBlock-incomp in the neighbourhood exploration of `LNS-1c` and limiting the maximum number of neighbours visited (*maxCombinations* parameter). Thus, the time spent in exploring a neighbour is considerably reduced, especially when k_f is large, and more steps of local search can be performed. We implemented such a variant and the results show that the average gap to RecBlock-incomp is halved. However, it is important to remark that providing solutions to the FSRP in short execution times is not a requirement and, as from the results we have discussed before, `LNS-1c` is able to return better solutions on average by exploiting the whole computation time allotted. Moreover, the `evo` and `ms` instances are generated according to biological models and are realistic, in the sense that a real-world instance would very likely have a similar structure. On the contrary, `random` instances are a purely artificial testbed and are meaningless with respect to biological applications.

6.2 Comparison against Back-and-Forth Iterated Greedy

We also compared `LNS-1c` with B&F IG, that attained good results on large-size random instances in a previous study [4].

B&F IG was run for 3 days. As it can be seen from boxplots in Figure 7, `LNS-1c` attains considerably better results than B&F IG. `LNS-1c` finds better solutions than B&F IG in most of instances: in 25 cases in `evo` and in 35 cases in both `ms` and `random`. In the few remaining cases, B&F IG returns results of the same quality as `LNS-1c`, except for one case in the `random` set, in which it finds a better solution—one unit lower than that returned by `LNS-1c`. These results show that `LNS-1c` can indeed considerably improve the initial solution provided by Iterated Greedy and that it is far more efficient, because it can successfully exploit the available computation time.

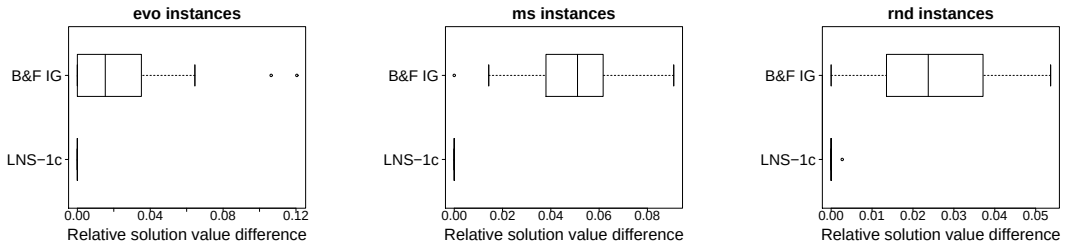

Figure 7: Boxplot of solution value relative difference between `LNS-1c` and Iterated Greedy.

7 Conclusion and future work

In this work, we have described LNS algorithms we designed for tackling large-size instances of the FSRP. The algorithms exploit RecBlock, a state-of-the-art complete solver for this problem. The best algorithm, named `LNS-1c`, performs a local search in which a move consists in reassigning the values of one or more founders. Given a founder matrix, its neighbours of rank k are all the founder matrices differing in k founders. The exploration of such large neighbourhoods is performed by RecBlock, that returns the provably best founder configuration among all the ones in the neighbourhood. `LNS-1c` achieves a performance considerably better than that of the state-of-the-art solvers. The algorithm also combines the property of *anytime solving* with that of completeness, if time is not limited. The experimental analysis has been performed on three sets of instances: `evo`, `ms` and `random`. `evo` and `ms` instances are created according to evolutionary models, so as to have realistic instances of variable size, whilst `random` instances have been considered just for comparison with previous works, being not meaningful from a biological perspective. Indeed, real-world instances are very likely to be far from random.

Further variants of LNS could be designed and embedded into higher level techniques, such as Iterated Local Search and Memetic Algorithms. Future work is also planned for tackling variants of the FSRP, such as the ones including mutation, noise and missing values. Furthermore, other variants including additional biological constraints, e.g., evenly distributed breakpoints or maximal number of breakpoints per recombinant, can be tackled.

Table 2: Detailed results of `LNS-1c`, RecBlock-compl, RecBlock-incompl and B&F IG on the `evo` instances.

number of	`LNS-1c`		RecBlock-comp		RecBlock-incomp		B&F IG	
founders	value	time (s)	value	time (s)	value	time (s)	value	time (s)
`evo-30_60`								
5	**145**	**4**	**145**	87	152	4	**145**	**4**
6	**94**	**53**	**94**	1205	97	28	**94**	96
7	**65**	**86**	**65**	14555	68	225	66	32
8	**45**	**353**	**45**	21288	47	1354	47	7
9	**36**	**51**	—	—	39	16704	**36**	200
10	**28**	**1**	—	—	—	—	**28**	**1**
`evo-30_90`								
5	**203**	**60**	**203**	153	210	6	204	3478
6	**118**	**52**	**118**	1508	128	39	**118**	2932
7	**69**	**19**	**69**	15484	75	293	**69**	**19**
8	**43**	**3**	**43**	4316	47	2285	**43**	**3**
9	**35**	**69**	**35**	33399	37	27359	37	3
10	**31**	28	—	—	—	—	**31**	**21**
`evo-30_150`								
5	**381**	893	**381**	**287**	392	16	388	44436
6	**230**	**72**	**230**	3066	244	65	234	88
7	**131**	**56**	**131**	23034	138	501	133	5067
8	**63**	**59**	**63**	104006	70	5650	64	566
9	**39**	**1**	**39**	43334	40	40989	**39**	**1**
10	**35**	12	—	—	—	—	**35**	**7**
`evo-50_100`								
5	**368**	**145**	**368**	448	394	13	371	143
6	**250**	**113**	**250**	9976	263	67	255	14419
7	**174**	**14706**	**174**	250975	198	357	180	28460
8	**124**	149	—	—	139	5084	132	102223
9	**99**	2507	—	—	114	36290	104	933
10	**83**	3696	—	—	—	—	84	17557
`evo-50_150`								
5	**522**	**132**	**522**	382	553	19	528	191564
6	**319**	**109**	**319**	10228	341	124	320	96282
7	**205**	**4**	**205**	88294	207	606	**205**	**4**
8	**135**	169	—	—	141	4695	138	12871
9	**101**	108	—	—	107	48300	102	77
10	**82**	291	—	—	—	—	85	201
`evo-50_250`								
5	**1126**	3060	**1126**	**1604**	1182	107	1160	9574
6	**726**	**1060**	**726**	25095	744	262	752	144651
7	**450**	259	—	—	466	1397	465	63669
8	**258**	603	—	—	291	14324	270	8746
9	**141**	12100	—	—	159	177159	156	3599
10	**83**	275	—	—	—	—	93	78290

Table 3: Detailed results of `LNS-1c`, RecBlock-compl, RecBlock-incompl and B&F IG on the `ms` instances.

number of	`LNS-1c`		RecBlock-comp		RecBlock-incomp		B&F IG	
founders	value	time (s)	value	time (s)	value	time (s)	value	time (s)
`ms-30_60`								
5	**124**	**209**	**124**	5368	130	8	126	128
6	**100**	98859	—	—	105	57	104	176786
7	**81**	17273	—	—	85	409	83	16
8	**70**	54798	—	—	76	6591	71	93
9	**60**	2002	—	—	65	72924	61	2406
10	**50**	38579	—	—	—	—	**50**	**21576**
`ms-30_90`								
5	**167**	**747**	**167**	2095	175	8	175	98780
6	**136**	**768**	**136**	163768	140	60	141	30
7	**114**	30934	—	—	119	570	117	225805
8	**97**	126402	—	—	102	8495	102	117982
9	85	216	—	—	**83**	226474	88	75
10	**74**	1648	—	—	—	—	77	6520
`ms-30_150`								
5	**251**	4986	**251**	**1805**	271	14	270	1412
6	**189**	**1421**	**189**	90048	204	93	203	43967
7	**153**	25361	—	—	159	1481	159	27490
8	**125**	7590	—	—	135	11381	133	8723
9	**103**	106022	—	—	—	—	109	116217
10	**88**	22794	—	—	—	—	92	33269
`ms-50_100`								
5	**310**	**2192**	**310**	166497	347	14	323	20093
6	**251**	18039	—	—	286	150	266	146944
7	**212**	442	—	—	220	1572	223	1739
8	**178**	51495	—	—	188	13538	187	2585
9	**155**	38758	—	—	163	218293	163	104530
10	**137**	30080	—	—	—	—	141	14612
`ms-50_150`								
5	**429**	48449	**429**	**27077**	453	35	468	2177
6	**346**	26957	—	—	376	114	368	24719
7	**286**	1958	—	—	312	1049	300	110129
8	**241**	130741	—	—	257	12294	254	11687
9	**203**	170493	—	—	—	—	212	35375
10	**174**	8253	—	—	—	—	183	250919
`ms-50_250`								
5	**613**	**2171**	**613**	18235	649	84	646	251801
6	**479**	48013	—	—	516	364	515	6793
7	**396**	16430	—	—	433	6383	418	212274
8	**336**	23916	—	—	360	28984	363	15497
9	**283**	243608	—	—	—	—	306	250687
10	**248**	7413	—	—	—	—	268	1906

Table 4: Detailed results of `LNS-1c`, RecBlock-compl, RecBlock-incompl and B&F IG on the `random` instances.

number of	`LNS-1c`		RecBlock-comp		RecBlock-incomp		B&F IG	
founders	value	time (s)	value	time (s)	value	time (s)	value	time (s)
`random-30_60`								
5	**372**	48427	**372**	**38490**	376	5	376	637
6	**324**	44255	—	—	333	30	327	214250
7	**293**	906	—	—	295	232	294	179134
8	**268**	96096	—	—	270	1482	270	185210
9	246	175659	—	—	**245**	7502	249	120582
10	229	190559	—	—	**225**	181678	230	2605
`random-30_90`								
5	**585**	72903	**585**	**58301**	594	7	595	177509
6	**516**	179754	—	—	526	63	527	24720
7	472	55418	—	—	**469**	383	477	184758
8	**426**	107173	—	—	**426**	**2437**	441	92253
9	399	12679	—	—	**389**	13334	406	174833
10	370	244167	—	—	**356**	108248	369	55441
`random-30_150`								
5	**976**	**134777**	**976**	169057	986	24	1000	35683
6	**865**	216875	—	—	876	106	890	8264
7	**778**	140918	—	—	781	499	812	96348
8	**710**	250463	—	—	713	4044	736	7448
9	666	87405	—	—	**648**	35617	685	76775
10	**619**	21046	—	—	—	—	639	136143
`random-50_100`								
5	**1213**	65968	—	—	1226	18	1226	47222
6	**1097**	60881	—	—	1108	63	1115	164682
7	1009	8769	—	—	**1000**	405	1024	31644
8	928	44145	—	—	**927**	2836	948	164238
9	875	113792	—	—	**862**	17875	892	109557
10	830	221118	—	—	**804**	167759	849	81393
`random-50_150`								
5	**1800**	195873	—	—	1815	23	1838	191857
6	**1622**	144474	—	—	1626	138	1673	59084
7	**1484**	221180	—	—	1490	783	1544	148594
8	1385	85140	—	—	**1381**	3353	1446	92349
9	1302	222181	—	—	**1280**	26654	1351	256516
10	**1240**	244166	—	—	—	—	1285	196005
`random-50_250`								
5	**3043**	101246	—	—	3074	117	3126	102337
6	**2725**	172785	—	—	2751	290	2845	110523
7	**2508**	251951	—	—	2512	1352	2628	119105
8	2330	176486	—	—	**2320**	5698	2455	104077
9	2204	244380	—	—	**2161**	46765	2318	180631
10	**2097**	257557	—	—	—	—	2194	205293

Acknowledgements

We thank Yufeng Wu for providing us the RecBlock source code and for his kind support. Andrea Roli acknowledges support from the "Brains (Back) to Brussels" 2009 programme funded by IRSIB – Institut d'encouragement de la Recherche Scientifique et de l'Innovation de Bruxelles. This work was partially supported by the META-X project, a *Actions de Recherche Concertée* funded by the Scientific Research Directorate of the French Community of Belgium. Thomas Stützle acknowledges support from the fund for scientific research F.R.S-FNRS of the French Community of Belgium of which he is a Research Associate. Christian Blum was supported by grant TIN2007-66523 (FORMALISM) of the Spanish government. Moreover, Christian Blum acknowledges support from the *Ramón y Cajal* program of the Spanish Ministry of Science and Innovation.

References

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

[2] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[3] V. Bafna and V. Bansal. The number of recombination events in a sample history: Conflict graph and lower bounds. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 1:78–90, 2004.

[4] S. Benedettini, C. Blum, and A. Roli. A randomized iterated greedy algorithm for the founder sequence reconstruction problem. In C. Blum and R. Battiti, editors, *Proceedings of the Fourth Learning and Intelligent OptimizatioN Conference – LION 4*, volume 6073 of *Lecture Notes in Computer Science*, pages 37–51. Springer, Heidelberg, Germany, 2010.

[5] M. Birattari. *Tuning Metaheuristics: A Machine Learning Perspective*, volume 197 of *Studies in Computational Intelligence*. Springer, Heidelberg, Germany, 2009.

[6] C. Blum, M. Blesa, A. Roli, and M. Sampels, editors. *Hybrid Metaheuristics: An Emerging Approach to Optimization*, volume 114 of *Studies in Computational Intelligence*. Springer, Heidelberg, Germany, 2008.

[7] C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308, 2003.

[8] Y. Caseau and F. Laburthe. Effective forget-and-extend heuristics for scheduling problems. In F. Focacci, A. Lodi, M. Milano, and D. Vigo, editors, *Electronic Proceedings of the Int. Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems – CPAIOR 1999*, 1999. Available at: http://www3.deis.unibo.it/Events/Deis/Workshops/Proceedings.html.

[9] M. Chiarandini, I. Dumitrescu, and T. Stützle. Very large-scale neighborhood search: Overview and case studies on coloring problems. In Blum et al. [6], pages 117–150.

[10] W.J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, New York, NY, 3rd edition, 1999.

[11] N. El-Mabrouk and D. Labuda. Haplotypes histories as pathways of recombinations. *Bioinformatics*, 20(12):1836–1841, 2004.

[12] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130:449–467, 2001.

[13] H.H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann Publishers, San Francisco, CA, 2005.

[14] R.R. Hudson. Generating samples under a Wright-Fisher neutral model of genetic variation. *Bioinformatics*, 18:337–338, 2002.

[15] R.R. Hudson and N.L. Kaplan. Statistical properties of the number of recombination events in the history of a sample of dna sequences. *Genetics*, 111:147–164, 1985.

[16] R.B. Lyngsø and Y.S. Song. Minimum recombination histories by branch and bound. In R. Casadio and G. Myers, editors, *Proceedings of the 5th Workshop on Algorithms in Bioinformatics – WABI 2005*, volume 3692 of *Lecture Notes in Computer Science*, pages 239–250. Springer, Heidelberg, Germany, 2005.

[17] S.R. Myers and R.C. Griffiths. Bounds on the minimum number of recombination events in a sample history. *Genetics*, 163(1):375–394, 2003.

[18] L. Perron, P. Shaw, and V. Furnon. Propagation guided large neighborhood search. In M. Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 468–481. Springer, Heidelberg, Germany, 2004.

[19] P. Rastas and E. Ukkonen. Haplotype inference via hierarchical genotype parsing. In R. Giancarlo and S. Hannenhalli, editors, *Proceedings of the 7th Workshop on Algorithms in Bioinformatics – WABI2007*, volume 4645 of *Lecture Notes in Computer Science*, pages 85–97. Springer, Heidelberg, Germany, 2007.

[20] A. Roli, S. Benedettini, T Stützle, and C. Blum. Additional material to the paper 'Large Neighbourhood Search Algorithms for the Founder Sequences Reconstruction Problem', 2010. Available at `www.lia.deis.unibo.it/~aro/research/fsrp/`.

[21] A. Roli and C. Blum. Tabu search for the founder sequence reconstruction problem: A preliminary study. In S. Omatu, M.P. Rocha, J. Bravo, F. Fernández-Riverola, E. Corchado, A. Bustillo, and J.M. Corchado, editors, *Proceedings of the 3rd International Workshop on Practical Applications of Computational Biology and Bioinformatics – IWPACBB'09*, volume 5518 of *Lecture Notes in Computer Science*, pages 1035–1042. Springer, Heidelberg, Germany, 2009.

[22] R. Ruiz and T. Stützle. A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research*, 177(3):2033–2049, 2007.

[23] R. Schwartz, A. Clark, and S. Istrail. Methods for inferring block-wise ancestral history from haploid sequences. In R. Guigó and D. Gusfield, editors, *Algorithms in Bioinformatics*, volume 2452 of *Lecture Notes in Computer Science*, pages 44–59. Springer, Heidelberg, Germany, 2002.

[24] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In M. Maher and J.-F. Puget, editors, *Principle and Practice of Constraint Programming – CP98*, volume 1520 of *Lecture Notes in Computer Science*, pages 417–431. Springer, Heidelberg, Germany, 1998.

[25] Y.S. Song, Y. Wu, and D. Gusfield. Efficient computation of close lower and upper bounds on the minimum number of recombinations in biological sequence evolution. *Bioinformatics*, 21(suppl_1):413–422, 2005.

[26] G. W. Thyson, J. Chapman, P. Hugenholtz, E. Allen, R. Ram, P. Richardson, V. Solovyev, E. Rubin, D. Rokhsar, and J. Banfield. Community structure and metabolism through reconstruction of microbial genomes from the environment. *Nature*, 428:37–43, 2004.

[27] E. Ukkonen. Finding founder sequences from a set of recombinants. In R. Guigó and D. Gusfield, editors, *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics – WABI 2002*, volume 2452 of *Lecture Notes in Computer Science*, pages 277–286. Springer, Heidelberg, Germany, 2002.

[28] Y. Wu. An analytical upper bound on the minimum number of recombinations in the history of SNP sequences in populations. *Information Processing Letters*, 109(9):427–431, 2009.

[29] Y. Wu and D. Gusfield. Improved algorithms for inferring the minimum mosaic of a set of recombinants. In B. Ma and K Zhang, editors, *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching – CPM 2007*, volume 4580 of *Lecture Notes in Computer Science*, pages 150–161. Springer, Heidelberg, Germany, 2008.

[30] Q. Zhang, W. Wang, L. McMillan, F.P-M. De Villena, and D Threadgill. Inferring genome-wide mosaic structure. *Bioinformatics*, pages 150–161, 2009.