Università degli Studi di Milano

Heuristic algorithms (laboratory sessions)

Roberto Cordone

– Academic year 2020/21 –

Contents

Chapter 5

Exchange metaheuristics

5.1 Introduction

This chapter discusses the application of exchange metaheuristics to the *Maximum Diversity Problem* (*MDP*). Exchange metaheuristic extend the basic scheme of exchange heuristics modifying its elements (the starting solution, the neighbourhood, the objective function or the selection rule) in order to proceed with the search after reaching a locally optimal solution. In the following, we will consider two such extensions, both based on the basic *steepest ascent* heuristic described in the previous chapter. The former is a *Variable Neighbourhood Search* (*VNS*) heuristic, that restarts the search from a new solution generated with a *shaking* procedure on the best known solution. The latter is a *Tabu Search* (*TS*) heuristic, that prolongs the search beyond local optima by looking for the minimum cost neighbour solution that respects suitable *tabu* conditions, designed to avoid a cyclic behaviour.

For the sake of simplicity, we will build the starting solution with the farthest-point constructive heuristic introduced in Chapter 3 and apply the steepest ascent heuristic with the *first-best* strategy that proved to be equally effective and more efficient in Chapter 4. The command line arguments will allow the user to choose which of the two metaheuristics to apply and the associated parameters:

- for the *VNS* metaheuristic, option `-vns`, followed by five parameters: the total number of neighbourhood explorations $t_{\max}$, the minimum neighbourhood index $s_{\min}$, the index variation δs, the maximum neighbourhood index $s_{\max}$ and the seed of the pseudorandom number generator;
- for the *TS* metaheuristic, option `-ts`, followed by three parameters: the total number of neighbourhood explorations $t_{\max}$, the tabu tenure for the reinsertion of deleted elements L^{in}, and the tabu tenure for the removal of added elements L^{out}.

```
parse_command_line(argc,argv,data_file,exchange_algo,&niter,&seed,
                   &s_min,&delta_s,&s_max,&l_in,&l_out);

load_data(data_file,&I);

create_solution(I.n,&x);
greedy_farthest(&I,&x);

start = clock();
```

```
if (strcmp(exchange_algo,"-vns") == 0)
  variable_neighborhood_search(&I,&x,"-fb",niter,s_min,delta_s,s_max,&seed);
else if (strcmp(exchange_algo,"-ts") == 0)
  tabu_search(&I,&x,"-fb",niter,l_in,l_out);
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);
```

5.2 Variable Neighbourhood Search

The *Variable Neighbourhood Search* metaheuristic applies a basic steepest ascent heuristic, and restarts it every time this terminates in a locally optimal solution. The restart is performed with a *shaking* procedure that modifies the current best known solution generating it at random in a suitable neighbourhood, whose size is the main parameter of the method[1]. The scheme is the following:

```
Algorithm VariableNeighbourhoodSearch($I, x^{(0)}, \ell, s_{\min}, \delta s, s_{\max}$)
$x :=$ SteepestAscent($x^{(0)}$); $x^* := x$;
$s := s_{\min}$;
For $l := 1$ to $\ell$ do
  $x' :=$ Shaking($x^*, s$);
  $x' :=$ SteepestAscent($x'$);
  If $f(x') > f(x^*)$
    then $x^* := x'$; $s := s_{\min}$;
    else $s := s + \delta s$;
  If $s > s_{\max}$ then $s := s_{\min}$;
EndWhile;
Return $(x^*, f(x^*))$;
```

This scheme is general enough to not require any specific adaptation for the *MDP*, except for the replacement of steepest descent with steepest ascent, as usual because it is a maximisation problem. However, we will also modify the termination condition, replacing the number of restarts ℓ with the total number of neighbourhood explorations $t_{\max}$ in order to get a better control of the computational time (of course, we could directly fix the total computational time, and that would be even more precise). To this purpose, we exploit the information on the number of neighbourhood explorations performed that is already provided by the steepest ascent procedure, but we also need to terminate if before reaching a locally optimal solution whenever the total number of available explorations has been consumed.

[1]This parameter is generally denoted as k (as in the slides of the theoretical lessons), but here we will denote it as s to distinguish it from the required number of points in the feasible solutions of the *MDP*.

For the sake of simplicity, we will adopt the steepest ascent heuristic discussed in the previous chapter, based on the single-swap neighbourhood N_{S_1} with the first-best exploration strategy. It is therefore straightforward to implement the VNS procedure as follows.

```
  truncated_steepest_ascent(pI,px,visit_strategy,niter,&iter);
  tot_iter = iter;

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

  s = s_min;
  while (tot_iter < niter)
  {
    shaking(pI,px,s,pseed);
    truncated_steepest_ascent(pI,px,visit_strategy,niter-tot_iter,&iter);
    tot_iter += iter;

    if (px->f > x_star.f)
    {
      copy_solution(px,&x_star);
      s = s_min;
    }
    else
    {
      s += delta_s;
      if (s > s_max) s = s_min;
    }
```

The variable tot_iter saves the cumulative number of neighbourhood explorations performed, in order to compare it with the maximum imposed value niter. Moreover, the truncated_steepest_ascent procedure coincides with the already implemented steepest_ascent procedure, with the additional termination condition of stopping as soon as the remaining niter-tot_iter explorations have been performed. The modified procedure is already provided in a modified localsearch.c library.

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

} while ( (delta_f > 0) && (*pniter < max_iter) );
```

The only part of the variable_neighborhood_search procedure that remains unimplemented is the shaking procedure, that receives the reference solution (that is the best known one, as we are applying the basic version of the *VNS*) the current value of parameter s and the seed of the pseudorandom number generator, and returns the perturbed solution that will be used as a starting point for the following application of the steepest ascent procedure. In order to implement it, we need to define a hierarchy of neighbourhoods from which to extract a random solution. In general, these neighbourhoods should be progressively increasing, in order to allow a controllable amount of intensification (using the first neighbourhoods) or diversification (using the last ones). In the specific case of the *MDP*, swaps are the most natural operation to generate neighbourhoods, due to the cardinality constraint, that guarantees the feasibility of every solution they generate, while proving the unfeasibility of the subsets generated by any other kind of operation. Therefore, we will adopt the hierarchy formed by the swap neighbourhoods N_{S_s}. Every solution in N_{S_s} is obtained performing s single swaps of a point in the current solution with a point out of it. A technical detail to be defined is whether we consider:

- s possibly overlapping swaps, in which a point deleted by one of the swaps can be added again by a following swap;
- s disjoint swaps, that is we swap s points from the starting solution with s points out of the starting solution.

The former definition allows to obtain also solutions that could be obtained with any number $s' \leq s$ of swaps (because swapping points i and j, followed by j and k is equivalent to swapping directly i and k). Therefore, it is a more general definition and it guarantees that each neighbourhood in the hierarchy include the previous ones. The latter definition restricts the neighbourhood only to disjoint swaps and yields disjoint neighbourhoods. However, it has the advantage that all neighbour solutions have a Hamming distance exactly equal to $2s$ from the reference solution x^*. This property seems particularly desirable from the point of view of controlling the size of the perturbation introduced (in the former case, a perturbation with a very large s might generate a solution very close to x^*, possibly even coincident with it). We therefore adopt the second definition. Generating a random s-swap is then straightforward: it requires to save in a vector the elements of the solution and extract s uniformly without repetitions. The same must be done for the elements out of the solution. Then, we can match the two sets of points in s pairs and swap the points to obtain the perturbed solution. There are several ways to do that, whose time and space complexity can be discussed in detail. Choosing one arbitrarily, we will write in a single vector of n elements the indices of all points: those in the solution in its first positions and the other ones in the last positions (using the incidence vector to distinguish them). Then, we will randomly extract s indices from the first and s from the second subvector, moving them to the ends of the vector to avoid generating duplicate terms and finally swap the corresponding points with the `swap_points` procedure. It can be argued that an *ad hoc* procedure could be more efficient (in particular for large values of s), but the shaking procedure is applied rarely enough to assume that it will take a negligible part of the overall computational time of the algorithm[2]

[2]This assumption should be verified, of course.

```
/* Build a vector with the indices of the internal points in the first k positions
   and the indices of the external point in the last n-k positions */
Indices = int_alloc(pI->n+1);
i_in = 1; i_out = pI->n;
for (i = 1; i <= pI->n; i++)
  if (px->in_x[i] == true)
    Indices[i_in++] = i;
  else
    Indices[i_out--] = i;

/* Select s internal points and move their indices to the first s positions of Indices */
for (i_in = 1; i_in <= s; i_in++)
{
  i = rand_int(i_in,pI->k,pseed);
  temp = Indices[i_in];
  Indices[i_in] = Indices[i];
  Indices[i] = temp;
}

/* Select s external points and move their indices to the last s positions of Indices */
for (i_out = pI->n; i_out >= pI->n-s+1; i_out--)
{
  i = rand_int(pI->k+1,i_out,pseed);
  temp = Indices[i_out];
  Indices[i_out] = Indices[i];
  Indices[i] = temp;
}

/* Perform s exchanges between the first and the last points of Indices */
for (i_in = 1, i_out = pI->n; i_in <= s; i_in++, i_out--)
    swap_points(get_point(Indices[i_in],pI),get_point(Indices[i_out],pI),px,pI);

free(Indices);
```

5.2.1 Time complexity estimation

The computational complexity of the *VNS* metaheuristic combines that of the steepest ascent heuristic with that of the shaking procedure. The former has already been estimated as $O\left(t_{\max}(n-k)k\right)$, with the modification that $t_{\max}$ is now the total number of neighbourhood explorations, fixed by the user. Applying the first-best exploration strategy means that the number of explored solutions should be $\leq (n-k)k$, though usually smaller (probably much smaller in the first iterations, when it is easier to improve the current solution, and nearly equal later when approaching the local optimum). The latter term is given by the initialisation in $O(n)$ time of the index vector, followed by the generation of s pseudorandom number (constant time, though probably not a very small constant) and s swaps, that take (as discussed in the previous chapter) $O(n)$ time each. Overall, this is $O(\ell ns)$ time, where $s \leq \min(k, n-k)$ (as discussed in the following) and ℓ is the number of restarts (unknown *a priori*). However, s and ℓ tend to be inversely correlated, since smaller perturbations probably imply shorter paths to the local optimum and more restarts, whereas larger perturbations imply longer paths to the local optimum and less restarts. A rough guess is that each shaking application should take a time comparable to a neighbourhood exploration, so that overall its contribution should be negligible with respect to the steepest ascent heuristic, but this needs to be verified experimentally.

5.2.2 Empirical evaluation

We can now evaluate the performance of the *VNS* metaheuristic. We have decided to use the total number of neighbourhood explorations as a termination condition. This value should be set so as to allow the heuristic to experiment with every possible working condition. However, we also want to get results in a reasonably short time. In the previous chapter, we have estimated that the average number of iterations performed from a starting solution to the corresponding local optimum ranges between 15 and 30 in our benchmark instances. It will probably be smaller for starting solutions generated with small perturations from the best known solution. Since $s \leq k$, it should take at most $30 \cdot 400$ neighbourhood explorations to reach the strongest perturbations on the largest instances. Therefore, $t_{\max} = 10\,000$ could be a good choice, but we will adopt $t_{\max} = 1\,000$ to fit the experiments in the space of a lesson.

In the first experiments, we will let parameter s autotune, by fixing $s_{\min}$, δs and $s_{\max}$ to their simplest values: $s_{\min} = 1$, $\delta s = 1$ and $s_{\max} = \min(k, n-k)$.

Computational time analysis

The detailed results show that the overall computational time (including the initialisation constructive procedure, the shaking procedure and the steepest ascent procedure) ranges from fractions of a second to less than four seconds. This is even shorter than the exchange heuristic initialised by the try-all heuristic, which is a quite promising fact, should the results prove at least as good. Let us remind that the steepest ascent procedure applied to the farthest-point procedure could not enhance it enough to overcome the advantage provided by the better try-all initialisation procedure, even if its computational time was smaller. We can investigate whether the use of *VNS* allows exploit the shorter computational time to get better results.

Figure 5.1 reports the logarithmic *scaling diagram* for the computational time of the *VNS* heuristic on the whole benchmark. The diagram shows a very regular polynomial increase with size, also thanks to the fixed number of iterations $t_{\max}$. The $O\left((n-k)k\right)$ theoretical estimate, with $k \propto n$, suggests a quadratic complexity, which is in extremely good accordance with the linear interpolation:

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

since $\alpha \approx 2.008$ and $\beta \approx 2.6 \cdot 10^{-6}$.

Solution quality analysis

Figure 5.2 reports the *SQD* diagram of the *VNS* metaheuristic, compared with that of the steepest ascent heuristic applied to the result of the farthest-point and of the try-all heuristics: the former allows the comparison with the result obtained stopping at the first local optimum; the latter allows a comparison with a heuristic taking a similar computational time. The improvement with respect to the former is clear (the average gap decreases from 0.93% to 0.33%): it is a strict dominance, given that they visit the same solutions, and the *VNS* proceeds when the competitor just terminates. Also with respect to the try-all initialisation the average gap is better (0.33% versus 0.35%) and the *SQD* diagram shows a much larger fraction

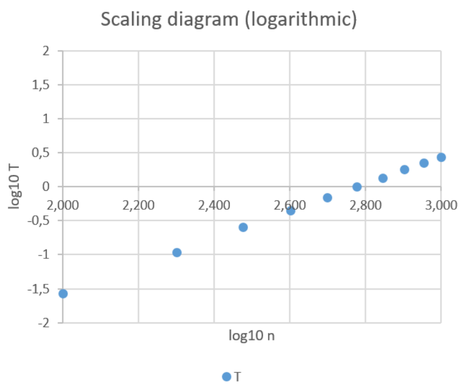

Figure 5.1: Scaling diagram in logarithmic scales for the *VNS* algorithm on the benchmark

of very good results, but also a larger fraction of bad results. Notice that the computational time is not the same: a true comparison should be made in perfectly equal conditions.

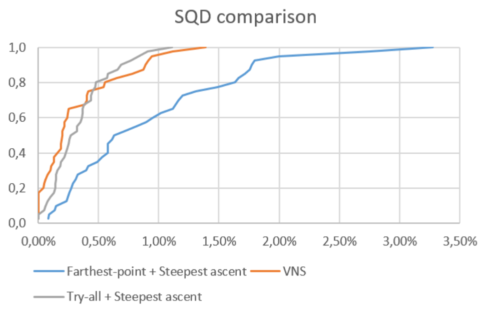

Figure 5.2: Solution Quality Distribution diagram for the *VNS* metaheuristic compared with the *steepest ascent* heuristic initialised with the *farthest point* and the *try-all* procedures

The boxplots reported in Figure 5.3 provide the same information: *VNS* provides both better and worse results with respect to the *steepest ascent* initialised by the try-all heuristic.

In order to have a rough idea of the corresponding computational times, Figure 5.4 provides the *RTD* diagram, from which it is apparent that *VNS* has some margin of further improvement for the slower runs, that correspond to the largest instances. On the contrary, it is already slower for the faster runs, that is the smallest instances. Therefore an equal-time comparison is clearly necessary to be fair. Performing one, however, is complicated when some of the competing algorithms have an intrinsic termination condition: we should build a time-limited version of the *VNS* heuristic, run the steepest ascent heuristic saving its computational time, and feed it to the *VNS* heuristic. This is beyond the scope of the course.

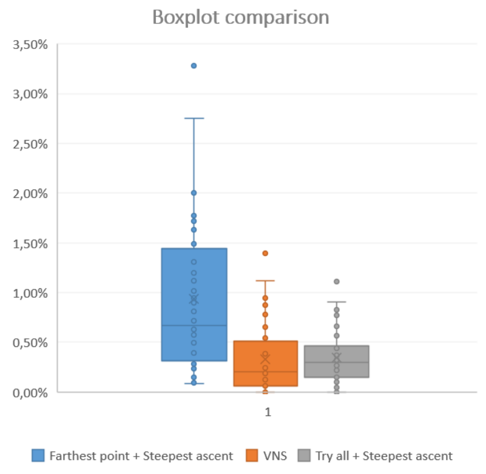

Figure 5.3: Boxplots for the *VNS* metaheuristic compared with the *steepest ascent* heuristic initialised with the *farthest point* and the *try-all* procedures

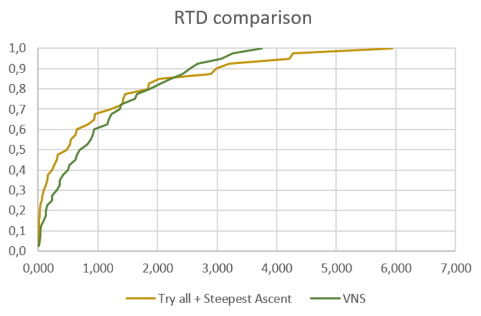

Figure 5.4: Solution Quality Distribution diagrams for the *VNS* metaheuristic and the *steepest ascent* heuristic initialised with the *try-all* procedure

5.2.3 Parameter tuning

It is often possible to improve the performance of an algorithm by tuning the values of its parameters. In the case of the *VNS*, this corresponds to increasing the smallest neighbourhood used to restart the search, or decreasing the largest one, or skipping some intermediate neighbourhoods. The last possibility is useful when the number of neighbourhoods is huge. In this case, s ranges from 1 to 400, that is quite large. As we have fixed a rather small total number of neighbourhood explorations $t_{\max}$, it is possible (and it can be easily verified printing the value of s at each shaking operation) that on the larger benchmark instances only small shakings are performed. In order to test this aspect, we compare some alternative parameter configurations. The test is very limited, considering the following configurations, chosen so as to cover rather extreme cases:

- $s_{\min} = 1$ and $s_{\max} = k$: the trivial configuration (probably not fully exploited, for the insufficient number of iterations);
- $s_{\min} = 1$ and $s_{\max} = 10$: an intensifying configuration, imposing rather small perturbations;
- $s_{\min} = 1$ and $s_{\max} = k/2$: a configuration avoiding only very large perturbations (this could coincide with the trivial fone if the insufficient number of neighbourhood explorations forbids to increase s beyond the upper bound);
- $s_{\min} = k/2$ and $s_{\max} = k$: a configuration producing strong perturbations since the beginning to diversify the search and try to avoid falling back in the reference solution;
- $s_{\min} = k$ and $s_{\max} = k$: a degenerate configuration in which the maximum possible perturbation is applied (all points must in the best known result are replaced by new random points);

The number of configurations and the number of iterations for each configuration are too limited to provide really significant results, but they are a good occasion for some intuitions and further discussion. A good way to formulate reasonable configurations is to print the values of s that improve the best known result. Increasing $s_{\min}$ and decreasing $s_{\max}$ to approach such values will modify the overall behaviour of the algorithm, because the same sequence of pseudorandom numbers will imply completely different choices, but it is probably a good idea: if no improvement is found out of a certain range of values of s, this probably means that the radius of the basins of attraction falls in a similar range, and therefore a good perturbation should be in the same range.

A quick glance at the computational times suggests that they are nearly independent from the parameters: the variations are usually below 10%, that in a period of few seconds is probably just due to random fluctuations. The quality of the results is roughly indicated by Table 5.1 which reports the average gaps with respect to the best known result. The diversifying configuration ($[k/2, k]$) is the best one, followed by the relaxed configuration ($[1, k]$) and the slightly intensifying one ($[1, k/2]$), that are very similar, as expected. The strongly intensifying and the strongly diversifying configuration appear to be the worst. It can be interesting to notice that the strongly diversifying configuration ($[k, k]$) actually proved the best in one of my personal contributions to the literature, which combined rather short runs of a *TS* metaheuristic with a *VNS* restart mechanism. This is not unreasonable, given that the *TS* procedure probably guarantees a good exploration of the region surrounding the current best known result (better than the steepest ascent procedure adopted here), so that stronger perturbations in the restart make sense.

$s_{\min}$	$s_{\max}$	Average gap
1	k	0.33%
1	10	0.45%
1	$k/2$	0.34%
$k/2$	k	0.22%
k	k	0.44%

Table 5.1: Average gaps with respect to the best known result of the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

Figures 5.5 and 5.6 show the *SQD* and boxplot diagrams of the five configurations, that clearly confirm the dominances suggested by the average gaps. Indeed, checking the results of the steepest ascent heuristic initialised with the try-all procedure, the diversifying configuration seems to be comparable even in the upper part of the diagram.

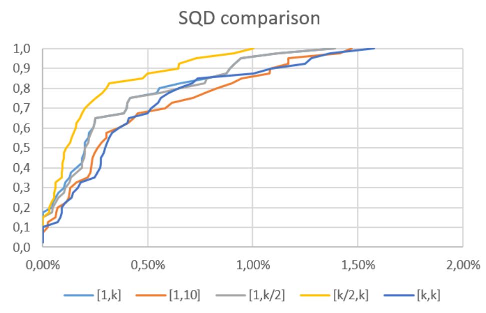

Figure 5.5: Solution Quality Distribution diagrams for the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

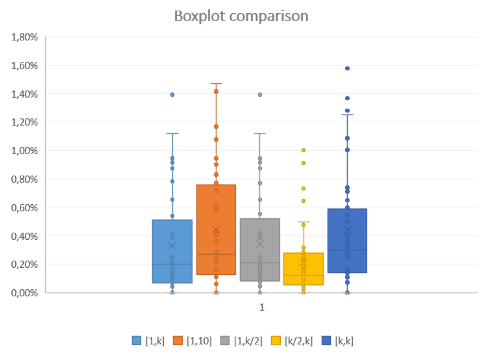

Figure 5.6: Boxplot diagrams for the *VNS* metaheuristic with different tunings of the shaking range $[s_{\min}, s_{\max}]$

Statistical tests

We can also compare the performance of the five configurations with statistical tests. Using Wilcoxon's test poses a methodological problem: the test is designed to compare two empirical populations. It is possible to use it on more than two algorithms, applying it to all pairs or choosing a reference heuristic and comparing the other ones to it. This, however, requires to handle with care the interpretation of the results. The p-value obtained, in fact, estimates the probability of observing the empirical results under the assumption that the two samples compared derive from the same population, that is, that the two configurations compared are equally effective. When p is small, this interpretation of the results can be rather safely rejected, but it must not be considered as straightforwardly false. If the test is applied n_t times, the probability that at least one of the interpretations drawn from the test is false becomes larger and larger. This means that our conclusions should be based on stricter requirements. The literature offer several methods to correct the estimates provided by a pairwise test. The simplest one is the *Bonferroni correction*, that is based on Boole's inequality for the *familywise error rate* (*FWER*):

$$FWER = P\left[\bigcup_{i=1}^{n_t} (p_i \leq \alpha)\right] \leq \sum_{i=1}^{n_t} P\left[(p_i \leq \alpha)\right]$$

In other words, the sum of the p-values obtained gives an overestimate of the p-value for the overall observation. A simple way to impose it every given threshold on the significancy of the results (for example, the classical 5%) consists in dividing such a threshold by the number n_t of the tests.

For example, let us check the hypothesis that the diversifying configuration is better than the other four. This corresponds to the four following different pairwise tests:

1. $[k/2, k]$ versus $[1, k]$

```
W+ = 470, W- = 91, N = 33, p <= 0.0007329
```

2. $[k/2, k]$ versus $[1, 10]$

```
W+ = 630, W- = 73, N = 37, p <= 2.743e-005
```

3. $[k/2, k]$ versus $[1, k/2]$

```
W+ = 499, W- = 96, N = 34, p <= 0.0005896
```

4. $[k/2, k]$ versus $[k, k]$

```
W+ = 612, W- = 18, N = 35, p <= 1.197e-006
```

The hypothesis is consistent with the observations even after applying Bonferroni's correction, since the sum of all p-values is 0.2% and each one is smaller than $5\%/n_t = 1.25\%$ (even a threshold much tighter than 5% would be respected).

Of course, other systems of assumptions, such as a full ordering among the configurations, could be checked, but they would probably be less interesting: we are mainly interested in finding the best performing configuration on the given benchmark.

5.3 Tabu search

The *Tabu Search* metaheuristic applies a basic steepest ascent heuristic, but modifies it introducing a limitation of the neighbourhood that forbids to accept as incumbent a solution already visited, but also (in the standard attribute-based version) a solution similar to the ones already visited. In order to adapt the *TS* metaheuristic to the *MDP*, we will initialise the search with the farthest-point heuristic and adopt the basic single-swap neighbourhood N_{S_1} as done for the *VNS*. This should make the comparison between the two approaches fairer, allowing to understand whether the problem is better attacked by restarting or prolonging the search after hitting a locally optimal solution. The general *TS* scheme can be easily adapted to the *MDP* with the usual replacements due to the maximising nature of the problem.

```
Algorithm TabuSearch$\left(I, x^{(0)}, t_{\max}, L\right)$
$x := x^{(0)}$; $x^* := x^{(0)}$;
$\bar{A} := \emptyset$;
For $t := 1$ to $t_{\max}$ do
  $x' := \emptyset$;
  For each $y \in N(x)$ do
    If $f(y) > f(x')$ then
      If Tabu$\left(y, \bar{A}\right) = false$ or $f(y) > f(x^*)$ then $x' := y$;
    EndIf
  EndFor
  $x := x'$;
  $\bar{A} :=$ Update$\left(\bar{A}, x', L\right)$;
  If $f(x') > f(x^*)$ then $x^* := x'$;
EndFor
Return $(x^*, f(x^*))$;
```

First, we must decide whether to apply the basic version of *TS* or the most common attribute-based one. In the *MDP* literature, the best performing algorithm proposed so far to solve the problem actually uses the basic version. However, this algorithm introduces several refinements, and we are interested in the *MDP* mainly as an example for the application of *TS* to general Combinatorial Optimization problems. We will therefore implement an attribute-based *TS* metaheuristic. It is rather natural to define a pair of complementary attribute sets, that are the presence and the absence of given points in the current solution. Setting $A = x$ and $A' = P \setminus x$ means that every time a point is deleted from the solution, it becomes tabu for a given number L^{in} of neighbourhood explorations to add it back; conversely, every time a point is added to the solution, it becomes tabu for a given number L^{out} of neighbourhood explorations to delete it.

Procedure *tabu_search* implements both tabu lists on a single integer vector `T`, each of whose components reports the last iteration at which the corresponding point has changed its status: if i currently belongs to x, `T[i]` is the iteration at which i has entered the solution; if i currently does not belong to x, `T[i]` is the iteration at which i has gone out of the solution. At the beginning, `T[i]` is set to `-MAX_INT` for all points, so that the check on the tabu status of any point states that the point is not tabu, and therefore free for exchanges. Procedure `explore_neighbourhood_with_tabu` differs from the `explore_neighbourhood` procedure implemented in the previous chapter for the steepest ascent heuristic in that it takes into account also the tabu status (in fact, it requires the current iteration index `iter`, vector `T`, the tabu tenures `Lin` and `Lout`, and the value of the best known solution `w_star.f`, to apply the aspiration criterium). The update of the tabu list simply amounts to saving the current iteration index in the two positions of vector `T` associated to the points swapped. We should also take into account the possibility that all moves in the neighbourhood are tabu. In that case, the neighbourhood exploration procedure should return the solution with the oldest tabu status. The current implementation simply returns no point and performs no move, remaining idle until some tabu expires. This is an inefficient implementation, that shall be corrected in future versions of the algorithm.

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

T = int_alloc(pI->n+1);
for (i = 1; i <= pI->n; i++)
  T[i] = INT_MIN;

for (iter = 1; iter <= max_iter; iter++)
{
  explore_neighbourhood_with_tabu(px,pI,visit_strategy,
                                  iter,T,l_in,l_out,x_star.f,
                                  &p_in,&p_out,&delta_f);
  if (p_in != NO_POINT)
  {
    swap_points(p_in,p_out,px,pI);
    T[get_index(p_in,pI)] = T[get_index(p_out,pI)] = iter;
    if (px->f > x_star.f) copy_solution(px,&x_star);
  }
}

free(T);

copy_solution(&x_star,px);
```

The exploration of the neighbourhood is performed exactly as in the steepest ascent heuristic, with the addition of a further check. If the currently explored swap is tabu (and this is checked by function `is_tabu`, then the move is performed only if it improves upon the best known one, that is if the improvement δf applied to the current solution value yields a value strictly better than the best known one (*aspiration criterium*). In this special case, in fact, the new solution is only apparently violating a tabu, and is on the contrary providing a precious overall improvement to the search process. We still apply the first-best exploration strategy, returning the first improving solution in the neighbourhood, but only if it is nontabu, or it satisfies the aspiration criterium.

```
*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) &&
         (!is_tabu(p_in,p_out,pI,iter,T,l_in,l_out,px->f+delta_f,fstar)) )
    {
      *pdelta_f = delta_f;
      *pp_in = p_in;
      *pp_out = p_out;
      if ( (delta_f > 0) && (strcmp(visit_strategy,"-fb") == 0) ) break;
    }
  }
```

Finally, the check of the tabu status simply consists in determining whether the current iteration index has reached or not the value at which the tabu expires. This must be checked both for the point i that is leaving the solution (and in that case the tabu tenure is `l_out`) and for the point j that is entering the solution (in that case the tabu tenure is `l_in`) .

```
if (f > f_star) /* aspiration criterium */
  return false;
else
  return ((iter <= T[get_index(p_in,pI)] + l_out) ||
          (iter <= T[get_index(p_out,pI)] + l_in));
```

5.3.1 Time complexity estimation

It is rather obvious that the computational complexity of the *TS* metaheuristic coincides with that of the basic steepest ascent heuristic, as all additional operations require constant time in their respective locations:

- the evaluation of the tabu status adds a constant number of operations to the evaluation of the cost of each explored solution;
- the update of the tabu list adds a constant number of operations to the execution of the move, that is the exploration of a neighbourhood;

The allocation, initialization and deallocation of vector `T` add $O(n)$ time to the overall algorithm. Therefore, the overall complexity remains $O\left(t_{\max}(n-k)k\right)$, where the number of neighbourhood explorations $t_{\max}$ is a parameter provided by the user as the termination condition.

5.3.2 Empirical evaluation

We can now evaluate the performance of the *TS* metaheuristic. We set the total number of neighbourhood explorations to $t_{\max} = 1\,000$, as for the *VNS* metaheuristic, in order to allow a meaningful comparison between them, even if a completely fair comparison would require to give them the same computational time.

Cyclic or erratic behaviours

Contrary to the *VNS*, where it is usually very easy to determine a default configuration for the parameters $s_{\min}$, δs and $s_{\max}$, the most influential parameters of the *TS*, that is the *tabu tenures* need to be tuned with a certain care since the beginning. Two basic complementary risks must be avoided:

- *cyclic behaviours*: if the tabu tenure is too short, the search can get stuck in a cyclic sequence of solutions, because the search is attracted by locally optimal solutions that have already been visited and the tabu expires before the cycle starts again;

- *erratic behaviours* or empty neighbourhoods: if the tabu tenure is too long, the search can wander in the solution space avoiding the more promising regions because these are too close to solutions that have already been visited; the neighbourhood can even become fully tabu.

Since the number of points out of the solution varies from $n-k=60$ to $n-k=900$, and the number of points in the solution varies from $k=10$ to $k=400$, the tenures should keep below these values, but above a few units, which is still a rather large range. The tenures could also possibly depend on the size of the instance. It is also likely that the tenure for reinsertion L^{in} should be larger than the tenure for redeletions L^{out}, because the candidate elements for insertion (the $n-k$ external ones) are more numerous than the candidate elements for deletion (the k internal ones). A simple way to verify the occurrence of the cyclic behaviours and (less evidently) of erratic ones, is to plot the profile of the objective function along the search.

In order to give a quick idea of how reasonable values are introduced, we will fix focus on the smallest instance (`n0100k010.txt`), and investigate different possible values for L^{in} while trivially setting $L^{\text{out}}=1$, so that the tabu mechanism is mainly based on the reinsertion of deleted elements. Figure 5.7 reports the profile of the objective function for the first 100 neighbourhood explorations. The configuration with $L^{\text{in}}=5$ exhibits a clearly periodic profile, suggesting that the tenure is too short. The one with $L^{\text{in}}=6$ appears much better, but long cycles (with a period of 106 iterations actually arise after a while). The configuration with $L^{\text{in}}=8$ hits the best known result ($f^*=3\,561$) several times during the whole run. The configuration with $L^{\text{in}}=30$ starts moving erratically in regions of worse quality. Finally, the configuration with $L^{\text{in}}=100$ seems to avoid good solutions and also shows sequences of iterations in which the value of the objective does not change because the whole neighbourhood is tabu ($L^{\text{in}}=100>n-k=90$) and our simple implementation trivially waits for the tabu to expire.

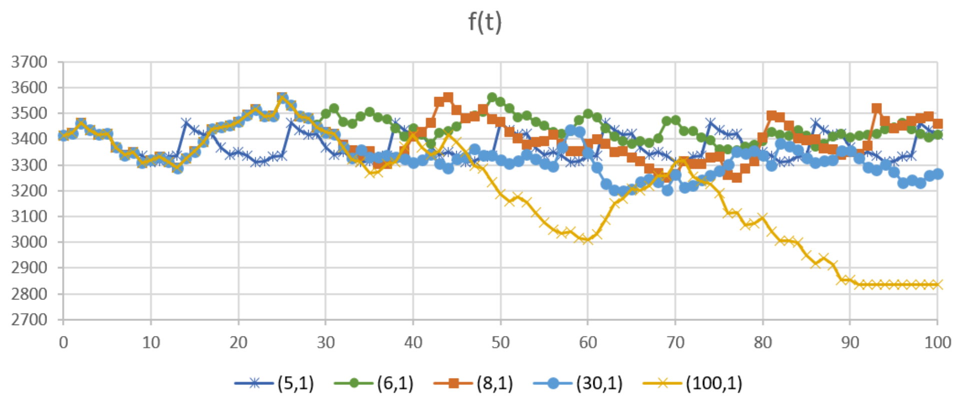

Figure 5.7: Profile of the objective function for the *TS* metaheuristic on instance `n0100k010.txt` with different values of tenures $\left(L^{\text{in}},L^{\text{out}}\right)$

Figure 5.8 reports the profile of the objective function for the whole run on the largest instance (`n1000k400.txt`). The configurations with the smallest values of L^{in} actually prove better, without exhibiting cyclic behaviours. This (absolutely nonobvious) behaviour is possibly due to the much larger size of the neighbourhood, that allows the algorithm to choose the incumbent in a larger set, and thus reduces the risk of repeating the same sequence of moves.

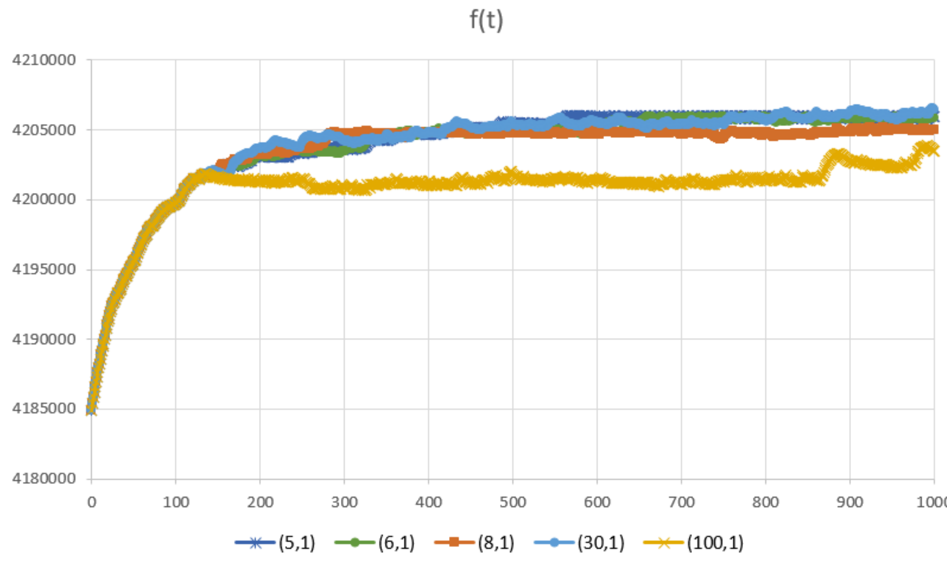

Figure 5.8: Profile of the objective function for the *TS* metaheuristic on instance `n1000k400.txt` with different values of tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

Computational time analysis

The overall computational time, including the initialisation procedure is very similar to that of the *VNS* metaheuristic: it ranges from fractions of a second to about four seconds. It is therefore another interesting candidate to provide an efficient solving approach. Figure 5.9 reports the logarithmic *scaling diagram* on the whole benchmark. The diagram shows a polynomial increase with size, in good accordance with the $O\left((n-k)k\right)$ theoretical estimate, which corresponds to a quadratic complexity when $k \propto n$. In fact, the linear interpolation is:

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

since $\alpha \approx 2.012$ and $\beta \approx 2.7 \cdot 10^{-6}$.

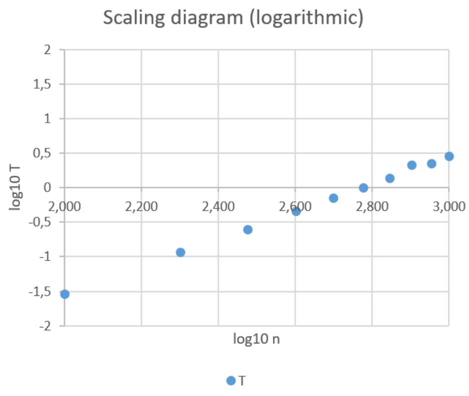

Figure 5.9: Scaling diagram in logarithmic scales for the *TS* algorithm on the benchmark

5.3.3 Parameter tuning

We now compare a small number of configurations in which both tenures L^{in} and L^{out} assume different values. For the sake of simplicity, we apply the same values to all instances of the benchmark, also based on the results of the previous experiments, which showed that rather short tenures are enough to avoid cyclic behaviours, and at the same time to allow the search to focus on good quality solutions, both on the smallest and the largest instance. We consider the six configurations obtained setting $L^{\text{in}} = 5$, 6 or 8 and $L^{\text{out}} = 1$ or 2. This is, of course, only a very simple illustrative investigation. Table **??** reports the average gaps over the whole benchmark. They are quite promising: most of them are smaller than the corresponding values obtained by the *VNS* metaheuristic; the larger tenures, in particular, provide the best gap.

L^{in}	L^{out}	Average gap
5	1	0.31%
6	1	0.26%
8	1	0.19%
5	2	0.17%
6	2	0.18%
8	2	0.14%

Table 5.2: Average gaps with respect to the best known result of the *TS* metaheuristic with different tunings of the tabu tenures L^{in} and L^{out}

Figure 5.10 reports the *SQD* diagram of the six configurations, confirming the better performance of the configuration with the larger tenures.

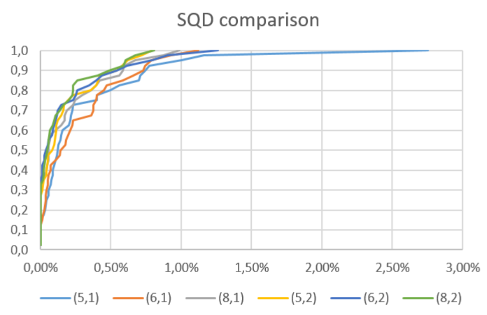

Figure 5.10: Solution Quality Distribution diagrams for the *TS* metaheuristic with different tunings of the tabu tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

The corresponding boxplots are reported in Figure 5.11: they seem to contradict the *SQD* diagram ($L^{\text{in}} = 1$ and $L^{\text{out}} = 8$ look better than $L^{\text{in}} = 2$ and $L^{\text{out}} = 8$), but that mainly depends on the automatic definition of outliers by Excel[3] It seems anyway justified to consider larger tenures as better. Indeed, the experiments should now include larger values, to determine whether it is possible to further improve the results, but we stop here.

The application of Wilcoxon's test to compare $\left(L^{\text{in}}, L^{\text{out}}\right)$ with the other configurations gives the following results:

[3]There seems to be a formula to label a value as an outlier. I do not know whether this is a statistic standard or just an Excel convention.

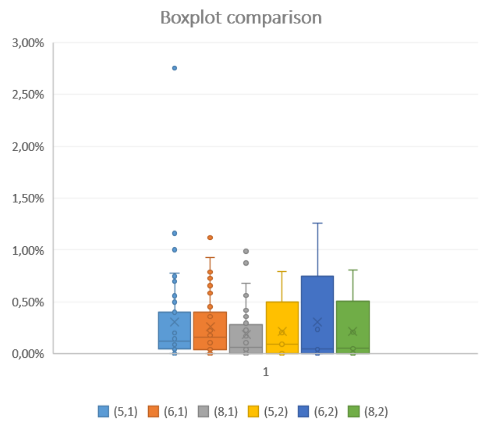

Figure 5.11: Boxplot diagrams for the *TS* metaheuristic with different tunings of the tabu tenures $\left(L^{\text{in}}, L^{\text{out}}\right)$

1. $(8,2)$ versus $(5,1)$

```
W+ = 432, W- = 64, N = 31, p <= 0.0003233
```

2. $(8,2)$ versus $(6,1)$

```
W+ = 494, W- = 101, N = 34, p <= 0.0008056
```

3. $(8,2)$ versus $(8,1)$

```
W+ = 261.50, W- = 173.50, N = 29, p <= 0.3469
```

4. $(8,2)$ versus $(5,2)$

```
W+ = 313, W- = 152, N = 30, p <= 0.09987
```

5. $(8,2)$ versus $(6,2)$

```
W+ = 248, W- = 187, N = 29, p <= 0.5165
```

which suggests that only the first two comparisons, with configurations having both tenures shorter, are significant (even applying the Bonferroni correction), whereas the other comparisons could easily be the result of a random sampling.

5.4 Comparison between *VNS* and *TS*

To conclude our experiments, we compare the *VNS* and the *TS* metaheuristic, with their best performing configurations, both with respect to the quality of the solutions and the computational time. The corresponding *SQD* and *RTD* diagrams are reported in Figures 5.12 and 5.13, and show that the *TS* metaheuristic is more effective while taking the same time as the *VNS* algorithm. The result is confirmed by the following response of Wilcoxon's test:

```
W+ = 131, W- = 464, N = 34, p <= 0.00454
```

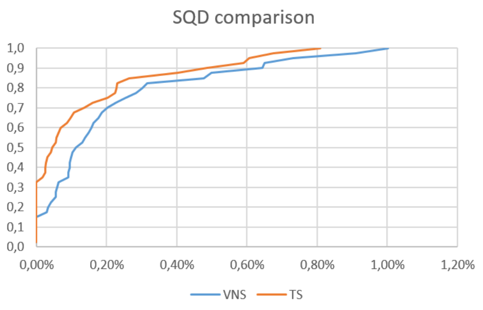

Figure 5.12: Solution Quality Distribution diagrams for the *VNS* metaheuristic and the *TS* metaheuristic

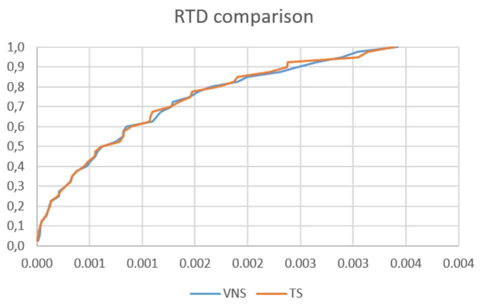

Figure 5.13: Run Time Distribution diagrams for the *VNS* metaheuristic and the *TS* metaheuristic

It is important not to overestimate the range of these conclusions. They refer to a not very large benchmark of a specific nature, to a computation consisting of $t_{\max} = 1\,000$ iterations, to a pair of configurations that have been obtained with a very short investigation. They are however a preliminary result of a certain soundness, obtained with an experimental methodology based on the formulation and verification of hypotheses.