Università degli Studi di Milano

Heuristic algorithms (laboratory sessions)

Roberto Cordone

– Academic year 2020/21 –

Contents

Chapter 6

Recombination metaheuristics

6.1 Introduction

In this chapter we consider a couple of recombination metaheuristics for the *Maximum Diversity Problem* (*MDP*), namely a *Path Relinking* (*PR*) and a *Scatter Search* (*SS*) approach. Recombination-based approaches do not have a common basic scheme, and they very frequently use randomization or memory. Therefore, the distinction between heuristics and metaheuristics is much less easy to draw than for constructive and exchange approaches. In general, however, they manipulate a reference set of solutions extracting from it suitable subsets and combining the elements of such subsets to generate new solutions. The aim is to retrieve promising portions from different solutions and integrate them in a better way, instead of labouriously modifying the bad portions of a single solution while keeping the good ones.

Since *PR* and *SS* algorithms are mainly deterministic methods based on the idea of exploiting as much as possible the information provided by the data, the initial population usually consists of locally optimal solutions generated by a previous exchange heuristic or metaheuristic. In order to make both methods virtually unlimited, we need a mechanism to generate new solutions even when the recombination mechanism fails to do so. For the sake of simplicity, we generate random solutions and improve each of them to local optimality, as we have done in Chapter 4, to restart the steepest ascent heuristic. In order to understand whether the performance of the two approaches depends on the recombination mechanism or not, we will compare them with the simple steepest ascent heuristic with random restart without applying any recombination procedure. In order to allow a fair comparison among these very different approaches, we will impose a limit on the total computational time as a termination condition.

The neighbourhood used in all three approaches will be the usual single-swap neighbourhood N_{S_1}, explored with the first-best strategy. We will discuss later in detail how the mechanism to generate new solutions is integrated in the schemes of *PR* and *SS*. These will be, therefore, more complicated than the ones reported in the theoretical lessons, which assumed the approaches to terminate as soon as the recombination operations fail to update the reference set. We will consider first *PR* and then *SS*, contrary to the theoretical lessons, because the former is simpler and this will allow to introduce more gradually different concepts and algorithmic components.

The command line arguments of the `main` procedure allow the user to choose which of the three metaheuristics to apply and the associated parameters:

- for the *PR* metaheuristic, option `-pr`, followed by three parameters: the total time $\tau_{\max}$, the cardinality of the reference set $|R|$, and the seed of the pseudorandom number generator;
- for the *SS* metaheuristic, option `-ss`, followed by four parameters: the total time $\tau_{\max}$, the cardinality of the best set $|B|$ and of the diverse set $|D|$, and the seed of the pseudorandom number generator;
- for the *random restart* metaheuristic, option `-rr`, followed by two parameters: the total time $\tau_{\max}$ and the seed of the pseudorandom number generator.

The structure of the `main` function is therefore the usual one, with the exception that no starting solution is generated, because the population is initialized inside the two recombination procedures.

```
parse_command_line(argc,argv,data_file,algo,&tauMax,&seed,&nb,&nd);

load_data(data_file,&I);

create_solution(I.n,&x);

start = clock();
if (strcmp(algo,"-ss") == 0)
  scatter_search(&I,&x,"-fb",tauMax,nb,nd,&seed);
else if (strcmp(algo,"-pr") == 0)
  path_relinking(&I,&x,"-fb",tauMax,nb,&seed);
else if (strcmp(algo,"-rr") == 0)
  random_restart(&I,&x,"-fb",tauMax,&seed);
end = clock();
tempo = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6f ",tempo);
print_sorted_solution(&x,I.n);
printf("\n");

destroy_solution(&x);
destroy_data(&I);
```

6.2 Path Relinking

The *Path Relinking* metaheuristic manages a *reference set*, that is composed of the best known solutions. We will denote it as B (instead of R) to stress the similarities with the design of the *Scatter Search* approach. The basic scheme of *PR* extracts pair of solutions from B and applies an auxiliary exchange procedure to move from the first to the second solution of each pair. Then, it identifies the best solution along each path thus determined and improves it with an exchange procedure. Usually, for the sake of simplicity, the exchange procedure used for both purposes is the same used to generate the solutions of the reference set. The only difference is that, when drawing the relinking path, the objective is to minimize the Hamming distance from the final solution, and only secondarily to optimize the objective of the problem. The locally optimal solution found for each pair is checked to decide whether it is worth inserting in the reference set. The process terminates when all solutions generated in this way are rejected, and the reference set is not updated.

Therefore, this scheme has an intrinsic termination condition. In order to prolong it indefinitely, if the candidate solutions generated by the relinking paths are less than a given number, new ones are generated at random and improved by steepest ascent. The same process is applied (with no candidate obtained by recombination) to generate the starting reference set. Then, these candidate solutions are tested for insertion in the reference set. This is not the only possible approach: many others are used in the literature, but this one has the advantage of being simple. An important distinction is between the *static update*, which collects the new solutions in a pool and tests them for insertion in the reference set at the end of the recombination phase, and the *dynamic update*, which immediately tests the new solutions. The former approach has the advantage of a simpler implementation and of exploiting all the generated solutions, whereas the latter can remove a solution from the reference set before using it for recombination. On the other hand, the latter approach is more aggressive, as it immediately exploits the new solutions, possibly leading to good results earlier. We will adopt the static update for the sake of simplicity.

The resulting scheme is the following:

```
Algorithm PathRelinking($I, \tau_{\max}, n_B$)
$B := \emptyset; P := \emptyset$;
While Time() $\leq \tau_{\max}$ do
  { Build or integrate the candidate population }
  While $|P| < n_B$ do
    $x :=$ RandomSolution($I$);
    $x :=$ SteepestAscent($x$);
    If $x \notin P$ then $P := P \cup \{x\}$;
  EndWhile
  { Test the candidate solutions for insertion in the reference set }
  For each $x \in P$ do
    If $x \notin B$ then $B :=$ UpdateBestSet($B, x$);
  EndFor
  { Recombine the solutions in the reference set into candidate ones }
  $P := \emptyset$;
  For each $(x, y) \in B \times B$ do
    $\Gamma_{x,y} :=$ FindRelinkingPath($x, y, I$);
    $z := \arg \max_{w \in \Gamma_{xy} \setminus \{x,y\}} f(w)$;
    $z :=$ SteepestAscent($z$);
    If $z \notin P$ then $P := P \cup \{z\}$;
  EndFor
EndWhile
$x^* := \arg \max_{x \in B} f(x)$;
Return $(x^*, f(x^*))$;
```

Function *Time()* returns the time elapsed from the beginning of the algorithm, in order to enforce the termination condition. In C, this is done with function `clock()`, that returns the number of time units[1] elapsed since the beginning of the execution. For the sake of simplicity, we will call this function only in the outer loop. If performing a single iteration is slow, due to the size and features of the instance, to the size of the reference set or any other factor, the time limit will be actually violated. This should be avoided checking the time also inside the inner loops. Of course, too frequent checks negatively affect the overall computational time.

The first inner loop builds a population of candidate solutions by generating random solutions and improving them with steepest ascent. For the sake of simplicity, the number of candidates is set large enough to fill the reference set (n_B), though in general it could be larger. Duplicate candidates are not accepted, because they would not provide any advantage in the following operations. There is therefore a risk that this loop could not terminate, if many identical locally optimal solutions are obtained. This is rather unlikely, however, with a sufficiently diversifying generation mechanism, such as a random extraction. Notice that in the following iterations of the outer loop, the random candidates will be used only to integrate the solutions generated by recombination when the latter are too few, and that the candidates derived from recombination could be more numerous than n_B, in which case this loop will simply be skipped. Additional parameters could be introduced to tune all these aspects.

The second inner loop tests each candidate solution to check whether it deserves to be included in the reference set B. This is done with a suitable procedure that, in the positive case, adds the solution to update the subset. At the end, the population is cleared, because all its useful elements have been added to the reference set.

The third inner loop considers each pair of solutions (x, y) from B, draws a relinking path from x to y and finds the best solution z on this path (excluding the two extremes). This is then improved by steepest ascent and saved in the population P. The exchange procedure will, as usual, exploit the single-swap neighbourhood, but it will use the global-best strategy when drawing the relinking path, and the first-best strategy when improving the solution. The reason is that the neighourhood used to move between candidate solutions is severely limited by the requirement to strictly reduce the Hamming distance from the final solution, so that a more careful exploration of the neighbourhood seems preferable. When all pairs of reference solutions have been considered, the algorithm starts a new iteration.

The procedures required by this phase of the implementation are in part available in the library `solutionpool`, that manages the allocation and deallocation of solution pools (`create_solutionpool` and `destroy_solutionpool`), the search for a given solution in a pool (`is_in_solutionpool`), the append of a new solution at the end of a pool with residual room (`add_solution_to_pool`), the insertion of a new solution in a pool of solutions sorted by increasing objective values (`update_best_set`), and the removal of all solutions from a pool (`clean_solutionpool`). The generation of a random solution, that was already implemented in Chapter 3, is provided in library `randomsolution`.

The practical implementation creates at the beginning and destroys in the end a current solution `x` and two pools of solutions `B` and `P`, representing the reference set and the population of candidate solutions. Two pools are required by the static update, that first builds all candidates and then tests them for insertion in the reference set. Pool `B` has maximum size `nb`; it is sorted by nondecreasing values of the objective, and solutions are added to it in the second loop by procedure

[1]Typically, they are milliseconds or microseconds, depending on the machine. QUESTO VA ANTICIPATO NEL COMMENTO AL MAIN DEL PRIMO CAPITOLO, DATO CHE SI USA ANCHE LI', LASCIANDO QUI SOLO UN RICHIAMO.

`update_best_set`. Pool `P` is unsorted, solutions are added at the end of the available space in the first and the third loop; its size is `nb*(nb-1)`, because the first loop stops when it reaches `nb` solutions and the third loop considers all pairs of solutions of `B`, though the actual number will probably be much smaller. At the end of the second loop, pool `P` is cleaned: that is simpler and more efficient than deallocating and reallocating it, exactly as it is preferable to clean the auxiliary solution `x` after its use[2]. The only procedure that is used and declared, but not yet implemented, is `find_relinking_solution`, that determines the relinking path and returns its best solution (excluding the two extremes x and y). This is then improved by steepest ascent and included among the candidate ones, if it is not a duplicate.

```
start = clock();
create_solution(pI->n,&x);
create_solutionpool(nb,&B);
create_solutionpool(nb*(nb-1),&P);

while (((double)clock() - start) / CLOCKS_PER_SEC < tauMax)
{
  /* Build or integrate the candidate population */
  while (P.card < nb)
  {
    clean_solution(&x,pI->n);
    generate_random_solution(pI,&x,pseed);
    steepest_ascent(pI,&x,visit_strategy,&iter);
    if (!is_in_solutionpool(&x,pI->n,&P)) add_solution_to_pool(&x,pI->n,&P);
  }

  /* Test the candidate solutions for insertion in the reference set */
  for (s = 1; s <= P.card; s++)
    update_best_set(P.S[s],pI->n,&B);
  clean_solutionpool(&P);

  /* Recombine the solutions in the reference set into candidate ones */
  for (s = 1; s <= B.card; s++)
    for (s2 = 1; s2 <= B.card; s2++)
      if (s != s2)
      {
        clean_solution(&x,pI->n);
        find_relinking_solution(B.S[s],B.S[s2],pI,"-gb",&x);
        steepest_ascent(pI,&x,visit_strategy,&iter);
        if (!is_in_solutionpool(&x,pI->n,&P)) add_solution_to_pool(&x,pI->n,&P);
      }
}
if (B.card > 0) copy_solution(B.S[1],px);

destroy_solution(&x);
destroy_solutionpool(&B);
destroy_solutionpool(&P);
```

The implementation of function `find_relinking_solution` builds upon the steepest ascent procedure. There are two main differences:

[2]An interesting point to discuss on the management of solutions pools is whether to add copies of the new solutions, as it is done at present, or move them physically into the pool. I have not pondered this point enough to take a clear decision.

1. the procedure minimizes first the Hamming distance from the final solution y, then the objective function (in case of ties);
2. the procedure terminates when solution y is reached.

The special features of the *MDP*, in particular its only constraint fixing the cardinality of the solution, implies that:

1. the only way to reduce the Hamming distance of the current solution z from y (always exactly by 2) is to swap a point in $z \setminus y$ with a point in $y \setminus z$;
2. it is possible to focus on the moves that reduce the Hamming distance and avoid all other moves of N_{S_1};
3. there are $|z \setminus y| \cdot |y \setminus z|$ such moves, that is a positive number, as long as $z \neq y$: the best of these moves will be performed.

Since the size of this neighbourhood and the number of moves is probably much smaller than for a typical single-swap, we will adopt the global-best visit strategy for the relinking path identification. An experimental comparison with the first-best strategy should be performed, but we will not do it for lack of time.

```
create_solution(pI->n,&z);
copy_solution(px,&z);

clean_solution(pz,pI->n);
do
{
  explore_neighbourhood_for_relinking(&z,py,pI,visit_strategy,&p_in,&p_out,&delta_f);
  if (p_in != NO_POINT)
  {
    swap_points(p_in,p_out,&z,pI);
    if (z.f > pz->f) copy_solution(&z,pz);
  }

} while (p_in != NO_POINT);
```

The neighbourhood is explored in procedure `explore_neighbourhood_for_relinking` exactly as in the steepest ascent heuristic, with two limitations to guarantee that the solutions explored are closer to the final one than the current solution: the deleted point `p_in` must not belong to the final solution `y`, and the added point `p_out` must belong to the final solution `y`. This is similar to what happens in *Tabu Search*, but much simpler, as it is just a straightforward reduction of the neighbourhood. Notice that, if $k < n/2$, it would be more efficient to scan `p_out` in `py` and check that it is not in `px`.

```
*pdelta_f = INT_MIN;
*pp_in = *pp_out = NO_POINT;
for (p_in = first_point_in(px); !end_point_list(p_in,px); p_in = next_point(p_in,px))
  if (!py->in_x[get_index(p_in,pI)])
    for (p_out = first_point_out(px); !end_point_list(p_out,px); p_out = next_point(p_out,
      if (py->in_x[get_index(p_out,pI)])
      {
        delta_f = evaluate_exchange(p_in,p_out,px,pI);
        if (delta_f > *pdelta_f)
        {
          *pdelta_f = delta_f;
          *pp_in = p_in;
          *pp_out = p_out;
        }
      }
```

6.2.1 Time complexity estimation

An *a priori* estimation of time complexity of the *PR* metaheuristic is quite complex, as it consists of procedures of a very different nature, that include conditions whose occurrence is usually unpredictable. Moreover, in our experiments the computational time is fixed by the user. However, the analysis can suggest the relative weight of the different components of the algorithm, and consequently on which procedures and parameters to focus in order to improve it.

The first inner loop, in which random candidates are generated and improved, is particularly hard to characterize: we know from Chapter 3 that random solutions are usually reduced to a locally optimal one in a number of neighbourhood explorations $t_{\max}$ that increases more than linearly with size. Of course, each exploration takes $O\left(k(n-k)\right)$ time. The main problem is that this loop ends when the population P includes at least n_B solutions, which can take zero time (if the relinking paths are enough to fill it) or a potentially infinite time (if the random initializations repeat over and over again the same locally optimal solutions). A very rough estimate can be $O\left(n_B t_{\max} k(n-k)\right)$ time.

The second inner loop tests all candidate solutions for insertion in B. In the worst case, the candidates could be $n_B(n_B-1)$; each one could require to scan all the reference solutions point by point (if all candidates and all reference solutions have the same value, an extremely unlikely case); finally, each candidate could be copied into the pool: overall, that would be $O\left(n_B^3 k + n_B^2 n\right)$ time. In practice, the candidate solutions are usually n_B and most of them are tested against some reference solutions using only the objective value (a single reference solution will be enough to reject bad candidates), so the time could be as low as $O\left(n_B\right)$.

The third inner loop draws a relinking path for $n_B\left(n_B-1\right)$ pairs of reference solutions. Each path takes at most k neighbourhood explorations and each exploration takes $O\left(k(n-k)\right)$ time, as usual (probably with smaller multiplying coefficients, due to the limitation of the neighbourhood). Then, the best solution along the path is improved by steepest ascent. Overall, this should be $O\left(n_B^2\left(k^2(n-k)+t_{\max}k(n-k)\right)\right)$ time, where $t_{\max}$ is probably smaller than for a random initialization. This is probably the most expensive component of the algorithm, though the first one could also be relevant. A profiler could confirm or disprove it empirically, but its use exceeds the scope of the present discussion[3].

6.2.2 Empirical evaluation

We can now evaluate the performance of the *PR* metaheuristic. We will experiment with typical values for the number of reference solutions n_B, ranging from 5 to 20. Smaller values would not be enough to generate enough new solution by recombination, whereas larger values would take too much time to systematically test all pairs and would also probably generate many duplicate or bad quality locally optimal solutions. Another possible experiment could be to compare the effect of using the global-best or the first-best strategy in the two exchange procedures

[3]Next year, may be.

that, respectively, improve the recombined and the random solutions or build the relinking paths. For the sake of briefness, we will not perform this analysis.

The only other parameter is the total time. From the previous chapters we know that for the largest instances of the benchmark 1 000 neighbourhood explorations correspond to about 4 seconds and that a randomly generated solution requires around 30 neighbourhood explorations. Each generation of the *PR* framework requires to improve about n_B^2 solutions, some of which are fully random, whereas most derive from the relinking paths. A very rough computation suggests that a generation could take about $20^2 \cdot 30/1\,000 \cdot 4 = 48$ seconds. This is quite long with respect to the time used in the previous chapters, though it is probably an overestimate, because the solutions derived from the relinking paths are probably quicker to optimize. As a compromise, we will set the time to $t_{\max} = 30$ seconds.

Notice that if the time limit is tested only at the beginning of each generation, it could expire before the generation has ended, but the algorithm would terminate only later, violating the limit. This happens also for the random restart approach, because the steepest ascent procedure runs to completion before testing the time limit. To avoid these violations, the condition should be tested more often; possibly, a truncated version of steepest ascent should be designed. However, for the sake of simplicity we will skip this step. Our experiments, in fact, show times ranging from 30 to 30.9 seconds: a more precise termination would be desirable, but is not strictly required for our rough analysis.

6.2.3 Parameter tuning

We now compare the effect on the quality of the solution of four different sizes of the reference set, namely $n_B = 5$, 10, 15 and 20. Table 6.1 reports the resulting average gaps over the whole benchmark. For comparison purposes, the table also reports the average gap obtained by the random restart approach. The best performing configurations set $n_B = 15$ and $n_B = 20$.

	PR				*RR*
n_B	5	10	15	20	
Gap	0.26%	0.20%	0.12%	0.12%	0.10%

Table 6.1: Average gaps with respect to the best known result of the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (*RR*)

Figure 6.1 reports the *SQD* diagram of the four configurations, confirming the relations suggested by the average gap (possibly with a prevalence of $n_B = 15$ on $n_B = 20$, since the latter looks more or less dominated by random restart, whereas the former does not).

The corresponding boxplots are reported in Figure 6.2, and approximately suggest the same conclusions.

Finally, applying Wilcoxon's test to compare the *PR* algorithm with $n_B = 15$ with the other tunings we obtain the following results:

1. $n_B = 15$ versus $n_B = 5$

```
W+ = 569, W- = 134, N = 37, p <= 0.001062
```

2. $n_B = 15$ versus $n_B = 10$

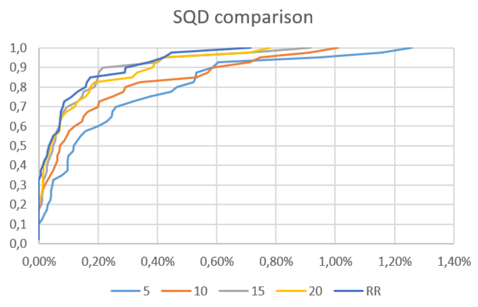

Figure 6.1: Solution Quality Distribution diagrams for the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (RR)

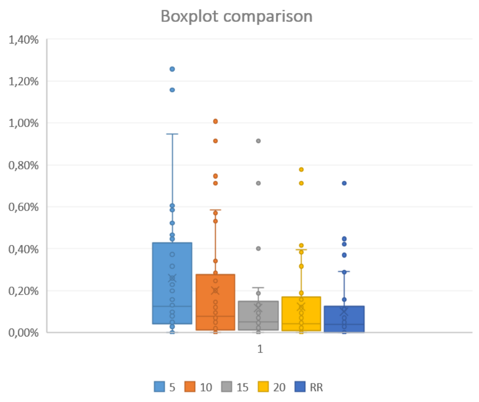

Figure 6.2: Boxplot diagrams for the *PR* metaheuristic with different cardinalities of the reference set (n_B) and of the random restart algorithm (RR)

```
W+ = 206, W- = 119, N = 25, p <= 0.2473
```

3. $n_B = 15$ versus $n_B = 20$

```
W+ = 144, W- = 109, N = 22, p <= 0.581
```

which suggests that only the first comparison is significant, whereas the other ones could easily be the result of a random sampling. So, the results are inconclusive.

Comparison with random restart

The comparison between the *PR* metaheuristic and the steepest ascent with purely random restart allows to estimate the contribution of recombination to the performance. In fact, the *PR* metaheuristic adopts the same method to initialize the population and to integrate it when recombination is not enough to fill the required minimum number of solutions. From the tables and pictures presented above, all configurations of the *PR* algorithm appear to be worse than random restart: even the best one has an average gap of 0.12% versus 0.10%, and a *SQD* diagram that is dominated in most of the range, though not for all values.

Wilcoxon's test however yields the following results:

1. *RR* versus $n_B = 5$

```
W+ = 411, W- = 184, N = 34, p <= 0.05337
```

2. *RR* versus $n_B = 10$

```
W+ = 301, W- = 294, N = 34, p <= 0.9591
```

3. *RR* versus $n_B = 15$

```
W+ = 247, W- = 314, N = 33, p <= 0.5554
```

4. *RR* versus $n_B = 20$

```
W+ = 185, W- = 221, N = 28, p <= 0.6903
```

While we could expect some tests to be inconclusive, it is rather surprising to find out that all of them are. Analyzing the detailed results, one can remark that random restart seems to perform better on smaller instances and worse on larger ones. Now, our application of Wilcoxon's test considers the absolute differences, whereas the *SQD* considers the relative differences, or percent gaps. The former favours the algorithm that is better on large instances (*PR*), whereas the latter favours the algorithm that is better on small instances (*RR*). This could perhaps explain the results. Anyway, the most correct interpretation is probably to suspend any judgment.

Another interesting insight can be obtained by plotting the values of the solutions met along the relinking paths: we can see that typically the objective is good (that is, large) in the two extreme solutions and decreases in the intermediate ones. Therefore, the best solution is usually the first or the last along the path, that is a neighbour of the given locally optimal solutions[4]: the subsequent exchange heuristic is unlikely to find better solutions. This means that the improvement is mainly due to the random generation step, and therefore increasing the time dedicated to this step by removing the recombination mechanism is actually profitable. In the largest instances, however, this is not always the case: the recombination mechanism starts yielding new solutions, whose quality is better than that of the random ones. These are however just speculations, working hypothesis for further studies, that should be verified checking the actual behaviour of the algorithm on some instances.

6.3 Scatter Search

The *Scatter Search* metaheuristic also manages a *reference set* R, that is however composed of two subsets: B includes the best known solutions, D includes the most diverse solutions. The basic scheme extracts pairs of solutions, one from B and one from $B \cup D$, and combines them with a suitable procedure. Then, it improves the new solution by an exchange procedure and checks whether the locally optimal solution thus obtained is worth inserting in B (first) or in D (then). The process terminates when all solutions generated in this way are rejected, and the reference set is not updated. This scheme has an intrinsic termination condition, at least if the recombination procedure is deterministic. In fact, applying it to the same set of reference solutions, it generates the same locally optimal solutions. If the recombination mechanism is randomized, different solutions could be obtained in different iterations, but the process is very intensifying, so that, even if it is profitable at first, in the long term it is likely to lead to repeated solutions and stagnation.

In order to implement a potentially unlimited algorithm with a user-defined time limit, we exploit the same mechanism used for *PR* in the previous section, that is also the same mechanism used to generate the starting population: new solutions are produced by generating random solutions and improving them with steepest ascent. Then, all solutions of the new population are tested for insertion in the reference set, first in B and then in D. For the sake of simplicity, we will adopt once again a static update mechanism.

The resulting scheme is the following.

[4]A possible idea could be to abandon the standard scheme of *PR* and improve an intermediate solution, instead of the best one along the relinking path.

```
*Algorithm* ScatterSearch$(I, \tau_{\max}, n_B, n_D)$
$B := \emptyset; D := \emptyset; P := \emptyset;$
*While* Time() $\leq \tau_{\max}$ *do*
  { Build or integrate the candidate population }
  *While* $|P| < n_B + n_D$ *do*
    $x :=$ RandomSolution$(I)$;
    $x :=$ SteepestAscent$(x)$;
    *If* $x \notin P$ *then* $P := P \cup \{x\}$;
  *EndWhile*
  { Test the candidate solutions for insertion in the reference set }
  *For each* $x \in P$ *do*
    *If* $x \notin B$ *then* $B :=$ UpdateBestSet$(B, x)$;
    *If* $x \notin B \cup D$ *then* $D :=$ UpdateDiverseSet$(D, x, B)$;
  *EndFor*
  { Recombine the solutions in the reference set into candidate ones }
  *For each* $(x, y) \in B \times (B \cup D)$ *do*
    $P := P \cup$ Recombine$(x, y, I)$;
  *EndFor*
*EndWhile*
$x^* := \arg\max_{x \in B} f(x)$;
*Return* $(x^*, f(x^*))$;
```

Notice the strong structural similarity with the scheme of *PR*. The first inner loop builds a population of candidate solutions by generating random solutions and improving them with steepest ascent. For the sake of simplicity, the number of candidates is set large enough to fill the reference set ($n_B + n_D$). As in *PR*, duplicate candidates are not accepted, because they would provide no advantage, and a (very limited) risk of not terminating the loop exists.

The second inner loop tests each candidate solution to check whether it deserves to be included in either of the two subsets B and D. This is done with suitable procedures that, in the positive case, also add the solution to update the subset. These procedures return `true` or `false` to indicate whether the candidate solution has been accepted or not. A solution added to B is not tested on D. Since the definition of "diverse" solution refers to both subsets, the update of the diverse set requires set B, or at least information on the Hamming distance of each current solution $y \in D$ from $B \cup D \setminus \{y\}$. At the end, the population P is cleaned, as in *PR*.

The third inner loop considers each pair of solutions (x, y) from $B \times (B \cup D)$ and recombines them to produce a solution z that is saved in pool P. Now the algorithm goes back to the random generation, in case the recombined solutions are not enough to provide the sufficient number of solutions to update the reference set.

The implementation can exploit many of the procedures already discussed: those provided in library `solutionpool` to manages solution pools, and the generation of random solutions. Of course, we need an additional pool `D`. As mentioned above, the update of the diverse set requires information on the Hamming distance: the two vectors `Hmin` and `Htot` provide, respectively, the minimum and total distance of each reference solution from the other ones. We use both the minimum and total distance to better discriminate between solutions: in fact, the Hamming distance is an integer value between 1 and k, and therefore it could easily assume identical values for several solutions. The update procedure should take care to keep these vectors sorted in the same way as the solutions in D.

Procedure `recombine_solutions` simply recombines two solutions into a third one in one of the several ways discussed in the theoretical lessons. In the present phase, we simply declare it without defining its content.

```
start = clock();
create_solutionpool(nb,&B);
create_solutionpool(nd,&D);
create_solutionpool(nb*(nb+nd),&P);

create_solution(pI->n,&x);

while (((double)clock() - start) / CLOCKS_PER_SEC < tauMax)
{
  /* Build or integrate the candidate population */
  while (P.card < nb + nd)
  {
    generate_random_solution(pI,&x,pseed);
    steepest_ascent(pI,&x,visit_strategy,&iter);
    if (!is_in_solutionpool(&x,pI->n,&P))
    {
     add_solution_to_pool(&x,pI->n,&P);
    }
    clean_solution(&x,pI->n);
  }

  /* Test the candidate solutions for insertion in the reference set */
  for (s = 1; s <= P.card; s++)
  {
    insert = update_best_set(P.S[s],pI->n,&B);
    if (!insert) insert = update_diverse_set(P.S[s],pI->n,&D);
  }

  /* Recombine the solutions in the reference set into candidate ones */
  for (s = 1; s <= B.card; s2++)
  {
    for (s2 = 1; s2 <= B.card; s2++)
      if (s != s2)
      {
        recombine_solutions(B.S[s],B.S[s2],pI,&x);
        add_solution_to_pool(&x,pI->n,&P);
      }
    clean_solutionpool(&P);

    for (s2 = 1; s2 <= D.card; s2++)
    {
      recombine_solutions(B.S[s],D.S[s2],pI,&x);
      add_solution_to_pool(&x,pI->n,&P);
    }
  }
}

if (nb > 0) copy_solution(B.S[1],px);

destroy_solutionpool(&B);
destroy_solutionpool(&D);
destroy_solutionpool(&P);
```

The update of the diverse set is similar to the update of the best set provided in library `solutionpool`. It is however more complicated because pool `D` is sorted with respect to the (minimum and total) Hamming distance from $B \cup D$. Therefore...

PROBLEM: THE DEFINITION OF POOL DOES NOT INCLUDE FIELDS FOR Htot AND Hmin. USING EXTERNAL VECTORS MAKES THE CODE MORE COMPLEX AND POSES THE QUESTION WHETHER THE FUNCTION SHOULD BE INCLUDED OR NOT IN THE LIBRARY, PLUS THE FOLLOWING ADDITIONAL PROBLEMS: WHEN ARE THE HAMMING DISTANCES OF THE RECOMBINED SOLUTION COMPUTED? WHERE ARE THEY COMPARED WITH THE HAMMING DISTANCES OF THE SOLUTIONS IN D?

```
TO BE COMPLETED
```

The core of the *SS* metaheuristic is the recombination procedure. Usually, its first step consists in initializing the new solution `x` with the intersection of the two parent solutions `x1` and `x2`. Then, the partial solution is augmented with elements drawn from the two parents. This can be performed in several ways, concerning two main aspects:

- whether the choice of the elements is random or greedy (or any semigreedy combination): the first approach allows a given pair of parent solutions to generate many different new solutions, the second usually generates better solutions;
- whether the elements are chosen alternatively from the two parents or freely: the first approach guarantees the largest possible distance from the two parent solutions, the latter allows a larger variety (in particular, if combined with choices at least partly random).

In the following, for the sake of simplicity, we will draw random elements alternatively from the two parent solutions, counting on the steepest ascent procedure as a tool to generate good solutions (rather than on a greedy choice) and on the recombination of good solutions as a tool to intensify the search (rather than on a biased extraction favouring one parent). It is in general possible that the constraints of the problem forbid to build a whole solution simply drawing random elements from the two parents. In the case of the *MDP*, however, this is possible, thanks to the very simple cardinality constraint. We have therefore simply to create an empty solution, add to it the points that belong to both parents (for example, by scanning one and checking which of its elements also belong to the other), put the other points of both solutions in suitable vectors from which they can be extracted at random, and perform the extraction as already done in several occasions in the previous chapters.

UPDATE: FIRST, SCAN A SOLUTION AND PUT IN THE DESTINATION THE POINTS THAT ARE ALSO IN THE OTHER SOLUTION. THEN, BUILD VECTORS FOR THE REMAINING CANDIDATE POINTS FROM THE TWO SOURCES. FINALLY, ALTERNATIVELY EXTRACT ONE POINT FROM EACH OF THE TWO SOURCES, AND ADD IT TO THE RECOMBINED SOLUTION. THERE IS MUCH IN COMMON WITH OTHER RANDOM EXTRACTIONS USED IN THE PREVIOUS CHAPTERS.

```
TO BE COMPLETED
```

6.3.1 Time complexity estimation

A time complexity estimation of the *SS* metaheuristic is quite complex, as it consists of procedures of a very different nature, including tests whose outcome is usually impossible to predict *a priori*. The generation of the reference set, for example, requires a random extraction in $O(kn)$ time, where coefficient n depends on the update of vector `D`. This apparently plays no role, and therefore seems damaging, but it still strongly accelerates the steepest ascent procedure, that probably gives a large contribution to the overall computational time: an assumption to be verified empirically. Then, it requires a steepest ascent procedure, whose complexity has already been characterized as $O(t_{\max}(n-k)k)$, where $t_{\max}$ is the number of neighbourhood exploration from a random solution to the local optimum in which steepest ascent terminates, that is an unknown function of n and k. The generation is repeated until B and D are both full, which requires at least $n_B + n_D$ iterations, but it could require many more, if the locally optimal solutions found are often the same. This depends strongly on chance and on the landscape of the problem (the *MDP* should have a sufficiently rugged landscape to generate many different local optima, but this is just a guess). Then, $n_B(n_B + n_D - 1)$ pairs of solutions are taken into account[5]. For each pair, the recombination procedure requires to find the intersection and the two set differences of the parent solutions (in time $O(k)$) and to add the points of the intersection and random points of the differences to build the new solution (in time $O(n)$ for each of the k points, once again for the update of vector `D`). Each solution must be checked for insertion in the best set B in time $O(k)$ (in the worst case, that becomes probably quite rare after a while, because it corresponds to improving most of the best solutions at every attempt). If this fails, it must be checked for insertion in the diverse set B in time WHO KNOWS? The main problem is that there is no way to predict how many times these operations will be repeated overall. Therefore, the theoretical analysis is mainly useful to determine the relative weight of the different contributions: the steepest ascent procedure appears to be the most computationally intensive, and this justifies the choice to keep vector `D` even if it slows down some of the other procedures.

6.3.2 Empirical evaluation

We can now evaluate the performance of the *SS* metaheuristic. We consider reference sets of the same size used for *PR*. In this case, however, we have two subset, B and D. For the sake of simplicity, we assume them to have the same size. Since the pairs of solution to be recombined are $n_B(n_B + n_D - 1)$, we will experiment with the following configurations: $n_B = n_D = 5$, $n_B = n_D = 10$ and $n_B = n_D = 15$. In this way, the number of pairs will not be very different from the one considered by *PR*.

OTHER POSSIBLE PARAMETERS: GREEDY CHOICE VERSUS RANDOM CHOICE; FREE CHOICE VERSUS ALTERNATE CHOICE

Computational time analysis

As for *PR* the computational time is fixed to 30 seconds overall.

TEST WHETHER THE TIME IS VIOLATED DUE TO THE CHECK MADE ONLY AT THE BEGINNING OF EACH ITERATION (PRESUMABLY NOT). IF THIS HAPPENS, ADD OTHER CHECKS.

[5]This means that every pair of best solutions is considered twice: once in each direction. The random choice of elements during the recombination suggests that such a repetition could be nonredundant, but this should be verified empirically.

6.3.3 Parameter tuning

AVERAGE RESULTS

Figure 6.3 reports the *SQD* diagram of the *SS* metaheuristic, compared with that of the steepest ascent heuristic applied to purely random solutions. This allows to estimate the contribution of recombination to the overall performance. In fact, the *SS* metaheuristic adopts the same method to initialize the population and to integrate it when recombination is not enough to fill the required minimum number of solutions

Figure 6.3: Solution Quality Distribution diagram for the *SS* metaheuristic compared with the *steepest ascent* heuristic initialised with random solutions with a time limit of 30 seconds

The boxplots reported in Figure 6.4 provide the same information: SUMMARY OF THE PICTURE

Figure 6.4: Boxplots for the *SS* metaheuristic compared with the *steepest ascent* heuristic initialised with random solutions with a time limit of 30 seconds

WILCOXON'S TEST

6.4 Comparison with *PR* and random restart