Solving the asymmetric traveling salesman problem using a combination of simulated annealing and parallel tempering algorithm

Łukasz Grabarski, Magdalena Jeczeń, Karolina Mączka, Mateusz Nizwantowski, Marta Szuwarska

June 2023

Abstract

The Asymmetric Traveling Salesman Problem (ATSP) poses a challenging optimization task to find the shortest route for a salesman to visit cities with varying travel costs. This paper presents the Parallel Tempering Simulated Annealing (PTSA) algorithm as a promising solution for ATSP. Metaheuristic algorithms, including Ant Colony Optimization, Simulated Annealing, and Lin-Kernighan, were explored, and PTSA emerged as the most effective option. We improve the algorithm by incorporating modifications such as initializing random and heuristic solutions, employing shuffle and swap transitions, and using multithreading. Performance testing demonstrates PTSA's superiority over alternative algorithms. The results show that PTSA provides satisfactory solutions for small problems and acceptable outcomes for larger instances. Additionally, long-term testing highlights the importance of extended runs for higher-dimensional problems.

Keywords: Simulated Annealing, Parallel Tempering, Asymmetric Traveling Salesman Problem

Contents

1 Introduction

The asymmetric traveling salesman problem (ATSP) is a classic combinatorial optimization problem that aims to find the shortest route for a salesman to visit a set of cities, where the travel costs between cities are not necessarily symmetric. This problem is known to be NP-hard, making it challenging to find an optimal solution in a reasonable amount of time.
We aim to provide valuable insights into addressing complex optimization problems as is the ATSP problem, offering a promising approach to find near-optimal solutions in a reasonable computational time.

Researchers have explored various techniques to solve the traveling salesman problem efficiently. We turned our attention to metaheuristic algorithms. We explored solutions such as the Ant Colony Optimization (ACO), Simulated Annealing, and Lin-Kernighan algorithms. We decided against ACO because of its poor performance compared to others. Lin-Kernighan, on the other hand, relies on the assumption of symmetric distances between cities. The Simulated Annealing approach sounded promising so we decided to delve into it and we found another likely-looking algorithm, Parallel Tempering Simulated Annealing (PTSA). We decided on implementing it with slight modifications. [3]

During our case study classes [2], we performed a performance test in the form of The Race and compared the results with two other teams. It is worth noticing that our implementation of PTSA gave the best results out of the three algorithms included in the test, which also consisted of the classic Simulated Annealing and Discrete Spider Monkey Optimization.

2 Algorithm

2.1 Simulated Annealing

Simulated Annealing is a widely used method in optimization problems. The SA approach treats the objective function as the energy of a thermodynamic system. It introduces a variable temperature into the objective function, allowing the system to escape local energy minima more easily at higher temperatures. By gradually decreasing the temperature, the system moves closer to thermodynamic equilibrium using the Metropolis-Hastings transition rules. Although SA improves the probability of crossing barriers at high temperatures, the waiting time in deep

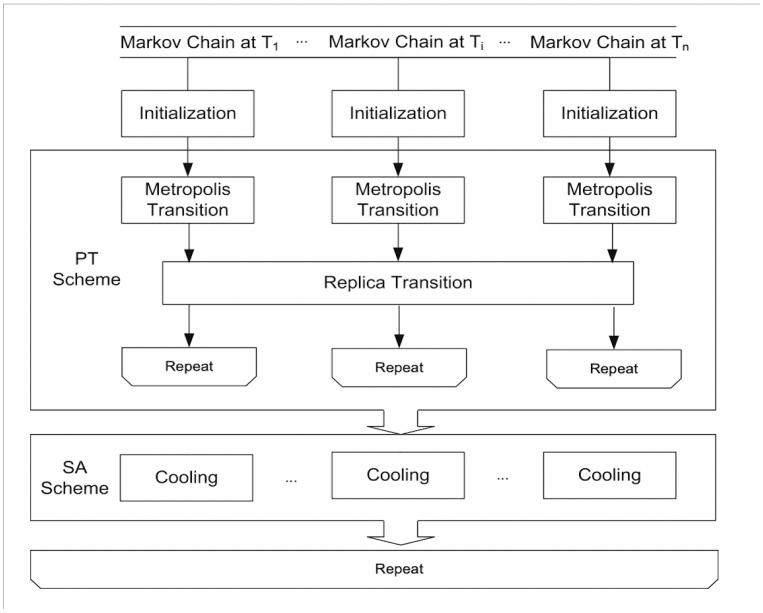
local minima increases as the temperature decreases, resulting in the need for careful tuning of the cooling rate.

2.2 Parallel tempering

Parallel Tempering is a Monte Carlo accelerated algorithm that is well-equipped to handle the challenges posed by complex energy landscapes with multiple local minima. It enhances the probability of crossing high barriers by incorporating a random walk in the temperature space during sampling. PT also exchanges system replicas at different temperature levels. A composite system is created with those subsystems (each has its own different temperature), allowing higher temperature subsystems to traverse barriers while lower temperature subsystems explore local minima. Unlike SA, PT allows the temperature of a single subsystem to increase or decrease. The consequence is an improved mixing rate of Markov Chain Monte Carlo samplers at different temperatures. PT reduces the relaxation time of the overall system, leading to fast global minimum convergence.

2.3 Parallel tempering - Simulated Annealing

The hybrid PT-SA method combines two approaches, PT and SA. The PT component aims to minimize the relaxation time of subsystems towards the equilibrium state within the composite system. The SA component lowers the temperature at each level, leading the composite system towards the target temperature and at the same time, it ensures that the composite system remains in a state close to thermodynamic equilibrium.



2.4 Metropolis transition

Firstly the paths are randomly shuffled. After that, the new solution is accepted with a certain probability. Below we present a function that determines whether it will be accepted.

Listing 1: Pseudocode for accepting a new solution

```
function acceptance:
  acceptance_probability =
  np.exp(-(new_solution_length - solution_length) / temperature)
  return uniform(0, 1) < min(1.0, acceptance_probability)
```

2.5 Replica transition

After that, the temperatures are swapped between different subsystems. As a consequence, the relaxation time of this algorithm is significantly reduced. We opted for a constant probability of swapping temperatures, which we later determined during the parameters' tuning.

Listing 2: Pseudocode for replica transition.

```
function replica_transition:
  if uniform(0, 1) < swap_states_probability:
```

```
3       temperatures[first_index], temperatures[second_index]
4       = temperatures[second_index], temperatures[first_index]
5     return temperatures
```

2.6 Cooling

Later on, each temperature is decreased by a certain amount but cannot fall below a minimum value to ensure that the algorithm continues to explore the search space.

Listing 3: Pseudocode for cooling.

```
1 function cooling:
2   new_temperature = cooling_rate * temperature
3   return max(new_temperature, min_temperature)
```

3 Modifications

3.1 Starting with heuristic solutions

While initializing initial solutions, some of the solutions we create randomly and the rest using the nearest neighbor algorithm [6].

In the nearest neighbor algorithm, we start from a city we will call the 'starting city'. Then we go to the city with the least distance from our starting city. Analogically, we find the next nearest city. We repeat the process until all cities are visited [9].

In the function better_nearest_neighbor_initial_solution(), we perform the nearest neighbor algorithm for every city as starting city. Then we sort them by solution length and save the best 10% of solutions.

Listing 4: Pseudocode for performing nearest neighbor algorithm for every city.

```
1 function better_nearest_neighbor_initial_solution:
2   heuristic_solutions = []
3   for every city as starting_city do:
4     unvisited = list of cities
5     current_city = starting_city
6     path = [current_city]
7     unvisited.remove(current_city)
```

```
8       while unvisited:
9         nearest_neighbor = nearest city to current city
10        unvisited.remove(nearest_neighbor)
11        path.append(nearest_neighbor)
12        current_city = nearest_neighbor
13      heuristic_solutions.append(path)
14    heuristic_solutions.sort()
15    return heuristic_solutions[: math.floor(len(distance_matrix) * 0.1)]
```

Whereas initializing a suboptimal solution, we take a random solution from those saved heuristic solutions.

Listing 5: Pseudocode for initializing heuristic solutions.

```
1 nearest_neighbor_solution =
2 better_nearest_neighbor_initial_solution(distance_matrix)
3 for n times:
4     if eps < probability_of_heuristic:
5       initial_solutions.append(random.choice(nearest_neighbor_solution))
```

We also compared the solutions obtained with the heuristic alone to the best known solutions. As we can see below, the heuristic on its own finds not that bad solutions.

Table 1: Best solutions' lengths obtained with the heuristic on its own compared with best known solutions' lengths.

Name	**Heuristic solution length**	**Best known solution length**
br17	39	39
ftv33	1590	1286
ftv35	1667	1473
ftv38	1759	1530
p43	5684	5620
ftv44	1844	1613
ftv47	2173	1776
ry48p	15575	14422
ft53	8584	6905
ftv55	1948	1608
ftv64	2202	1839
ft70	41815	38673
ftv70	2287	1950
kro124p	43316	36230
ftv170	3582	2755
rbg323	1702	1326
rbg358	1747	1163
rbg403	3497	2465
rbg443	3858	2720

3.2 Implementing two types of metropolis transition

For metropolis transition, we have implemented two different approaches: shuffle transition and swap transition. Each state is assigned only one of them. Both of those functions also have a given transformation length, which states the length of the path in the solution that will be changed.

3.2.1 Shuffle transition

In the shuffle transition, from a given solution we randomly choose a path of transformation length. We do it by randomly choosing the starting index first and then going to the next cities from the solution until we have a path of the given

length. Since the solutions are cycles, if we get to the 'end' of the solution, we wrap to the first city.

For example, if we have a solution [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] with a transformation length equal to 3 and randomly chosen starting index 9, then we end up with a path [9, 10, 1].

Having chosen a path, we shuffle it and replace the old path in the solution with the new path. In our example, if our new path after shuffling were [1, 10, 9], then our new solution would be [9, 2, 3, 4, 5, 6, 7, 8, 1, 10].

Listing 6: Pseudocode for shuffle transition.

```
function transition_function_shuffle:
  list_to_shuffle = []
  l = length(solution)
  start_index = random index
  repeat
  for i from 0 to transformation_length - 1:
    list_to_shuffle[i] = solution[(start_index + i) mod l]

  shuffle(list_to_shuffle)

  new_solution = copy(solution)
  for i from 0 to transformation_length - 1:
    new_solution[(start_index + i) mod l] = list_to_shuffle[i]

  new_solution_length = cycle_length(new_solution, distance_matrix)
  return new_solution, new_solution_length
```

3.2.2 Swap transition

In the swap transition, from a given sequence we randomly choose two non-overlapping paths of transformation length. We do it by randomly choosing two starting indices first and then going to the next or previous cities from the solution until we have a path of the given length.

By default, we take the next cities, however, if it is not possible, we take the previous cities. Firstly, we check if there are enough cities between the first index and the second index to make a path of transformation length going right from the first index. Secondly, we check if there are enough cities between the second index and the first index on the cycle to make a path of transformation length going right from the second index. If the first condition is not satisfied but the second is, we take the cities before the city with the first index and end up with the first

path ending with the city with the first index. Otherwise, if the first condition is satisfied but the second is not, we take the cities before the city with the second index and end up with the second path ending with the city with the second index.

For example, we are given a solution [1, 2, 3, 4, 5, 6, 7, 8, 9, 10] with a transformation length equal to 3. If we draw starting cities 2 and 6, we end up with paths [2, 3, 4] and [6, 7, 8]. If we draw starting cities 5 and 6, we end up with paths [3, 4, 5] and [6, 7, 8]. If we draw starting cities 1 and 10, we end up with paths [1, 2, 3] and [8, 9, 10].

Having chosen two paths, we swap them in the given solution. In this way, in the first case, we get the new solution [1, 6, 7, 8, 5, 2, 3, 4, 9, 10]. In the second case, we get [1, 2, 6, 7, 8, 3, 4, 5, 9, 10]. Finally, in the third case, we get [8, 9, 10, 4, 5, 6, 7, 1, 2, 3].

Listing 7: Pseudocode for swap transition.

```
function transition_function_swap:
  l = length(solution)

  start_index_first_path = random index
  start_index_second_path = random index

  if (not first_path_can_go_right) and second_path_can_go_right:
    start_index_first_path =
    start_index_first_path - transformation_length + 1

  else if first_path_can_go_right and (not second_path_can_go_right):
    start_index_second_path =
    start_index_second_path - transformation_length + 1

  first_path = path from start_index_first_path
  second_path = path from start_index_second_path

  new_solution = solution.swap(first_path, second_path)
  new_solution_length = cycle_length(new_solution, distance_matrix)

  return new_solution, new_solution_length
```

3.3 Speeding metropolis transition up with multithreading

Metropolis transition is applied to every state separately. This way, updating states are independent of each other. Therefore, applying multithreading here becomes quite an obvious way to speed up the process [4].

We moved updating states, i.e. applying metropolis transition, to a new function, in which we update them and save them to a shared list of solutions with a lock to prevent from reading not saved data. We start threads and join them in the pt_sa() function.

Listing 8: Pseudocode for adding multithreading to metropolis transition.

```
function update_state:
  solution, solution_length = metropolis_transition()
  with lock:
    solutions[state], solutions_lengths[state] = solution, solution_length

function pt_sa:
  ...
  threads = []
    lock = threading.Lock()
    for state in range(n):
      thread = new Thread(update_state)
      thread.start()
      threads.append(thread)
    for thread in threads:
      thread.join()
  ...
```

3.4 Skipping solutions close to the best solution in replica transition

In replica transition, when we come across a solution close to the current best solution, we do not want to change it too much since we are already on the right track. Therefore, if one of the solutions in the states chosen for swapping is close to the best solution, we skip swapping there.

Listing 9: Pseudocode for skipping solutions close to the best solution in replica transition.

```
first_solution_close_to_best =
```

```
2  solutions_length[first_index] <= closeness * best_solution_length
3    second_solution_close_to_best =
4    solutions_length[second_index] <= closeness * best_solution_length
5
6    if first_solution_close_to_best or second_solution_close_to_best:
7      pass
8    else:
9      ...
```

3.5 Running algorithm multiple times during 5 minutes period

While testing our algorithm with different durations of execution, we noticed that a longer duration does not necessarily mean better solutions. Therefore, we set out to test whether it would be better in some cases to run the algorithm from the very beginning a few short times rather than one long time.

In order to do that, we ran the algorithm for every problem and saved the best found solutions after 0.25, 0.5, 1, 1.5, 2, 2.5 and 3 minutes. We also compared our solutions with the best known solutions and saved how much time was needed to get a solution worse by 1%, 2%, 3%, 5%, 10%, 15%, 20%, 30% and 50% than best known solutions. However, if the time exceeded 186 seconds, we skipped it. If we want to divide our entire time of execution (5 minutes), then we need to divide it into at least two iterations (2.5 minutes each). Hence, if the best solution still changes after 2.5 minutes, we should run the algorithm only once for the entire time (5 minutes). We considered 186 seconds to be enough over 2.5 minutes to ensure checking that. We saved all that information into a data frame.

Table 2: The lengths of the best solutions found after a given time.

Name	**0.25 min**	**0.5 min**	**1 min**	**1.5 min**	**2 min**	**2.5 min**	**3 min**
br17	39	39	39	39	39	39	39
ftv33	1382	1382	1382	1382	1382	1382	1382
ftv35	1556	1555	1555	1555	1555	1555	1555
ftv38	1618	1618	1618	1618	1618	1618	1618
p43	5642	5642	5642	5642	5642	5642	5642
ftv44	1768	1768	1768	1768	1768	1768	1768
ftv47	2041	2041	2041	2041	2041	2041	2041
ry48p	15256	15256	15256	15256	15256	15256	15256
ft53	7937	7937	7937	7937	7937	7937	7937
ftv55	1758	1758	1758	1758	1758	1758	1758
ftv64	2042	1974	1974	1974	1974	1974	1974
ft70	40708	40708	40708	40708	40708	40708	40708
ftv70	2217	2216	2216	2216	2216	2216	2216
kro124p	41235	40212	39757	39757	39757	39757	39757
ftv170	3492	3430	3430	3428	3428	3428	3428
rbg323	1689	1654	1632	1602	1573	1566	1542
rbg358	1691	1626	1572	1534	1525	1494	1482
rbg403	3229	3127	2942	2847	2826	2787	2753
rbg443	3557	3466	3298	3231	3188	3147	3120

Table 3: The time needed to get solutions worse by a given percentage.

Name	**50%**	**30%**	**20%**	**15%**	**10%**	**5%**	**3%**	**2%**	**1%**
br17	0.002	0.003	0.006	0.006	0.006	0.007	0.007	1.348	1.348
ftv33	0.004	0.004	0.097	0.188	0.313	NA	NA	NA	NA
ftv35	0.004	0.004	0.004	0.005	0.724	NA	NA	NA	NA
ftv38	0.004	0.004	0.005	0.005	0.665	NA	NA	NA	NA
p43	0.005	0.005	0.006	0.006	0.006	0.006	0.007	0.007	0.269
ftv44	0.005	0.006	0.006	0.006	0.279	NA	NA	NA	NA
ftv47	0.006	0.006	0.292	6.948	NA	NA	NA	NA	NA
ry48p	0.006	0.006	0.007	0.007	0.007	NA	NA	NA	NA
ft53	0.007	0.008	0.863	1.688	NA	NA	NA	NA	NA
ftv55	0.008	0.009	1.155	3.897	3.897	NA	NA	NA	NA
ftv64	0.012	0.012	0.013	6.159	17.557	NA	NA	NA	NA
ft70	0.017	0.018	0.018	0.018	0.019	NA	NA	NA	NA
ftv70	0.018	0.018	0.019	3.819	NA	NA	NA	NA	NA
kro124p	0.037	0.038	0.039	1.869	38.743	NA	NA	NA	NA
ftv170	0.156	1.246	NA	NA	NA	NA	NA	NA	NA
rbg323	1.109	1.109	96.622	NA	NA	NA	NA	NA	NA
rbg358	7.057	132.615	NA	NA	NA	NA	NA	NA	NA
rbg403	2.061	18.825	50.190	104.045	NA	NA	NA	NA	NA
rbg443	2.680	19.533	75.796	166.865	NA	NA	NA	NA	NA

In order to compare the results of our test, we made some plots. In the first one, we compared for every problem how much time in seconds was needed to reach a certain best solution deficit ratio[1], meaning the solution worse by a certain percentage than the best known solution for the given problem. As we can see, for problem rbg403, after only a few seconds, the best solution deficit ratio of the solution is about 50%. However, after about 20 seconds, it becomes about 30%, after about 50 seconds, it becomes about 20%, and after about 100 seconds, it becomes 15%. For the problems kro124p, rbg323, rbg358 and rbg443, the best solution deficit ratio also improves even after a longer time. Nevertheless, in other cases, the solutions stop improving after 30 seconds.

[1]The variable name coined by Chat GPT [5].

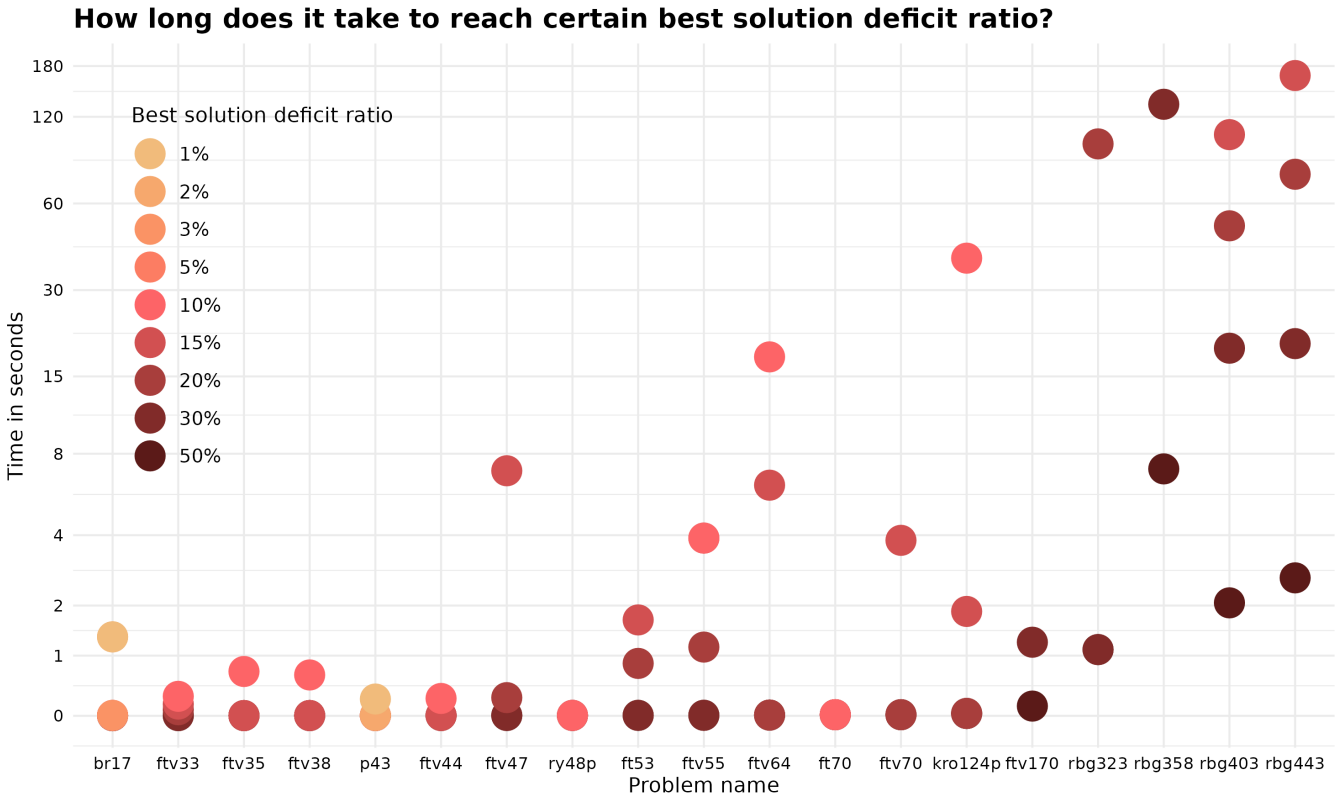


Figure 1: Comparison of the time needed to reach a certain best solution deficit ratio.

In the second plot, we compared for every problem what deficit ratio was reached after certain amounts of time. As shown, in most cases the best deficit ratio was reached after 0.25 minutes. Regardless, in some cases, the best solution deficit ratio increases after a longer time. For the problems rbg323, rbg358, rbg403 and rbg443, the best deficit ratios were reached after 3 minutes. Additionally, in the case of the problem kro124p, the best deficit ratio was reached after 1 minute, and in the case of the problems ftv64 and ftv170, the best deficit ratio was reached after 0.5 minutes.

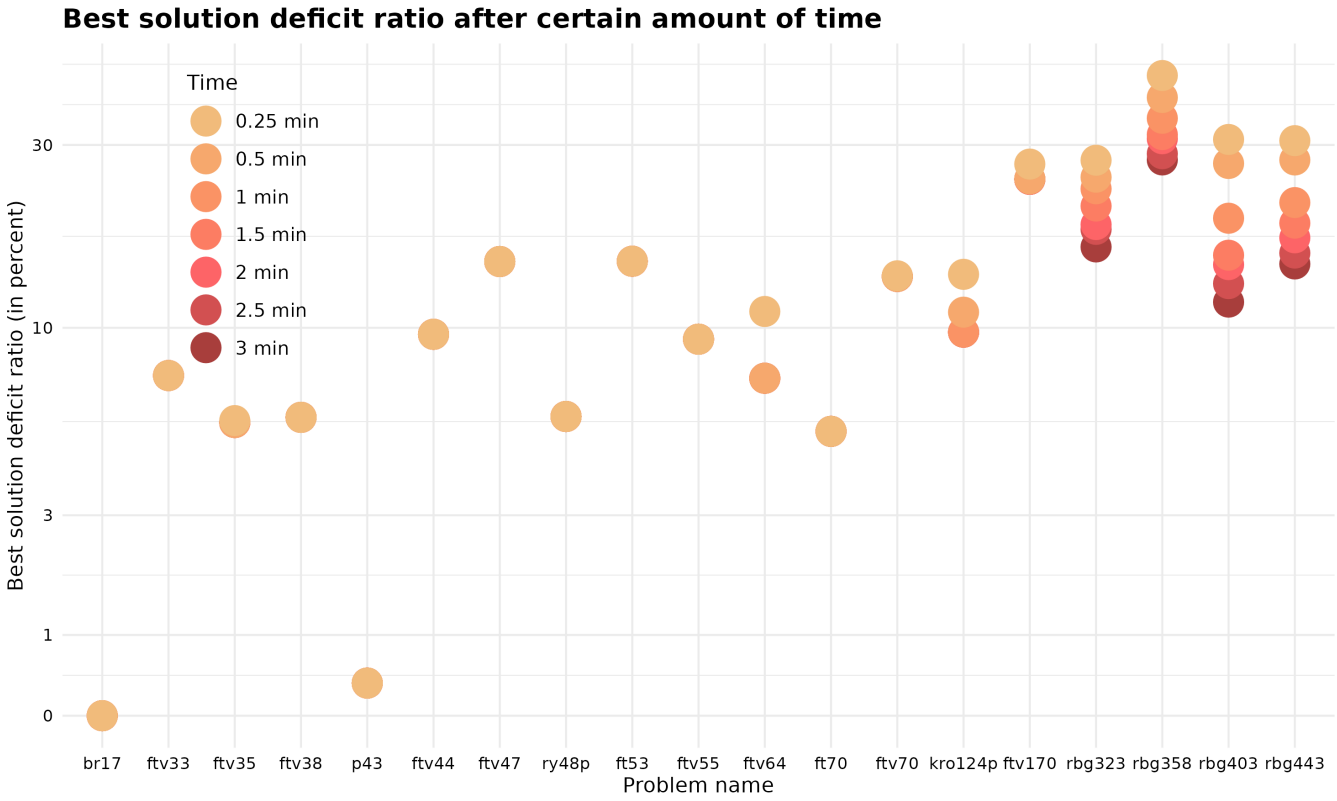


Figure 2: Comparison of the best solution deficit ratio reached after certain amounts of time.

We noticed a trend that the problems with dimensions above 300 usually need more time to get to the best deficit ratio, therefore we decided to run the algorithm 2 times for 2.5 minutes for the problems with dimensions above 300 and 10 times for 30 seconds for the rest of the problems.

Listing 10: Pseudocode for running algorithm a different number of times depending on the problem dimension.

```
function run_algorithm:
  N = len(distance_matrix)
  if N < 300:
    num_of_runs = 5 min / 30 s
  else:
    num_of_runs = 5 min / 2.5 min
  for num_of_runs times:
```

```
8       solution, solution_length = pt_sa(distance_matrix, **parameters)
9       if solution_length < best_solution_length:
10        best_solution, best_solution_length = solution, solution_length
11    return best_solution, best_solution_length
```

4 Parameters

- n - number of states.

  We expected that 100 or more states will be final parameter,

- min_temperature - minimal temperature achievable by states,

- max_temperature - maximal temperature achievable by states.

  It was hard to estimate what the optimal parameters for temperature will be, we did not have any previous experience with Simulated Annealing. Initially, we set the minimum as 0.1 and the maximum as 100,

- probability_of_shuffle - probability of using shuffle transition function in Metropolis Transition, alternatively swap transition function is used.

  In the beginning, it was set to 0.5, which means that half of the states were modified with the shuffle transition function and the rest with the swap transition function. Our intuition behind this was that both transition functions are equally good, but they produce different results. That is the reason why we implemented two versions of the transition function,

- probability_of_heuristic - probability of using heuristic as an initial solution in a state, alternatively random initial solution is used.

  Initially, it was set to 0.3, meaning that 30% of our states were initialized using one of the heuristic solutions. Tuning parameters showed that the bigger this probability is, the better. It is likely due to time restraints (5 minutes), starting from random solution explores solutions space better, but is more time-consuming. The more heuristic initial solutions there are the more stable algorithm is, but it is also more likely to be stuck in the local optimum,

- a - first parameter of the beta distribution,

- b - second parameter of the beta distribution.

To give an idea of how a and b influence beta distribution here plot from Wikipedia[8].
Note that notation is slightly different, a = α and b = β:

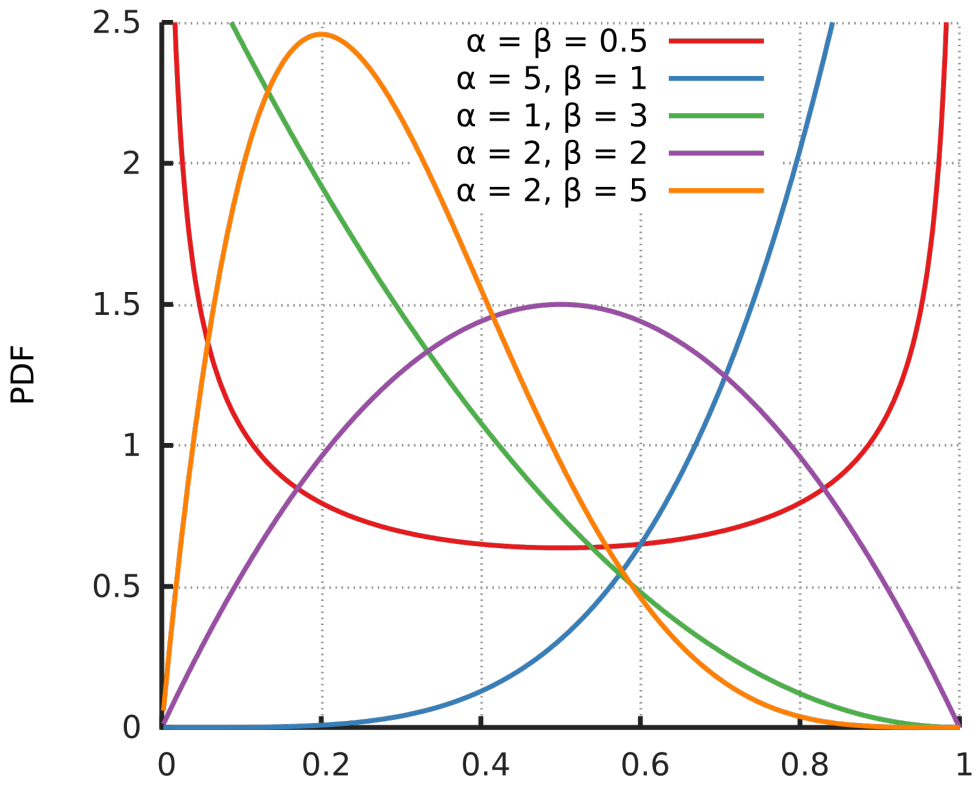


Figure 3: Density of beta distribution with different α and β

We started with a = 1 and b = 1, this set of parameters produces uniform distribution. We thought it was the safest way of doing this, but still gives us a field to maneuver,

- duration_of_execution_in_seconds - number of seconds the PTSA algorithm runs, it is set to 300 seconds which is equal to 5 minutes,
- k - number of executed PT faze between two SA steps,
- max_length_percent_of_cycle - how long path we modify in Metropolis Transition,

This parameter is hard to understand, so an example should help to understand how it behaves.

Let's assume that our max temperature is set to 100 and max_length_percent_of_cycle is set to 0.3. Then in a state where the current temperature equals 90, the length of the path modified in the transition function will be equal to $l = \text{size\_of\_the\_problem} \cdot \frac{90}{100} \cdot 0.3$. If this state uses a swap transition function, then two paths length l will swap places, and if a state uses the shuffle transition function, then the path of length l is shuffled. Note that this implementation requires temperatures to be always positive.

The general equation for length modification in the transition function is:

$$l = \text{size\_of\_the\_problem} \cdot \frac{\text{current\_state\_temperature}}{\text{max\_temperature}} \cdot \text{max\_length\_percent\_of\_cycle}$$

- swap_states_probability - the probability of swapping temperature with another randomly chosen state in Replica Transition,

  We expected this probability to be close to 0, not higher than 0.05. For each Replica Transition, the algorithm tries to make n swaps of temperature with swap_states_probability probability. We thought that swapping temperatures too often would not be beneficial,

- closeness - states with solution shorter than current_best · closeness do not participate in Replica Transformation,

  The idea behind this logic is to allow good solutions to converge. It cannot be too large because then at some point Replica Transform part will not occur anymore, and the algorithm will perform multiple instances of Simulated Annealing. In the beginning, it was set to 1.5, so states with solutions better than 1.5 · current_best_solution did not swap temperature with other states.

- cooling_rate - the speed at which states are cooled.

  new_temperature = cooling_rate · old_temperature

  Temperature is updated in cooling faze, which occurs every k PT fazes. We expected this parameter to be in the range of (0.99, 1). Otherwise, we would fast drop in temperature to a minimum and each time just slightly modify the solution, without proper exploration of the solution space.

5 Testing parameters

5.1 Methodology

The first, greedy idea for selecting the best parameters to solve both small and large problems was grid search. However, it is easy to notice that if we wanted to test the algorithm on all 19 problems and, let's say, check 6 values for each of the 10 parameters, there would be a total of 68,400 algorithm runs. Even with the algorithm limited to just 1 second of execution time, this would result in 19 hours of uninterrupted work. Clearly, with the current hardware resources available to the team, testing large problems with over 100 vertices within just 1 second would not be reliable.

Hence, there arose a need to find another way of parameter testing. We adopted the ratio of the difference between the length of the best known solution and our solution to the best known solution as a measure of solution quality. This universal method allowed us to compare the results.

$$Best\ solution\ deficit = \frac{computed\_solution\_lenght - best\_known\_solution}{computed\_solution\_lengh} \cdot 100\% \tag{1}$$

The process of improving the parameters went through 4 iterations, each following the same procedure. Firstly, a list of values to be tested was determined for each parameter. Then, each value was individually substituted for the default value of the parameter. Each algorithm invocation checked how changing the value of one parameter affected the result for all problems. Since testing the change of one parameter required 19 algorithm runs (one for each problem), we limited the execution time to 2-10 seconds depending on the iteration. At the end of each iteration, we analyzed the collected data and selected the new parameter value that yielded the smallest average deviation from the best known result. Overall, the algorithm was invoked 1 691 times for parameter testing.

5.2 Dependence on the problem size

The distribution of results, divided into small problems (below 50 vertices), medium problems (50 to 100 vertices), and large problems (above 100 vertices), is as follows:

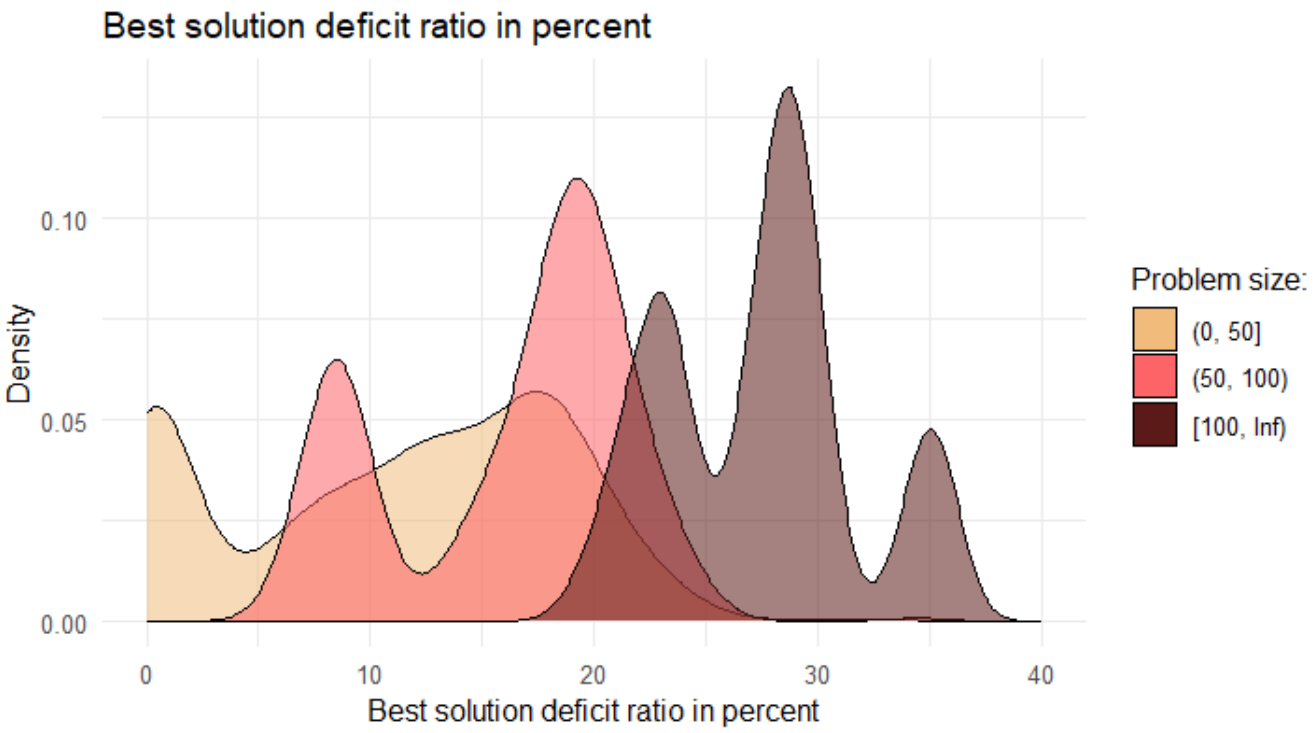


Figure 4: Distribution of best solution deficit ratio.

Among the small-sized problems, we basically have 2 subgroups. The first consists of problems that are solved perfectly, with an error ranging from 0% to 5%, while the second subgroup has errors not exceeding 10% to 17%. Solving medium-sized problems already incurs a greater error. The best results have an error rate ranging from 7% to 10%, while the remaining more frequently occurring cases have an error rate of about 20%. In the case of the largest problems, the best result achieved an error rate of 23%. The majority of large problems achieved results between 27% and 30%, while the most challenging of the largest problems were up to 35% relatively worse than the best result.

On average, our algorithm yielded a result that was 18.36% worse than the best known solution.

5.3 Changing parameters impact

The distribution of relative deviations from the best result for different parameter values is as follows:

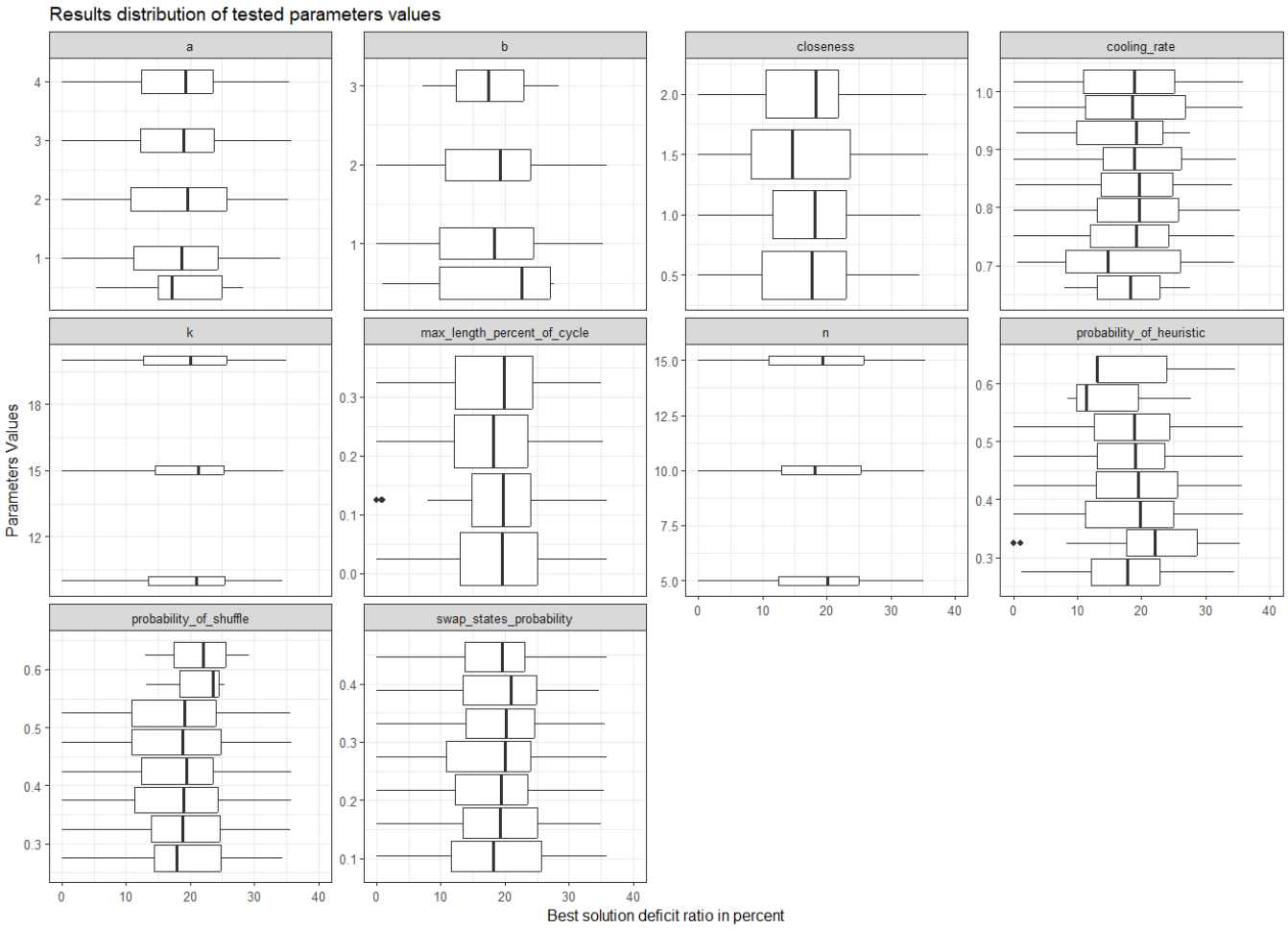


Figure 5: Distribution of best solution deficit ratio grouped by changed parameter.

The parameter 'a' yielded the best results for a value of 0.5, while the parameter 'b' performed best when set to 3. The 'closeness' parameter showed better results for a value of 1.5. In the case of the cooling rate, the average error was the lowest for a value of 0.7, although the values were considerably dispersed. Tests on the 'k' and 'n' parameters confirmed our assumption that larger values produce better results. Changing the 'max length percent per cycle' parameter yielded similar results, with a slightly better value of 0.2. The 'probability of heuristic' parameter yielded significantly better results when set to values greater than 0.5, specifically 0.57 and 0.62. On the other hand, the 'probability of shuffle' and 'swap states

probability' parameters yielded better results with decreasing values, specifically 0.27 and 0.1, respectively.

In summary, after analyzing various parameter values, it can be concluded that their changes did not result in significant differences in the results. Thus, the developed algorithm proved to be very stable when it came to parameter testing. It is possible that changes in values may have a greater impact when the algorithm operates over a longer period of time, such as 1 or 5 minutes. However, due to a lack of sufficient computational resources, we did not consider those cases.

6 Results

6.1 Best solution depending on time

We checked how the best solution depends on time. Therefore, we took one small problem, one medium problem and one big problem. Then for each of them we performed our algorithm, running it for 5 minutes and in the meantime we collected the new best solutions and the timestamps from the start of the algorithm in which they were found. Then we combined the resulting 15 data frames, for each of these 3 problems, by calculating the average of the best solution for each of the timestamps in data frames.

Then, to visualize our results, we scaled the mean column to from 0 to 1, and made a scatterplot of the scaled mean best solution depending on time. The values on the y-axis have also been logarithmized to visualize the effects better. We also put a line connecting these points on the graph, so that you can see the trend in which the best solution behaves.

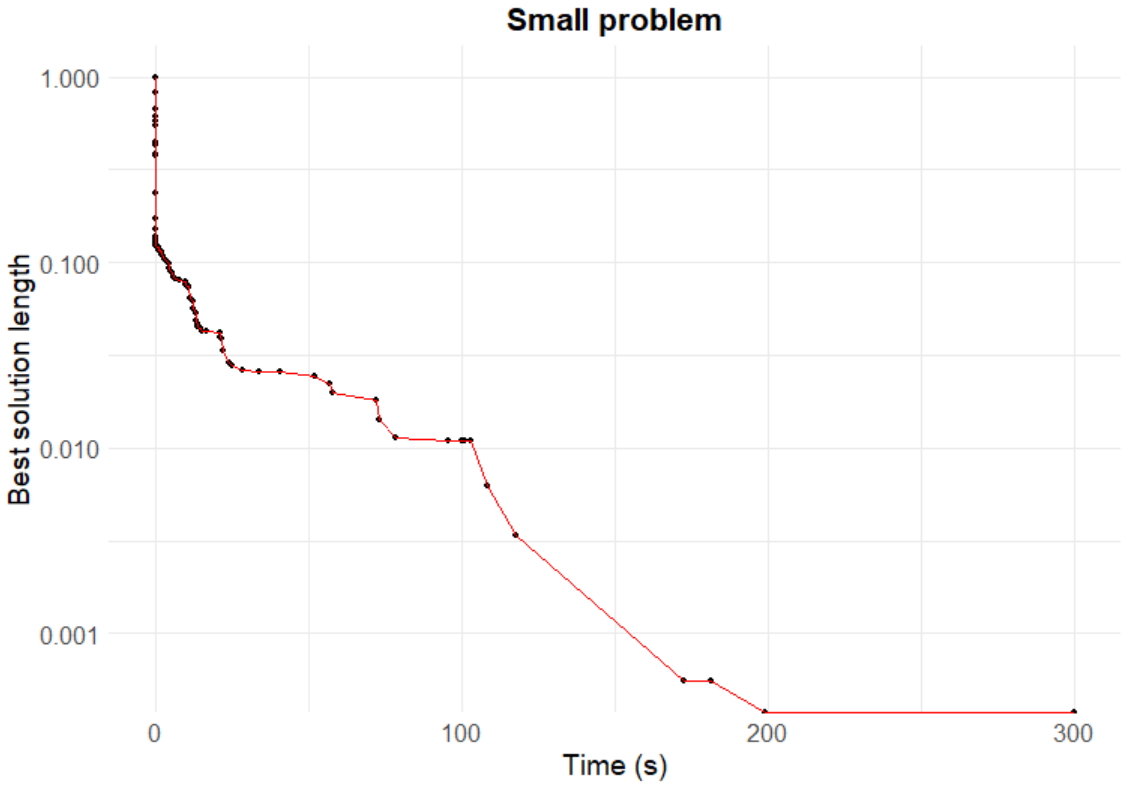


Figure 6: Best solution length for small problem depending on time

We can see that for a small problem at the very beginning, the best solution decreases very quickly over time. This is of course due to the specification of our algorithm (the ptsa method is fired several times within 5 minutes). On average, the best solution for this small problem is found after only 2/3 of our algorithm time.

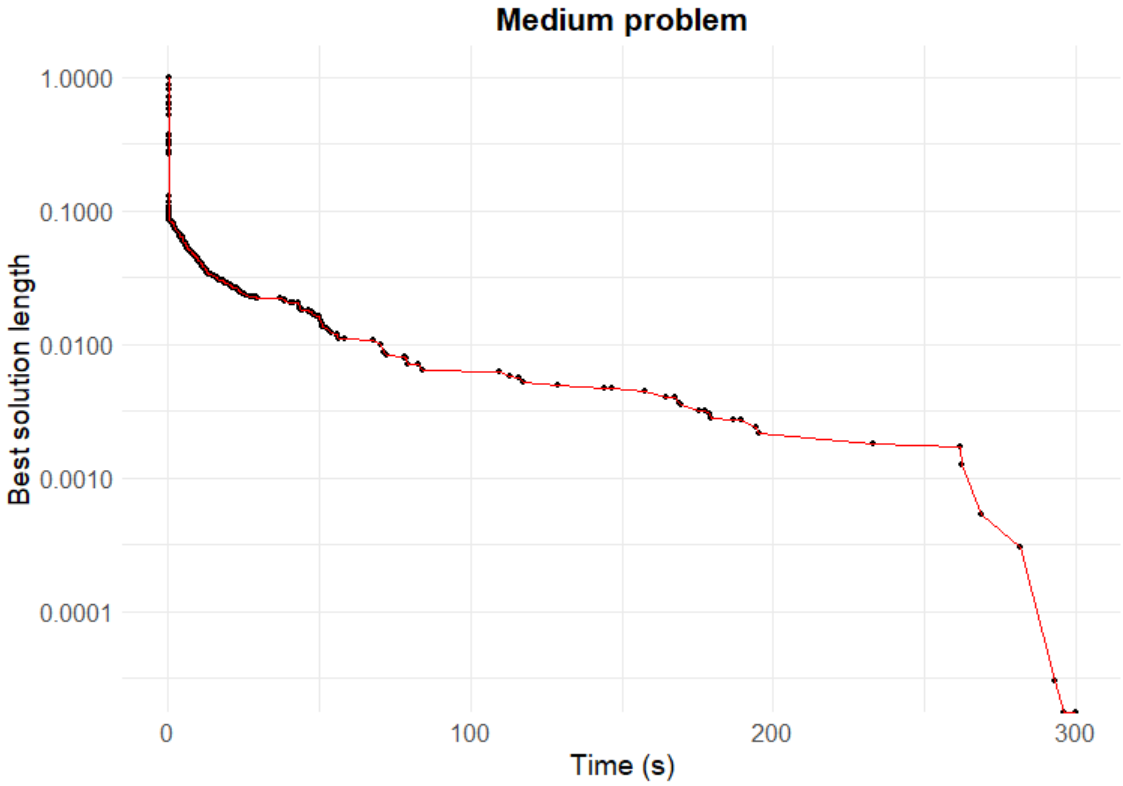


Figure 7: Best solution length for medium problem depending on time

For the average problem, the graph looks a bit different. As with a small problem, the best solution drops off quickly at the beginning. However, on average, the best solution decreases significantly at the very end of our algorithm.

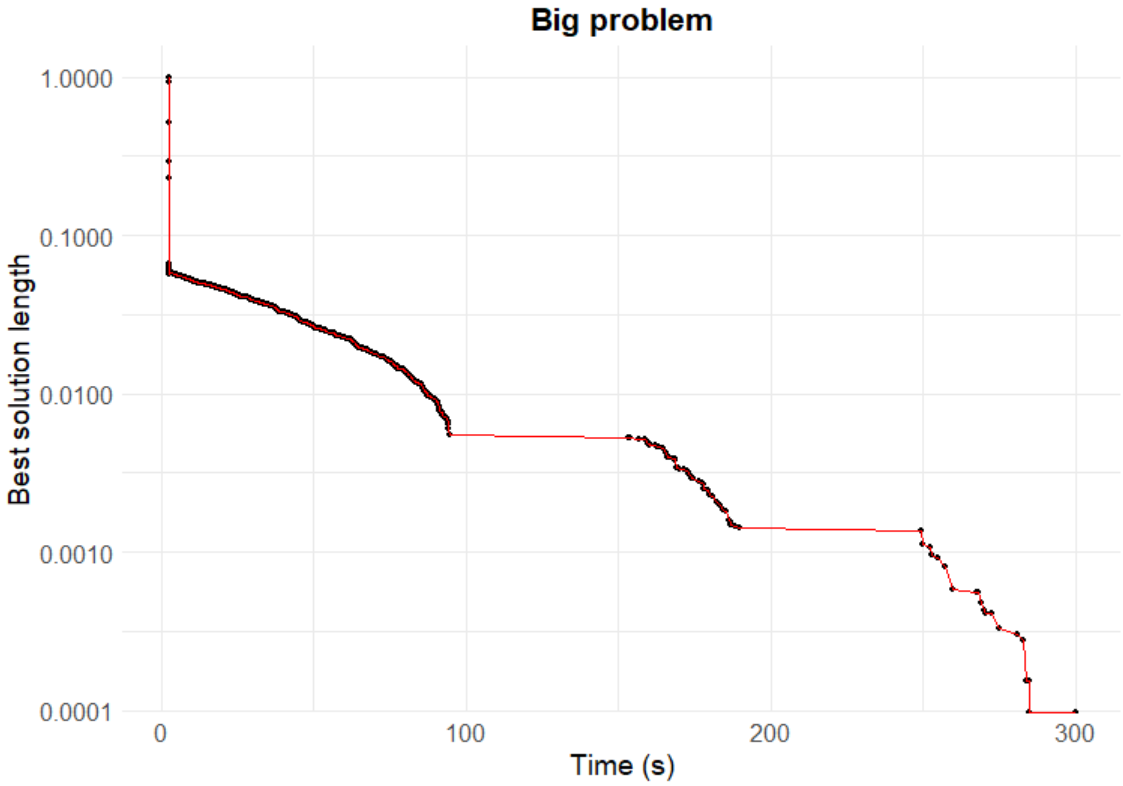


Figure 8: Best solution length for big problem depending on time

For a large problem, the decreases in the graph are slightly more evenly distributed. As before, it decreases significantly at the beginning, but then also the best solution decreases relatively evenly until the end.

6.2 Our best results

The table below shows our best results in running the algorithm for 5 minutes so far for each problem. It also shows the best known results for a problem and how many percent worse is our solution.

Table 4: Solutions' lengths obtained with our final algorithm, in 5 minutes.

Name	**Best known solution length**	**Our best known solution length**	**Best solution deficit ratio (in percent)**
br17	39	39	0.0000000
ftv33	1286	1382	7.4650078
ftv35	1473	1534	4.1412084
ftv38	1530	1614	5.4901961
p43	5620	5627	0.1245552
ftv44	1613	1713	6.1996280
ftv47	1776	1931	8.7274775
ry48p	14422	15105	4.7358203
ft53	6905	8019	16.1332368
ftv55	1608	1782	10.8208955
ftv64	1839	1976	7.4497009
ft70	38673	40597	4.9750472
ftv70	1950	2174	11.4871795
kro124p	36230	40276	11.1675407
ftv170	2755	3379	22.6497278
rbg323	1326	1527	15.1583710
rbg358	1163	1441	23.9036973
rbg403	2465	2742	11.2373225
rbg443	2720	3067	12.7573529

We can see that when it comes to smaller problems, our algorithm is doing a pretty good job. Our solution is worse by around 10% than the best known one for these cases. The algorithm handles bigger problems slightly worse, where our solution is worse up to 24%.

7 Cython

In order to improve the speed of our algorithm, we decided to compile our code in C language. Due to limited time instead of translating our Python code to C, we used Cython language. Since we had specified the data types in all our functions, it was rather easy to make Cython files (extension .pyx) from our Python files. Then we made a setup.py file and ran it to compile those files [7, 1]. At last, we ran our algorithm using the functions compiled in C. We ended up with such results:

Table 5: Solutions' lengths obtained with the use of Cython.

Name	**Best known solution length**	**Our solution length**	**Best solution deficit ratio (in percent)**
br17	39	39	0.0000000
ftv33	1286	1397	8.6314152
ftv35	1473	1556	5.6347590
ftv38	1530	1608	5.0980392
p43	5620	5631	0.1957295
ftv44	1613	1726	7.0055797
ftv47	1776	1937	9.0653153
ry48p	14422	15117	4.8190265
ft53	6905	7934	14.9022448
ftv55	1608	1788	11.1940299
ftv64	1839	1976	7.4497009
ft70	38673	40387	4.4320327
ftv70	1950	2174	11.4871795
kro124p	36230	40722	12.3985647
ftv170	2755	3451	25.2631579
rbg323	1326	1560	17.6470588
rbg358	1163	1527	31.2983663
rbg403	2465	2851	15.6592292
rbg443	2720	3168	16.4705882

The best solution deficit ratio ranges from 0% to around 31% with a mean of around 11%. We compared our Cython solutions with the ones obtained with Python. As we can see below, the results are mixed. For part of the problems Cython solutions are better but for the rest Python solutions are better. In the case

of the problems with the highest dimensions (170 dimension and higher) Cython deficit ratios are dramatically higher, however, Cython's solution for some smaller problems e.g. ftv35 is improved in comparison with the Python one. That said, for most of the problems Python's and Cython's deficit ratios are rather close. All things considered, we chose the Python approach during The Race and for the final results.

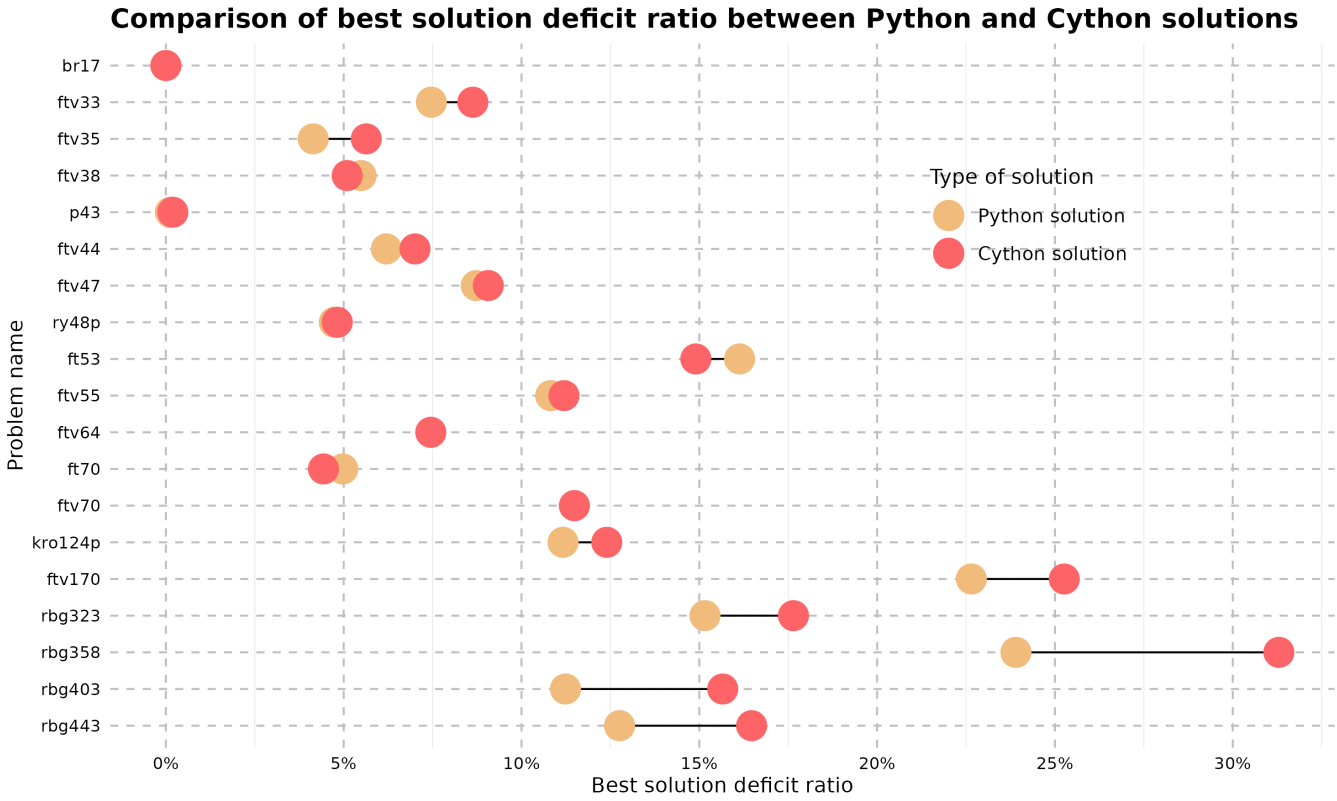


Figure 9: Comparison of best solution deficit ratio between solutions' lengths obtained with Python and Cython.

8 Comparison with another team

PT_SA algorithm have been compared with a Spider Monkey Optimization - algorithm created by other team.

In Spider Monkey Optimization algorithm, every spider monkey represents a Travelling Salesman Problem solution; and Swap Sequence (SS) and Swap Operator (SO)-based operations are considered for interaction among monkeys to find the optimal TSP solution. An SS is a group of SOs. Each contains a pair of indexes on the TSP tour. A new tour is generated transforming a TSP tour by swapping cities indicated by SOs of an SS. The SS generation to update a monkey for an improved solution is the key feature of the method and SS of a particular spider monkey is generated with interaction with other members of the group. In the case of updating a solution of spider monkey with generated SS, a Partial Search (PS) technique is deployed to achieve the best outcome with full or partial SS.

Both algorithms were tested in a short race formula - with a 60 seconds time limit for a single problem. The results are provided below.

Table 6: Comparison with Monkey team.

Name	**PT_SA**	**Monkey**	**Solutions difference ratio (in percent)**
br17	39	39	0.0000000
ftv33	1421	1860	17.5817618
ftv35	1636	1608	7.2011326
ftv38	1618	1849	26.1394102
p43	5645	5682	9.5003519
ftv44	1891	2175	14.7012163
ftv47	2128	2566	5.4388133
ry48p	15180	16375	14.7012163
ft53	8378	10280	16.5413534
ftv55	1788	2594	10.0671141
ftv64	2347	3422	21.6446527
ft70	41674	47893	14.6608315
ftv70	2285	3645	18.9902287
kro124p	44723	95092	0.4428698
ftv170	3730	14902	17.3316708
rbg323	1604	5389	26.9930948
rbg358	1593	5920	16.0993873
rbg403	2938	6642	17.0225747
rbg443	3278	6957	4.9934124

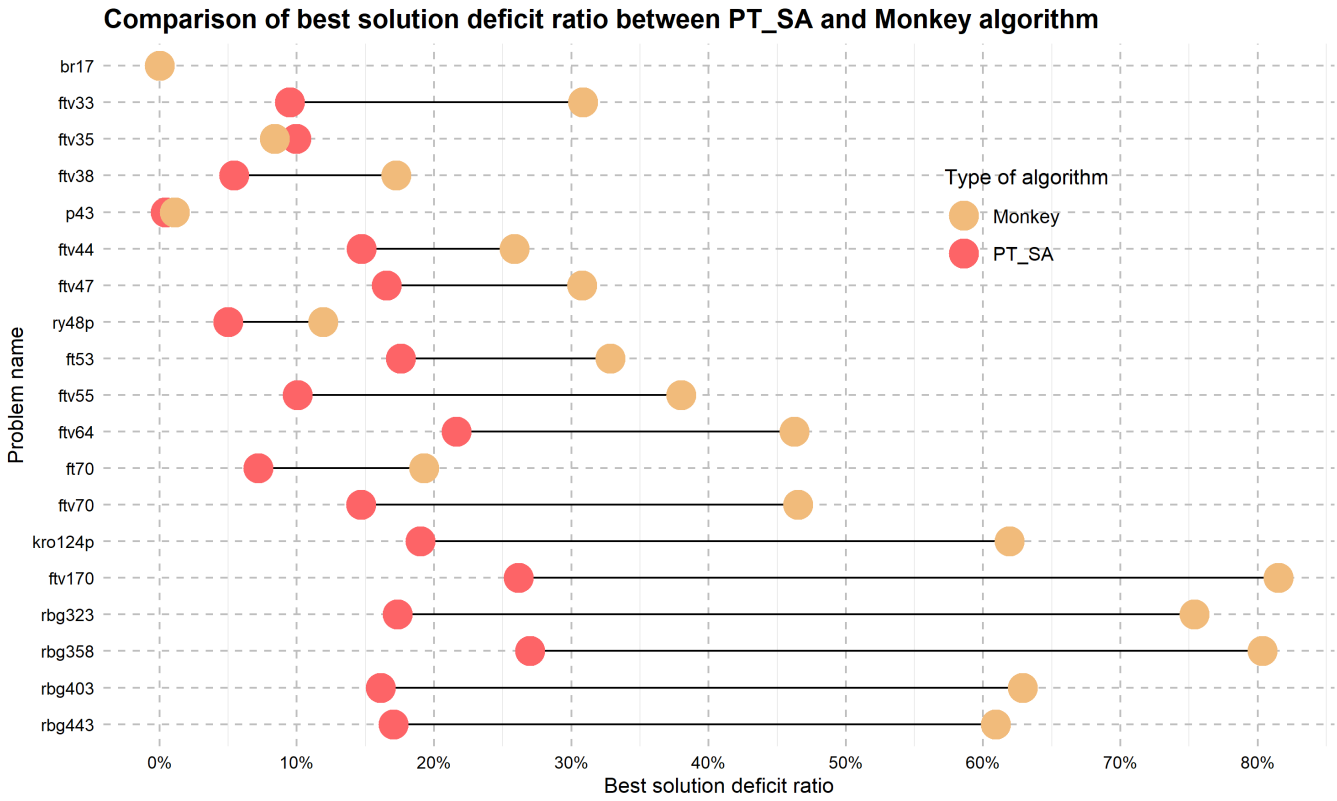


Figure 10: Comparison of best solution deficit ratio between solutions' lengths between PT_SA and Monkey algorithms.

In 18 of 19 cases PT_SA algorithm was better than SMO.

9 Long-term results

All the time before we were running our algorithm for up to 5 minutes since we were focused on The Race. However, we were curious about what results we would achieve after a longer time. Therefore, we also ran the algorithm for every problem for 30 minutes, and here are the results:

Table 7: Solutions' lengths obtained after 30 minutes run of the algorithm.

Name	**Best known solution length**	**30 min solution length**	**Best solution deficit ratio (in percent)**
br17	39	39	0.0000000
ftv33	1286	1443	12.2083981
ftv35	1473	1624	10.2511881
ftv38	1530	1616	5.6209150
p43	5620	5637	0.3024911
ftv44	1613	1768	9.6094234
ftv47	1776	1965	10.6418919
ry48p	14422	15146	5.0201082
ft53	6905	8136	17.8276611
ftv55	1608	1788	11.1940299
ftv64	1839	2105	14.4643828
ft70	38673	40706	5.2568976
ftv70	1950	2246	15.1794872
kro124p	36230	40099	10.6789953
ftv170	2755	3379	22.6497278
rbg323	1326	1446	9.0497738
rbg358	1163	1310	12.6397248
rbg403	2465	2525	2.4340771
rbg443	2720	2824	3.8235294

The best solution deficit ratio ranges from 0% to around 23% with a mean of around 9%. We compared our long-term solutions with the ones obtained with multiple runs of the algorithm during 5 minutes period. In the case of the problems with the highest dimensions (323 and above), the best solution deficit ratios after 30 minutes run are radically improved. Unfortunately, in terms of smaller problems, the results are substantially worse, probably due to bad random seeds. If we had had 30 minutes during the race, we would have run the algorithm for the problems with dimensions 300 or higher once for 30 minutes and for others stayed with running multiple times for 30 seconds.

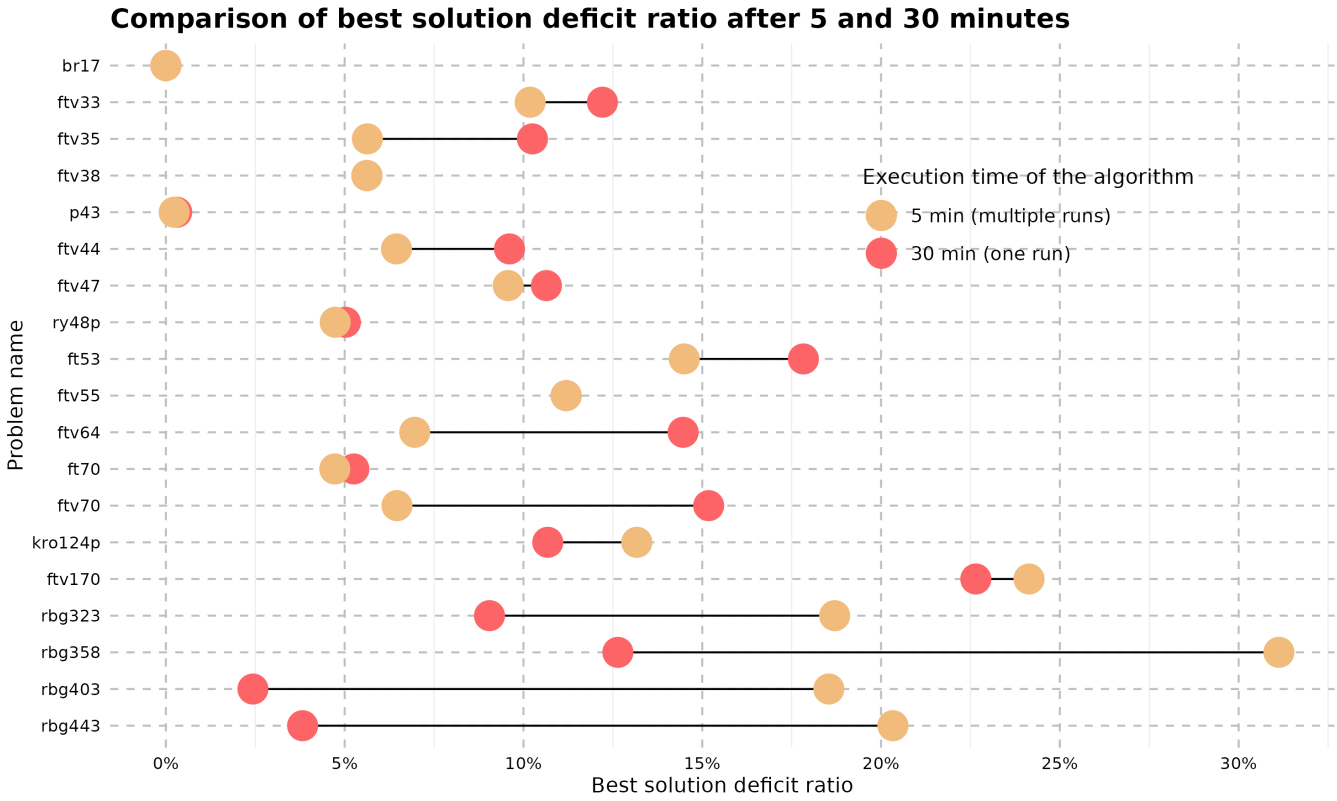


Figure 11: Comparison of best solution deficit ratio between solutions' lengths obtained with 30 minutes run of the algorithm and multiple runs during 5 minutes period.

10 Summary

We implemented the Parallel Tempering Simulated Annealing (PTSA) algorithm, a noteworthy approach for solving complex optimization problems, for the Asymmetric Travelling Salesman Problem and present the results demonstrating its effectiveness.

The PTSA is a metaheuristic algorithm that is used for solving optimization problems. Simulated Annealing explores the solution space by randomly changing the current solution and accepting or rejecting the new one based on a certain probability criterion. Combined with the Parallel Tempering approach, it facilitates

escaping from local minima by running different instances of the SA algorithm at once and also exchanges temperatures between replicas.

We modified the original algorithm with a few ideas. One of the significant changes was initializing not only random solutions but also heuristic ones calculated with the nearest neighbor algorithm. We also implemented two types of metropolis transition: shuffle transition and swap transition, which gives our states more room for change. Additionally, we sped up the metropolis transition by using multithreading with a thread for every state. Furthermore, we decided to skip solutions close to the best solution in the replica transition. At last, we run tests to determine whether we would get better solutions via running the algorithm multiple short times instead of one long run. After thorough analysis, we indeed chose the multiple runs approach. We agreed to run the algorithm 2 times for 2.5 minutes for the problems with dimensions above 300 and 10 times for 30 seconds for the rest of the problems.

We introduced 13 parameters to modify the behavior of our algorithm accordingly to our desire. Those are n, min_temperature, max_temperature, probability_of_shuffle, probability_of_heuristic, a, b, duration_of_execution_in_seconds, k, max_length_percent_of_cycle, swap_states_probability, closeness, cooling_rate. Then we used them to fine-tune the performance of our algorithm.

To identify the best parameters, we created a relative indicator called the "Best Solution Deficit Ratio" to compare the results. In order to conduct the testing, we performed 1 691 invocations of the algorithm. Based on the analysis of the collected data, we selected the best parameters. Our algorithm proved to be highly stable, as parameter changes resulted in differences in the "Best Solution Deficit Ratio" of approximately 1%.

We also checked how does best solution length depend on time for small, medium and big problems. The results showed that on average in each of these cases, the best solution decreases rapidly at the beginning. For a small problem, the ultimate best solution is found rather quickly. Until 2/3 of the 5 minute time maximum. In the medium problem, the ultimate solution was found on average at the end, in the last iteration of our PTSA method. For big problems, the solution was decreasing quite evenly throughout time.

The final results were quite satisfying. When it came to small problems, our 5 minute algorithm did a pretty good job. For bigger problems, the results were slightly worse but still satisfying.

In order to improve the speed of our algorithm, we tried compiling our code in C language using Cython. However, the results on average were worse than Python ones, therefore, we decided to stay with the Python approach.

In comparison to the Spider Monkey Optimization algorithm, PT_SA were better in 18 of 19 tested problems.

We also did a long-term test running the algorithm for all problems for 30 minutes each. After that long-term run results for problems with higher dimensions were significantly better so it would be highly recommended to perform longer runs for those problems. Unfortunately, we did not have that much time during The Race.

11 Possibilities for development

Even after The Race, there are things to do and a few ideas to explore. We will list the most important ones and try to explain why we need them in the first place, why we think they would change something, and if it would be beneficial.

Ideas to implement:

- We think that swapping solutions instead of temperatures might make a difference. Why is that? Because then different transition functions operate on the same solutions, we believe this would facilitate exploring solution space better. Thanks to the modular implementation of our algorithm it is a relatively easy task. Although we do not know if it would make a difference with the algorithm tuned for 5 minutes.

- We experimented with Cython with mixed results. For sure, there is a place for improvement when it comes to performance. An obvious solution is to rewrite everything in some different language known for its blazing-fast speed, like Rust or C++. We know that our project inspired our colleague to do just that. He is trying to recreate it in Rust. Unfortunately, we do not know how it is going.

- To have greater control over the behavior of our algorithm, we would introduce a new parameter called number_of_replica_transitions_swap. Now, the number of swaps is equal to n. This unwanted correlation might blur the real optimal value of n, and that might be the reason why it is beneficial to run the algorithm multiple times with a lower n value during these 5 minutes. Instead, we would just increase n, which would result in simpler code.

- From the beginning, we assumed that an exponential decrease in temperature was the way to go. But that might not be the case. There are multiple modifications of Simulated Annealing that use other cooling functions. It might be worthwhile to check how some of them perform in our algorithm.

- Given how effective nearest neighbor initial solutions are, other heuristics might be used alongside existing ones to boost variety and provide even better initial solutions. Again, thanks to our very modular code, it is easy. What might be challenging, though, is figuring out an idea for heuristics. The nearest neighbor is by far the most popular, and we are not sure if the alternative even exists.
- In the end, we would test parameters more thoroughly. Our current approach to executing the algorithm for 2 - 10 seconds gives a bias to rely heavily on initial heuristic solutions and drop temperature quickly. That way, we create an initial solution and just slightly fine-tune it. Right now, we are finding fast local optimums, but then we are stuck. We believe changing our testing methodology would significantly improve algorithm performance, especially on larger problems.

References

[1] Peter Baumgartner. *An Introduction to Just Enough Cython to be Useful*. https://www.peterbaumgartner.com/blog/intro-to-just-enough-cython-to-be-useful/. [Online; accessed 17 May 2023]. 2022.

[2] Adam Chojecki et al. *2023Lato-WarsztatyBadawcze*. GitHub repository. 2023. URL: https://github.com/PrzeChoj/2023Lato-WarsztatyBadawcze.

[3] Łukasz Grabarski et al. *Warsztaty_Badawcze*. GitHub repository. 2023. URL: https://github.com/nizwant/Warsztaty_Badawcze.

[4] Leonardo Jelenković and Joško Poljak. "Multithreaded Simulated Annealing". In: (Jan. 1998). URL: https://www.researchgate.net/publication/229004865_Multithreaded_Simulated_Annealing.

[5] OpenAI. *ChatGPT*. Artificial Intelligence Language Model. 2021. URL: https://chat.openai.com.

[6] Md. Azizur Rahman and Hasan Parvez. "Repetitive Nearest Neighbor Based Simulated Annealing Search Optimization Algorithm for Traveling Salesman Problem". In: *Open Access Library Journal* 8 (2021), pp. 1–17. ISSN: e7520. DOI: https://doi.org/10.4236/oalib.1107520. URL: https://www.scirp.org/journal/paperinformation.aspx?paperid=109634.

[7] JetBrains s.r.o. *Cython support*. https://www.jetbrains.com/help/pycharm/cython.html. [Online; accessed 17 May 2023]. 2023.

[8] Wikipedia contributors. *Beta distribution*. https://en.wikipedia.org/wiki/Beta_distribution. [Online; accessed 2 June 2023].

[9] Wikipedia contributors. *Nearest neighbour algorithm*. https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm. [Online; accessed 3 April 2023].