Università degli Studi di Milano

Heuristic algorithms (laboratory sessions)

Roberto Cordone

– Academic year 2020/21 –

Contents

Chapter 4

Exchange heuristics

4.1 Introduction

This chapter discusses the application of exchange heuristics to the *Maximum Diversity Problem* (*MDP*). Exchange heuristic start from a given feasible solution $x^{(0)}$ (typically obtained with a constructive heuristic, or metaheuristic, or a random generation process) and try to improve the current solution x iteratively by adding a suitable subset A and deleting a suitable subset D of elements of the ground set. Of course, A consists of external elements ($A \subseteq B \setminus x$) and D of internal ones ($D \subseteq x$). The possible pairs of subsets are determined by a rule that takes the form of a *neighbourhood* function $N : X \to 2^X$, associating each feasible solution $x \in X$ with a subset of feasible neighbour solutions $N(x)$. The choice of the *incumbent*, that is the neighbour solution that replaces the current one, is done optimising a suitable selection criterium, that nearly always is the objective function value.

In the following, we will consider some alternative initialisation procedures (namely, the farthest point and the try-all constructive heuristics and a purely random generation). We will also adopt the most natural neighbourhood for the *MDP*, that is the *single-swap neighbourhood* N_{S_1}, which includes all the subsets obtained from x deleting a single element i and adding a single element j:

$$N_{S_1}(x) = \{x' = x \setminus \{i\} \cup \{j\} \text{ with } i \in x, j \in P \setminus x\}$$

Notice that $N_{S_1}(x) = N_{H_2}(x)$, that is, it coincides with the collection of feasible subsets having Hamming distance equal to 2 from x. However, the collection of all subsets at Hamming distance equal to 2 also includes the ones obtained adding or deleting two points, which are unfeasible. The single-swap neighbourhood, on the contrary, automatically satisfies the cardinality constraint that characterises the *MDP*, and this implies the strong advantage that the feasibility of the subset obtained with any swap operation is guaranteed *a priori* and needs not be verified. The exploration of the neighbourhood, therefore, simply consists in the computation of the objective function for each neighbour solution.

Thanks to the cardinality constraint, and to the lack of other complicating constraints, neighbourhood N_{S_1} always includes exactly $k(n-k)$ solutions, and this induces a strong relation between the number of neighbourhood explorations and the computational time (at least, if the neighbourhood is fully explored).

Finally, concerning the selection criterium, we will adopt the objective function, thus implementing the basic exchange heuristic known as *steepest ascent* (for maximisation problems as the *MDP*). We will discuss its theoretical and empirical computational complexity, and we will improve it with a standard trick to allow

the evaluation of quadratic objective functions in cardinality-constrained problems. We will compare the results of the different initialisation procedures and we will tune the size of the neighbourhood with the adoption of the *first-best* exploration strategy as opposed to the *global-best* one.

The `main` function, then, allows to choose from the command line which of the three initialization procedures to apply (with option `-gp` for the *farthest-point*, `-ga` for the try-all heuristic and `-r` followed by a negative integer seed for the random initialisation), and which of the two neighbourhood exploration strategies to apply (with option `-gb` for the *global-best* and `-fb` for the *first-best* strategy). The steepest ascent heuristics also returns the number of neighbourhood explorations performed, because we are going to investigate the influence of the exploration stategy on this value and its relation with the computational time. Apart from printing the number of iterations, all other operations are the same introduced for the previous heuristics.

```
parse_command_line(argc,argv,data_file,init_algo,visit_strategy,&seed);

load_data(data_file,&I);

create_solution(I.n,&x);

start = clock();
if (strcmp(init_algo, "-gf") == 0)
  greedy_farthest(&I, &x);
else if (strcmp(init_algo, "-ga") == 0)
  greedy_tryall(&I, &x);
else if (strcmp(init_algo, "-r") == 0)
  generate_random_solution(&I, &x, &seed);
steepest_ascent(&I,&x,visit_strategy,&niter);
end = clock();
tempo = (double) (end - start) / CLOCKS_PER_SEC;

printf("%s ",data_file);
printf("%10.6f ",tempo);
printf("%8d ",niter);
print_sorted_solution(&x,I.n);
printf("\n");

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

4.2 The steepest ascent heuristic

The *steepest ascent* heuristic chooses the following solution from the neighbourhood of the current one by optimising a selection criterium $\phi(x,i,j)$ that is simply the value of the objective function $f(x \cup \{i\} \setminus \{j\})$ or, to be more precise, half of its variation:

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = \sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

This implies the following adaptation to the *MDP* of the general scheme:

Algorithm SteepestAscentMDP$\left(I, x^{(0)}\right)$
$x := x^{(0)}$;
Stop := *false*;
While Stop = *false* *do*
$\tilde{x} := \arg \max_{i \in x, j \in P \setminus x} \left(\sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij} \right)$;
If $f(\tilde{x}) \leq f(x)$ *then* Stop := *true*; *else* $x := \tilde{x}$;
EndWhile;
Return $(x, f(x))$;

The implementation of this algorithm is nearly straightforward. The main difference is that, for the sake of efficiency, the procedure `explore_neighbourhood` that selects the incumbent returns a pair of points (i^*, j^*) to be exchanged, and the resulting variation of the objective, instead of a whole solution $\tilde{x}$. Therefore, if the incumbent improves the current solution (that is, the variation is negative), the update operation $x := \tilde{x}$ is obtained performing the exchange suggested with procedure `swap_points`.

```
*pniter = 0;
do
{
  explore_neighbourhood(px,pI,visit_strategy,&p_in,&p_out,&delta_f);
  if (delta_f > 0)
  {
    (*pniter)++;
    swap_points(p_in,p_out,px,pI);
  }
} while (delta_f > 0);
```

It could be remarked that swapping points j and i is equivalent to deleting point j and adding point i, so that we do not actually need an additional procedure. However, implementing this procedure separately has the advantage to avoid any instruction that is useless when the two operations must be performed together (to give a trivial example, the cardinality of the solution remains unvaried, instead of being decremented and incremented).

The exploration of the neighbourhood trivially consists in two nested loops, as j scans the current solution x and i scans its complement, taking advantage of the corresponding lists. For each pair of solutions, the procedure estimates the variation of the objective function $\delta f(x, i, j)$ calling a suitable procedure `evaluate_exchange` and saves the best exchange and the corresponding variation of the objective.

```
*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))
  for (p_out = first_point_out(px); !end_point_list(p_out,px); p_out = next_point(p_out,px))
  {
    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;
```

```
    }
  }
```

The procedure that evaluates each single exchange does not physically perform it. *Do not perform moves only to evaluate them* is a specific application of a general fundamental principle in the design of heuristic algorithms: *avoid all useless operations*. It simply computes

$$\sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

as follows[1].

```
delta = 0;

delta  = dist_from_x(p_out,px,pI);
delta -= dist_from_x(p_in,px,pI);
delta -= pI->d[get_index(p_in,pI)][get_index(p_out,pI)];

return delta;
```

4.2.1 Time complexity estimation

The computational complexity of the steepest ascent heuristic derives from three main sources:

1. the number of neighbourhood explorations $t_{\max}$ performed to reach the local optimum in which the search terminates;
2. the number of neighbour solutions (or, in general, subsets) whose objective value (and, in general, feasibility) is evaluated;
3. the computational time required to evaluate the objective value (and, in general, the feasibility) of each neighbour solution (or subset).

The first term is in general unknown and hard to estimate (unless for upper estimates such as the total number of feasible solutions, that are very loose). For the single-swap neighbourhood N_{S_1}, the number of neighbour solutions is exactly $k\,(n-k)$. The feasibility is automatically guaranteed, and the evaluation of the objective requires to compute the distance of two points from the current solution, that is $O\,(k)$ time. The resulting overall estimate is $O\left(t_{\max}\,(n-k)\,k^2\right)$.

4.2.2 Empirical evaluation

We can now evaluate the performance of the *steepest ascent* heuristic with the global-best strategy.

Computational time analysis

From the detailed results, we can remark that the overall computational time (including both the initialisation constructive procedure and the following exchange

[1] This is just a detail, but it is probably better to add and remove d_{ij} rather than checking at every step whether $k = j$ or not.

procedure) ranges from fractions of a second to a couple of minutes. This is two orders of magnitude larger than the time required for the constructive heuristic alone (up to half a second), and therefore mostly depends on the exchange phase. It is comparable to the try-all heuristic only for the smaller instances, and one order of magnitude smaller for the larger ones (in fact, the try-all heuristic repeats the basic constructive heuristic n times, with n ranging from 100 to 1 000). The exchange heuristic is therefore less efficient than the constructive (and destructive) ones, but more efficient than the try-all heuristic.

Figure 4.1 reports the semilogarithmic *scaling diagram* for the computational time of the steepest ascent heuristic on the whole benchmark. The diagram shows the expected polynomial increase with size. The $O\left(t_{\max} n k^2\right)$ theoretical estimate, with $k \propto n$, suggests an overall $O\left(t_{\max} n^3\right)$ complexity. The linear interpolation:

$$T_A = \beta n^\alpha \Leftrightarrow \log T_A = \alpha \log n + \log \beta$$

suggests that $\alpha \approx 4.3$ and $\beta \approx 5.75 \cdot 10^{-12}$. If we assume a cubic complexity for the neighbourhood exploration, this would imply that $t_{\max}$ increases slightly more than linearly with n. To test more precisely this conclusion we can compute and plot the ratio T/t_{max} of the total time T on the number of neighbourhood explorations t_{max} (see the yellow graph in Figure 4.1) and make an interpolation on it. Since $\alpha \approx 2.7$ and $\beta \approx 3.6 \cdot 10^{-9}$, it seems that the cubic estimate for the time required to explore a single neighbourhood is excessive, and that $t_{\max}$ is more than linear in n, though not quadratic. One can also notice that the first diagram is much more irregular than the first one, meaning that $t_{\max}$ is not strictly dependent on n. Of course, we could also directly interpolate t_{max} as a function of size: a quick look at the detailed results for each fixed value of n suggests that t_{max} indeed strongly depends on k, increasing more than linearly: it becomes about 10 times larger as k goes from $0.1\,n$ to $0.4\,n$.

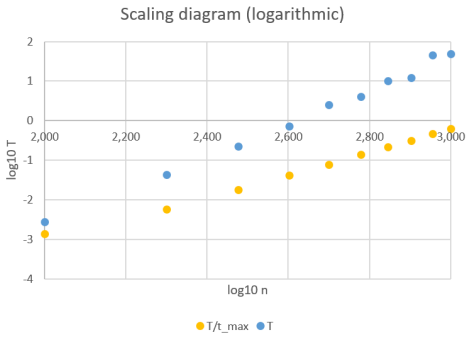

Figure 4.1: Scaling diagram in logarithmic scales for the steepest ascent algorithm on the benchmark

Solution quality analysis

Figure 4.2 reports the *SQD* diagram, compared with that of the initialisation farthest-point procedure. Of course, the former strictly dominates the latter, as it receives the solution in input and proceeds by improving it with local search. What is interesting is the amount of the improvement, that is strong, but not huge: the average gap δ decreases from 1.52% to 0.97%. This suggests that the *MDP* has many local optima of various quality and with small basins of attraction. In fact, the try-all heuristic still performs better than the steepest ascent. Of course, as it

takes much more time, we can't say that the latter is dominated. However, it is still impossible to dismiss the try-all heuristic as a viable approach. One can notice that the steepest ascent heuristic has a larger probability of finding very small gaps (below 0.3%), though not of finding the best known result. This suggests a region of stronger stability, possibly corresponding to the capacity of improving good initial results.

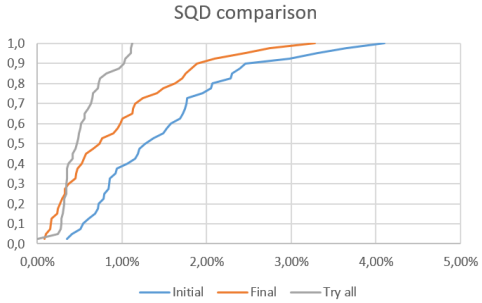

Figure 4.2: Solution Quality Distribution diagram for the *steepest ascent* and the *farthest point* heuristics

The boxplots reported in Figure 4.3 provide the same information.

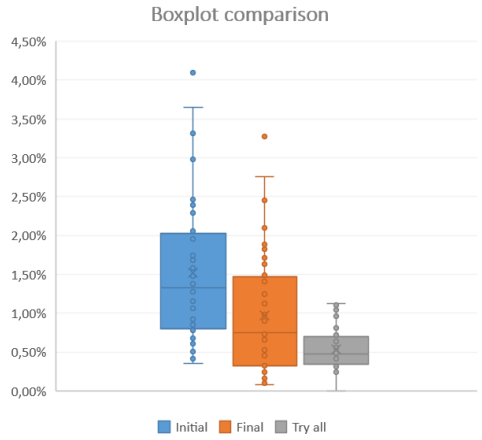

Figure 4.3: Boxplots for the *steepest ascent* and the *farthest point* heuristics

We do not apply statistical tests to this comparison, because by definition the dominance of steepest ascent with respect to the farthest point heuristic is strict and the tests would not add anything to this fundamental information.

4.2.3 Constant-time neighbour evaluation

From theory we know that the variation of a quadratic objective function implied by a simple swap of elements can be estimated in constant time exploiting the formula

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = \sum_{k \in x} d_{jk} - \sum_{k \in x} d_{ik} - d_{ij}$$

by saving in a suitable vector D_i the total distance of each point $i \in P$ from the current solution x, both for internal and external points. In fact, given this vector

$$\delta f(x,i,j) = \frac{1}{2}\left(f(x \cup \{i\} \setminus \{j\}) - f(x)\right) = D_j - D_i - d_{ij}$$

can be computed in two operations. Of course, whenever the current solution x changes into $x' = x \setminus \{i\} \cup \{j\}$, the vector D must be updated. This takes time $O(n)$ applying the following formula

$$D_k := D_k - d_{ik} + d_{jk} \qquad \text{for all } k \in P$$

Overall, this reduces the time to explore a single neighbourhood from $O\left((n-k)\,k^2\right)$ to $O\left((n-k)\,k\right)$, at the cost of adding an $O(n)$ term, that is dominated.

From the implementation point of view, we need to decide where to store vector D. The most natural approaches are either to keep it explicitly as a variable in procedure `steepest_ascent` or to "hide" it in solution x. In the former case, we will have to pass it as an argument to procedure `explore_neighbourhood`. In the latter case, we will have to update all the functions that manipulate objects of type `solution_t`. The choice mainly depends on whether we think that the vector will be used outside of the exchange heuristic or not. As we are going to use it also in the exchange metaheuristics, we will adopt this approach. Moreover, it make sense to go back to constructive heuristics and metaheuristics and evaluate whether the computational trick would provide an advantage also in those algorithms. Indeed, the $O\left(nk^2\right)$ complexity of the constructive heuristics depends on applying k times the basic constructive step, in which for all external points (hence, the $O(n)$ term) the distance from the current solution is estimated in time $O(k)$. The introduction of vector D would remove that term, and therefore the total time to $O(nk)$ at the cost of an additional $O(n)$ term, that is dominated. All the required work auxiliary work has been done in library `solution2`, that we will use in the following instead of `solution`. This will also require to modify accordingly the inclusion directive in file `local_search.h`.

Adding vector D to the `solution_t` data structure under the form of a dynamic integer vector (`int *`) requires to update the creation, destruction, copy and check procedures, as well as the manipulation procedures (`add_point`, `delete_point` and `swap_points`). After this update, it is possible to implement the evaluation of the exchange simply as follows.

```
delta  = px->D[get_index(p_out,pI)];
delta -= px->D[get_index(p_in,pI)];
delta -= pI->d[get_index(p_in,pI)][get_index(p_out,pI)];

return delta;
```

On the other hand, the `swap_points` procedure must also update the elements of vector D (in $O(n)$ time), but it can save $O(k)$ time for the update of the objective value.

Solving again the whole benchmark leads to exactly the same results as above, as expected. The computational time is however much smaller: in Figure 4.4, the logarithmic scale clearly shows a decrease in the slope, corresponding to a reduction of the exponent in the polynomial dependence of the computational time on the number of points[2]. Just to have an intuitive idea of the improvement, the

[2]Actually, α decreases from 4.3 to 2.9 for the overall time and from 2.7 to 1.3 for the time per iteration, which seems rather too much, but I have had no time to check the numbers and to investigate the reason of this behaviour.

computational times now range from fractions of a second to 1 second, instead of two minutes. This impressive result derives from having reduced the theoretical worst-case complexity from $O\left(t_{\max} n k^2\right)$ to $O\left(t_{\max} n k\right)$. The time is not strictly reduced by a factor of k (from 10 to 400 in the benchmark) because the contribution of secondary terms previously overwelmed by the evaluation of the value for the neighbour solutions now can becomes perceivable.

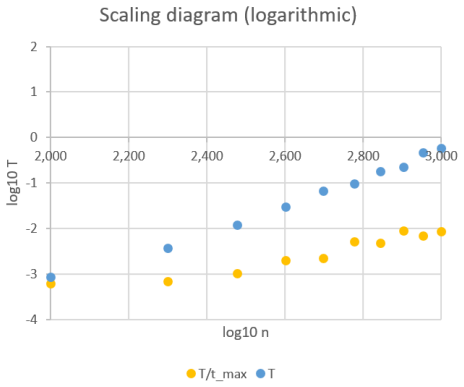

Figure 4.4: Scaling diagram in logarithmic scales for the steepest ascent algorithm with the constant-time or the linear-time evaluation of neighbours

4.2.4 Comparison of initialisation procedures

We now investigate whether the initial solution $x^{(0)}$ exerts a long-term influence on the quality of the final solution returned by the exchange heuristic; in other words, whether good starting solutions tend to fall into the basin of attraction of good local optima. This would recommend the adoption of a good constructive procedure, provided that its computational time is not excessive. To investigate also this aspect, we consider a third variant in which the initial solution is obtained with the "try-all" heuristic[3].

Figure 4.5 shows the three *SQD* diagrams for the initial solutions and for the final ones. The diagrams are in semilogarithmic scale because the gaps are widely distributed (part of the diagrams is missing because zero values cannot be represented on a logarithmic axis). This allows to appreciate the relative improvement obtained by the exchange procedure with respect to each different starting point This is particularly strong in the case of the random initialisation, whereas it becomes smaller for the constructive initialisations. In particular, it can be noticed that the worse starting solutions are only slightly improved (in the case of the try-all heuristic, the second-worse solution is not improved, so that the diagram appears to reach the upper bound nearly for the same gap). It is however clear that better starting solutions tend to be associated with better final solutions: the exchange heuristic is not strong enough to overcome the initial advantage.

Of course, the computational times are also relevant for the choice. Notice that we can apply the try-all heuristic only because the constant-time evaluation

[3]It is clear that this heuristic could be fully exploited by running an exchange procedure on each of the starting solutions it provides, but we are now focusing on the improvement power of a single run of an exchange heuristic. Applying the exchange heuristic to each solution generated by the "try-all" heuristic would be more on the line of a multi-start exchange metaheuristic. The same can be said about the classical *GRASP* mechanism that applied an exchange procedure to each solution generated by the semigreedy algorithm.

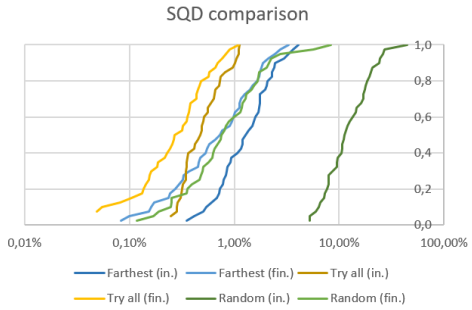

Figure 4.5: Comparison of different initialisation procedures

procedure can be extended to the constructive phase, so that this heuristic no longer takes several minutes to provide the starting solution for the larger instances. Indeed, its overall time requirement (constructive and improvement phase) ranges from fractions of a second to about 5 seconds, as opposed to about 1 second for the farthest point initialisation and 1.6 seconds for the random initialisation. The try-all initialisation is slower due to the more refined constructive phase; in fact, the number of neighbourhood explorations before reaching the local optimum is on average 15. The random initialisation has an extremely fast initialisation, but an average of 100 iterations. That implies the final longer time with respect to the farthest point heuristic, which only makes 25 iterations before hitting the local optimum.

Figure 4.6 provides the *RTD* diagrams on the benchmark. We remind that such a diagram makes little sense for benchmarks collecting instances of different size, but it allows meaningful comparisons between different algorithms.

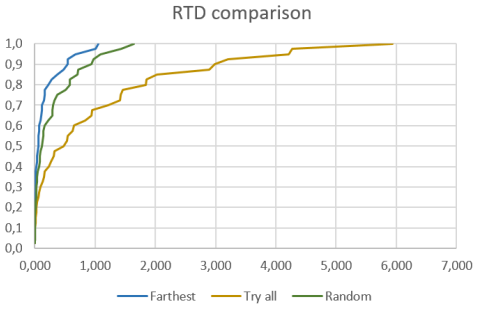

Figure 4.6: Comparison of the overall running time of the exchange heuristic for different initialisation procedures

This suggests that a random initialisation is not a good idea (at least for the steepest ascent heuristic), whereas it is still open whether the farthest point or the try-all initialisation (or other non fully random ones) could be more effective.

4.2.5 Neighbourhood tuning: *global-best* versus *first-best*

Finally, we experiment with the idea of tuning the exploration of the neighbourhood, that is terminating it as soon as an improving solution is found with respect to the current one. This is known as *first-best* strategy, as opposed to the classical

global-best strategy that visits the whole neighbourhood and returns the overall best solution it contains. The rationale is to accept a smaller improvement in each step of the search, in exchange for a much smaller computational time, that allows to perform many more steps, possibly getting earlier to the same local optimum. It must be noticed that, changing the rule that determines the following visited solution in general also changes the basins of attraction, and therefore the final local optimum reached. This could be worse or better than in the global-best case. To make things even more complex, the local optimum returned also depends on the order in which each neighbourhood is visited, a fact that could be perhaps exploited somehow (possibly, starting with the most promising elements based on the distance D from x), with a possible increase of the computational cost. We are not going to explore this line of research: the neighbourhood will be explored scanning x and $P \setminus x$ with the corresponding lists, exactly as in the global-best strategy.

In order to impose the first-best strategy, an extremely simple modification must be made to the neighbourhood exploration procedure: as soon as an exchange with a positive effect is found, the procedure terminates returning that exchange.

```
*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))
  for (p_out = first_point_out(px); !end_point_list(p_out,px); p_out = next_point(p_out,px
  {
    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;
      if ( (delta_f > 0) && (strcmp(visit_strategy,"-fb") == 0) ) break;
    }
  }
```

For the sake of simplicity, we apply the farthest-point initialisation heuristic. The detailed results show that the number of iterations tends to be larger with respect to the global-best strategy, as expected, but the difference is not strong (30 versus 25 on average). In fact, the computational time is shorter, and the results comparable. Figure 4.7 shows that the two algorithms have rather similar performance with respect to the solution quality (perhaps, the first-best strategy is slightly better).

Figure 4.8, however, shows that the first-best strategy is clearly faster. This could be enough to suggest to adopt it instead of the classical global-best strategy.

In order to further support this choice, we can compare the two sets of results with Wilcoxon's test. The results concerning the solution quality are:

```
W+ = 168, W- = 297, N = 30, p <= 0.188
```

and suggest that, while there is a slight predominance of the first-best strategy with respect to the global-best, it would not be unlikely that such a predominance be due only to a random extraction (the p-value is 18.8%, that is quite large).

On the other hand, the results for the computational time are:

```
W+ = 677.50, W- = 25.50, N = 37, p <= 9.094e-007
```

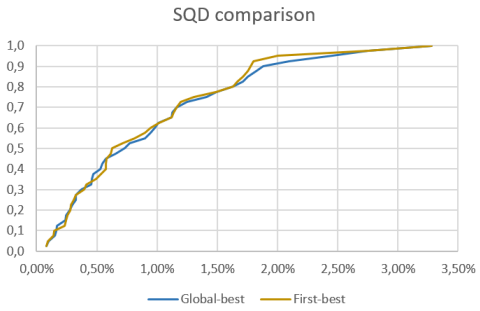

Figure 4.7: Comparison of the solution quality of the exchange heuristic with the global-best and first-best neighbourhood exploration strategies

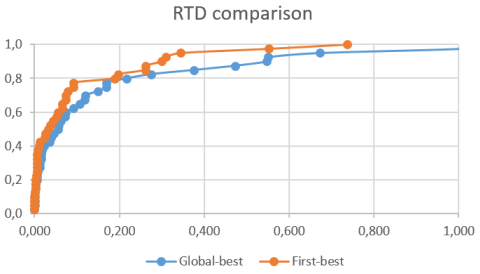

Figure 4.8: Comparison of the computational time of the exchange heuristic with the global-best and first-best neighbourhood exploration strategies

and indeed suggest that the global-best strategy takes a longer time, even if the first-best strategy requires more neighbourhood explorations (the p-value is approximately 10^{-6}).