Università degli Studi di Milano

Heuristic algorithms (laboratory sessions)

Roberto Cordone

– Academic year 2020/21 –

Contents

Chapter 3

Constructive metaheuristics

3.1 Introduction

This chapter discusses the application of constructive metaheuristics to the *Maximum Diversity Problem* (*MDP*). Constructive metaheuristic try to improve the results of a basic constructive heuristic by running it repeatedly with the introduction of mechanisms that modify its final result. In the end, of course, the algorithm returns the best of the solutions found during the process. The main mechanisms used by metaheuristic algorithms to enhance a constructive heuristic are:

1. the use of different selection criteria, typical of *multi-start* algorithms;
2. the use of *random choices*, typical of *GRASP*;
3. the use of *memory*, typical of the *Ant System*

In the following, we will implement *GRASP* and *Ant System* algorithms for the *MDP*, based on the constructive heuristics discussed in the previous chapter[1]. In the literature, these two approaches require the introduction of exchange procedures to improve the solutions generated by the constructive mechanism. In order to focus on the latter, however, we will avoid them.

Correspondingly, the `main` function allows to choose from the command line which of the two algorithms to apply (with option `-grasp` for the *GRASP* and `-as` for the *Ant System*) and to provide the numerical values of the following parameters:

- for the *GRASP* heuristic:
  - the total number of iterations ℓ
  - the randomness parameter μ
- for the *Ant System* heuristic:
  - the total number of iterations ℓ
  - the randomness parameter q
  - the oblivion parameter ρ

plus the seed required to initialise the pseudorandom number generator.

The other operations (loading the data, allocating and deallocating the data and the solution, determining the computational time and printing the results on

[1] Presently, the chapter only includes the *GRASP* algorithm.

the screen) are the same as for the constructive heuristics, except for the fact that also the parameters are printed, so that the report keeps trace of how each single solution was obtained to help guarantee the reproducibility of the results[2]

```
parse_command_line (argc,argv,data_file,algo,&iterations,&mu,&q,&rho,&seed);

load_data(data_file,&I);
//print_data(&I);

create_solution(I.n,&x);

inizio = clock();
if (strcmp(algo,"-grasp") == 0)
  grasp(&I,&x,iterations,mu,&seed);
else if (strcmp(algo,"-as") == 0)
  ant_system(&I,&x,iterations,q,rho,&seed);

fine = clock();
tempo = (double) (fine - inizio) / CLOCKS_PER_SEC;

printf("%s ",data_file);
for (arg = 2; arg < argc; arg++)
  printf("%s ",argv[arg]);
printf("%10.6lf ",tempo);
print_solution(&x);
printf("\n");

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

3.2 Greedy Randomized Adaptive Search Procedure

The *Greedy Randomized Adaptive Search Procedure* (*GRASP*) is a development of the classical *semigreedy* algorithm, that we will actually implement. Its basic idea is to modify the scheme of constructive algorithms by replacing the deterministic choice of the element that provides the best value of the selection criterium

$$i^* := \arg \min_{i \in \Delta^+(x)} \varphi_A(i, x)$$

with a stochastic choice $i^*(\omega)$. This requires to define a probability distribution on set $\Delta^+(x)$, that should be biased so as to favour the best elements over the worst ones:

$$\varphi_A(i, x) \leq \varphi_A(j, x) \Leftrightarrow \pi_A(i, x) \geq \pi_A(j, x)$$

The following pseudocode provides the basic scheme of *GRASP* for maximisation problems, adapted to the specific application to the *MDP* by translating the termination condition into a check on the cardinality of the current subset, by returning the last visited subset (as it is the only feasible one) and by replacing the search for a minimum cost solution with the search for a maximum value one. The search procedure after each constructive phase is also neglected:

[2]Of course, the code could change and yield different results for the same parameter values,

Algorithm 1 GRASP

```
1: procedure GRASP(I, ℓ, μ)
2:     x* := ∅; f* := 0;                          ▷ Best solution found so far
3:     for l := 1 to ℓ do
4:         x := ∅;
5:         while |x| < k do                       ▷ Randomised constructive procedure
6:             φ_i := Σ_{j∈x} d_ij for all i ∈ P \ x;      ▷ φ_i := (f(x ∪ {i}) − f(x))/2
7:             i* := BiasedRandomExtraction(P \ x, φ, μ);
8:             x := x ∪ {i*};
9:         end while
10:        if f(x) > f* then
11:            x* := x; f* := f(x);
12:        end if
13:    end for
14:    return (x*, f*);
15: end procedure
```

The algorithm performs ℓ iterations, where ℓ is a number chosen by the user, based on the available time. It is easy to recognise in the inner **while** a modified version of the basic constructive heuristic presented in the previous chapter. The algorithm still computes all values of $\varphi_A(i, x)$, but, instead of choosing the largest one, it selects one at random based on a biased probability distribution (with parameter μ) that favours the largest ones. At the end of each iteration, the current solution possibly updates the best known one.

The scheme is quite similar to the one used for the greedy "try-all" heuristic, as it requires to create and iteratively fill and empty a current solution, while updating the best known one, that will be returned in the end. We just replace the function `best_point_to_add(px,pI)` used in the basic greedy heuristic, with a function `biased_random_point_to_add(px,pI,pseed,pmu)` that performs the evaluation of the selection criterium $\varphi_A(i, x)$ based on the current solution `x` and on the instance `I`, the extraction of a pseudorandom number depending on the `seed` and the biased stochastic selection of a new point `i` to add based on the parameter `mu` provided by the user.

```
create_solution(pI->n,&x);
for (iter = 1; iter <= iterations; iter++)
{
  while (get_card(&x) < pI->k)
  {
    i = biased_random_point_to_add(&x,pI,mu,pseed);
    add_point(i,&x,pI);
  }

  if (x.f > px->f) copy_solution(&x,px);
  clean_solution(&x,pI->n);
}
destroy_solution(&x);
```

but this should no longer be the case when the implementation has reached a sufficiently stable status.

3.2.1 Choice of the basic constructive heuristic

In the previous chapter, we have considered four alternative (though very similar) constructive heuristics, and a destructive one. we now discuss which of the heuristics should be adopted as the core of the *GRASP* approach.

We will leave aside the destructive heuristic, because it is conceptually different from the other ones, though it would indeed be interesting to compare the constructive and destructive approaches in a metaheuristic allowing to give them equal time[3]. We will also exclude the try-all heuristic, that was the best-performing one, but also the slowest, and in a sense it is already a sort of multi-start metaheuristic, using the initial point as a parameter whose value changes at every iteration. Since a metaheuristic is intrinsically less efficient than the basic heuristic on which it is based, we prefer to choose a simple and fast mechanism, rather than a slow and complex one, at least for a first experiment. There is always time to introduce complications and refinements, if justified by theory or by experience.

The basic heuristic had a strong drawback: it could only generate solutions including point 1. Is this drawback still present in a randomized version? The answer depends on the probability distribution selected. We remind that at the first step the selection criterium is equal to zero for all points. Therefore:

- a scheme based on a *Heuristic Biased Stochastic Sampling* (*HBSS*) would choose any point, but it would assign larger probabilities to the first points;
- a scheme based on a *Restricted Candidate List* (*RCL*) would choose with uniform probability one of the first points.

In the first case, all solutions can be obtained, but there is a bias towards those including the points with small indices, and such a bias is not justified by any good reason. In the second case, the points with larger indices could even be impossible to reach for some instances (that depends also on the values of the distance function). Since for the sake of simplicity, we are going to test only a *RCL* approach, the basic greedy heuristic is not a good choice, unless with some additional correction.

The farthest-point and the farthest-pair heuristic still introduce a bias, or a deterministic advantage, in favour of points with a large total distance, or pairs of very distant points. Such a bias is less unreasonable, but still not provably justified. Moreover, in the case of point pairs, since the number of possible distances is not huge, several pairs of points could have the same distance, and therefore the discrimination between them would end up being based on their indices, and that would not be reasonable.

A very simple idea to avoid an index-based bias at the first iteration of the procedure could be to select the first point at random with uniform probability. At the following step[4], the choice would be stochastically biased in favour of the point that is farthest from a first one selected uniformly at random. That is similar to the farthest-pair heuristic, but different in that at least one of the two indices would be selected at random, without any index-based bias. For the sake of simplicity, we will apply this idea. This means that when x is empty the procedure directly selects one of the points with probability $\pi_i = 1/n$ without computing the selection criterium (that would be trivially equal to zero for all points $i \in P$).

When x is not empty, we compute the selection criterium for all external points and proceed to a biased random extraction from the available alternatives. Instead

[3] That's for a future laboratory, but it could be a good exercise.

[4] The following discussion is unrevised brainstorming: I do not expect it to be very clear, or even correct, but I think that the elements discussed are indeed relevant for the behaviour of the algorithm.

of simply saving the maximum of these values, we save them progressively in a vector `phi` and the corresponding points in a vector `P`, returning the final length `num` of the two vectors, that corresponds to the number of possible extensions $|\Delta^+(x)| = P \backslash x$[5].

```
if (get_card(px) == 0)
  i = get_point(rand_int(1,pI->n,pseed),pI);
else
{
  P = point_alloc(pI->n+1);
  phi = int_alloc(pI->n+1);

  num = compute_selection_criterium(px,pI,P,phi);
  i = biased_random_extraction(P,phi,num,mu,pseed);

  free(P);
  free(phi);
}

return i;
```

The selection criterium is still the value of the objective, that is

$$\varphi_A(i,x) = f(x \cup \{i\}) = \sum_{j \in x \cup \{i\}} \sum_{k \in x \cup \{i\}} d_{jk}$$

replaced for the sake of efficiency by half of its variation $\delta f(x,i) = (f(x \cup \{i\}) - f(x))/2$

$$\delta f(x,i) = \sum_{j \in x} d_{ji}$$

It is therefore computed with the function `dist_from_x(i,px,pI)` that was implemented in the previous chapter.

```
cnt = 0;
for (i = first_point_out(px); !end_point_list(i,px); i = next_point(i,px))
{
  cnt++;
  P[cnt] = i;
  phi[cnt] = dist_from_x(i,px,pI);
}

return cnt;
```

3.2.2 Pseudorandom number extraction

In order to generate random numbers, we exploit a classical pseudorandom number generator (the `ran1` generator described in the *Numerical recipes in C*). This is a function that receives in input a *seed*, that is a negative integer number, modifies that number (this is why the seed is passed by reference) and returns in output a real number ω that tends to assume a uniform distribution in the range $[0;1]$ as the function is repeatedly called. The first value of the seed is selected by the user, fed to the algorithm in the command line and determines the overall sequence

[5] Strictly speaking, this value is therefore already known, so that it would not be necessary to retrieve it from the computation.

of numbers generated. This is why the numbers of the sequence are denoted as *pseudorandom*. The ability to generate the same sequence in all runs is fundamental for the repeatability of the approach and the reproducibility of the results. In short, it is a basic condition for the scientific investigation of the problem.

3.2.3 Biased point selection

Given the pseudorandom number ω, it is necessary to determine the corresponding point in $\Delta^+(x) = P \setminus x$ with a biased scheme that favours the points with larger values of ϕ_i. This mechanism depends on the probability distribution adopted. In the following, for the sake of simplicity, we will adopt a *value-based Restricted Candidate List* (*RCL* scheme, in which:

1. a real parameter $\mu \in [0;1]$ is used to fix an intermediate threshold $\bar{\varphi}(x,\mu)$ between the minimum and maximum values available for ϕ_i;
2. the points i such that ϕ_i is better than (i.e., above) the threshold enter the *RCL*;
3. a point is selected from the *RCL* with uniform probability.

Other schemes commonly adopted in *GRASP* heuristics employ a cardinality-based *RCL*, a linearly decreasing or an exponentially decreasing probability profile on all external points. All of them assign decreasing probabilities to the possible choices $i \in \Delta^+(x)$ sorted by nondecreasing values of ϕ. In short, if i_r is the element in position $r = 1, \ldots, |\Delta^+(x)|$ in the ranking, $\phi(i_1, x) \geq \phi(i_2, x) \geq \ldots \geq \phi(i_{|\Delta^+(x)|}, x)$. Moreover, the typical schemes adopted in *GRASP* heuristics define probabilities based on the ranking of the choices, and not on the absolute values of the selection criterium, that is $\phi(i_r, x)$ can be expressed as a function of r and $|\Delta^+(x)|$.

Value-based *RCL*

This scheme computes an adaptive threshold depending on the values of the selection criterium on the available extension.

$$\bar{\varphi}(x,\mu) = (1-\mu)\,\varphi_{\min} + \mu\varphi_{\max}$$

where

$$\varphi_{\min}(x) = \min_{i \in \Delta_A^+(x)} \varphi_A(i,x) \quad \text{and} \quad \varphi_{\max}(x) = \max_{i \in \Delta_A^+(x)} \varphi_A(i,x)$$

and defines a *RCL* as

$$\mathrm{RCL}(x,\mu) = \{i \in \Delta^+(x) : \varphi(i,x) \geq \bar{\varphi}(x,\mu)\}$$

Then, it assigns the elements keeping under the threshold a uniform probability, and the following ones a zero probability:

$$\pi_{i_r} = \begin{cases} \dfrac{1}{|\mathrm{RCL}(x,\mu)|} & \text{if } r \leq |\mathrm{RCL}(x,\mu)| \\ 0 & \text{if } r > |\mathrm{RCL}(x,\mu)| \end{cases}$$

The parameter $\mu \in [0;1]$ tunes the randomness of the choice, with $\mu = 0$ yielding a deterministic heuristic and $\mu = 1$ a random walk.

In order to implement this scheme, the procedure `biased_random_extraction` scans the vector `phi` a first time to determine its minimum and maximum values $\varphi_{\min}$ and $\varphi_{\max}$. Then, it computes the threshold based on parameter μ that discriminates from the other points the elements of the *RCL*, according to condition[6]

$$\varphi_i \geq (1-\mu)\,\varphi_{\max} + \mu\varphi_{\min}$$

Scanning again the *RCL* allows to determine the number of its elements and to move them at the beginning of the vector of points `P`. For the sake of efficiency, the points are copied overwriting the previous elements of the vector, because there is no need to keep the whole original content: only the elements of the *RCL* are necessary.

Identification of the selected point

Now, in order to extract one of them with uniform probability, it is enough to compute $\lceil \omega\,|\mathrm{RCL}(x,\mu)| \rceil$ and take the corresponding element of vector `P`. In *RCL*-based schemes, it is enough to multiply ω by the size of the *RCL* and round up the result. This provides the position of the point in the list, and therefore allows to access it directly.

The other schemes mentioned above require to first translate the pseudorandom number ω into a ranking position, and then to identify the point corresponding to that position in a sorted vector. Let us focus on the first phase. It is always possible to identify the ranking position that corresponds to ω by summing the probabilities associated to the subsequent rankings, and stopping when the sum reaches ω: the corresponding position is the required one. In practice, however, it is usually not necessary to perform this sum explicitly, since the structure of the values allows to compute the correct ranking more quickly, just as in the case of the *RCL* it was enough to multiply ω by the size of the list[7]

After computing the ranking position, one must still identify the corresponding point according to the order by nondecreasing values of ϕ. It is never required to fully sort vector `phi`: the extraction of the k-th largest element from a vector can in fact be performed in linear time applying suitable algorithms. We do not give details here since we limit our experiments to the *RCL* scheme.

```
phiMin = INT_MAX;
phiMax = -1;
for (cnt = 1; cnt <= num; cnt++)
{
  if (phi[cnt] < phiMin) phiMin = phi[cnt];
  if (phi[cnt] > phiMax) phiMax = phi[cnt];
}

barphi = (1-mu) * phiMax + mu * phiMin;

RCLsize = 0;
for (cnt = 1; cnt <= num; cnt++)
```

[6] The condition is complementary to the one given in the slides, because the selection criterium must be maximised, but μ still measures randomness.

[7] I need to work on this point, that is not clearly discussed in any paper or textbook at the best of my knowledge. I am pretty sure that a linear probability profile allows to give a closed-form quadratic expression of the cumulated probability in each ranking position r, and therefore to find the position corresponding to ω in constant time solving a second-order equation. For the exponential profile, something more sophisticated is required.

```
    if (phi[cnt] >= barphi)
    {
      RCLsize++;
      P[RCLsize] = P[cnt];  /* overwriting P for the sake of efficiency */
    }

  cnt = rand_int(1,RCLsize,pseed);

  return P[cnt];
```

3.2.4 Empirical evaluation

We can now evaluate the performance of the *GRASP* heuristic. Contrary to what we have done in the previous chapter, having acquired a certain understanding of what the heuristics are doing, we will try to avoid producing meaningless diagrams.

Computational time analysis

An *a priori* worst-case asymptotic analysis of the computational time can be based on the similar analysis made for the basic deterministic greedy heuristic. First of all, the constructive heuristic is run for a given number of iterations ℓ. This number is a relevant parameter, that could constant and totally unrelated to the size of the problem, but could also be chosen depending on it, with the idea that larger instances could require more iterations to be explored properly, or vice versa that larger instances allow less iterations because thery require a longer time for each run, and the overall time is limited. In general, therefore, the expression of the time complexity will include ℓ. The basic deterministic heuristic required time $O(nk^2)$ for each run. The randomised version requires additional time for the generation of the pseudorandom number (that can be assumed as constant), for the identification of the minimum and maximum values of the selection criterium and the construction of the *RCL* (that can be assumed as linear), for the biased random selection of the new point (constant time). Consequently, we can estimate an additional linear time per each of the k iterations of the constructive method: the resulting $O(nk)$ term is dominated asymptotically. Other additional terms are given by the comparison of each of the ℓ solutions obtained with the best known one and, possibly, the update of the latter. All these terms are asymptotically dominated, but they could have a perceivable influence on the empirical evaluation.

For the first experiment, we set $\ell = n$. The choice is clearly arbitrary, but it is motivated by the idea to allow each of the n points to be chosen with a reasonable probability as the starting point of the constructive heuristic. It also aims to obtain a computational time comparable to that of the "try-all" heuristic, which also had n repetitions of the basic constructive scheme. This should allow to better estimate the impact of the additional operations required by the randomisation (not the maintenance of the best known solution, that occurs also in the deterministic heuristic).

Figure 3.1 reports the *scaling diagram* for the whole benchmark, obtained setting $\mu = 0.1$. It can be noticed, however, that according to the theoretical analysis parameter μ should have very little influence on the computational time. The detailed reports show a limited slow-down as μ increases, but the difference is not significant. The diagram shows the expected increase of the computational time with size, and its logarithmic version in Figure confirms its polynomiality. The $O\left(\ell nk^2\right)$ theoretical estimate, with $\ell = n$ and $k \propto n$, suggests an overall $O\left(n^4\right)$

complexity, that is confirmed by the linear interpolation:

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

with $\alpha \approx 3.969$ and $\beta \approx 3.4 \cdot 10^{-10}$.

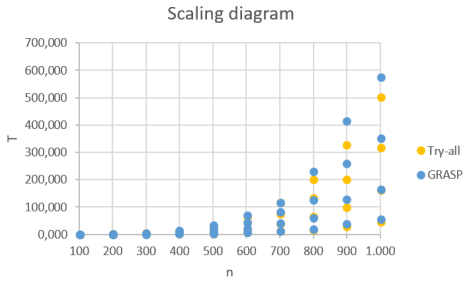

Figure 3.1: Scaling diagram for the greedy algorithm on the benchmark

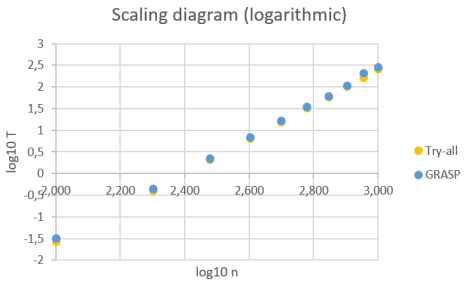

Figure 3.2: Scaling diagram in logarithmic scales for the greedy algorithm on the benchmark

The two figures report also the scaling diagram of the greedy "try-all" heuristic, that was expected to have a very similar performance with respect to the computational time. In fact, the two profiles are very similar, with the *GRASP* heuristic only slightly higher, confirming that the additional operations to randomise the selection of points affect very little the overall complexity.

Solution quality analysis

Without forgetting that the benchmark is rather small and specific, we can now draw the *SQD* diagram (see Figure **??**), to compare the different parameter tunings with one another. We consider the following tunings: $\mu \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$, after some preliminary experiments showed that larger values provided worse results. The same diagram can be used to compare *GRASP* with the algorithms already developed. In particular, the figure shows the profile of the "try-all" heuristic, that is the most similar one in terms of behaviour (applying a sequence of n different constructive heuristics) and computational time. The diagram is not very clear, but the versions with larger values of the randomness coefficient μ seem to perform slightly worse (the trend becomes clearer for larger values). Indeed, the performance of the "try-all" heuristic is similar to that of the smaller values of μ and better than the other ones. This is rather disappointing, and poses the question

whether the values tested are too large and should be reduced. In order to be sure, one would need to check the typical length of the *RCL* during the search. A rough estimate, based on the (unproved, but not unreasonable) assumption that the values of the selection criterium are uniformly distributed between $\varphi_{\min}$ $\varphi_{\max}$ is that the typical size decreases from μn to $\mu(n-k)$. For $\mu = 0.01$, n ranging from 100 to 1 000 and k ranging from $0.1\,n$ to $0.4\,n$, this corresponds to sizes ranging from 1 to 9, that do not seem so large.

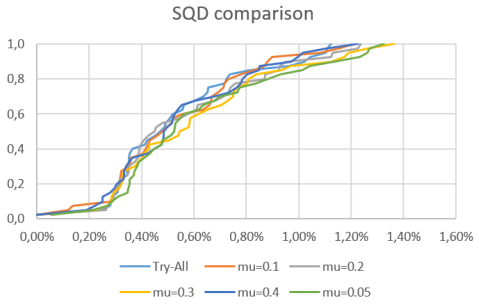

Figure 3.3: Solution Quality Distribution diagram for the *GRASP* algorithm (with $\mu \in \{0.01, 0.02, 0.03, 0.04, 0.05\}$) and the greedy "try-all" heuristic

The boxplots reported in Figure 3.4 provide a similar, perhaps slightly clearer, intuition. It should anyway be noticed that the *GRASP* heuristic can be prolonged for more than $\ell = n$ iterations, probably improving the final results, whereas the deterministic heuristic cannot. As well, we could experiment with shorter runs, to determine whether the results obtained actually could require a lower number of iterations. Anyway, the results remain unpromising.

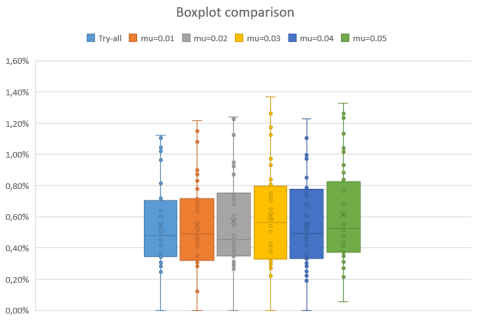

Figure 3.4: Boxplots for the *GRASP* algorithm (with $\mu \in \{0.1, 0.2, 0.3, 0.4\}$) and the greedy "try-all' heuristic'

Statistical tests We now apply Wilcoxon's test to determine whether it can discriminate between the results of different parameter tuning of *GRASP* and the deterministic competitor. The results are:

- for $\mu = 0.01$

```
W+ = 338.50, W- = 402.50, N = 38, p <= 0.6478
```

- for $\mu = 0.02$

```
W+ = 430.50, W- = 310.50, N = 38, p <= 0.3882
```

- for $\mu = 0.03$

```
W+ = 523.50, W- = 256.50, N = 39, p <= 0.06345
```

- for $\mu = 0.04$

```
W+ = 455.50, W- = 324.50, N = 39, p <= 0.3644
```

- for $\mu = 0.05$

```
W+ = 634.50, W- = 145.50, N = 39, p <= 0.0006617
```

In short, the difference, at first not statistically significant, tends to become more significant as μ increases (with an exception for $\mu = 0.04$. The only comparison that seems to exhibit a true dominance is the one between the deterministic heuristic and $\mu = 0.05$. The message is clearly not to exceed with randomicity.

Influence of the random seed An aspect that must be analysed when using random steps is the influence that randomness has on the final result obtained. If we run the algorithm a single time, in fact, the quality of the results achieved could be easily due to a lucky, or unlucky, choice of the random seed. In order to estimate the role of randomness, we should run the algorithm for several times, with different random seeds, and compare the results thus obtained. A simple index is given by the average quality of the solution with respect to a sufficiently large number of runs (at least 10, possibly more), but other indices of distribution are certainly relevant: the maximum and minimum values obtained, or the medians and quartiles. The description is very similar in principle to that we have given with respect to the benchmark instances (numerical indices, boxplots, *SQD* diagrams), but it considers single instances and variable seeds, instead of a single seed and random instances.

In order to give an idea of this kind of investigation, we select a single instance, that we consider as significant for some reason, and we run the algorithm for a given number of times with different seeds. We will consider the instance `n0600k060`, because it is the instance on which the *GRASP* algorithm with $\mu = 0.01$, that is the best performing one on average (though with nonsignificant differences with other ones) obtains the largest gap: $\delta_A(I) = 1.21\%$ (see the maximum of the second boxplot in Figure 3.4. The question investigated is whether this bad result was typical or derived from a particularly unlucky, or particularly lucky, choice of the random seed. In order to establish this, since the algorithm takes about 5 seconds to solve the instances, we run it 100 times with different random seeds, ranging from -1 to -100. Figure 3.5 provides the *SQD* with respect to the random seed. The gap δ is never huge, but indeed it varies in a rather large range (between 0.6% and 1.4%). The red line in the picture, that corresponds to the single run of the previous phase of experiments, suggests that its result was indeed rather unlucky. This suggests that the *GRASP* algorithm (at least on this instance) is rather unstable, that its results could be in practice better (but also worse) than the ones discussed above, that were obtained in a single (not necessarily representative) run, and that the conclusions drawn from such results should be handled with much care.

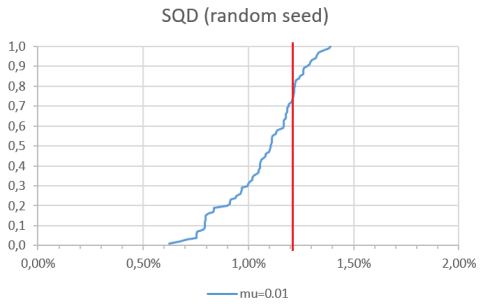

Figure 3.5: Solution Quality Distribution diagram for the *GRASP* algorithm with $\mu = 0.01$ on instance `n0600k060` with 100 different random seeds: the red line corresponds to the single run of the previous phase of experiments.

3.3 Ant System

The *Ant System* (AS) is a development of the classical *cost perturbation* algorithm, as well as of the *semigreedy* algorithm. Its basic idea is to modify the scheme of constructive algorithms by replacing the deterministic choice of the element that provides the best value of the selection criterium with a stochastic choice, influenced by additional information provided by the memory of previously found solutions. In addition, the AS considers a *population* of algorithms that work in parallel, iteration by iteration. Among the several variants of AS, we are going to implement the one that:

- defines the *visibility* of a point $i \in P \setminus x$ as the selection criterium used in the constructive and the $GRASP$ heuristic, that is

$$\eta(i,x) = \varphi(i,x) = \sum_{j \in x} d_{ji}$$

- maintains in a suitable vector a *trail* function $\tau(i)$ that depends only on the point i to be added, and is progressively updated during the execution;
- combines visibility and trail by multiplying them, as they both tend to associate larger values to better options;
- tunes the randomness of the choice with a parameter q, so that the choice is made selecting with probability $1 - q$

$$i^* = \arg\max_{i \in P \setminus x} \varphi(i,x)\,\tau(i)$$

  and with probability q a random point with probability distribution

$$\phi_i = \frac{\varphi(i,x)\,\tau(i)}{\sum_{j \in P \setminus x} \varphi(j,x)\,\tau(j)}$$

- applies a local update to the trail to diversify the search after each individual has built a solution;
- applies a global update to the trail to intensify the search on the points that belong to the best known solution found in the whole process.

The following pseudocode provides the scheme of this variant of the *AntSystem* adapted to the *MDP* as already done for *GRASP*. Also in this case the search procedure that should be run to improve the solutions built is neglected:

The algorithm performs ℓ iterations, in each of which it generates h different solutions; ℓ and h are numbers chosen by the user, based on the available time. It is easy to recognise in the inner **while** a modified version of the basic constructive heuristic presented in the previous chapter. The algorithm still computes all values of $\varphi_A(i,x)$, but, instead of choosing the largest one, it selects one at random based on a biased probability distribution (with parameter μ) that favours the ones with largest values of φ and τ. TALK ABOUT THE LOCAL TRAIL UPDATE At the end of each iteration, the current solution possibly updates the best known one. TALK ABOUT THE GLOBAL TRAIL UPDATE

$\tau = 1$ PERCHE' L'AGGIORNAMENTO E' Q-f?

Algorithm 2 AntSystem

```
1: procedure AntSystem(I, ℓ, q, ρ)
2:     x* := ∅; f* := 0;                                  ▷ Best solution found so far
3:     τ_i = τ_0 for all i ∈ P;
4:     for l := 1 to ℓ do
5:         for g := 1 to h do
6:             x := ∅;
7:             while |x| < k do                           ▷ Randomised constructive procedure
8:                 φ_i := f(x ∪ {i}) − f(x) for all i ∈ P \ x;
9:                 i* := BiasedRandomExtraction(P \ x, φ, τ, μ);
10:                x := x ∪ {i*};
11:            end while
12:            if f(x) > f* then
13:                x* := x; f* := f(x);
14:            end if
15:            τ := LocalTrailUpdate(x, τ, ρ);
16:        end for
17:        τ := GlobalTrailUpdate(x*, τ, ρ);
18:    end for
19:    return (x*, f*);
20: end procedure
```