

A Randomized Iterated Greedy Algorithm for the Founder Sequence Reconstruction Problem

Stefano Benedettini¹, Christian Blum², and Andrea Roli¹

¹ DEIS, *Alma Mater Studiorum* Università di Bologna, Campus of Cesena, Italy
{s.benedettini|a.roli}@unibo.it

² ALBCOM Research Group, Universitat Politècnica de Catalunya, Barcelona, Spain
cblum@lsi.upc.edu

Abstract. The problem of inferring ancestral genetic information in terms of a set of founders of a given population arises in various biological contexts. In optimization terms, this problem can be formulated as a combinatorial string problem. The main problem of existing techniques, both exact and heuristic, is that their time complexity scales exponentially, which makes them impractical for solving large-scale instances. Basing our work on previous ideas outlined in [1], we developed a randomized iterated greedy algorithm that is able to provide good solutions in a short time span. The experimental evaluation shows that our algorithm is currently the best approximate technique, especially when large problem instances are concerned.

1 Introduction

Technical advances in sequencing of genetic material has led to a rapid growth of available DNA sequences and haplotyped sequences. Given a sample of sequences from a population of individuals (for example, humans) one may study the evolutionary history of those individuals on the basis of their genetic information. This is important, for example, for the discovery of the genetic basis of complex diseases. In case the population from which the sample sequences are taken has evolved from a relatively small number of founders, the evolutionary history can be studied by trying to reconstruct the sample sequences as fragments from the set of founder sequences. This genetic model, which is central to the problem tackled in this paper, was used, for example, in [2,3]. Many findings from biological studies support the validity of this model, as, for example, [4]. The major problem is that neither the number of founder sequences, nor the founder sequences themselves, may be known. Ukkonen [2] proposed a computational problem that, given the number k of founder sequences, consists in finding a set of k sequences such that the set of sample sequences, also called recombinants, can be reconstructed using as few fragments as possible. This problem is known as the *founder sequence reconstruction problem* (FSRP) or the *minimum mosaic problem* [3] and it is NP-complete [5]. A technical description of the problem is given in the following section.

Existing techniques. The first algorithm that was developed for the FSRP is based on dynamic programming [2]. However, this algorithm does not scale well when the number of founders or the number/length of the recombinants grows. The authors of [3] proposed an exact solver based on tree search, called RECBLOCK. This solver, currently the state of the art for what complete solvers are concerned, can also be applied as a heuristic with varying levels of sophistication. While the results of RECBLOCK are very good for rather small numbers of founders, even the heuristic variants still do not scale well when, for example, the number of founders grows. This was our motivation for the development of a simple constructive heuristic and a tabu search algorithm in [1]. Although the proposed tabu search solver proved superior both to the simple constructive heuristic and to the heuristic variants of RECBLOCK, it showed two major drawbacks:

- Running times are still too long, that is, the computation of the employed neighborhood structure is quite time intensive.
- Considering the elevated running times, the improvement over the simple constructive heuristic was not impressive.

These two drawbacks provided us with the motivation for the work presented in this paper. First, we wanted to study if the simple constructive heuristic proposed in [1] can be improved by the incorporation of look-ahead techniques. Second, we wanted to explore the possibilities of extending the simple constructive heuristic in order to obtain an iterated greedy algorithm [6], which is a metaheuristic based on the construction and partial destruction of solutions. This was done with the intention of developing a fast algorithm that does not need any time-intensive neighborhood search routine for providing good solutions.

The remainder of the paper is organized as follows. In Section 2 we technically introduce the FSRP. Section 3 is devoted to the introduction of the simple constructive heuristic extended by a look-ahead mechanism, while in Section 4 is introduced the randomized iterated greedy algorithm. The subsequent part of the paper is devoted to the experimental evaluation. We followed a two-phase experimental analysis of our algorithms, consisting of a parameter tuning phase as described in Section 5.1, and a comparison to the state of the art as presented in Section 5.2. Finally, conclusions and an outlook to future work are given in Section 6.

2 The Founder Sequence Reconstruction Problem

The founder sequence reconstruction problem (FSRP) can technically be described as follows. Given is a set of m *recombinants* $\mathcal{C} = \{C_1, \dots, C_m\}$. Each recombinant C_i is a string of length n over a given alphabet Σ : $C_i = c_{i1}c_{i2} \dots c_{in}$ with $c_{ij} \in \Sigma \forall j$. In this work we will consider a typical biological application where the recombinants are haplotyped sequences and $\Sigma = \{0, 1\}$. The symbols 0 and 1 encode the two most common alleles of each haplotype site.

1 1 0 1 1 0 1		b b b b b c c
1 0 1 0 0 0 1	0 1 1 0 1 0 0	c c c c c c c
0 1 1 1 1 1 1	1 1 0 1 1 1 1	a a a b b b b
0 1 1 0 1 0 0	1 0 1 0 0 0 1	a a a a a a a
1 1 0 0 0 1 1		b b b c c b b
(a) Set of recombinants $\mathcal{C}$	(b) Set of founders $\mathcal{F}$	(c) Decomposition

Fig. 1. (a) shows a set of 5 recombinants in matrix form. Assuming that the number of founders is fixed to 3, (b) shows a valid solution as a matrix of 3 founders. Denoting the first founder by "a", the second founder by "b", and the third one by "c", (c) shows a decomposition of the recombinants matrix into fragments taken from the founders. This decomposition produces the minimum number of breakpoints points, namely 4. Note that breakpoints are marked by vertical lines. This example is reproduced from [3].

A *candidate solution* to the problem consists of a set of k *founders* $\mathcal{F} = \{F_1, \dots, F_k\}$. Each founder F_i is a string of length n over the alphabet Σ : $F_i = f_{i1}f_{i2} \dots f_{in}$ 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 $p_i \leq n$ fragments (that is, strings) $Fr_{i1}Fr_{i2} \dots Fr_{ip_i}$, 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 *polynomial time* (see [3]) a so-called *minimal decomposition*. This is a decomposition where $\sum_{i=1}^n p_i - n$ 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 optimization goal considered in this paper is the following one. Given a fixed k , that is, a fixed number of founders, find a valid solution $\mathcal{F}^*$ that minimizes $f(\cdot)$. For an example, see Fig. 1.

3 A Simple Constructive Heuristic With Look-Ahead

In the following we outline a randomized version of the simple constructive heuristic proposed in [1] extended by a look-ahead technique. This algorithm, which can be applied in a multi-start fashion as shown in Alg. 1, is henceforth denoted by GREEDY. In the following we explain in detail the working of function `ComputeRandomizedSolution()`. Note that throughout the presentation of our algorithms, the set of recombinants $\mathcal{C}$ is regarded a matrix with m rows and n columns. In the same way, a solution $\mathcal{F}$ is a matrix with m rows and k columns.

Initialization and filling of the first column. The solution construction process starts by filling the first column of $\mathcal{F}$, which is done as follows. First, the fraction

Algorithm 1 GREEDY

```
1: INPUT:  $nt \geq 1, lh \geq 1, rnd \in [0, 1]$ 
2:  $\mathcal{F} \leftarrow \text{ConstructRandomizedSolution}()$ 
3: while termination conditions not met do
4:    $\mathcal{F}' \leftarrow \text{ConstructRandomizedSolution}()$ 
5:   if  $f(\mathcal{F}') < f(\mathcal{F})$  then  $\mathcal{F} \leftarrow \mathcal{F}'$  endif
6: end while
7: OUTPUT:  $\mathcal{F}$ 
```

p of 0-entries in the first column of $\mathcal{C}$ is derived. Then, two counters are introduced; counter n_0 for the 0-entries in the first column of $\mathcal{F}$, and counter n_1 for the 1-entries in the first column of $\mathcal{F}$. Both counters are initialized to 1 to ensure at least one 0-entry, respectively one 1-entry. Finally, a random number q from $[0, 1]$ is drawn $k - 2$ times. In case $q \leq p$ counter n_0 is incremented, n_1 otherwise. The first column is then composed of n_0 0-entries, followed by n_1 1-entries. After filling the first column, some data structures are initialized. For each row i of $\mathcal{C}$ is kept a variable cp_i that stores the position of the last breakpoint. These variables are initialized to 0, because no breakpoint exists yet. More specifically, $cp_i = 0$, for $i = 1, \dots, m$. Moreover, a variable rep_i is kept that stores the index of the founder that represents row i of $\mathcal{C}$ after the last breakpoint cp_i . For all rows of $\mathcal{C}$ with a 0-entry in the first column this variable is initialized to 0, while for each row of $\mathcal{C}$ with a 1-entry the respective variable is initialized to $n_0 + 1$, that is, the first row of $\mathcal{F}$ with a 1-entry in the first column. More specifically, $rep_i = 0$ if $c_i = 0$, and $rep_i = n_0 + 1$ otherwise.

Filling of a column. After filling the first column of $\mathcal{F}$ and the initialization of the data structures, solution $\mathcal{F}$ is completed iteratively by filling one column after another. In the following we first outline the mechanism without look-ahead procedure. Let us assume that the first $j - 1$ columns are already filled, and let us denote the corresponding partial solution by $\mathcal{F}_1^{j-1}$. Accordingly, the current column to be filled is column j . The positions of column j are filled one after the other, starting from row 1. For filling position f_{ij} , let n_0 be the number of rows of $\mathcal{C}$ that are represented by founder i and that have a 0-entry in position j . More specifically, n_0 is the number of rows r of $\mathcal{C}$ with $rep_r = i$ and $c_{rj} = 0$. Correspondingly, n_1 is the number of rows r of $\mathcal{C}$ with $rep_r = i$ and $c_{rj} = 1$. The actual setting of f_{ij} depends on parameter $rnd \in [0, 1]$ that determines the amount of stochasticity that is introduced into the solution construction. A random number q is drawn uniformly at random from $[0, 1]$. If $q < rnd$, f_{ij} is set to 1 with probability $\frac{n_1}{n_1 + n_0}$, and to 0 otherwise. If $q \geq rnd$, f_{ij} is set to 1 in case $n_1 > n_0$, and $f_{ij} = 0$ in case $n_0 > n_1$. Otherwise (that is, in case $n_0 = n_1$) a value for f_{ij} is chosen uniformly at random. This means that, even when $rnd = 0$, there is still some randomness in the solution construction, introduced by cases where $n_0 = n_1$.

If, after assigning a value to f_{ij} , row i can not be represented anymore by its current representant, one may try to change its representant by an equally good

Algorithm 2 General iterated greedy (IG) algorithm

```
1:  $s \leftarrow \text{GenerateInitialSolution}()$ 
2: while termination conditions not met do
3:    $s_p \leftarrow \text{Destruction}(s)$ 
4:    $s' \leftarrow \text{Construction}(s_p)$ 
5:    $s \leftarrow \text{AcceptanceCriterion}(s, s')$ 
6: end while
7: OUTPUT: best solution found
```

one. In case $f_{ij} = 0$, this concerns all rows r of $\mathcal{C}$ with $rep_r = i$ and $c_{rj} = 1$; similarly in case $f_{ij} = 1$. For all these rows r of $\mathcal{C}$ a new representing founder l (where $i < l \leq k$) that can equally represent r starting from breakpoint cp_r is searched, that is, a row l in $\mathcal{F}$ (where $i < l \leq k$) such that $c_{rs} = f_{ls}$, for all $s = cp_r, \dots, j - 1$. In case such a founder l can be found, rep_r is set to 1, and the search for an alternative representant for row r is stopped.

As a last step, after filling all the positions of column j , the variables cp_r and rep_r must be updated for all rows r of $\mathcal{C}$ for which $f_{rep_r j} \neq c_{rj}$. In such a case, the founder i with the minimum l such that $c_{rs} = f_{is}$, for all $s = l, \dots, j$ must be determined. After identifying such a founder i , cp_r is set to 1, and rep_r is set to i .

Look-ahead variant. The look-ahead variant of GREEDY depends on parameters nt (the number of trials) and lh (the look-ahead size). Let us assume that a partial solution $\mathcal{F}_1^{j-1}$ is given, which means that column j must be filled. For that purpose, nt matrices $\{J_1, J_2, \dots, J_{nt}\}$ each one composed of k rows and $\min\{lh, m - j\}$ columns are generated. This is done on the basis of the data structures as given by $\mathcal{F}_1^{j-1}$. Note that each matrix J_i represents a possible extension of $\mathcal{F}_1^{j-1}$ by $\min\{lh, m - j\}$ columns. For each matrix J_i the optimal number of breakpoints bps_i obtained by appending J_i to the partial solution $\mathcal{F}_1^{j-1}$ is computed. Let $I = \arg\min\{bps_i\}$. The column to be appended to the partial solution $\mathcal{F}_1^{j-1}$ is then selected to be the first column of J_I .

4 A Probabilistic Iterated Greedy Algorithm

Several examples from the literature have shown that constructive heuristics may be improved by a simple metaheuristic framework known as an iterated greedy (IG) algorithm; see, for example, [6,7,8]). An IG algorithm starts with a complete solution. While some termination conditions are not met, it iteratively alternates between the partial destruction of the incumbent solution (destruction phase) and the re-construction of the resulting partial solution in order to obtain again a complete solution (construction phase). The general pseudo-code is provided in Alg. 2.

The idea for our IG algorithm for the FSRP is based on the fact that solutions to a problem instance can be constructed from left to right, as explained in the

Algorithm 3 BACKFORTH: An iterated greedy (IG) for the FSRP

```
1: INPUT:  $nt \geq 1$ ,  $lh \geq 1$ ,  $rnd \in [0, 1]$ ,  $d \in [0, 0.5]$ ,  $r \geq 1$ 
2:  $\mathcal{F} \leftarrow \text{ConstructRandomizedSolution}()$ 
3:  $right \leftarrow \text{TRUE}$ 
4: while termination conditions not met do
5:    $count = 0$ 
6:    $improved = \text{FALSE}$ 
7:   while  $count < r$  and  $improved = \text{FALSE}$  do
8:      $d_c = \lfloor d \cdot n \rfloor$ 
9:     while  $d_c \leq \lfloor (1 - d) \cdot n \rfloor$  and  $improved = \text{FALSE}$  do
10:       $\mathcal{F}_p \leftarrow \text{Destruction}(\mathcal{F}, d_c, right)$ 
11:       $\mathcal{F}' \leftarrow \text{ReconstructRandomizedSolution}(\mathcal{F}_p, d_c, right)$ 
12:      if  $f(\mathcal{F}') < f(\mathcal{F})$  then  $\mathcal{F} \leftarrow \mathcal{F}'$ ,  $improved \leftarrow \text{TRUE}$  endif
13:       $d_c \leftarrow d_c + 1$ 
14:    end while
15:     $count \leftarrow count + 1$ 
16:  end while
17:   $right \leftarrow \text{not } right$ 
18: end while
19: OUTPUT:  $\mathcal{F}$ 
```

previous section, but also from right to left. Based on this idea we developed the IG algorithm that is pseudo-coded in Alg. 3 and henceforth denoted by BACKFORTH. In the following, the details of this algorithm are explained in depth.

Function `ConstructRandomizedSolution()` uses the constructive heuristic outlined in Sec. 3 for generating an initial solution from left to right. In the main loop, the algorithm tries to improve upon the current solution $\mathcal{F}$ either by removing columns from the right, or by removing columns from the left. This is done in function `Destruction( $\mathcal{F}, d_c, right$ )`, where $right$ is a boolean variable that controls the switch between both variants. More specifically, when $right = \text{TRUE}$ columns are removed from the right hand side, and from the left hand side otherwise. The number of columns that are removed in function `Destruction( $\mathcal{F}, d_c, right$ )` is controlled by a parameter $d \in [0, 0.5]$. The actual number d_c of columns to be removed is first set to $\lfloor d \cdot n \rfloor$. If this does not prove to be successful, d_c is incremented until an upper limit of $\lfloor (1 - d) \cdot n \rfloor$ is reached. This procedure is repeated at most $r \geq 1$ times, where r is another parameter of the algorithm. The use of function `Destruction( $\mathcal{F}, d_c, right$ )` produces a partial solution $\mathcal{F}_p$. Departing from this partial solution, function `ReconstructRandomizedSolution( $\mathcal{F}_p, d_c, right$ )` produces a complete solution $\mathcal{F}'$, employing the constructive heuristic outlined in Sec. 3. In case the newly produced solution is better than the current solution, variable $right$ switches value and the algorithm tries to improve the current solution from the other side.

5 Experimental Evaluation

Algorithms GREEDY and BACKFORTH were implemented in C++, compiled with GCC 3.4.6 and options `-O3 -fno-rtti -fno-exceptions` enabled. All experiments were performed on a cluster composed of dual core 2GHz Intel XeonTM processors with 6Gb of cache and 8Gb of RAM. As it is useful to distinguish between the randomized algorithm versions (that is, when $rnd > 0$) and the quasi-deterministic algorithm versions (that is, when $rnd = 0$) we henceforth refer to the randomized versions as GREEDY-RND, respectively BACKFORTH-RND. Before we present a comparison of our algorithms to the state of the art, we first report on experiments that we performed in order to find a suitable parameter setting.

5.1 Parameter Tuning

For our experimentation we used the same benchmark set as introduced in [1]. This set is composed of randomly generated instances with $m \in \{30, 50\}$ recombinants and $n \in \{2m, 3m, 5m\}$ sites. More specifically, the benchmark set consists of five instances per combination of m and n . The generated instances are valid and not reducible, that is, no columns can be removed without affecting the optimal solution. Concerning our four algorithm types, that is, GREEDY, GREEDY-RND, BACKFORTH, and BACKFORTH-RND, we need to determine the best parameter values for each algorithm type. Moreover, we would like to infer how a certain parameter affects the behaviour of an algorithm type. From now on we refer to an algorithm type coupled with a specific parameter setting as an *algorithm instantiation*.

In the following we remind the reader about the parameters of the different algorithms. First, all algorithms need a setting for (1) the number of trials (nt), (2) the look-ahead size (lh), and the amount of randomness used in the solution construction (rnd). In addition, algorithm BACKFORTH requires a setting for parameters d (which controls the number of columns to be removed in the destruction phase) and r (the number of rounds for trying to improve the incumbent solution). Regarding nt and lh , we tested the following combinations: (1, 1), (5, 1), (5, 2), (5, 5), (10, 1), (10, 2), (10, 5). The amount of randomness, rnd , may be selected from $\{0.0, 0.01, 0.1, 0.2\}$. For what concerns parameter d we allowed the following settings: $d \in \{0.1, 0.25, 0.4\}$. Finally, parameter r was set to 5 after initial experiments. These options result in 72 different algorithm instantiations. For the tuning experiments we selected 12 problem instances, one for each combination of m and n , as a *training set*. We applied each algorithm instantiation 10 times to each instance of the training set, for $k \in \{3, 5, 7, 10\}$. For $k = 3$ we excluded the combinations of with $nt > 5$ since the number of possible columns for a 3-founder solution is six. For each run we used a computation time limit of 50 seconds for the instances with 30 founders, and a computation time limit of 100 seconds for the ones with 50 founders.

For analyzing the results we employed a rank-based procedure as described in the following. In order to study if the relative performance of the different algorithm instantiations depends on the number k of founders, we performed the same analysis for each number of founders. For each problem instance we computed the average over the 10 applications of each algorithm instantiation. Then, for each problem instance, we ordered the 72 algorithm instantiations according to the average they achieved, that is, the algorithm instantiation with the best average obtains rank 1, etc. Ties were given the same rank. Then, for each algorithm instantiation we computed the average rank by averaging over all instances of the training set. Afterwards, the 72 algorithm instantiations were ordered according to their average rank. This approach is particularly useful when dealing with problem instances where the objective function values are in different scales, like in our case. In Tab. 1 for each algorithm type (that is, GREEDY, GREEDY-RND, BACKFORTH, and BACKFORTH-RND) and each founder we report (1) the position of the first occurrence in the final ranking of the algorithm instantiations, (2) its average ranking on all problem instances and (3) its description consisting of algorithm type and relevant parameter settings. These figures clearly demonstrate that the best performing algorithm instance, on average, is BACKFORTH-RND with $nt = 10$ (5 in case $k = 3$), $lh = 1$, $d = 0.1$, and $rnd = 0.2$. In other words, the iterated greedy algorithm has clear advantages over the multi-start heuristic. Moreover, when algorithm BACKFORTH is concerned, randomness is much more useful than in the case of algorithm GREEDY.

In order to show also the qualitative difference between the four algorithms shown in Tab. 1, we compare the average solution qualities they achieved in Fig. 2. This is done in the following way. For each of the 10 applications we summed up the result achieved on all problem instances from the test set. This provides us with 10 values for each of the four algorithm instantiations. These 10 values are shown in the form of box-plot for each algorithm instantiation. The graphics clearly support the conclusions that we have drawn from the results of Tab. 1.

Finally, note that the best algorithm instantiation in the comparison does not use look-ahead (that is, $lh = 1$). This seems counter-intuitive at first. Especially, because the best performing algorithm instantiations of the other three algorithm types all use a look-ahead value greater than one. The reason seems that, given a limited amount of time, instead of look-ahead it is better to explore as many solutions as possible. The graphics of Fig. 3 support this claim. Boxplots show the performance of the best instance of BACKFORTH-RND on the whole training set over the 10 runs when varying the values of the (nt, lh) parameters. The y-axis reports the average solution value. As shown in the graphics, incrementing the look-ahead is detrimental to the algorithm performance, while incrementing the number of trials is beneficial. When $k = 10$ this phenomenon can also be observed for other algorithm instantiations as shown in Tab. 1(d).

Table 1. Rank-based analysis of the tuning experiments. The results are presented separately for each different number k of founders.

(a) Ranking for $k = 3$.

Rank	Average rank	Algorithm type	nt	lh	d	rnd
1	1.167	BACKFORTH-RND	5	1	0.1	0.2
30	32.667	BACKFORTH	5	5	0.1	
43	39.833	GREEDY-RND	5	5		0.1
44	40.167	GREEDY	5	5		

(b) Ranking for $k = 5$.

Rank	Average rank	Algorithm type	nt	lh	d	rnd
1	1.500	BACKFORTH-RND	10	1	0.1	0.2
40	37.000	BACKFORTH	10	5	0.1	
72	60.000	GREEDY-RND	10	5		0.01
74	61.333	GREEDY	10	5		

(c) Ranking for $k = 7$.

Rank	Average rank	Algorithm type	nt	lh	d	rnd
1	1.667	BACKFORTH-RND	10	1	0.1	0.2
17	18.167	BACKFORTH	10	5	0.4	
68	53.000	GREEDY-RND	10	5		0.1
70	54.167	GREEDY	10	5		

(d) Ranking for $k = 10$.

Rank	Average rank	Algorithm type	nt	lh	d	rnd
1	3.167	BACKFORTH-RND	10	1	0.1	0.2
3	6.833	BACKFORTH	10	2	0.1	
64	50.833	GREEDY	10	2		
67	51.667	GREEDY-RND	10	5		0.01

5.2 Comparison to the State of the Art

We tested the best algorithm instantiation as determined by the parameter tuning phase—henceforth denoted by BEST—against all techniques used in [1]. For consistency reasons we maintain the same algorithm notifiers in the result tables as used in [1]. This means that *heuristic* actually refers to GREEDY with $nt = 1$, $lh = 1$, and $rnd = 0$. Moreover, *TS* refers to the tabu search presented in [1]. The remaining algorithms are three variants of RECBLOCK: (a) an exact version (*rec-exact*), (b) a sophisticated heuristic variant (*rec-heuristic*), and (c) the lightest heuristic version (*rec-DOC1*).

Remember that the benchmark set consists of 60 problem instances as outlined at the beginning of Sec. 5.1. Each instance was considered in combination with different numbers of founders, more specifically, we considered $k \in \{3, \dots, 10\}$. Then, as a first experiment we applied BEST for one hour to each

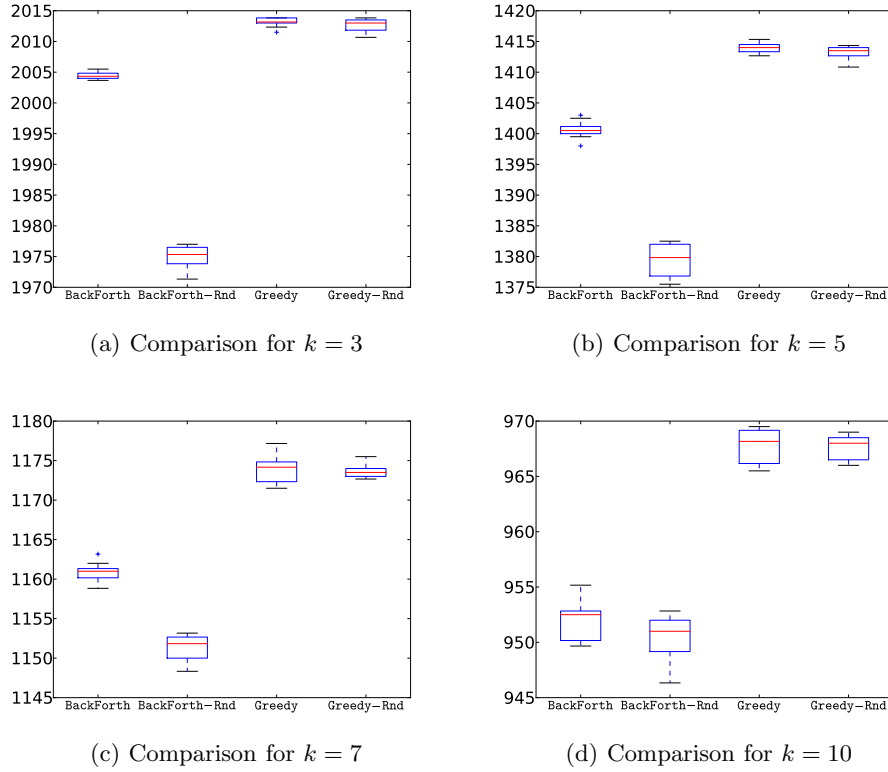


Fig. 2. A comparison regarding the solution quality of the best algorithm instantiations of the four algorithm types.

combination of an instance and a founder number k exactly once. Results are summarized in Tabs. 2 and 3 in which the average solution qualities and the corresponding standard deviations are reported. Statistics are taken over the five instances per combination of the number of recombinants (m) and sites (n). Note that the results in [1] were also obtained with a computation time limit of one hour for each run. Even though the results from [1] were obtained on different processors, they are comparable because the processors have a similar speed.³

The results show that our algorithm achieves, in each case, a better performance than *rec-D0C1*, *heuristic*, and *TS*. This is remarkable, because *TS* is build upon a sophisticated neighborhood structure, whereas our randomized iterated greedy algorithm is very un-sophisticated in comparison. For what concerns

³ Consider that all algorithms implemented in this paper are single-threaded and do not take advantage of parallel architectures; for what RAM is concerned, the algorithms in this paper use less than 5Mb.

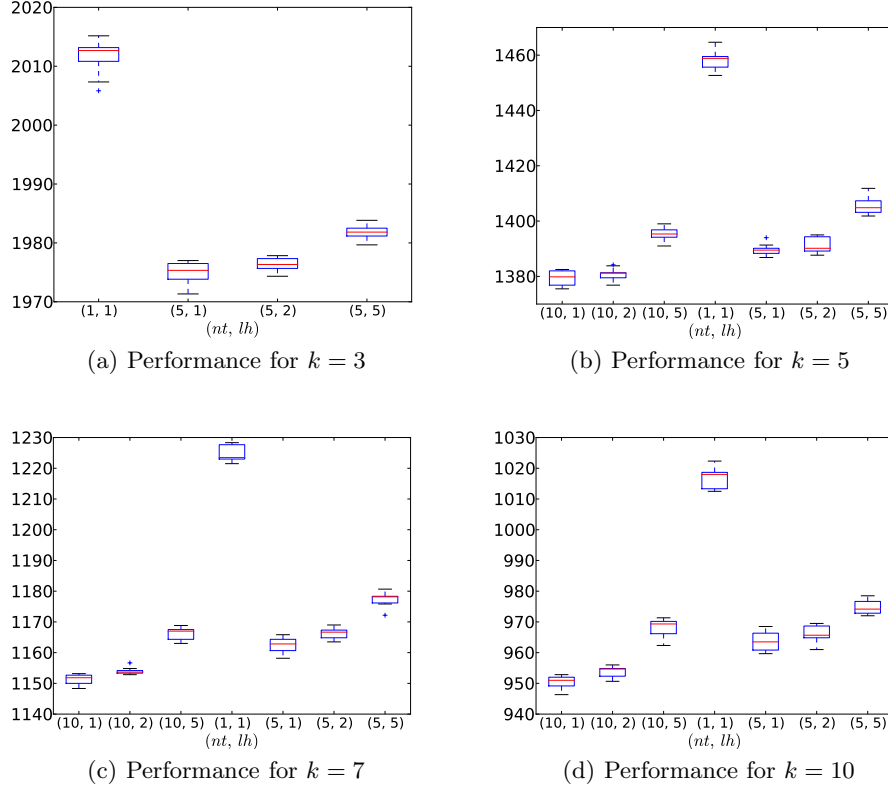


Fig. 3. A comparison of algorithm instantiations of BACKFORTH-RND with varying values of nt and lh .

the comparison to *rec-exact* and *rec-heuristic*, our algorithm is generally inferior. However, *rec-exact* and *rec-heuristic* fail rather soon (that is, with growing problem size) to produce any solution within the allotted time of one CPU hour.

In order to study the development of the solution quality obtained by our algorithm over time, we run the algorithm also with other computation time limits. In particular, we were interested in the behaviour of our algorithm when computation time limits are much more restrictive. Table 4 shows three additional computation time limits that we used. The resulting algorithm instantiations are denoted by BEST I, BEST II, and BEST III. In Fig. 4 we show the results in the following form. Results are averaged over all instances with the same number of recombinants. For each different founder number we show the results of the BEST with the four different computation time limits in terms of the percent deviation with respect to the results achieved by *TS*. First, it is interesting to note that in all cases the percent deviation is positive, which means that with all tested computation time limits our algorithm is better than *TS*. This is es-

Table 2. Results for instances with 30 recombinants. Results are averaged over 5 random instances. The symbol ‘—’ indicates that no solution was returned. Standard deviations are reported in brackets.

30 recombinants sites , founders	rec-exact	rec-heuristic	rec-D0C1	heuristic	TS	BEST
60 , 3	573.8 (12.38)	579.4 (11.5)	604 (16.11)	594.2 (13.08)	583 (11.79)	574.6 (11.52)
60 , 4	445.4 (5.59)	450.2 (6.53)	494.2 (18.27)	479.6 (9.18)	459.6 (7.5)	448.2 (5.34)
60 , 5	—	385.2 (7.85)	425.4 (10.06)	412.2 (8.87)	395.8 (9.36)	384.6 (7.42)
60 , 6	—	340.6 (5.18)	383.6 (5.13)	367.6 (6.88)	352 (6.6)	339.6 (6.34)
60 , 7	—	303.6 (5.64)	353.8 (10.06)	335.2 (7.22)	318.2 (6.76)	306.6 (4.76)
60 , 8	—	274.6 (3.71)	331 (8.75)	311.6 (5.77)	291.2 (4.38)	281.8 (4.53)
60 , 9	—	—	307.4 (10.29)	288.6 (6.47)	270.4 (4.51)	258.8 (6.49)
60 , 10	—	—	294 (9)	268.4 (4.56)	251.8 (4.32)	237.8 (5.71)
90 , 3	877.2 (2.95)	885.2 (3.96)	917.8 (12.83)	910.8 (8.01)	892 (4.58)	879.6 (1.50)
90 , 4	684.2 (3.27)	689.4 (4.34)	749.4 (5.81)	741.6 (7.16)	711.8 (4.02)	690.0 (3.63)
90 , 5	—	596.2 (4.49)	653 (14.23)	645.6 (3.21)	618.6 (3.78)	600.2 (4.79)
90 , 6	—	525 (2.45)	584.2 (7.85)	580.2 (4.32)	552.8 (4.76)	532.8 (3.19)
90 , 7	—	469.4 (3.91)	542 (22.29)	529.8 (6.76)	500.4 (4.16)	482.4 (3.44)
90 , 8	—	424.4 (2.7)	498.8 (17.47)	491 (4)	461.2 (2.17)	444.4 (1.36)
90 , 9	—	—	469.8 (6.1)	456.2 (4.92)	427.8 (3.9)	409.2 (2.40)
90 , 10	—	—	438.2 (7.05)	427 (4.85)	398.8 (3.35)	377.6 (3.38)
150 , 3	1468.8 (21.7)	1482.6 (17.87)	1533.4 (16.46)	1529 (16.12)	1500.6 (18.65)	1480.0 (18.74)
150 , 4	1140.4 (9.42)	1154.4 (5.18)	1249 (18.72)	1253.2 (12.77)	1200.8 (10.76)	1157.6 (9.09)
150 , 5	—	991.6 (8.2)	1083.8 (20.68)	1090.8 (9.88)	1041.6 (10.78)	1005.8 (7.03)
150 , 6	—	876.2 (6.26)	971.2 (3.49)	980 (4.8)	932 (9.14)	899.4 (5.92)
150 , 7	—	—	888.8 (12.03)	897 (4.47)	848.2 (6.42)	817.8 (3.43)
150 , 8	—	—	819.2 (5.36)	831.8 (4.6)	783.2 (4.71)	748.2 (5.42)
150 , 9	—	—	770.2 (12.64)	773 (3.39)	727.6 (3.71)	700.6 (4.76)
150 , 10	—	—	715.2 (9.52)	724.8 (2.68)	676.6 (3.78)	646.6 (4.59)

pecially remarkable for the shortest computation time limits of 50 seconds per run for instances with 30 recombinants and 100 seconds per run for instances

Table 3. Results for instances with 50 recombinants. Results are averaged over 5 random instances. The symbol ‘—’ indicates that no solution was returned. Standard deviations are reported in brackets.

50 recombinants sites , founders	rec-exact	rec-heuristic	rec-D0C1	heuristic	TS	BEST
100 , 3	1765.4 (16.96)	1784.4 (14.64)	1837.8 (31.03)	1821.2 (18.02)	1789 (15.18)	1775.0 (14.57)
100 , 4	1377.6 (10.88)	1392.2 (9.39)	1481.8 (24.63)	1483.8 (8.23)	1425.2 (13.95)	1388.8 (11.23)
100 , 5	—	1225.2 (14.72)	1305 (17.36)	1301.2 (15.06)	1260.6 (14.43)	1232.2 (10.61)
100 , 6	—	1095.8 (13.92)	1177.6 (12.16)	1188.4 (15.08)	1140.2 (11.21)	1115.6 (12.67)
100 , 7	—	997.8 (10.99)	1087.8 (15.9)	1101.4 (9.89)	1049.4 (9.13)	1027.6 (10.33)
100 , 8	—	920.4 (9.71)	1026.8 (6.3)	1034.8 (9.78)	976 (9.62)	960.4 (9.54)
100 , 9	—	—	963.8 (14.82)	976.2 (13.59)	915 (11.73)	897.0 (6.03)
100 , 10	—	—	918.8 (6.76)	928.4 (10.64)	868 (8.34)	851.2 (3.49)
150 , 3	2631.2 (22.88)	2660.6 (22.74)	2740.8 (29.3)	2722.6 (23.99)	2677.4 (23.56)	2647.6 (22.92)
150 , 4	2056.8 (5.72)	2078.8 (6.91)	2194.2 (26.48)	2240.6 (6.88)	2148.2 (8.41)	2091.6 (10.48)
150 , 5	—	1823.2 (8.32)	1936.8 (12.74)	1965 (9.46)	1894.8 (8.35)	1857.2 (9.02)
150 , 6	—	1635.8 (12.85)	1759.6 (9.66)	1794.8 (6.8)	1717.8 (7.16)	1683.2 (12.67)
150 , 7	—	1493.2 (11.19)	1644 (12.53)	1668 (9.22)	1578.8 (10.18)	1554.4 (8.01)
150 , 8	—	—	1528.8 (13.24)	1562.8 (10.01)	1475.2 (10.96)	1450.4 (5.12)
150 , 9	—	—	1443.8 (6.69)	1479.2 (14.74)	1386 (8.86)	1365.6 (7.91)
150 , 10	—	—	1376.8 (15.59)	1403.2 (11.56)	1314.8 (5.81)	1299.0 (6.03)
250 , 3	4421 (22.06)	4466.2 (20.46)	4597.8 (33.69)	4601.6 (15.53)	4514.8 (11.95)	4475.8 (18.48)
250 , 4	3448.67 (4.73)	3490.8 (10.76)	3728.8 (8.53)	3813.6 (7.54)	3634.2 (13.88)	3542.6 (15.29)
250 , 5	—	3071.4 (15.98)	3258.4 (33.25)	3344 (21.12)	3218.8 (11.69)	3151.6 (22.97)
250 , 6	—	2754.4 (14.17)	2967.8 (24.77)	3046.8 (11.37)	2915.8 (17.31)	2864.2 (21.39)
250 , 7	—	2510.6 (9.4)	2735.6 (20.89)	2832 (13.82)	2686.6 (11.8)	2643.6 (11.46)
250 , 8	—	—	2570.6 (22.06)	2648.8 (17.77)	2504.8 (12.93)	2472.6 (10.09)
250 , 9	—	—	2422 (30.24)	2505.8 (14.79)	2358 (9.67)	2324.8 (10.36)
250 , 10	—	—	2304.4 (28.06)	2378.8 (7.22)	2237.2 (7.6)	2203.8 (5.49)

with 50 recombinants. Finally, as expected, the graphics show clearly that our algorithm improves with growing computation time limits.

On the negative side, the obtained solution quality does not improve impressively over time. Therefore we conclude that, although BEST is a valuable and scalable heuristic, it does not take the best possible advantage of larger time limits. That would suggest that our algorithm is especially suited either as a fast heuristic upper bound on medium- and large-sized problem instances or as an improvement step in the context of some population-based metaheuristic, such as a memetic algorithm or ant colony optimization. From this point of view, the combination of our algorithm with a learning-based method should be quite promising since it can be expected that when learning is incorporated, larger running times can be exploited more beneficially.

Table 4. Alternative computation time limits (in seconds).

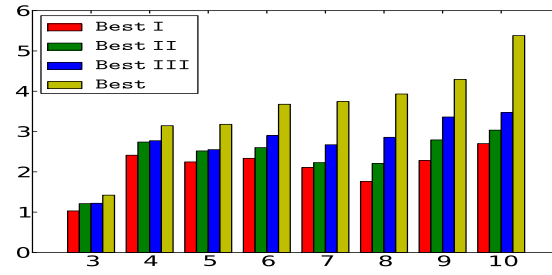
Number of recombinants	Notifier		
	BEST I	BEST II	BEST III
30	50	100	180
50	100	200	600

6 Conclusions and Outlook

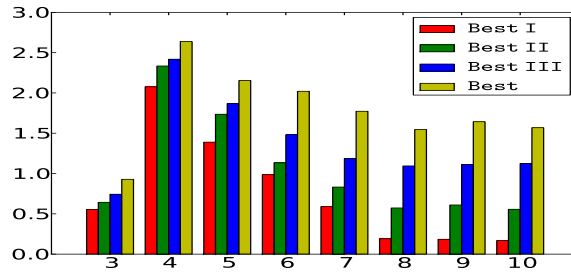
In this paper we have proposed a IG heuristic for tackling large size instances of the FSRP. Results on random instances show that our IG method outperforms the heuristic version of RECBLOCK on large size instances. Since this algorithm starts from a fully constructed solution, it might be possible to use it also as a fast black-box procedure incorporated into some high-level framework. In this regard, we are currently working on an integration of this techniques in a Hypercube ACO metaheuristic.

References

1. Roli, A., Blum, C.: Tabu search for the founder sequence reconstruction problem: A preliminary study. In: IWANN '09: Proceedings of the 10th International Work-Conference on Artificial Neural Networks, Berlin, Heidelberg, Springer-Verlag (2009) 1035–1042
2. Ukkonen, E.: Finding founder sequences from a set of recombinants. In Guigó, R., Gusfield, D., eds.: Proceedings of WABI 2002 – Proceedings of the 2nd Workshop on Algorithms in Bioinformatics. Volume 2452 of Lecture Notes in Computer Science., Springer Verlag, Berlin (2002) 277–286
3. Wu, Y., Gusfield, D.: Improved algorithms for inferring the minimum mosaic of a set of recombinants. In: Proceedings of CPM 2007 – Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching. Volume 4580 of Lecture Notes in Computer Science., Springer Verlag, Berlin (2008) 150–161



(a) Instances with $m = 30$



(b) Instances with $m = 50$

Fig. 4. The y-axis shows the percent deviation of algorithms BEST I, BEST II, BEST III, and BEST over TS . Results are averaged over the instances with the same number of recombinants. The x-axis ranges over different numbers of founders.

4. Thyson, G.W., Chapman, J., Hugenholtz, P., Allen, E., Ram, R., Richardson, P., Solovyev, V., Rubin, E., Rokhsar, D., Banfield, J.: Community structure and metabolism through reconstruction of microbial genomes from the environment. *Nature* (428) (2004) 37–43
5. Rastas, P., Ukkonen, E.: Haplotype inference via hierarchical genotype parsing. In: *Proceedings of WABI2007 – 7th Workshop on Algorithms in Bioinformatics*. (2007)
6. St T.: Iterated local search for the quadratic assignment problem. *European Journal of Operational Research* **174**(3) (2006) 1519 – 1539
7. Ruiz, R., St T.: A simple and effective iterated greedy algorithm for the permutation flowshop scheduling problem. *European Journal of Operational Research* **177**(3) (2007) 2033 – 2049
8. Ruiz, R., St T.: An iterated greedy heuristic for the sequence dependent setup times flowshop problem with makespan and weighted tardiness objectives. *European Journal of Operational Research* **187**(3) (2008) 1143 – 1159