# Lab report:
# Biologically Inspired Algorithms and Models

Part II: Extending search optimization algorithms with simulated annealing and tabu search for QAP

May 29, 2024

Thursday classes, 16:50.
Code available on a public Github repository: link

I declare that this report and the accompanying source code have been prepared solely by the above author, and all contributions from other sources have been properly indicated and are cited in the bibliography. Content added or altered for the second report is marked in Mahogany

# 1 Introduction

## 1.1 Problem description

This report discusses solving the Quadratic Assignment Problem (QAP) using local search methods. QAP was first described in "Assignment Problems and the Location of Economic Activities" [4]. The article focuses on optimizing the cost of business processes involving multiple locations. The quadratic assignment problem itself was represented in the context of attempting to minimize transportation costs between $n$ facilities (plants), where each must receive a specified number of units of some resource from others. $n$ locations to which plants can be assigned are available. Each plant must be assigned to a location, and every location must have exactly one plant assigned.

A problem instance is specified by two matrices, denoted in this report by $\mathbf{A}$ and $\mathbf{B}$. $\mathbf{A}$ stores the information about the cost of transporting goods from one location to another – for example, if $\mathbf{A}_{ij} = 5$, then moving a single unit from location $i$ to location $j$ costs 5. $\mathbf{B}$ represents how many units' worth of commodities must be transported from one plant to any other – thus, $\mathbf{B}_{ij} = 3$ means that plant $j$ requires 3 units from plant $j$.

A single solution to an instance of the Quadratic Assignment Problem is represented as a permutation, denoted by $\pi$. The permutation is of length $n$. The numbers within the permutation identify plants, while their order – to which locations they were assigned. Therefore, a permutation $\pi = [4, 1, 2, 3]$ would indicate that plant 4 was assigned to location 1, plant 1 to location 2, 2 to 3 and 3 to 4. With this in mind, the objective of the problem can be expressed as finding the permutation $\pi$ which minimizes the expression given in Equation (1), where $\pi(i)$ denotes the $i$th element of permutation $\pi$

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\mathbf{A}_{ij}\cdot\mathbf{B}_{\pi(i)\pi(j)} \tag{1}$$

The problem is relevant to optimizing the configuration of any system with clear required interactions between its constituents, and a way to measure their impact. This could mean assigning machines to tasks, minimizing the amount of expected load in a network, or, naturally, identifying locations for facilities that would minimize the cost of transporting goods between them.

QAP is known to be a challenging problem to solve. It was proven to be NP-hard in "P-Complete Approximation Problems" [5]. Searching the solution space is infeasible, due to its sheer size of $n!$. A detailed overview of the different varieties of QAP and known ways of tackling them can be found in [1].

## 1.2 Chosen instances and implementation overview

The following eight instances were chosen to be used in the experiments conducted for this report: `tai10a`, `tai20a`, `tai30b`, `tai40b`, `tai50b`, `tai60a`, `tai80a`, `tai100b`. While the implementation proved itself to be capable of handling the largest instances in the dataset (with 256 facilities and locations), this selection was made mostly due to there being solutions for all of its instances and a uniform growth in instance size, which ought to be conducive to providing clear visualizations and facilitating drawing conclusions.

All of the methods for solving QAP were implemented in Rust, version 1.76.0. This was the first project the author ever wrote in the language. The code mostly follows procedural programming design, with structs being used to represent problem instances and solutions. To maximize performance, critical data was stored in constant-size stack-allocated arrays [3]. They thus had to be sized to accommodate the largest instance in the dataset, regardless of which one the program was currently working on. By default, the arrays were filled with zeroes. Upon data being loaded, only the elements needed for the instance were updated. This meant that all procedures had to have access to the size of the current instance to prevent accessing the artificially introduced zero values. All running times are reported for the code compiled with the maximum `opt-level = 3` and no debug information (the default configuration for Cargo's `--release` build profile) [2]. The entire implementation is single-threaded. Whenever there was an option, memory was sacrificed to improve performance. "Move deltas" are calculated to reduce the time spent evaluating solutions.

Simulated annealing and tabu search methods implemented for this report follow the same guiding principles as the local search approaches. Simulated annealing was written to extend the greedy algorithm (it immediately moves to the first solution it accepts) while tabu search is more akin to steepest (it visits

the best solution among all that are found to be legal). Whenever possible, the algorithm parameters were set to reflect the recommendations given with the assignment specification.

### 1.2.1 Simulated annealing

The parameters were set up to match the proposed configuration closely. The **starting temperature** was set to accept a move from the 95th percentile from a sample of the solution space with at least a 50% probability. The "at least" is a consequence of how the delta used to compute the 95th percentile was estimated – it was set to the value of the delta at position ceil($0.95 \cdot$ num_sampled_deltas) within the vector of deltas sorted in ascending order. The samples were computed independently for every run of the algorithm. Sample deltas were obtained by generating *instance size* random solutions and calculating deltas for 10 available moves. **Markov chain length** was set to a constant 5, but due to the nature of this implementation, that still resulted in the number of evaluated solutions being required to change the temperature to be linearly proportional to neighborhood size (this will be explained later). The **cooling constant $\alpha$** was set to 0.9. The **stopping condition** is a bit more involved. The search can be terminated in two situations:

- No solution within a neighborhood is accepted
- Both of the following are true: the temperature decreased to or below a specified threshold (explained later); a given number of solutions, since the incumbent best solution was found, was visited (no improvement to the best solution discovered by the search so far was found for a number of iterations of the main search loop).

The **final temperature** threshold is set to express a situation, in which the probability of accepting *any* deterioration in the current solution is 0.01. The process of computing the final temperature is shown in Equation (2). Note that 1 is the smallest possible change that can make the solution worse, because all instance parameters are provided in integers, and the evaluation function exclusively performs addition and multiplication, two operations under which the set of integers is closed.

$$\frac{-\text{max accepted deterioration}}{\ln \text{acceptance probability}} = \frac{-1}{\ln 0.01} \approx \frac{-1}{-4.605} \approx 0.217 \tag{2}$$

To explain why the Markov chain length is set to a constant, the process of evaluating neighboring solutions must be explained. It operates similarly to the greedy procedure – it goes through the neighborhood of the current solution, in random order. But instead of moving on to the first improving solution, it moves on to the first solution that is accepted by the algorithm, based on the basic simulated annealing principle (comparing $\exp\left(\frac{-\Delta}{T}\right)$ to a random variable). Because of how the code operated, and how everything fit into the framework of greedy search, it was easier and more readable to increment the number of iterations at the current temperature level after a new solution was accepted, rather than after every delta calculation. One might argue that this approach is also somehow fairer – it gives every solution in the neighborhood a more even chance, given that they use the same temperature value. That means that every "unit" necessary to move towards the completion of the Markov chain was proportional to the neighborhood size – the exploration of an entire neighborhood might be necessary to accomplish this. Keeping with the spirit of using the recommended parameters, the number of iterations since the last change of the incumbent solution was set to be 10 times larger than the length of the Markov chain – a constant 50.

### 1.2.2 Tabu search

The tabu list was implemented to ban moves that were made recently. Since swapping element $i$ with element $j$ is the same as swapping $j$ with $i$, to speed up the process of updating the list, by convention it was assumed that the swap of $i$ and $j$ would be stored at position $\min(i,j), \max(i,j)$ within a fixed size, 2D, stack-allocated "tabu matrix". Its diagonal and lower triangular elements were set to 0 and never accessed by the algorithm. After conducting a search through the entire neighborhood of the current solution, and identifying the best legal move, the tabu list was updated – the position of the move was set to the value of the tenure and all others were decremented (but not below 0). Tabu search operated with the **tenure** (the number of algorithm steps for which the solution remained tabu) set to floor(instance_size/4).

There was only one **aspiration criterion**, the simplest possible one – a tabu move would be accepted if and only if it would result in obtaining a solution better than the best found so far. To prevent all of the tabu operations from making the algorithm impractically slow, the entire neighborhood was not evaluated at every iteration of the search. Instead, the mechanism of **elite candidate list** was used, with a fixed number of **10** candidates. The principal reason for this was to have the candidate moves stored in a stack-allocated array, which can not be accomplished in Rust when the array size is unknown at compile time. Candidate list size also allows for controlling the search behavior – with 1 candidate, the algorithm behaves almost exactly like steepest. Increasing the number of candidates speeds up the search, but increases the chance of selecting highly suboptimal moves. The list of candidates is computed whenever the search finds that there are no legal candidates left. The list is created by finding the best num_candidates moves in the neighborhood of the current solution. The best of these is then used as the next move. The delta of the worst of these is set to be the acceptance delta for a candidate moved to be deemed good enough. If at any point, changes to the current solution cause the candidate move to give a delta worse than this threshold, it is removed from the list of candidate moves. There was a single **stopping condition** – if no improvement over the best-known solution found so far was discovered within **instance size** changes of the current solution, the search terminated.

### 1.2.3 Heuristic

The heuristic used in comparisons to other methods is a simple "rank alignment heuristic". It creates the permutation by assigning the rows of the matrix of interactions with the smallest sums to be multiplied by rows of the matrix of costs with the largest sums and vice versa. The quality of the solutions it obtained was not good, but even the authors of [4] wrote: "Unfortunately, our tentative and heuristic exploration into this problem (...) also meets with a negative outcome".

## 2 Neighborhood operator

All methods that use a neighborhood operator (Steepest Local Search, Greedy Local Search, Random Walk, Simulated Annealing, Tabu Search), use the simple 2-OPT neighborhood. Such neighborhood is defined as all solutions that can be obtained from the starting permutation by swapping two of its elements. This means that a permutation of length $n$ has $\frac{n \cdot (n-1)}{2}$ neighbors.

## 3 Algorithm comparison

Experiments conducted for this section consisted of 500 runs of each algorithm for every instance. The timeouts for the random algorithms have been set according to the schema described in Table 1, based on the running times of greedy and steepest.

| Instance | Timeout [$\mu$s] |
|---|---|
| tai10a | 100 |
| tai20a | 200 |
| tai30b | 1600 |
| tai40b | 5200 |
| tai50b | 12500 |
| tai60a | 13000 |
| tai80a | 35000 |
| tai100b | 240000 |

Table 1: Search timeouts for random search and random walk, in microseconds

When any average value is plotted, it is surrounded by whiskers indicating the standard deviation of the statistic.

In all plots, a particular search algorithm will always be represented with a particular color – random search with orange, random walk with olive, greedy with blue, steepest with red, simulated annealing with cyan, and tabu search with magenta. This is also illustrated in Figure 1

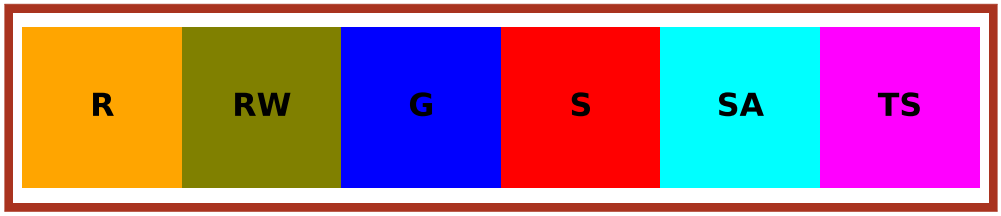


Figure 1: Colors used to represent different search types in all subsequent plot: random search, random walk, greedy, steepest, simulated annealing, tabu search

## 3.1 Quality

The quality measure was defined as the ratio of the cost at the global optimum, and the cost of the solution (see Equation (3)). This means that the best possible value is 1.0, while the worst approaches 0.

$$\text{Solution quality} = \frac{\text{Optimal cost}}{\text{Solution cost}} \tag{3}$$

This goes against the proposition of calculating quality as the distance from the optimum, but I find the term quality to refer to something desired in large values rather than small. Such formulation of the metric also makes it "naturally normalized" between problem instances, allowing for easier comparisons. The results can be observed in Figure 2.

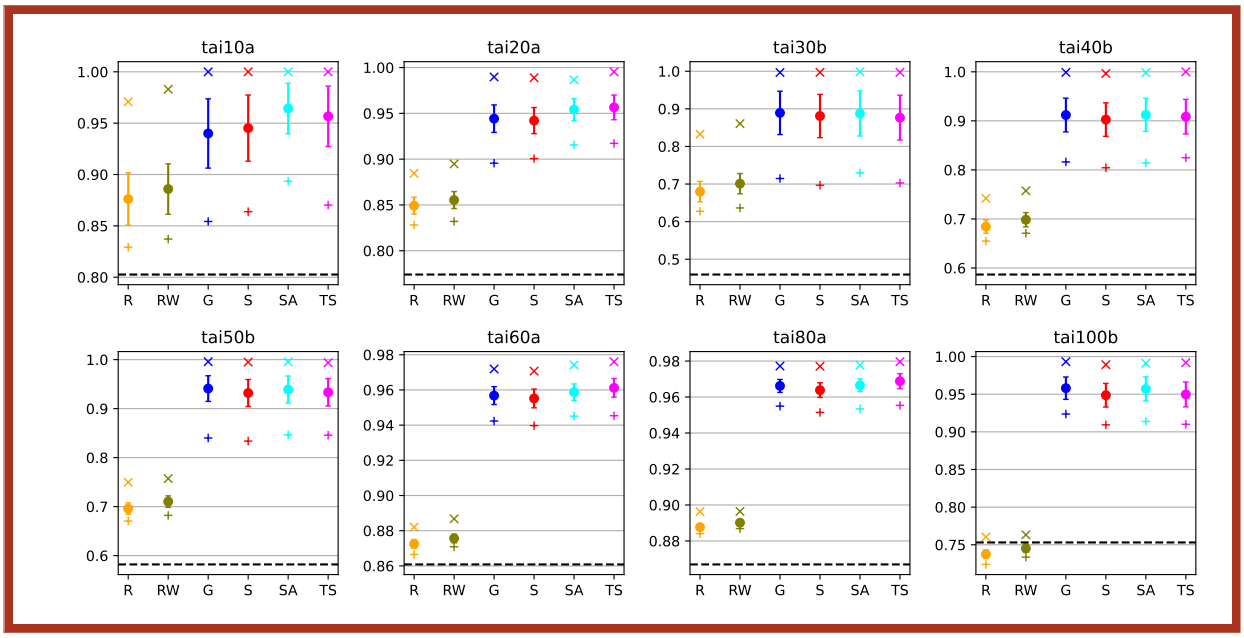


Figure 2: Min, Mean, and Max qualities for algorithm runs on all instances. The scales must be different to accommodate the heuristic solution (quality indicated by the dashed line) Colors used: random search, random walk, greedy, steepest, simulated annealing, tabu search

As expected, local search methods obtain solutions of significantly better quality than random approaches. The difference becomes more apparent as the size of the instance increases. Greedy consistently finds solutions with better quality than steepest, but not by much. The range of quality values does not appear to depend just on instance size. All methods produce significantly more concentrated results for

`tai60a` and `tai80a`. Simulated annealing and tabu search did not improve the solution quality by much, except for a few cases. SA did particularly well on `tai10a` while TS excelled on `tai60a`. Both of the new methods had noticeably higher minimum and average qualities on `tai20a`. Other than these cases, the qualities of their solutions were comparable with their parent methods. The quality of the heuristic solution is consistently very poor – worse than those obtained by random methods. There is no trend in those quality values (for a clearer visualization, see Figure 9). Do note, however, that the heuristic method obtained its solution very fast, which will be discussed in Section 3.2. This also does not mean that the heuristic solutions are worse than random – just that within time limits based on the running time of greedy and steepest, random methods can find solutions better than the heuristic. An illustration of this can be seen in Figure 25.

## 3.2 Running time

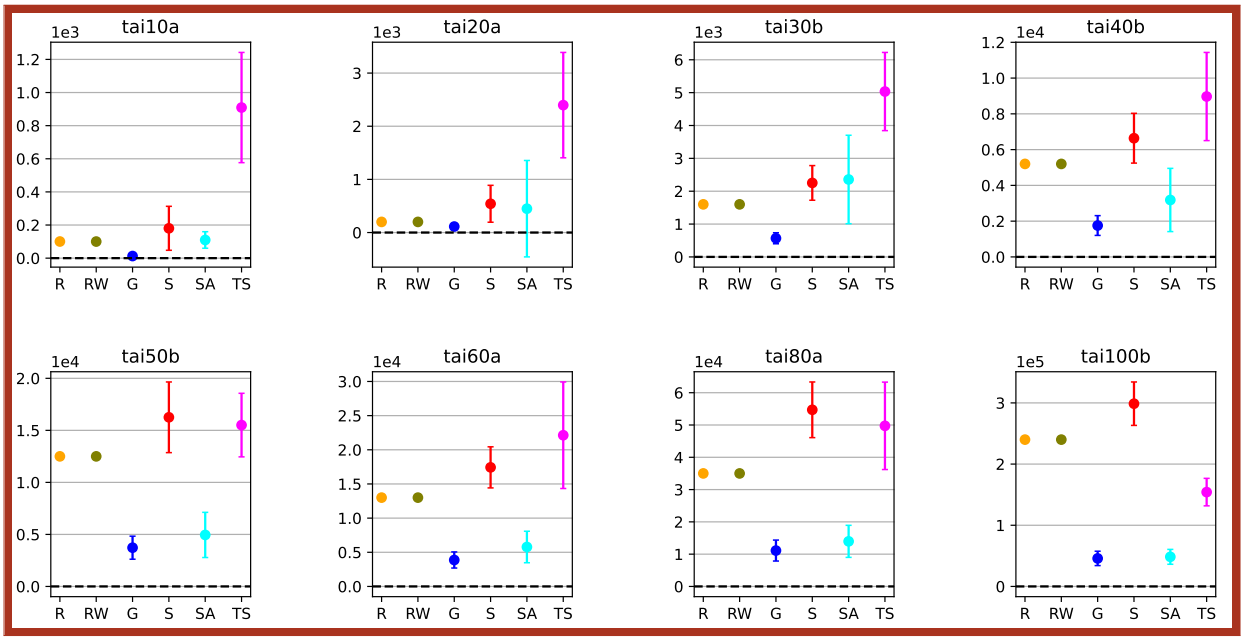


Figure 3: Running time for all algorithms, in microseconds. Mind the scaling factors above the subplots

Figure 3 shows the average running time for all methods. Steepest is shown to be slower than greedy on all considered instances. The running times of random search and random walk are highly consistent, matching the specified limits (see Table 1). The high standard deviations for steepest can be explained by its extreme maximum values of running time (shown in Figure 21). The cause for steepest's long running times will be discussed in Section 3.5. The running time of simulated annealing is consistently slightly longer than that of greedy, with sizable standard deviations. Both of these can be explained by the additional random element introduced to the algorithm over the default greedy. Accepting worse solutions, which allows for a more thorough exploration of the solution space, naturally translates to increasing the chance of needing to spend more time before a local optimum is found (the few exceptionally long runs of SA can be seen in Figure 20). On the other hand, tabu search is more complicated. It is by far the slowest on small instances, but as instance size increases, it becomes faster relative to its peers (in the end, its times are firmly between those of simulated annealing and random methods). This is due to using a fixed number of elite candidate moves, regardless of instance size (see Section 1.2.2 for an explanation of the mechanism). In small instances, the overhead resulting from the tabu-exclusive operations plays a major role. But as instances grow, focusing on only 10 candidates gives an increasing advantage in speed, reducing the need to traverse the entirety of large neighborhoods less often.

Note how fast the heuristic approach is – it requires its separate plot, shown in Figure 4. The time to create a heuristic solution appears to be growing linearly, but due to how it is implemented, the trend would be expected to follow $\mathcal{O}(n^2)$, because all elements of the matrices specifying the instance must be

added to calculate the row sums, which then have to be sorted.

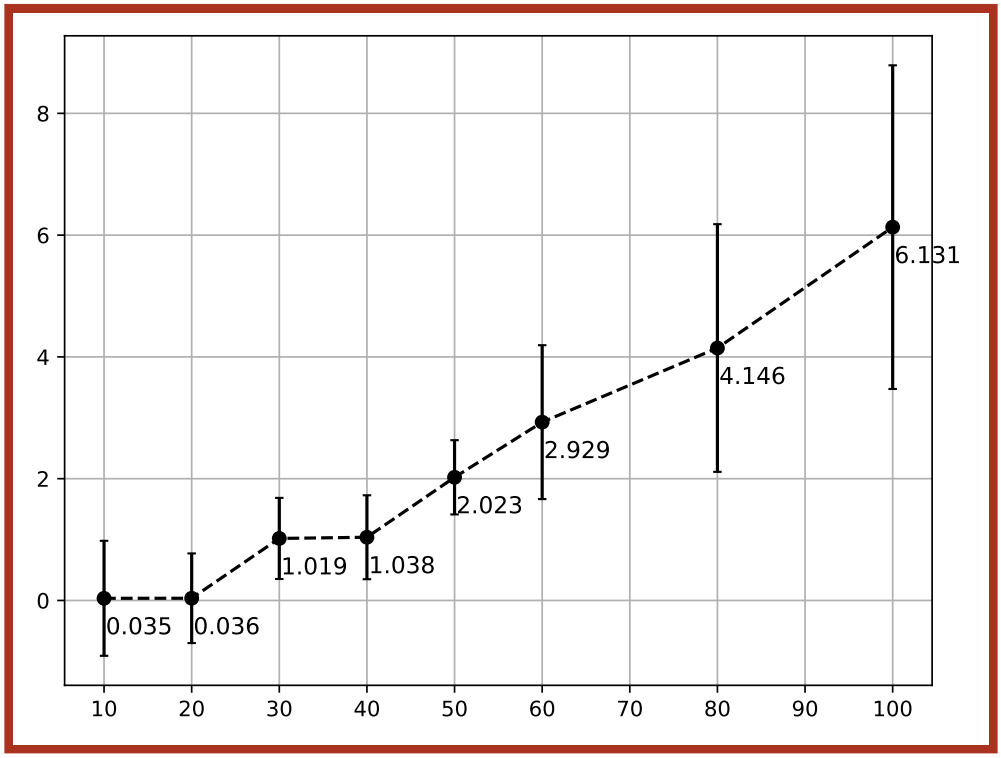


Figure 4: Average running times of the heuristic algorithm, in microseconds. Based on 10000 runs on each instance

### 3.3 Efficiency

The first idea for a measure of efficiency was to divide the quality (defined in Equation (3) as the ratio of the optimal cost to the cost of a solution) by the number of evaluated solutions. However, this led to very small, uninterpretable values, which differed wildly between instances. To mitigate this, a "normalization factor" equal to the neighborhood size of the instance was introduced, by which the previous value is multiplied. This gives the final formula as shown in Equation (4).

$$\text{Efficiency} = \frac{\text{Quality} \cdot \text{Neighborhood size}}{\text{Number of evaluated solutions}} = \frac{\text{Optimal cost} \cdot \text{Neighborhood size}}{\text{Solution cost} \cdot \text{Number of evaluated solutions}} \tag{4}$$

This definition makes it easy to understand what algorithm would obtain an efficiency of 1.0 - one that finds the global optimum while evaluating as many solutions as there are in one neighborhood. This is not the maximum value, but any values close to 1.0 are unlikely to be obtained for non-trivial instances, even less so exceeding it. Note that neither greedy nor steepest can ever exceed 1.0 because before they stop, they must search through at least one full neighborhood. This boundary was the deciding factor in adopting this measure as default.

The results of applying this efficiency measure are presented in Figure 5. The plot highlights one issue of this metric — it can reward inefficient algorithms. Ones that evaluate a relatively low number of solutions, and stumble across good ones by chance, as is the case with random search. Random search is deemed the most efficient, and by a significant margin, for all instances except the smallest `tai10a`. To see how few solutions are evaluated by random search, examine Figure 11. Steepest is found to be no better than random walk. The impact of the number of evaluated solutions is far too large compared to that of solution quality, in terms of which we would expect (and can confirm, e.g. by examining Figure 2) steepest to be significantly better than random walk. This issue could be alleviated by exponentiating the fractions $\frac{\text{Neighborhood size}}{\text{Number of evaluated solutions}}$ or $\frac{\text{Optimal cost}}{\text{Solution cost}}$ to some powers, reflecting if more focus is to be paid to solution quality, or how few evaluations the algorithm performs.

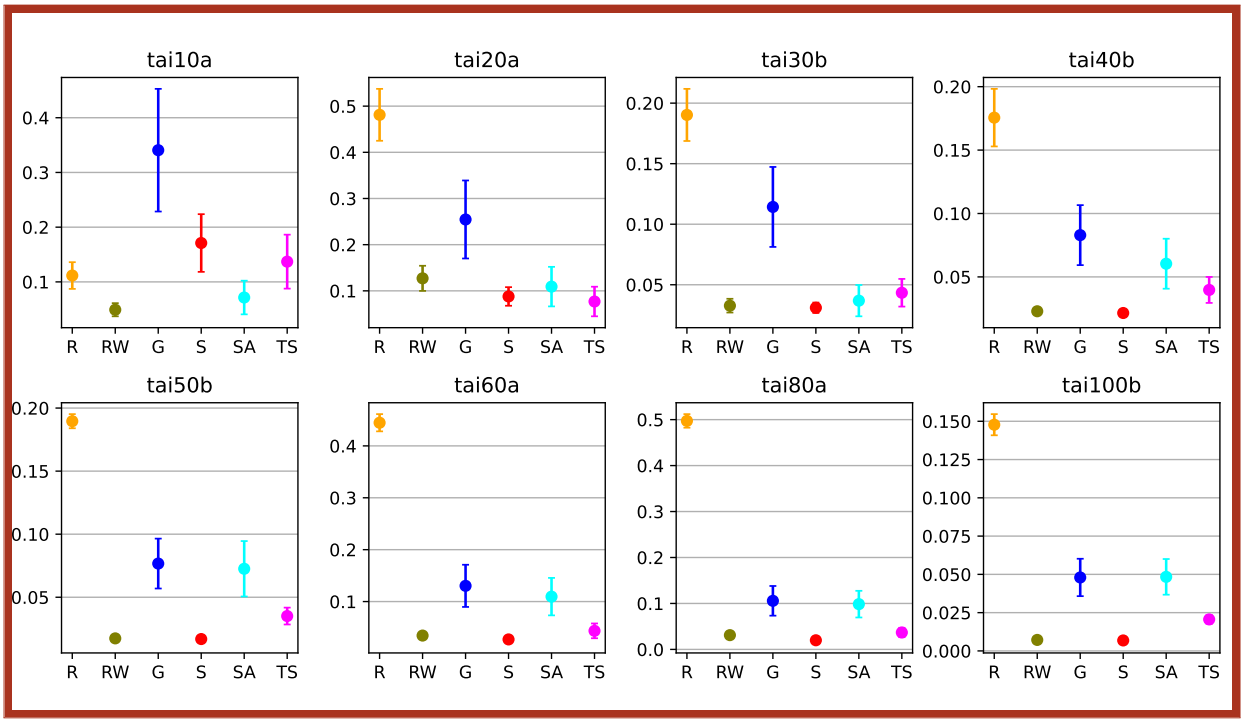


Figure 5: Average efficiency values, using the number of evaluated solutions

A different way of handling this is to replace the *Number of evaluated solutions* term with the number of algorithm steps. The new formula is presented in Equation (5).

$$\text{Efficiency}_{\text{steps}} = \frac{\text{Quality} \cdot \text{Neighborhood size}}{\text{Number of algorithm steps}} = \frac{\text{Optimal cost} \cdot \text{Neighborhood size}}{\text{Solution cost} \cdot \text{Number of algorithm steps}} \tag{5}$$

Results obtained with this formula are presented in Figure 6. This penalizes random algorithms, as they change their current solution as many times, as they evaluate, while more sophisticated search algorithms move to other solutions more carefully, after making a number of evaluations. However, this comes at a loss of interpretability – there is no obvious candidate to replace *Neighborhood size*. A good replacement would be somehow connected to the number of solutions a "good" algorithm is supposed to move to. Choosing $n!$ would usually lead to enormous efficiency values, even for poor algorithms.

These efficiency values are more in line with expectations. Steepest being the best is unsurprising, with its exhaustive search within the neighborhood of any current solutions leading to every change of current solution bringing the maximum decrease in solution cost – what one might consider the very definition of efficiency. Random approaches are left far behind greedy and steepest, with their efficiency values being unaffected by the change in the formula.

One might be interested in a third formula that considers the algorithm's running time, rather than the operations it performs. While it might be subject to some noise due to the operation of computers, and can not be reliably normalized between instances, that is the metric that a user of the algorithm can experience and corresponds to practical constraints more. This third formula is presented in Equation (6). In this case, the running time is given as the number of microseconds the execution took.

$$\text{Efficiency}_{\text{running time}} = \frac{\text{Quality} \cdot \text{Neighborhood size}}{\text{Running time}} = \frac{\text{Optimal cost} \cdot \text{Neighborhood size}}{\text{Solution cost} \cdot \text{Running time}} \tag{6}$$

The results of applying this metric are presented in Figure 7. Greedy comes out as the best by far. This is because of its combination of high quality (significantly better than random) and short running time (significantly shorter than steepest). Random methods and steepest obtain very similar efficiency values. Random methods have an advantage in running time (because the limit they are given is set to be

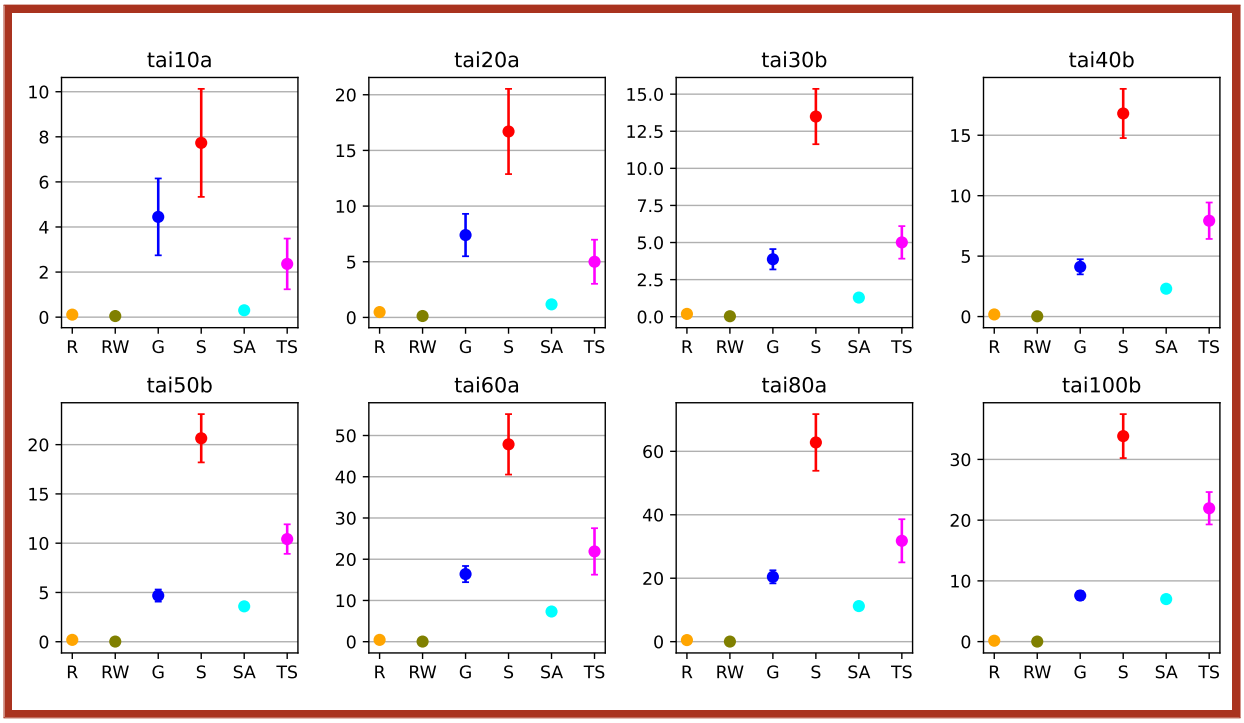


Figure 6: Average efficiency values computed using the number of algorithm steps (changes of current solution)

smaller than the time for steepest), but steepest finds better solutions. These factors seem to somewhat equalize within the metric. The relationships in all types of efficiency metrics between simulated annealing and tabu search reflect those of greedy and steepest. Despite introducing different extension mechanisms, according to the efficiency definitions discussed here, neither of SA and TS can match the vanilla versions. The potential increase in quality comes at a price.

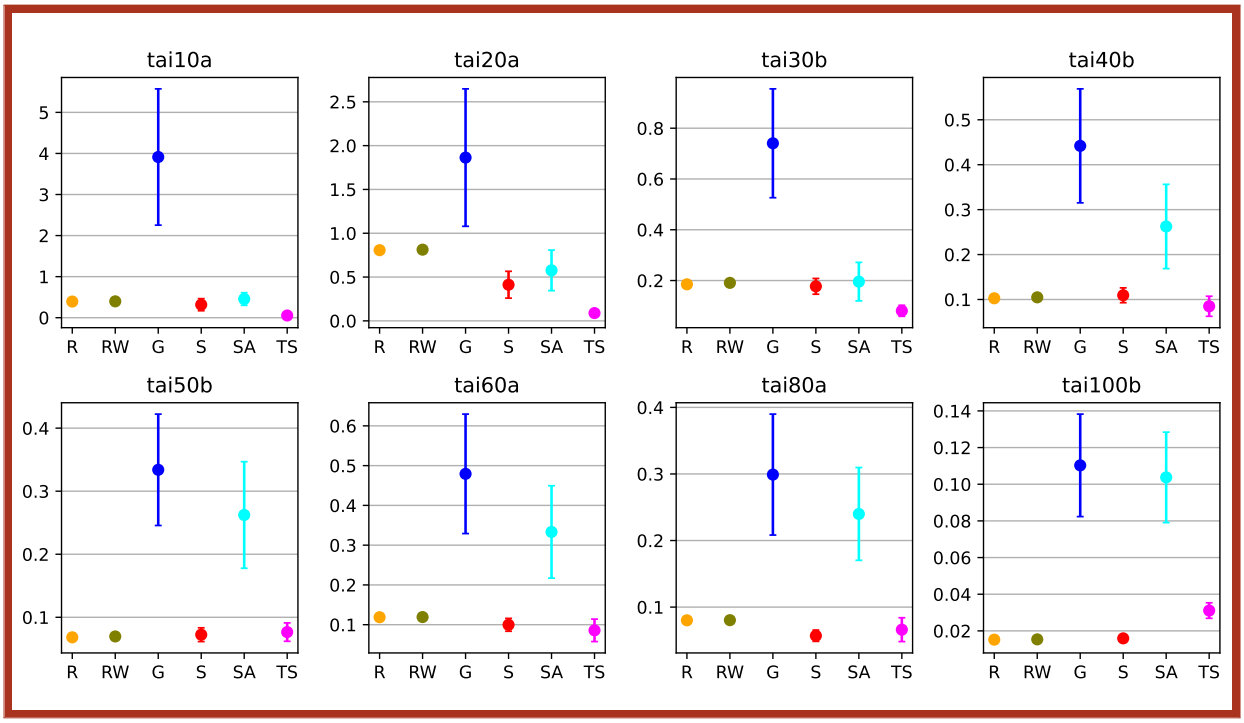


Figure 7: Average efficiency values computed using the running time. Keep in mind that steepest had more time than both random methods, which had more time than greedy

### 3.3.1 Efficiency of the heurisitc

The efficiency of the heuristic solution was not presented in these plots or discussed in the text. This is because it is not a search algorithm that can be repeated many times, or explore the solution space. It simply generates a fixed solution given the instance parameters, guided by principles deemed worthy by the programmer's intuition. This allows for generating the solution very quickly (as shown in Section 3.2). This lack of exploration makes the efficiency measure uninterpretable when the same formulas involving *Neighborhood size*. It is also unclear if the number of visited or evaluated solutions should be set to 0 or 1. I have thus decided to estimate the efficiency of the heuristic algorithm using the formula described in Equation (7). The efficiency values are very inconsistent. To try to explain them, Figure 9 was prepared – it compares the quality of the heuristic solution to the amount of time to generate said solution. While there is a clear dependency between the size of the instance and the amount of time it took to generate the solution, solution quality is highly unpredictable, leading to chaotic efficiency values.

$$\text{Efficiency}_{\text{heuristic running time}} = \frac{\text{Quality}}{\text{Running time}} = \frac{\text{Optimal cost}}{\text{Solution cost} \cdot \text{Running time}} \tag{7}$$

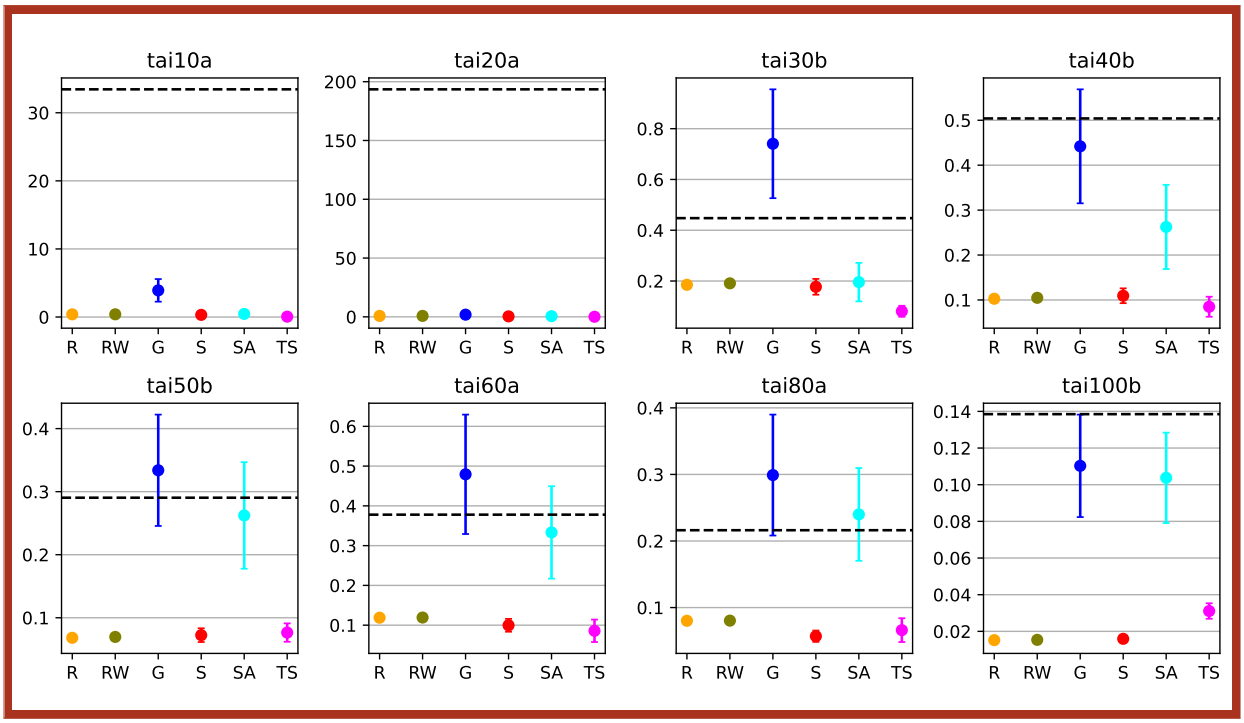


Figure 8: Efficiency of algorithms based on running time, including the efficiency of the heuristic algorithm (black dashed line)

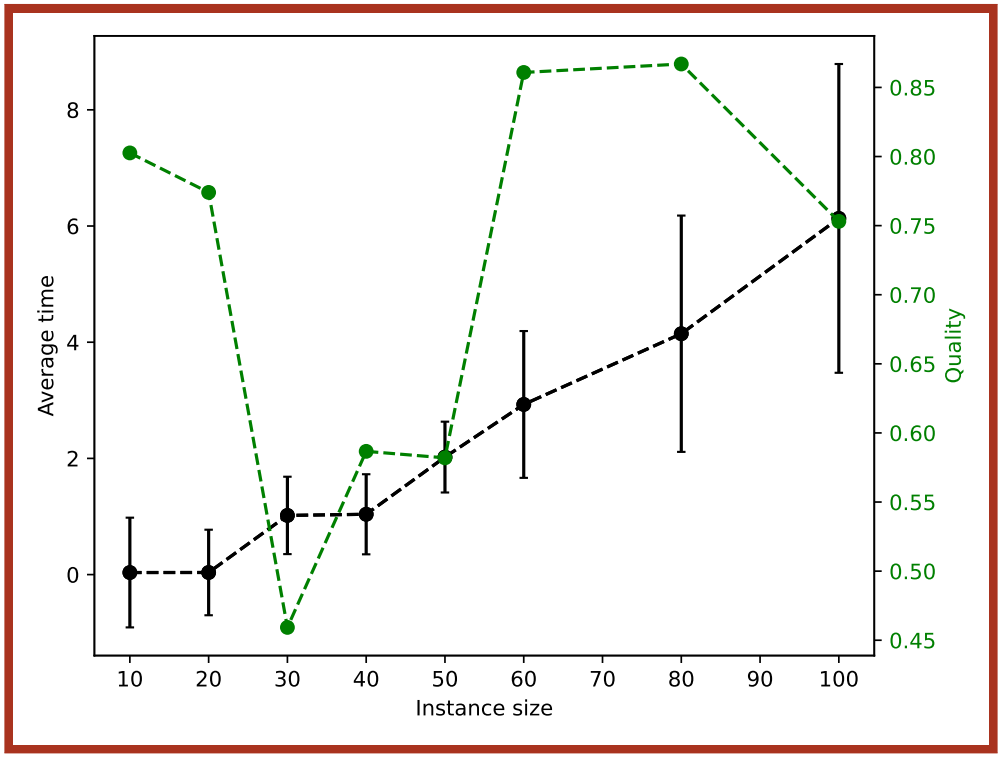


Figure 9: Running time and quality of the heuristic algorithm. Running time based on 10000 repeats per instance

### 3.4 Greedy, Steepest – Average number of steps

Steepest makes significantly fewer changes to its current solution than greedy, and this discrepancy increases with the size of the instance (in general – see Figure 10). This is because the size of the neighborhood, which steepest must search through scales quadratically with instance size. There is no such clear dependency between how many solutions greedy must evaluate before it "takes a step" and instance size. Steepest is also more stable in terms of how many steps it takes, with significantly smaller standard deviations of the statistic. Instances `tai60a` and `tai80a` cause the algorithms to buck the trend of increasing the number of steps with the size of the instance – this is likely due to the specifics of these instances, perhaps they have many "densely-spaced" local optima, which can be reached in few steps. It is likely to be an unusual scenario – the pretty clear difference between the two algorithms in their willingness to change the current solution discussed earlier in this paragraph would suggest so. A plot comparing all four methods based on local search can be viewed in Figure 24.

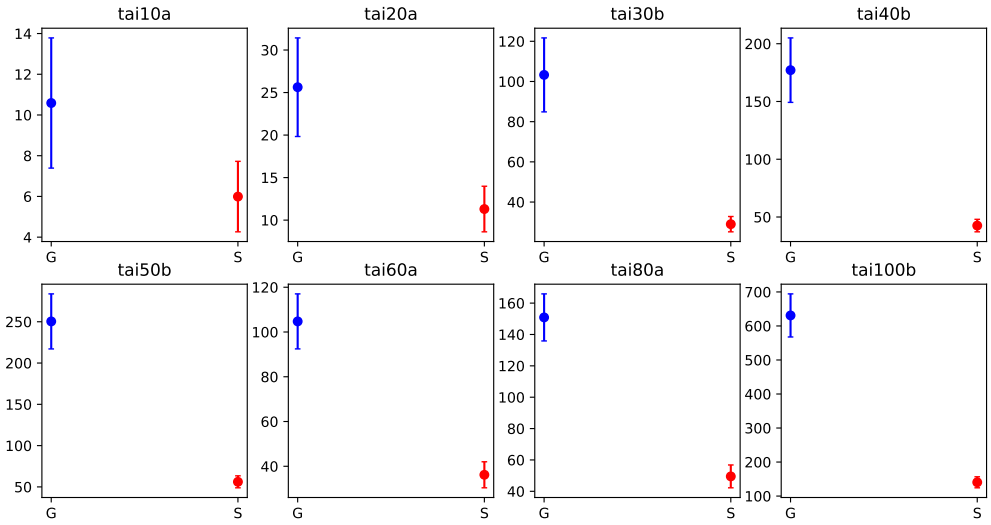


Figure 10: Average number of algorithm steps (changes of current solution)

### 3.5 Average number of evaluated solutions

The measurements obtained in the experiments are shown in Figure 11. Random search evaluates the least solutions because it has to always evaluate the entire solution from scratch, without using deltas. The comparatively larger number for `tai10a` is due to it being given a long time for the search, and the instance being small enough for greedy and steepest to reliably identify local optima in fewer evaluations. Additionally, small instances might introduce more relative overhead due to all of the auxiliary operations that steepest and greedy must perform, causing the number of evaluated instances to be less correlated with running time for small instances. Other than that, on average, steepest evaluates more solutions than random walk – this is just because the time limit for random techniques was set to be somewhere between the runtime of greedy and steepest, meaning that RW had less time available. If it had the same time limit as steepest, it would be expected to make more evaluations, due to having to perform fewer auxiliary operations than steepest. This large number of evaluations that steepest must perform is the primary reason for its long running time. Greedy performs fewer evaluations than random walk – it eagerly moves to other solutions, quickly ending up in some local optimum, without diligently exploring its neighborhood. For this metric, `tai60a` and `tai80a` seem to defy the trend again, this time with the impact of instance size on the number of evaluations. In this case, they do so less dramatically than in the case of the number of steps, especially for `tai80a`. I would conjecture that this has to do with larger

neighborhood sizes, which are taken into account when counting the number of evaluated solutions instead of the number of steps.

Simulated annealing consistently evaluates more solutions than greedy – this makes intuitive sense, as one would expect allowing for deterioration to make the route to local optima longer. What I did not expect however, is that this discrepancy decreases with instance size. For small instances, SA performs even more evaluations than steepest (and changes its current solution an enormous number of times – see Figure 24). This might be because in small instances local optima are expected to be easier to reach, and steepest can converge on them while SA is still trying to explore the solution space. For small instances, tabu performs more evaluations than steepest, but for larger instances – significantly fewer than its parent method. I would again attribute this to the fixed size of the candidate list.

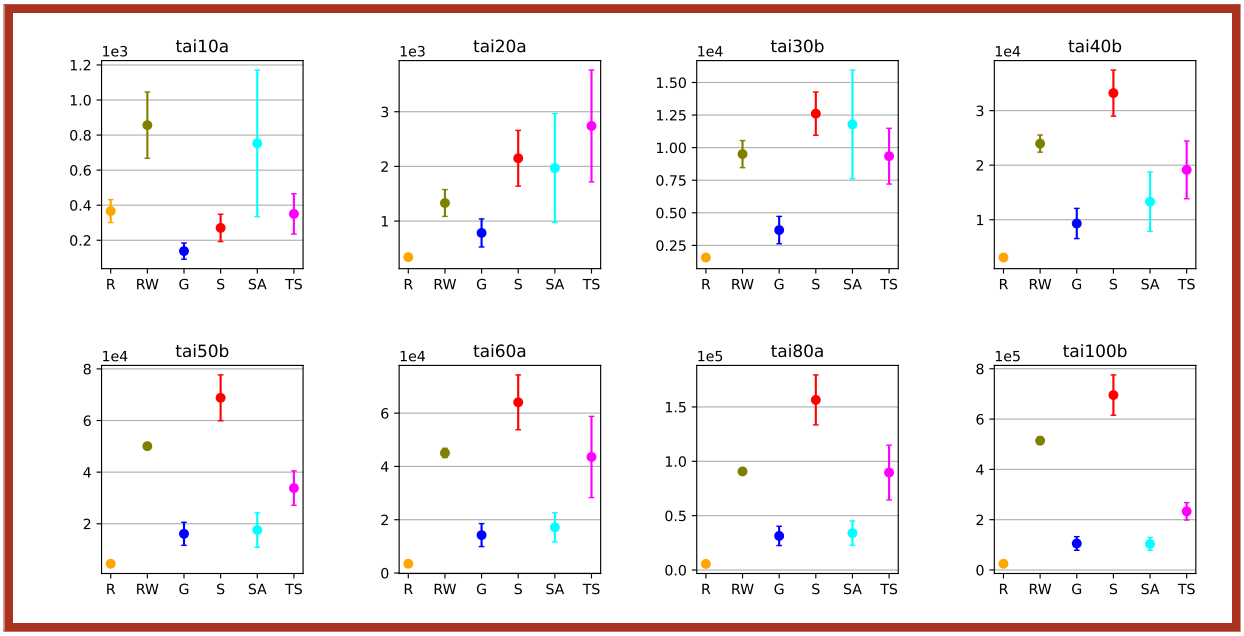


Figure 11: Average number of solutions evaluated by the model, partial evaluations (move delta calculations) included. Mind the scaling factors above the subplots

# 4 Greedy, Steepest - initial solution vs. final solution

No selection of instances was made, as all appear to behave in a very similar way (see Figure 12). Both greedy and steepest show no correlation between the quality of the starting and final solution, with the strongest correlation coefficient being $-0.104$ (greedy on `tai30b`). Even when the same experiment is performed for the largest instance in the entire dataset, `tai256c`, the correlation gets weaker, not stronger (see Figure 13). I would expect the impact of the initial solution to be larger for bigger instances, as they could have more local optima (different than global), and a search algorithm would explore a relatively smaller section of the solution space away from the start. The surprisingly narrow range of quality values for this large instance will be discussed more in Section 6 covering the correlation between solution quality and similarity to the global optimum. While the correlation remains nowhere to be seen `tai10a` and `tai30b` have a slightly different distribution of points – one can notice a sort of ceiling forming at quality 1.0, rather than there just being a few disjoint examples in this region of the y-axis. This suggests that for those instances, solutions of a cost close to the global optimum were found more consistently than in others. For `tai100b`, it can be observed particularly clearly, that greedy outperforms steepest – this is in line with the quality measurements presented in Figure 2. This is not however something that seems to be dependent strictly on instance size, as no such transition is to be found in plots for smaller instances, and Figure 13, based on the largest instance in the dataset does not support this.

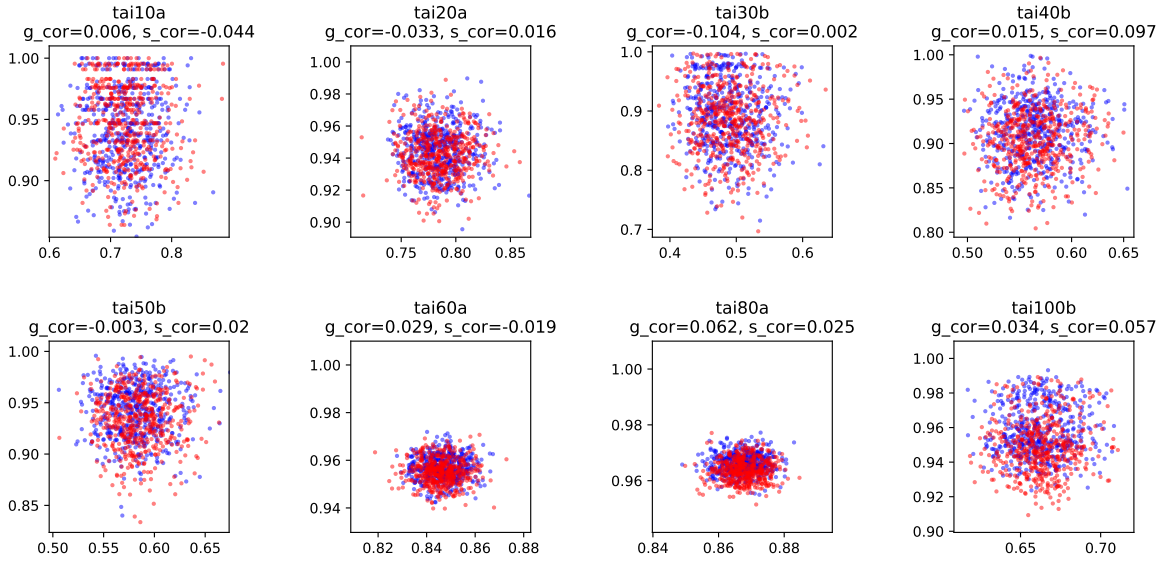


Figure 12: Quality of the initial solution (on the x-axis) vs the quality of the local optimum (y-axis) Colors: greedy, steepest

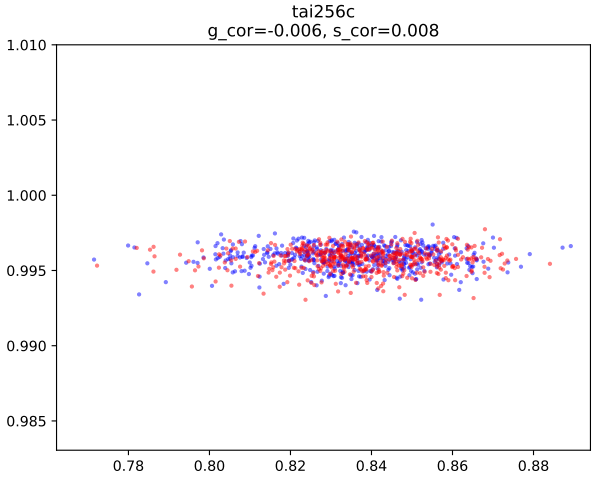


Figure 13: Quality of the initial solution vs the quality of the local optimum for tai256c Colors: greedy, steepest

# 5 Effectiveness of restarts

Repeats of both greedy and steepest lead to finding significantly better solutions in the long run. The instance that seems to benefit from these the most is `tai80a`. Repeating the algorithms gives more benefit for larger instances, which makes sense, giving their larger solution space – on `tai10a` there seems to be little benefit to performing more than 200 runs, while for the larger instances, improvements are found well after 400th restart. An interesting observation is that with larger instance sizes, greedy appears to identify better solutions, especially in terms of average cost. Figure 14, Figure 15, and Figure 16 show the experimental results of solution costs within some number of first restarts (on the x-axis).

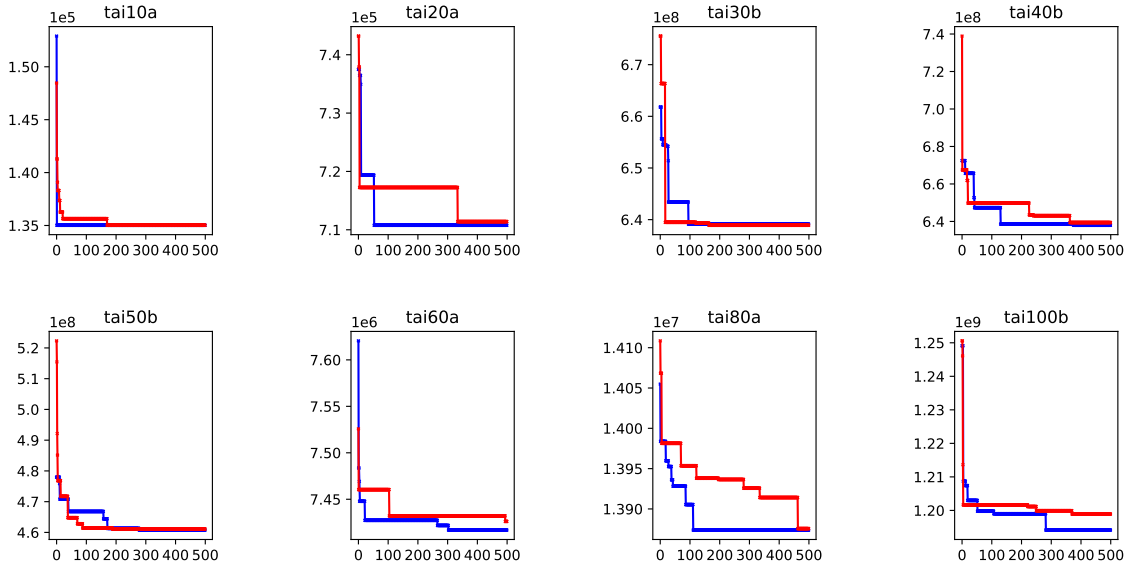


Figure 14: Cost of the best solution found within a given number of restarts Colors: greedy, steepest

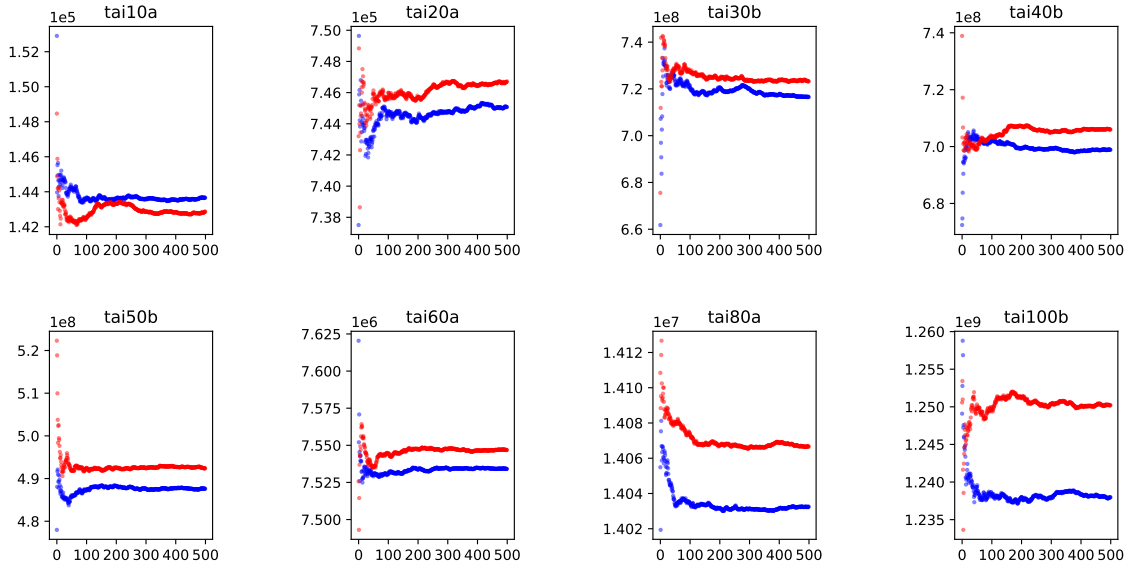


Figure 15: Average solution cost within a given number of restarts Colors: greedy, steepest

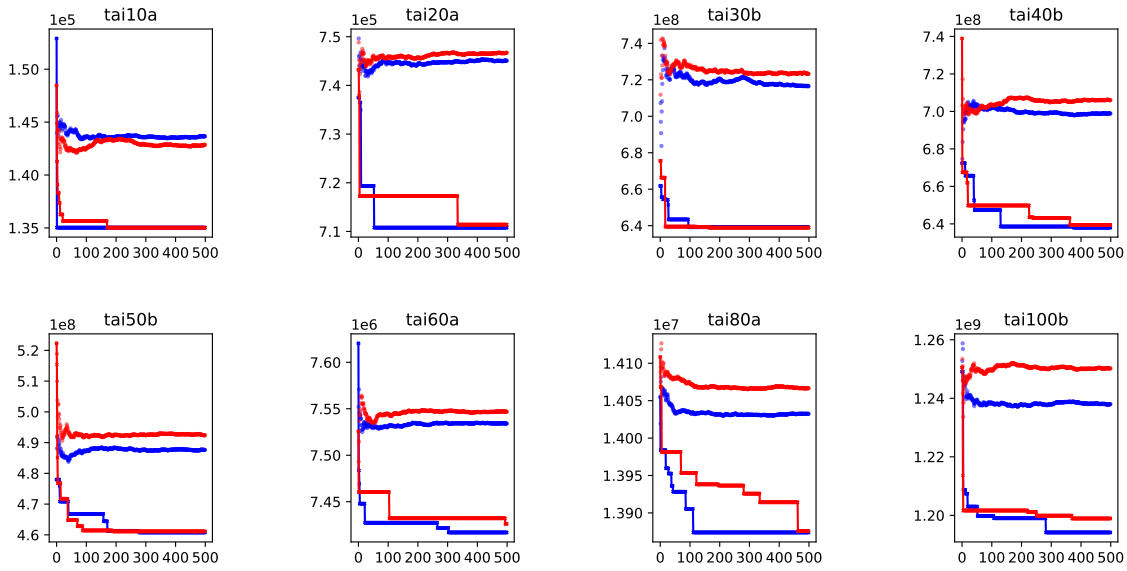


Figure 16: Average and best costs together Colors: greedy, steepest

# 6 Similarity of local optima to global

The similarity was measured by counting the number of places in the permutation that had the same element at the same location and dividing by the permutation length. I find it quite surprising how dissimilar many local optima are to the global one – similarity of 0 is the most common value (see Figure 17). For some instances, it can be seen that solutions of higher quality tend to be similar to the global optimum (e.g. `tai30b` or `tai100b`), but this doesn't hold for others. Again, there seems to be no significant difference between greedy and steepest in this regard. Repeating the experiment for `tai256c` (Figure 18) suggests that larger instances do not cause good local optima to be more similar to the global optimum. Perhaps that might be due to there being more solutions that are very close in cost to the global optimum, which can be found by the search, while the region of the global optimum remains elusive. Most local optima are completely different (the only instance in which any of the 1000 runs give some shared solutions is `tai10a`), and most of the time the algorithms can't find solutions with more than a few shared permutation elements. This might suggest that QAP (or at least the examined instances) can have many vastly different local optima, which can be nothing like the global optimum. This relates to the narrow range of solution qualities for `tai256c` – even though the search is long, and some very good solutions are found, the size of the solution space makes it so that local optima can be vastly different than the global while maintaining their quality. All of this together indicates that QAP is not globally convex.

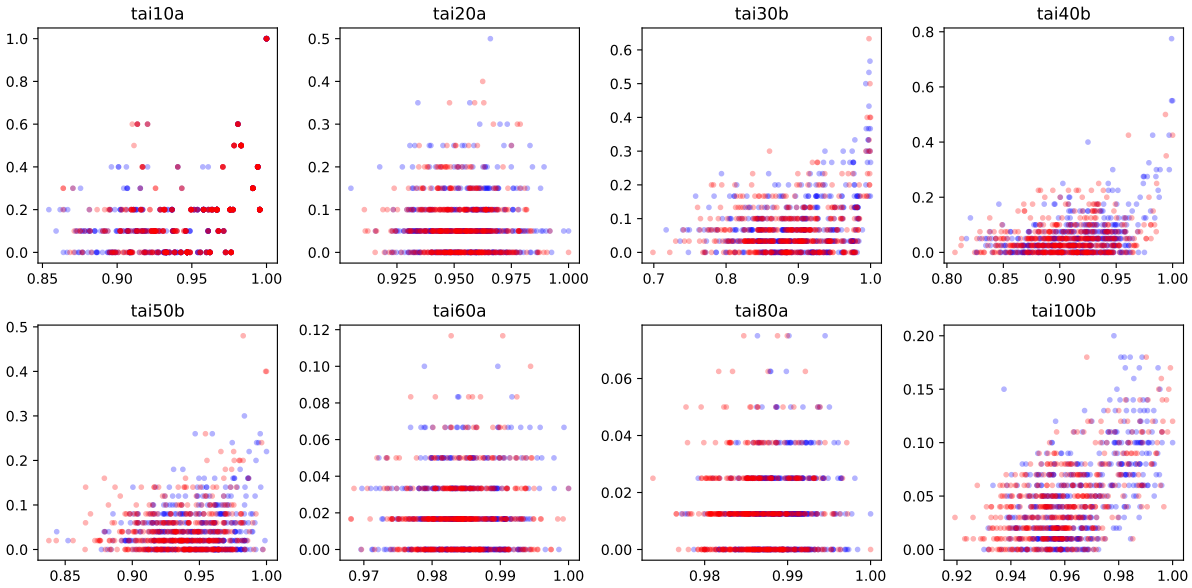


Figure 17: x-axis: solution quality, y-axis: similarity to the global optimum Colors: greedy, steepest

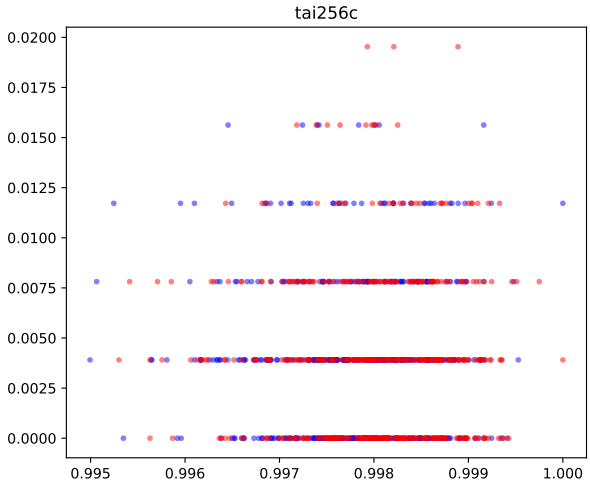


Figure 18: Solution quality vs similarity to global optimum for tai256c Colors: greedy, steepest

# 7 Similarity of local optima to not worse ones

This section was modified, because there was a mistake in the code generating the plot in report 1 – solutions were compared to not-better ones, instead of not-worse.

Figure 19 shows the relationship between the quality of local optima identified by greedy and steepest (on the x-axis) and the average similarity of these optima to all other non-worse (with cost that is equal or smaller) local optima. All plots share the same interesting characteristic – the spread of similarity value decreases with the increase in quality. Note that the similarity values correspond to having between 0 and 2 shared permutation elements. This supports the idea that QAP has multiple very diverse local optima. Since the y-axis values are computed based on averages of lists of varying sizes, there can be more granularity to the results than just three options. This might also explain the concentration of similarity values in solutions of good quality – they have fewer not-worse ones to compare themselves to. This means that there are fewer possible average similarity values for them, and, with there being fewer good solutions, even those limited thresholds they could occupy, are left vacant. Taking all of this into account, the points are probably arranged in a shape akin to the normal distribution bell, arising from representing averages of a random distribution.

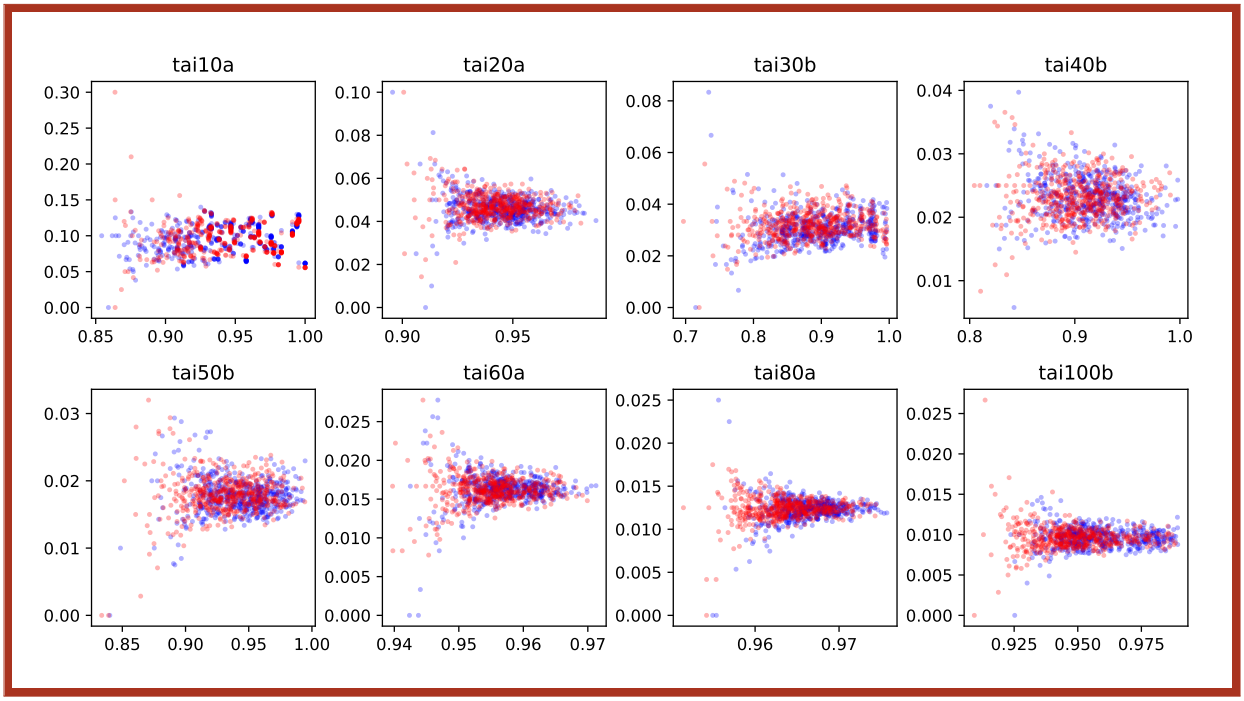


Figure 19: x-axis: quality of a local optimum, y-axis: average similarity to all non-worse local optima
Colors: greedy, steepest

# 8 Quality across algorithm evaluations

So far within this report, the algorithms were analyzed by looking at the best solutions they obtained. This section will address this, by taking a peek at the changes in solution qualities over the course of algorithm runs. Figure 20 shows the changes in qualities of the best solutions found by the methods during every run (y-axis) and when this incumbent solution was found in terms of the number of solutions evaluated by the procedures (x-axis). What surprised me at first was that greedy looked more steep than steepest. There is nothing wrong with the implementation, or the plots. The phenomenon occurs because while the humps in quality made by greedy are smaller, they occur in a quicker succession, meaning that over the same number of evaluated solutions, it has every right to get to better ones than steepest. One might think

of it as steepest spending a long time evaluating many bad solutions within its current neighborhood to be sure that it will choose the best one, while greedy follows the path of least resistance. Random search is barely visible, despite the points being drawn in random order – this is because it evaluates relatively few solutions, and they are not of high quality, meaning that all points corresponding to the permutations it identified are located in the most crowded region of the plot. Some vertical stripes can be observed in the trends for steepest and tabu search. These correspond to the intervals of length equal to the neighborhood size. Simulated annealing and greedy exhibit very similar behavior – despite SA visiting more solutions, its willingness to accept deterioration means that improvements might be visited just as often as in the baseline approach. Some extraordinarily long runs of SA can be seen, especially for smaller instances. These might have resulted from randomly accepting a sizeable deterioration, and then the algorithm having to fight its way back to a local optimum. Tabu search appears to be firmly located between steepest and the two greedy-based methods.

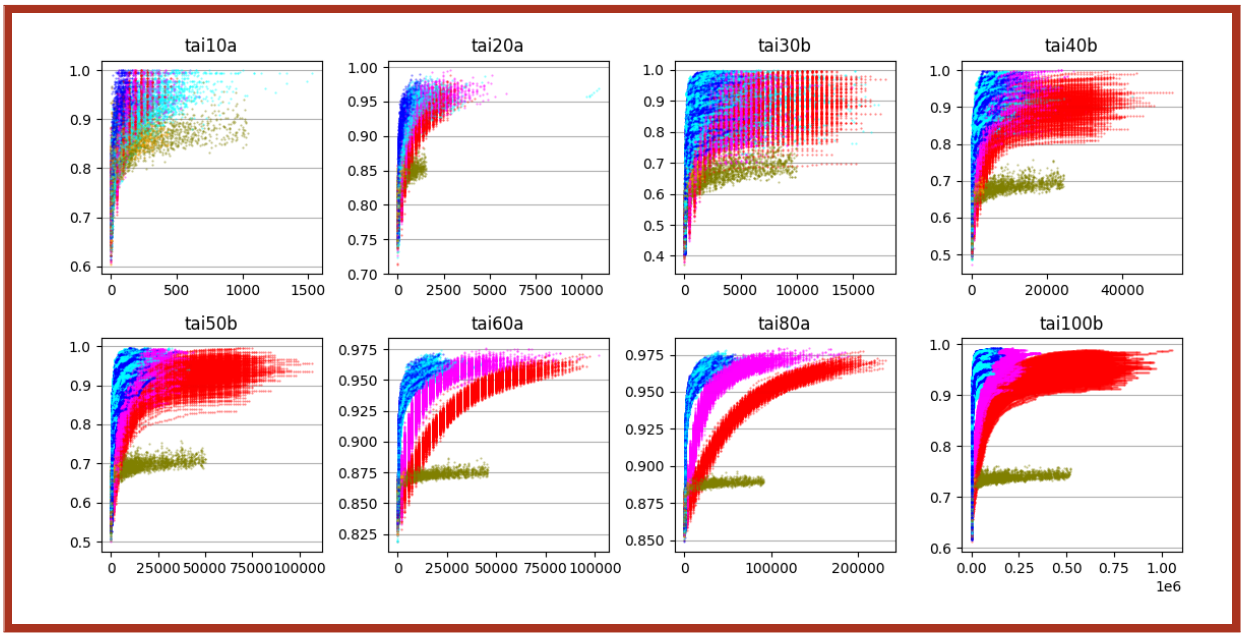


Figure 20: Quality of the incumbent best solution found in each of the 500 runs of the methods against the number of evaluations until the solution was found

# 9 Conclusions

The below list discusses the characteristics of the algorithms that can be inferred from the conducted experiments.

- Random search can give surprisingly good results, especially when taking into account the number of evaluated solutions (see the discussion about efficiency in Section 3.3).
- Random walk is the best of the tested algorithms in terms of exploring the largest part of the solution space (it evaluates the most solutions in a given amount of time – see Section 3.5).
- Even poor heuristics can be valuable if time is of the essence (Section 3.1, Section 3.3.1, Figure 25)
- Greedy tends to converge to a local optimum within fewer evaluations, which translates to less time – it is a better tool than steepest at identifying multiple local optima quickly (Section 3.2 about running time and Section 3.5 about the number of evaluated solutions, also Section 8).
- Often time larger instances make the differences between algorithms more clear. Not everything about an algorithm's performance can be predicted from instance size, as the numerous cases of `tai60a` and `tai80a` breaking the trend established by instances of size 10, 20, 30, 40, 50, (for example in Section 3.4 when comparing the number of steps of greedy and steepest). However, larger instances highlight the advantages of greedy and steepest over random, showcase the differences in the number of evaluations between them, or their "cumulative average solution cost" (Section 3.5 and Section 5 respectively).
- Greedy and steepest react essentially the same to the quality of the initial solution (Section 4) and the similarity of the local optima to the global one is very similar (Section 6).
- Simulated annealing and tabu search preserve many characteristics of the methods they extend (happens many times, different aspects are shown in Section 3).
- Simulated annealing can improve the quality of the identified local optima, but it comes at a considerable cost to the computational resources required (Section 3.1, Section 3.3).

One thing that I found to be an overarching conclusion is that **QAP is difficult**.

- There seems to be little to no correlation between how similar solutions are, and how good they are – low similarities between local and global optima (see Section 6).
- Having a better starting solution does not appear to improve the search (Section 4).
- Heuristic approaches give solutions of poor quality and are unpredictable (Section 3.3.1)
- While not QAP-specific, the large solution space of $n!$ certainly makes itself known, with how quickly the running times of local search algorithms increased.

# 10 Difficulties

- Implementing the algorithms in a new language was surprisingly easy.
- Designing a good heuristic. I tried two: first made sure that rows of matrix $\mathbf{A}$ with the largest sums were multiplied by the rows of matrix $\mathbf{B}$ with the smallest sum and vice versa; second admitted the smallest product into the solution, then excluded all products involving any of these two rows from the pool of available products, and then repeated until a solution was created. The first one (based on row sums), to my surprise, proved to be better. Both gave very bad results (worse than random – see Figure 23 where the quality of the better heuristic solution is highlighted). Setting a single element in the permutation has a much larger impact on the cost than in TSP, that might be why building a solution iteratively, as guided by a heuristic gives such poor results.
- Drawing good conclusions – many experiments showed that there seems to be no dependency between something, or some instances would invalidate some conclusion.
- Tuning algorithm parameters while making sure that they respect the specification given in the lectures

# 11 Final remarks

In some points I showed plots for all instances rather than selecting just a few – I did that, because I could not justify selecting any interesting representatives, and I think I managed to be efficient enough with the available space.
I think that it could be possible to run the experiments for all instances. It would make the analysis of the results more challenging. Some instances do not have optimal solutions provided. But that could remove the selection bias that comes with picking just eight instances, all created by the same author.
I am aware of the time discrepancy in running times between simulated annealing and tabu search - but since the discrepancy changed with instance size, I decided not to make the process of tuning hyperparameters any longer than it already was for myself.
I know the discussion of efficiency for SA and TS was very brief, but I didn't think there was too much interesting to say there, and this section, and the report, were already quite sizeable.
The history/efficiency in running time plot had to be a raster graphic, sorry - a pdf was over 25MB! Using lines might have made identifying individual runs possible, but it made the overall legibility of an already crowded graph even worse.

# 12 Additional plots

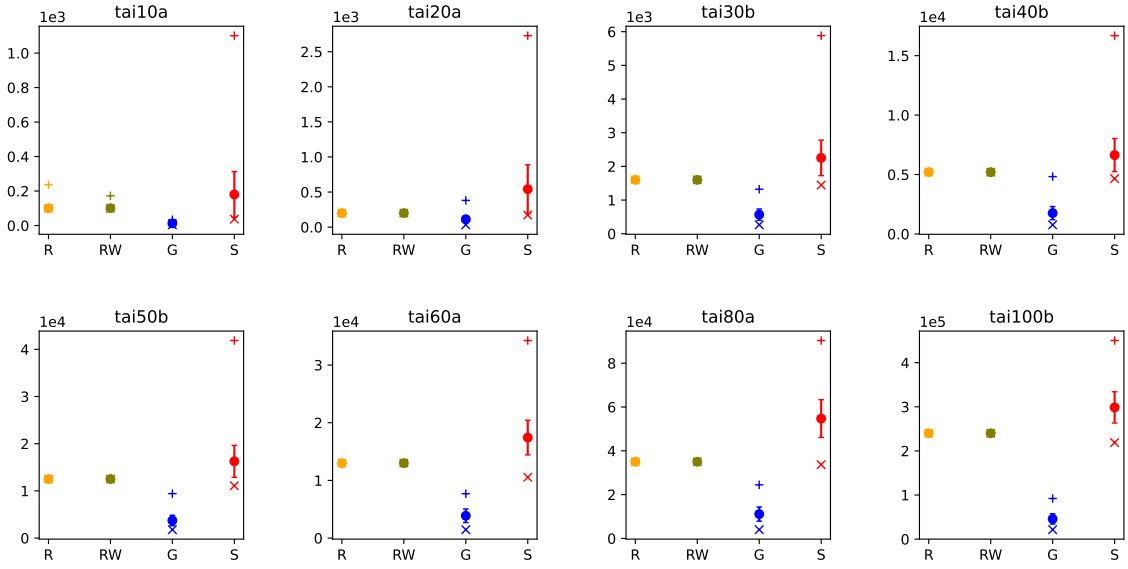


Figure 21: Min, mean, max running times. Shows the large maxima for steepest

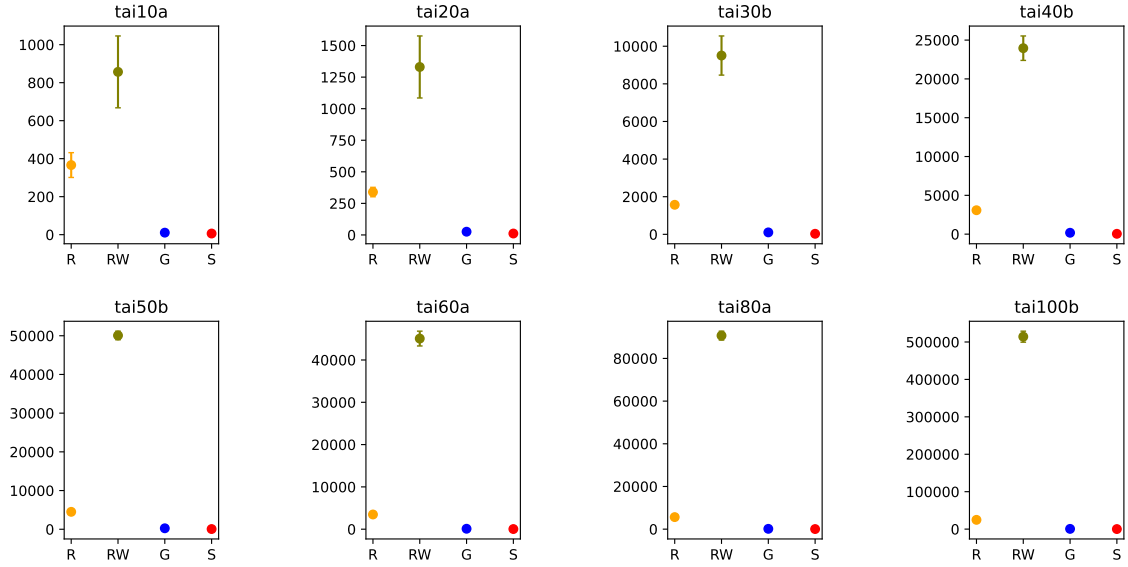


Figure 22: Average number of algorithm steps (changes in current solution) for all algorithms

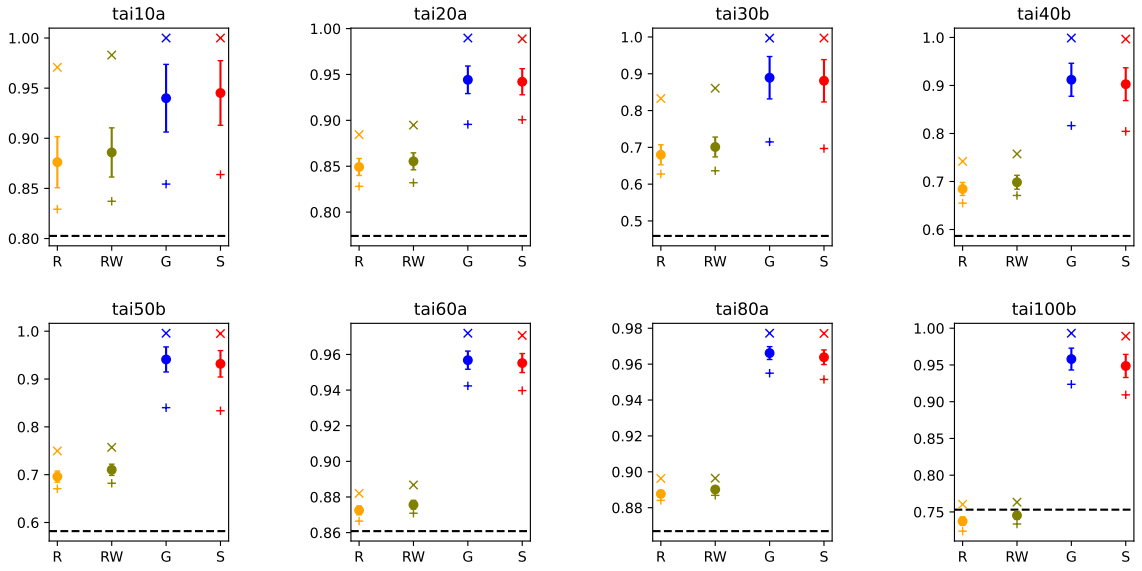


Figure 23: Quality plot with the horizontal line indicating the quality of the heuristic solution

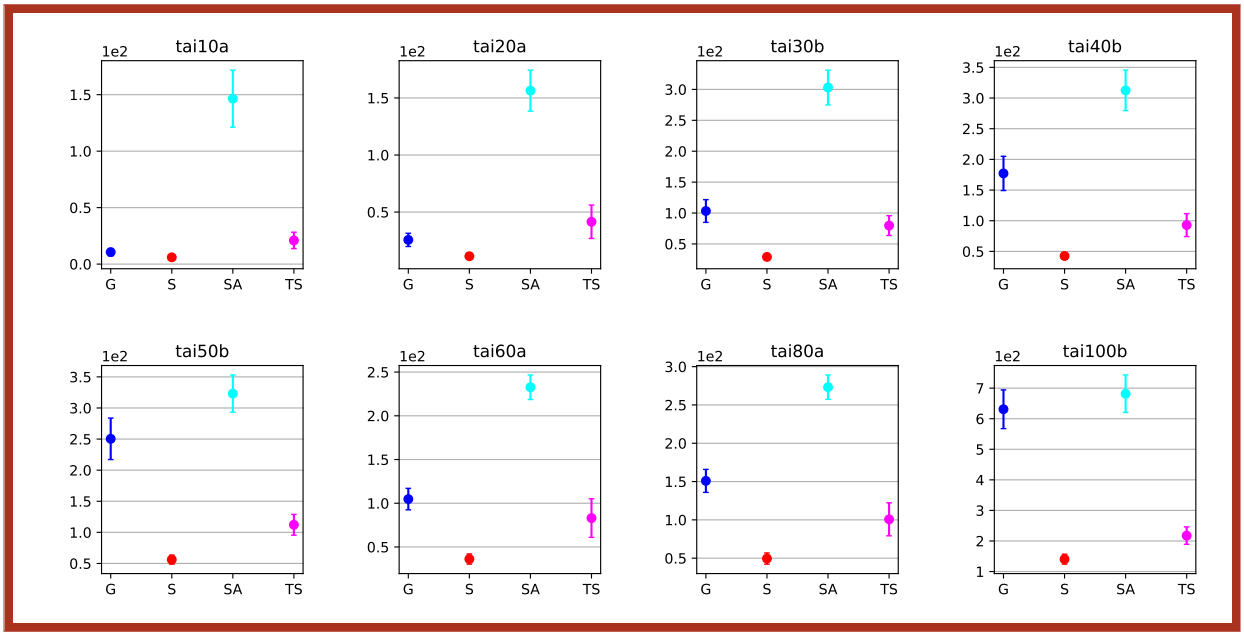


Figure 24: Average number of algorithm steps (changes of the current solution) for all algorithms based on local search

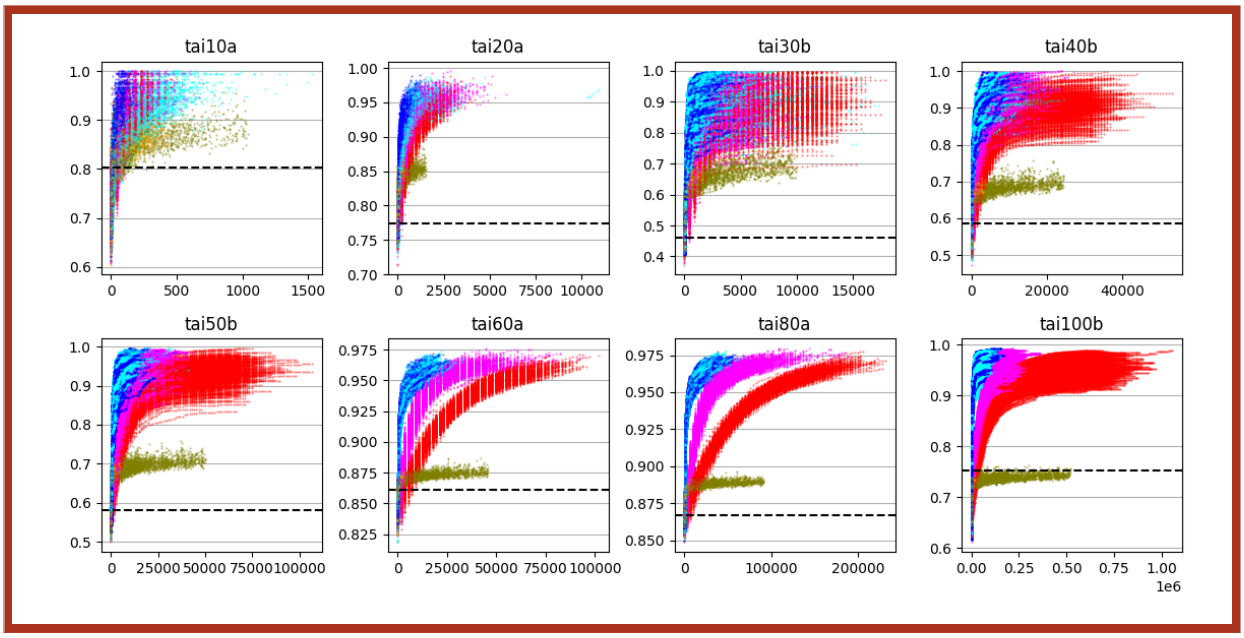


Figure 25: Solution quality across runs of different methods over the number of evaluations. The quality of the heuristic solution is marked with a black, dashed line

# References


[1] Rainer Ernst Burkard, Eranda Çela, P.M. Pardalos, and L.S. Pitsoulis. *The quadratic assignment problem*, volume 2, pages 241–337. Kluwer Academic Publishers, Netherlands, 1998.

[2] The Rust Community. *Profiles*, chapter 3.5. The Rust Foundation, 2024. URL: `https://doc.rust-lang.org/cargo/reference/profiles.html?highlight=--release#release`.

[3] Steve Klabnik, Carol Nichols, and contributors from the Rust Community. *Array types*, chapter 10.1.6. The Rust Foundation, 2024. URL: `https://doc.rust-lang.org/reference/types/array.html`.

[4] Tjalling C Koopmans and Martin Beckmann. Assignment problems and the location of economic activities. *Econometrica: journal of the Econometric Society*, pages 53–76, 1957.

[5] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, jul 1976. `doi:10.1145/321958.321975`.