Hanoi University of Science and Technology
SoICT

PROBLEM 7 REPORT

FUNDAMENTALS OF OPTIMIZATION

TEAM 6:

Mai Việt Bảo

Nguyễn Nam Khánh

Lê Duy Anh

Võ Tá Quang Nhật

Supervisor: Dr. Bùi Quốc Trung

Contents

Abstract

In this report, we are trying to propose several algorithms in order to solve the problem 7 from mini-project topics. In each algorithm, we demonstrate the idea, modeling and conclusion of its performance. The implementation are attached to the assignment.

We also had noted our mistakes pointed out by the supervisor in the representation.

1 Introduction

Problem Statement:

CBUS

There are n passengers $1, 2, \ldots, n$. The passenger i want to travel from point i to point $i + n(i = 1, 2, \ldots, n)$. There is a bus located at point 0 and has k places for transporting the passengers (it means at any time, there are at most k passengers on the bus). You are given the distance matrix c in which $c(i, j)$ is the traveling distance from point i to point j $(i, j = 0, 1, \ldots, 2n)$.
Compute the shortest route for the bus, serving n passengers and coming back to point 0.

We had also created some testcases for the problem which used for the algorithms below. They are attached to the assignment.

2 Proposed Solutions

2.1 Backtracking - Branch and Bound

Backtracking is the first direct and naive solution that we first think of to solve the problem.

It tries all configurations that satisfy the constraints and see which one is the cheapest. We construct a combination step-by-step, from the ground up:

- Initially, we have an empty configuration: a bus that is currently at 0 and has picked up no passenger. We choose a city to visit. Choice is made by adding it to the current configuration (as a list) and marking the city visited.
- At each subsequent step, based on the current configuration (current state of the bus), we have the choice of either picking up or dropping off passengers. Add that city of choice to the current configuration.
- Whenever the configuration is complete, compare that with the best solution we are keeping track of. If the new solution is better, replace.
- Backtracking happens when we have considered all candidates for a position. For instance when there's no new candidate for k^{th} position, we remove/unmark the state at that position and move on to other candidates of the previous $k-1^{\text{th}}$ position.

This algorithm is exact, meaning it can find the optimal solution. But this brute-forcing technique is computationally intensive. Running time for this approach lies within a polynomial factor of O(n!), making this technique impractical even for N=10 or 20 cities.

The second exact algorithm we can use is Branch and Bound.

Branch is the action of breaking down the problem into tiny subproblems that are easier to solve. In our case, the subproblem is to construct subsets of a complete route, much like Backtracking's process of adding cities into subsets till they become a possible solution. Thus this step is exactly like the Backtracking implementation.

Bound is the action of using a bounding function to eliminate extensive searches into subproblems that cannot surpass the best solution so far. In our case of minimizing the path length, the lower bound (the shortest our current configuration can possibly be) is this function:

$$g(N) = d + \text{emin} \times (2N - l)$$

where:

d: the current accumulated path length

emin = the shortest edge to any of the unvisited cities

$2N - l$: number of unvisited cities, where l is the length of current route.

It's only possible when all the edges to unvisited cities are of that length, which is quite unrealistic. If that bound is still higher than the current record of shortest path, we can safely end the current search and backtrack to other choices of visit.

These exact algorithms can solve to optimality, but only suitable for a very small number of cities. As depicted in Table 1, in a N=5 problem, Branch and Bound cut down a visible amount of computing time compared to Backtracking, though both algorithms could not solve any problem larger than that.

	Backtracking	Branch and Bound
Best distance found	37	37
Time taken	0.19479	0.04803

Table 1: Performance of Backtracking and BnB at a N=5 problem

2.2 Constraint Programming

If we step back for a more general look, this problem is classified as Capacitated Single Vehicle Routing Problem with Pickup and Delivery (CSVRPPD)[1]. In our case, the constraints could be broken down as the following:

- Depot point: 0.
- Simply defined pairs of pickup and deliveries: pickups at i and deliveries at i+N.
- Simply defined capacity constraint: minimum 0 and maximum K, +1 for all pickup points and -1 for all delivery points.

[1]In the original presentation, our classification of this problem as CVRPPD was pointed out by our supervisor to be misleading. We fully acknowledged it and further specified the problem in this report as Single Vehicle Routing Problem (SVRP).

- "Hidden" constraint for an optimal route: each city is entered and exited exactly once.

We turn the problem into a series of linear constraints:

- Decision variables: $x_{ij} \in \{0, 1\}\ \forall i, j \in \{0, ..., 2N\}$
- Objective function: $\Sigma_i \Sigma_j c_{ij} x_{ij}$ min
- Flow control: $\begin{cases} \Sigma_i x_{ij} = 1\ \forall j \in \{0, ..., 2N\} \\ \Sigma_j x_{ij} = 1\ \forall i \in \{0, ..., 2N\} \end{cases}$
- Subtour elimination (DFJ):

$$\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1\ \forall Q \subset \{0, ..., 2N\}, |Q| \geq 2$$

- Capacity constraint:

$$\sum_{i=1}^{n} \sum_{j=2}^{n} q_j x_{ij} \leq K,\ q_j = \begin{cases} 1 & \text{if } j \leq N \\ -1 & \text{otherwise} \end{cases}$$

- Pickup - drop-off order constraint: $T_i < T_{i+N}$, T_i is the accumulated path length from 0 to i in the current solution.

We used the old pywrapcp's routing solver instead of CP-SAT's since the last 2 constraints resulted in too many linear constraints, which negatively impacted the solver's performance. Meanwhile, the old solver has tools that can take them into consideration with just a few lines of code.

Using the OR-Tools, the solver works great till N=100 (or 201 cities). Result of our implementation and its runtime is depicted in Table 2. From that point onward, it takes extremely long, eventually quitting and returning 'inf'.

	N=10 (21 cities)	N=100 (201 cities)
Best distance found	41	122
Time taken	0.03501	18.97257

Table 2: Performance of pywrapcp at larger problem sizes

It seems that exact algorithms have been taking up a huge amount of computational power while only able to work with smaller problems. We move on to heuristic algorithms, which instead of considering every feasible solution, they try to arrive at a good, reasonable solution.

2.3 Nearest Neighbor

This is a constructive heuristic algorithm that builds the solution step-by-step like the first 2 algorithms. But instead of considering all feasible solutions, it will only create one solution and always (naively and greedily) choose the nearest unvisited city for the next move. Table 3 shows that Nearest Neighbor could quickly return a solution to a problem up to N=1000 (or 2001 cities).

Best distance found	12176
Time taken	0.43269

Table 3: NN's solution for a N=1000 problem

In exchange for the speed, the solution given by Nearest Neighbor is nowhere near optimality. Table 4 compares the solution of this to the optimal one by Backtracking, where their solutions show a stark gap in score.

	Backtracking	Nearest Neighbor
Best distance found	37	49
Time taken	0.19479	0.0

Table 4: Performance of Backtracking and NN at a N=5 problem

However, Nearest Neighbor's solution could be use as a pretty good starting point for these subsequent Local Search heuristic algorithms.

2.4 Hill climbing - Beam search

Both these algorithms belong to a family of algorithms called Local Search heuristic, which are especially helpful in solving computationally hard optimization problems. Local Search starts with a feasible initial solution. From there, concerned feasible solutions are mutated to form "neighbor" solutions that could better the objective value, hence the name "Local Search". After repeatedly expanding and jumping from solution to solution, the search can be terminated given a specific condition.

Hill climbing starts with a arbitrary state, then tries to find a better state in the previous state's neighbor. If none of the neighbors is better, the algorithm is terminated.

In order to find neighbors of a solution, we use the k-opt technique. k-opt intentionally removes k edges from the current solution and adds k new ones, creating a new and (in most cases) better solution.

The most basic operation is 2-opt, of which the idea is conveyed in Figure 1:

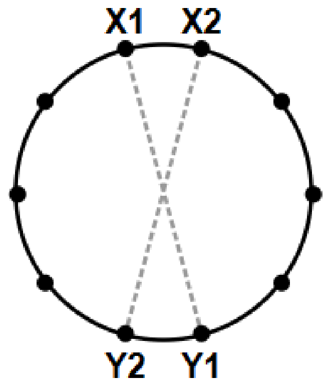


Figure 1: 2-opt edges replacement visualization

We will replace edges when

$$d(X1, Y1) + d(X2, Y2) < d(X1, X2) + d(Y1, Y2)$$

where $X1Y1$ and $X2Y2$ are newly added edges, $X1X2$ and $Y1Y2$ are removed edges.

For 3-opt, every 3-edge manipulation results in $2^3 - 1 = 7$ new neighbors, which is visibly much more computationally intensive than 2-opt. We consider 2.5-opt, which is 2-opt and 2 simple cases of 3-opt.

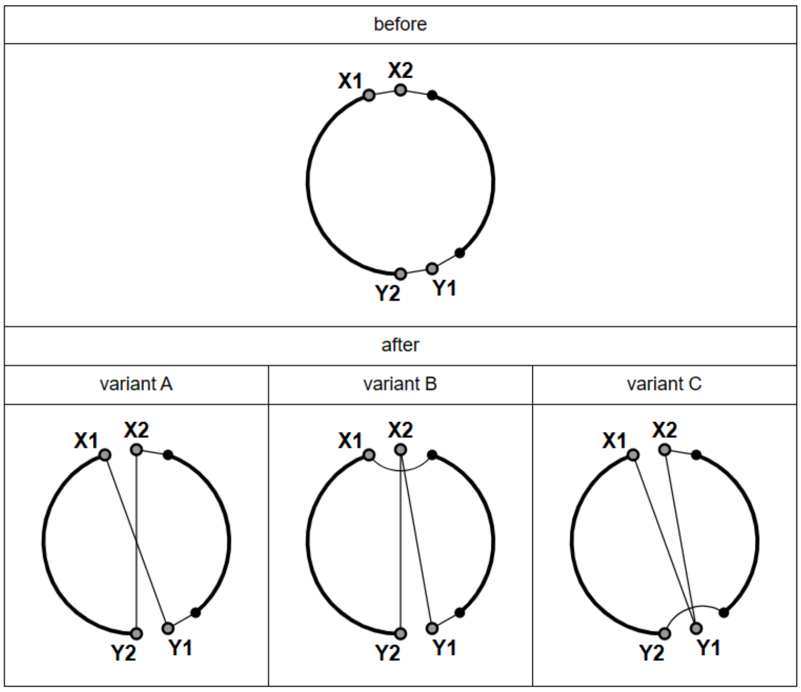
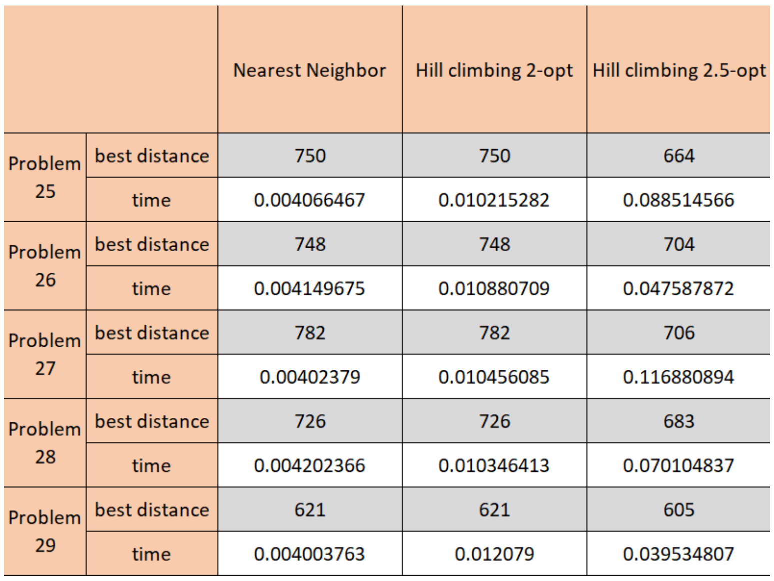


Figure 2: 2.5-opt neighbor possibilities

In order to check the performance, we set Hill climbing with both 2-opt and 2.5-opt against NN in 5 randomly generated N=100 problems. The results are depicted in Figure 3.



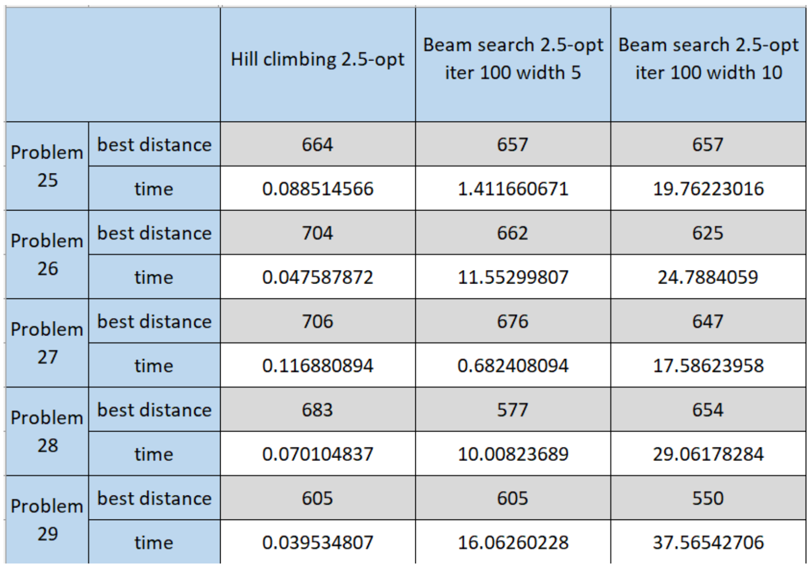
		Nearest Neighbor	Hill climbing 2-opt	Hill climbing 2.5-opt
Problem 25	best distance	750	750	664
	time	0.004066467	0.010215282	0.088514566
Problem 26	best distance	748	748	704
	time	0.004149675	0.010880709	0.047587872
Problem 27	best distance	782	782	706
	time	0.00402379	0.010456085	0.116880894
Problem 28	best distance	726	726	683
	time	0.004202366	0.010346413	0.070104837
Problem 29	best distance	621	621	605
	time	0.004003763	0.012079	0.039534807

Figure 3: Hill climbing 2-opt and 2.5-opt

In our limited testing, 2-opt unfortunately was not able to further improve Nearest Neighbor's solution. On the other hand, 2.5-opt could find better solution with a fractional increase in computing time. This also shows that a good neighborhood searching method would come a long way, as 2.5-opt could help a greedy-in-nature heuristic algorithm like Hill climbing find a much better solution than the initial one. With that said, we will use 2.5-opt for subsequent local search algorithms.

Beam search is another Local Search heuristic algorithm. Instead of focusing on one best neighbor, it keeps track of and expand a set number of most promising neighbors in an iteration. That number is β - beam width.

We tested Beam search with $\beta = 5$, $\beta = 10$, each in 100 iterations, and gauged their improvement over Hill climbing. The results are in Figure 4.



		Hill climbing 2.5-opt	Beam search 2.5-opt iter 100 width 5	Beam search 2.5-opt iter 100 width 10
Problem 25	best distance	664	657	657
	time	0.088514566	1.411660671	19.76223016
Problem 26	best distance	704	662	625
	time	0.047587872	11.55299807	24.7884059
Problem 27	best distance	706	676	647
	time	0.116880894	0.682408094	17.58623958
Problem 28	best distance	683	577	654
	time	0.070104837	10.00823689	29.06178284
Problem 29	best distance	605	605	550
	time	0.039534807	16.06260228	37.56542706

Figure 4: Beam search with $\beta = 5$ and 10

Beam search immediately gives better solutions than Hill climbing, in exchange for more computing time - this is to be expected. Beam width plays a crucial role in finding better results, in exchange for even more computing power, thus one must know and suitably balance between speed and result optimality.

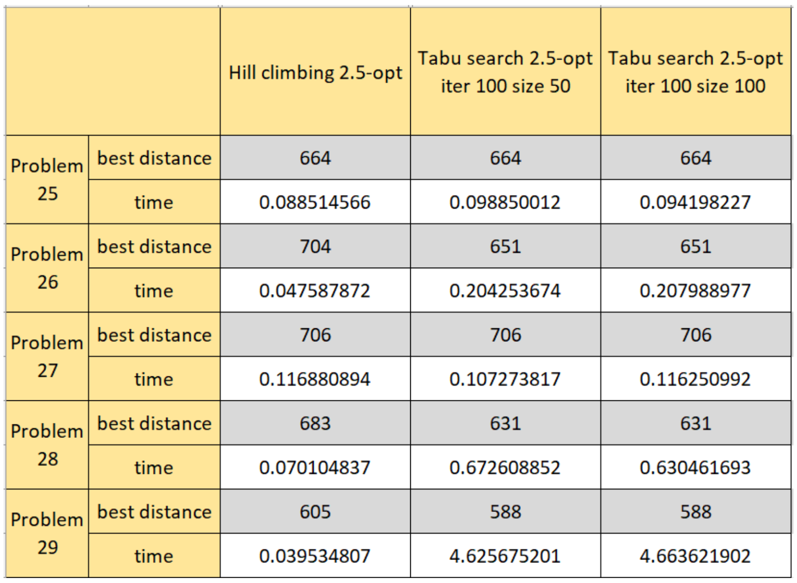
2.5 Tabu search - Tabu Beam

These algorithms belong to the higher-level Local Search-based metaheuristic algorithm. These are additional heuristic method on top of the existing Local Search heuristic techniques, designed to assist or guide simple Local search heuristic algorithms toward better solutions.

We start with Tabu search. Like Hill climbing, it expands the best neighbor, but enhances the performance by relaxing some rules:

- Accept "best in the neighborhood but worse than the best ever" neighbors.
- Keep track of a tabu list of previously best neighbors of predetermined size to avoid revisiting.

We tested Tabu search with tabu list size of 50 and 100, each in 100 iterations, and compared them with Hill climbing. The result is in Figure 5.



		Hill climbing 2.5-opt	Tabu search 2.5-opt iter 100 size 50	Tabu search 2.5-opt iter 100 size 100
Problem 25	best distance	664	664	664
	time	0.088514566	0.098850012	0.094198227
Problem 26	best distance	704	651	651
	time	0.047587872	0.204253674	0.207988977
Problem 27	best distance	706	706	706
	time	0.116880894	0.107273817	0.116250992
Problem 28	best distance	683	631	631
	time	0.070104837	0.672608852	0.630461693
Problem 29	best distance	605	588	588
	time	0.039534807	4.625675201	4.663621902

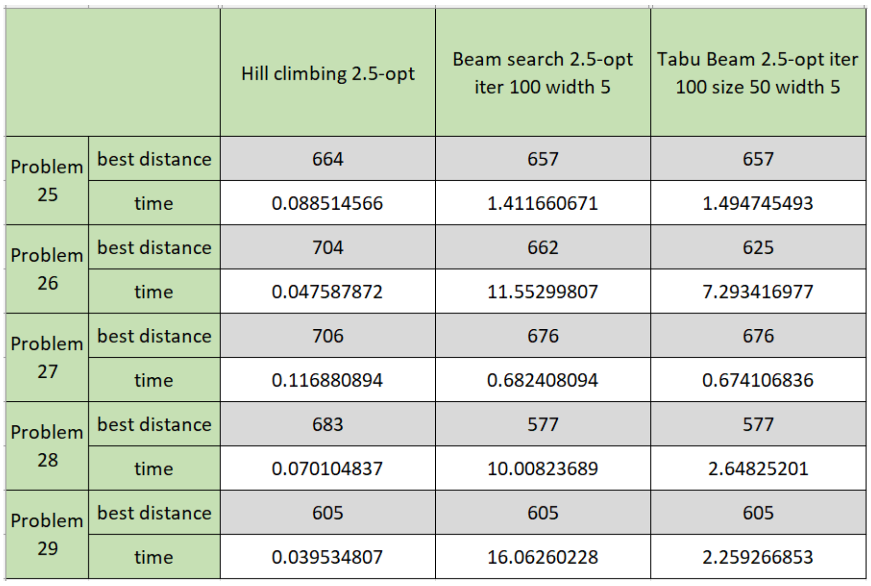
Figure 5: Tabu search with tabu list sized 50 and 100

Tabu search, thanks to the relaxed rules, finds better solutions than Hill climbing, all while keeping the increase in computing time pretty negligible. On top of that, we found that increasing the tabu list size did not yield better results, at least in our test cases with 100 iterations. With order-of-magnitude larger number of iterations, increasing tabu list size will better show its importance in avoiding repetitions.

Seeing the effect of tabu list, we tried to combine Beam search and Tabu search. This time, tabu list is used to enhance Beam Search. Tabu list helps eliminate recurring solutions in the beam, e.g. the solution where 2 pairs of edge are repetitively added/removed.

We tested Tabu Beam hybrid search with $\beta = 5$ and $\beta = 10$, each kept a tabu list sized 50 and ran in 100 iterations.

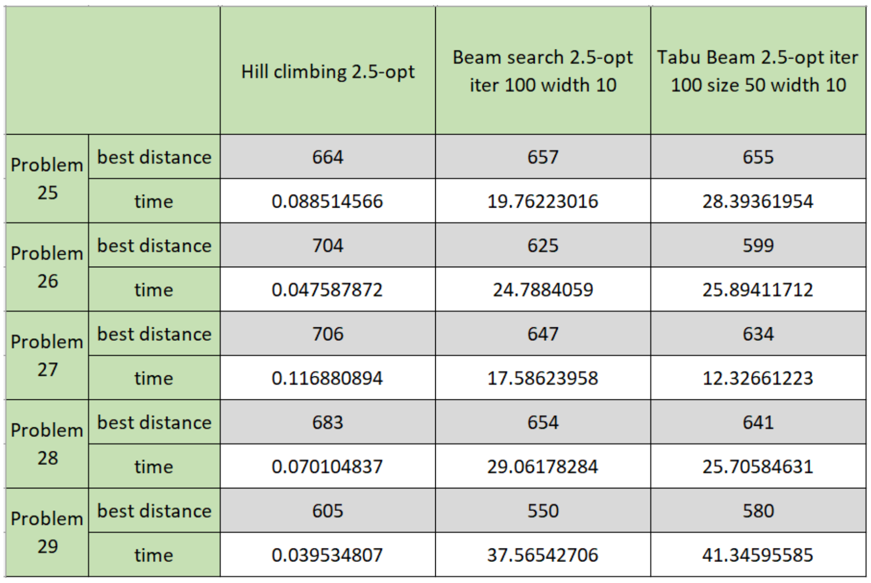
We measured their improvement over basic Beam search with similar parameters.



		Hill climbing 2.5-opt	Beam search 2.5-opt iter 100 width 5	Tabu Beam 2.5-opt iter 100 size 50 width 5
Problem 25	best distance	664	657	657
	time	0.088514566	1.411660671	1.494745493
Problem 26	best distance	704	662	625
	time	0.047587872	11.55299807	7.293416977
Problem 27	best distance	706	676	676
	time	0.116880894	0.682408094	0.674106836
Problem 28	best distance	683	577	577
	time	0.070104837	10.00823689	2.64825201
Problem 29	best distance	605	605	605
	time	0.039534807	16.06260228	2.259266853

Figure 6: Tabu Beam search with $\beta = 5$

The first comparison's result are in Figure 6. With beam width of 5, Tabu Beam does not seem to give better results than simple Beam search in most cases. However, Tabu Beam reaches terminating condition much faster compared to Beam search.



		Hill climbing 2.5-opt	Beam search 2.5-opt iter 100 width 10	Tabu Beam 2.5-opt iter 100 size 50 width 10
Problem 25	best distance	664	657	655
	time	0.088514566	19.76223016	28.39361954
Problem 26	best distance	704	625	599
	time	0.047587872	24.7884059	25.89411712
Problem 27	best distance	706	647	634
	time	0.116880894	17.58623958	12.32661223
Problem 28	best distance	683	654	641
	time	0.070104837	29.06178284	25.70584631
Problem 29	best distance	605	550	580
	time	0.039534807	37.56542706	41.34595585

Figure 7: Tabu Beam search with $\beta = 10$

The second comparison's result are in Figure 7. With beam width of 10, Tabu Beam can discover better result than simple Beam search, at the cost of even higher computing time.

From a wide variety of comparison, we can safely conclude that Tabu heuristic can greatly improve Beam search, and beam width is the decisive factor for the algorithm.

2.6 Simulated Annealing

Simulated annealing is a better version of hill climbing. It's not always trying to find a better state than the previous one. Sometimes, a worse state is acceptable with a probability decreasing by time.

In this algorithms, instead of using k-opt technique as hill climbing algorithms above (which we can do that), we will use something else different: random walk. This method create the new neighbor as the result of one of the mutating functions below with equal probability:

- inverse(route): invert the order between 2 cities in the route
- swap(route): swap two arbitrary cities
- insert(route): select random point j in the route and insert it to the i^{th} position
- swap_routes(route): select a subroute $[a : b+1]$ and insert in at another position

The acceptance probability decreases proportionally to the decrease of time, also the less different the new state is, the more likely it will be accepted.

Those are the intuitive reasons behind how the acceptance probability is:

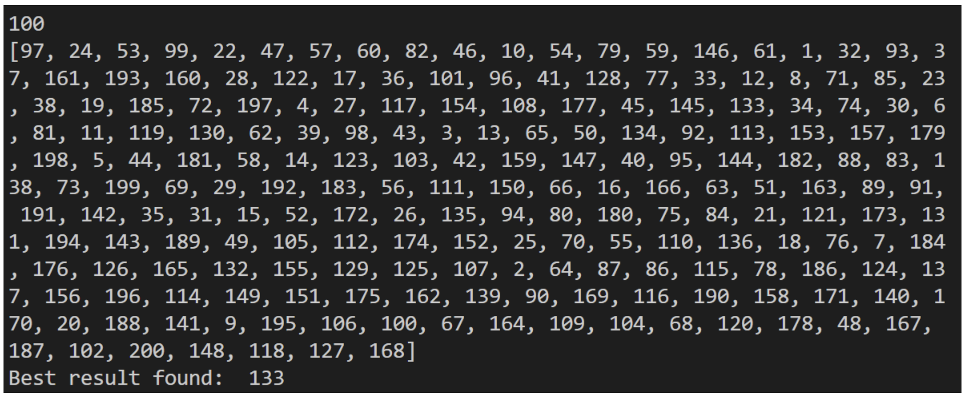
$$p = e^{\frac{-\Delta}{T}}$$

where:

Δ: the difference between current state and the new one

T: the temperate variable which decrease by time.

This algorithm performs better than Hill climbing since it produces routes that tend to the global optimum.



```
100
[97, 24, 53, 99, 22, 47, 57, 60, 82, 46, 10, 54, 79, 59, 146, 61, 1, 32, 93, 3
7, 161, 193, 160, 28, 122, 17, 36, 101, 96, 41, 128, 77, 33, 12, 8, 71, 85, 23
, 38, 19, 185, 72, 197, 4, 27, 117, 154, 108, 177, 45, 145, 133, 34, 74, 30, 6
, 81, 11, 119, 130, 62, 39, 98, 43, 3, 13, 65, 50, 134, 92, 113, 153, 157, 179
, 198, 5, 44, 181, 58, 14, 123, 103, 42, 159, 147, 40, 95, 144, 182, 88, 83, 1
38, 73, 199, 69, 29, 192, 183, 56, 111, 150, 66, 16, 166, 63, 51, 163, 89, 91,
 191, 142, 35, 31, 15, 52, 172, 26, 135, 94, 80, 180, 75, 84, 21, 121, 173, 13
1, 194, 143, 189, 49, 105, 112, 174, 152, 25, 70, 55, 110, 136, 18, 76, 7, 184
, 176, 126, 165, 132, 155, 129, 125, 107, 2, 64, 87, 86, 115, 78, 186, 124, 13
7, 156, 196, 114, 149, 151, 175, 162, 139, 90, 169, 116, 190, 158, 171, 140, 1
70, 20, 188, 141, 9, 195, 106, 100, 67, 164, 109, 104, 68, 120, 178, 48, 167,
187, 102, 200, 148, 118, 127, 168]
Best result found:  133
```

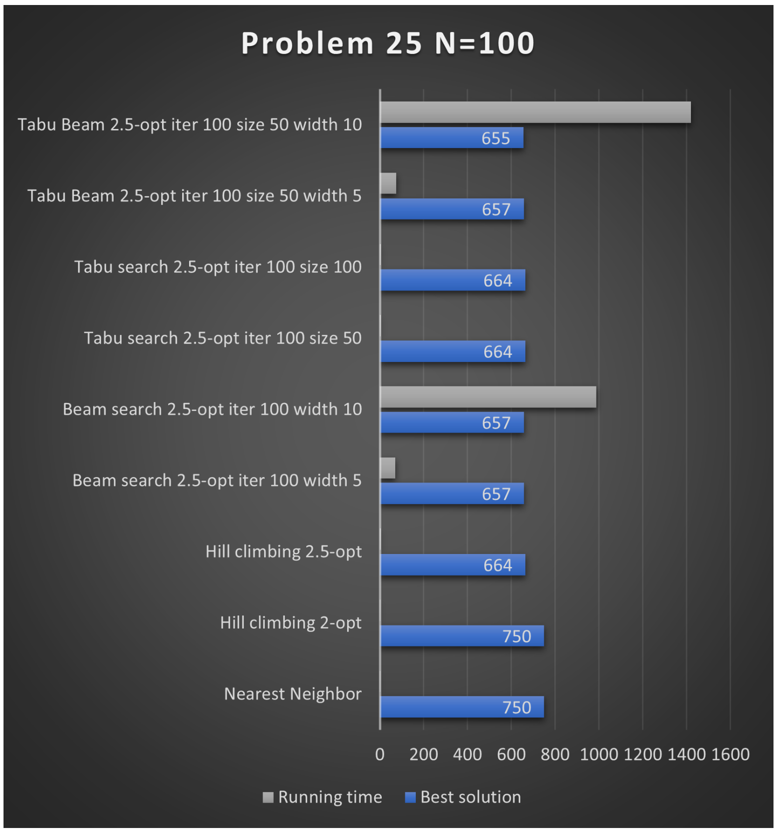
Figure 8: result for n = 100, time limit is 180s

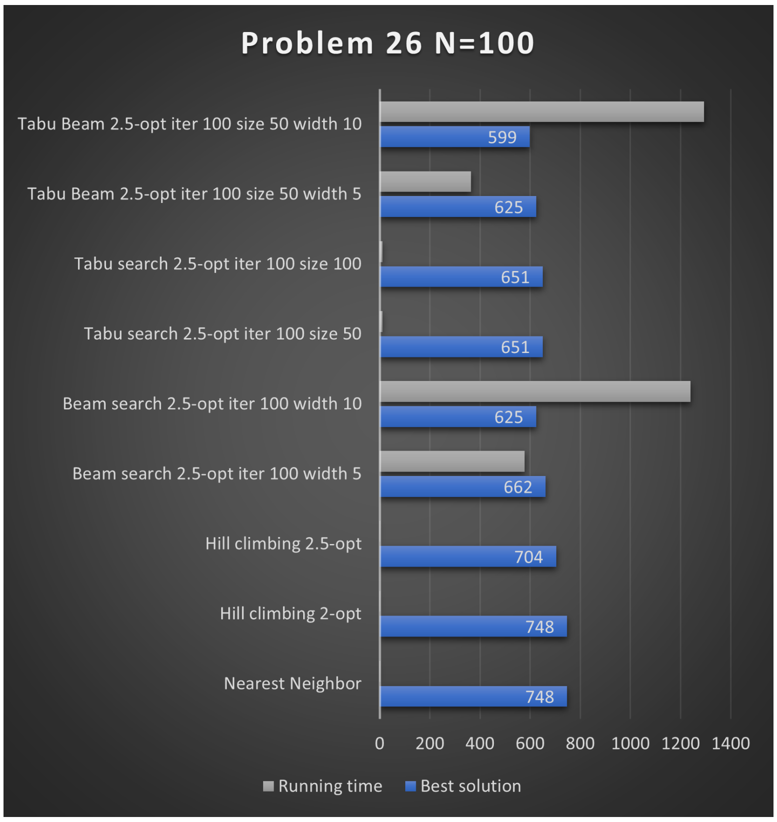
But there are some disadvantages:

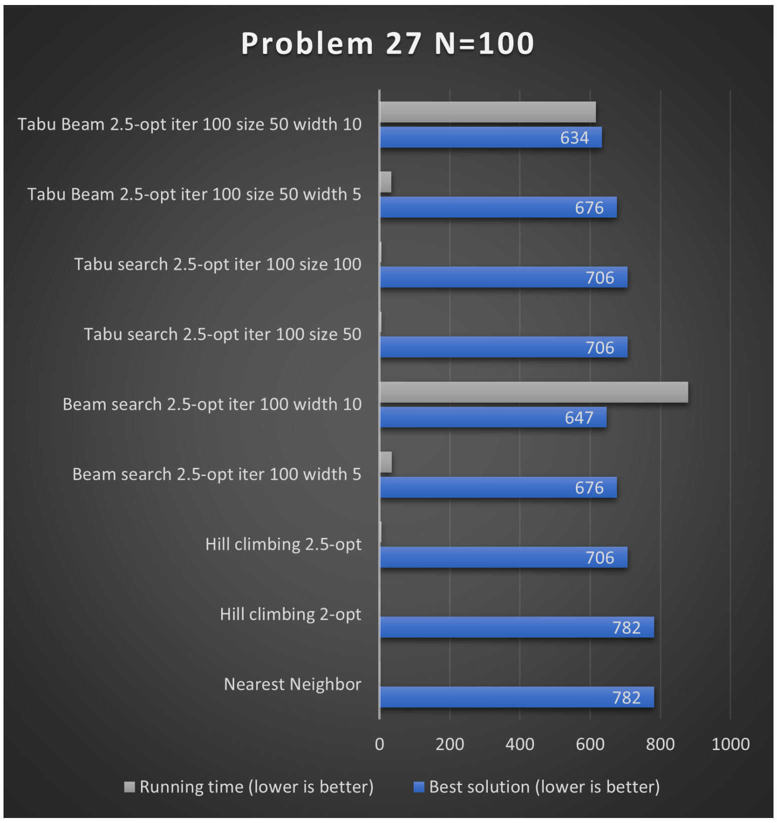
- The initial temperature and its decreasing speed need to be adjusted manually.
- Potential increase in computational time

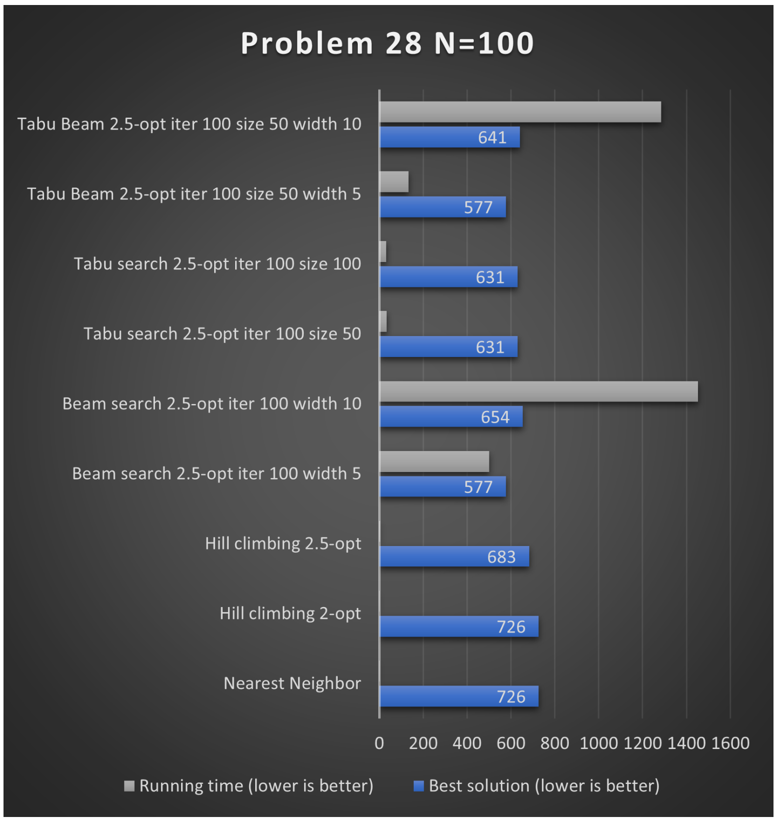
3 Result

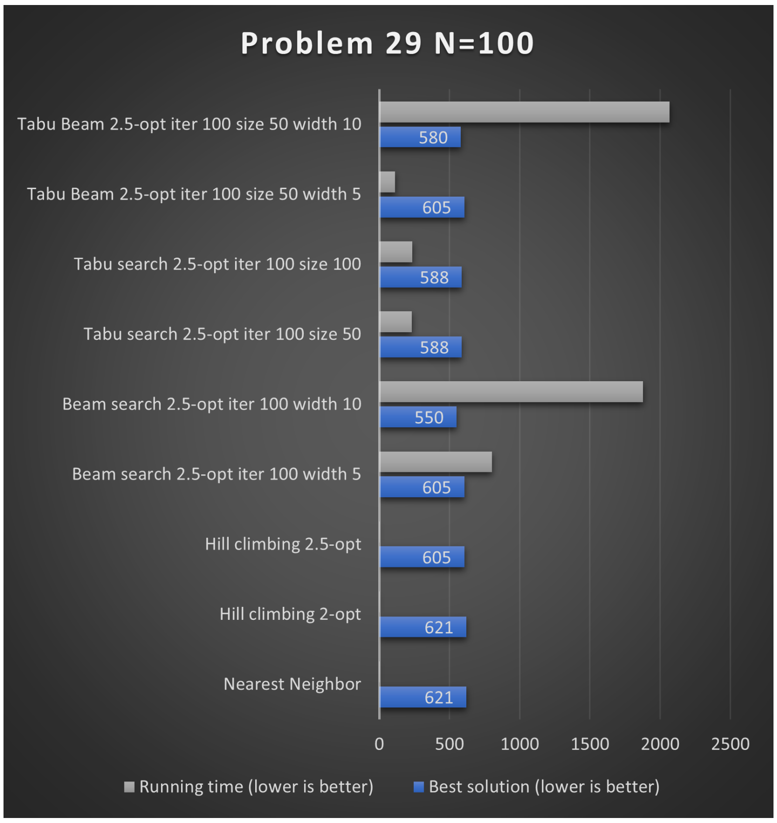
In order to compare some algorithms in terms of optimality and computational time, we had ran more test (with $N = 100$) and had the result as the figures on next pages. Given the same 2.5-opt neighborhood searching method and tabu list sized 50, Tabu-Beam gave the best solution with the longest computing time. Beam search variants took insanely larger amount of time compared to other concerned algorithms. Finally, Simulated Annealing could not guarantee good-and-consistent-enough results to be put on comparison, though the algorithm can tend to optimal solutions given plenty of running time (and luck).


Problem 25 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
Tabu Beam 2.5-opt iter 100 size 50 width 5
Tabu search 2.5-opt iter 100 size 100
Tabu search 2.5-opt iter 100 size 50
Beam search 2.5-opt iter 100 width 10
Beam search 2.5-opt iter 100 width 5
Hill climbing 2.5-opt
Hill climbing 2-opt
Nearest Neighbor
655
657
664
664
657
657
664
750
750
0 200 400 600 800 1000 1200 1400 1600
Running time
Best solution


Problem 26 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
Tabu Beam 2.5-opt iter 100 size 50 width 5
Tabu search 2.5-opt iter 100 size 100
Tabu search 2.5-opt iter 100 size 50
Beam search 2.5-opt iter 100 width 10
Beam search 2.5-opt iter 100 width 5
Hill climbing 2.5-opt
Hill climbing 2-opt
Nearest Neighbor
599
625
651
651
625
662
704
748
748
0 200 400 600 800 1000 1200 1400
Running time
Best solution


Problem 27 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
634
Tabu Beam 2.5-opt iter 100 size 50 width 5
676
Tabu search 2.5-opt iter 100 size 100
706
Tabu search 2.5-opt iter 100 size 50
706
Beam search 2.5-opt iter 100 width 10
647
Beam search 2.5-opt iter 100 width 5
676
Hill climbing 2.5-opt
706
Hill climbing 2-opt
782
Nearest Neighbor
782
0 200 400 600 800 1000
Running time (lower is better)
Best solution (lower is better)


Problem 28 N=100
Tabu Beam 2.5-opt iter 100 size 50 width 10
641
Tabu Beam 2.5-opt iter 100 size 50 width 5
577
Tabu search 2.5-opt iter 100 size 100
631
Tabu search 2.5-opt iter 100 size 50
631
Beam search 2.5-opt iter 100 width 10
654
Beam search 2.5-opt iter 100 width 5
577
Hill climbing 2.5-opt
683
Hill climbing 2-opt
726
Nearest Neighbor
726
0 200 400 600 800 1000 1200 1400 1600
Running time (lower is better)
Best solution (lower is better)



4 Conclusion

This problem can be solved by various optimization techniques, from exact algorithms to heuristics. But there is a trade-off between accuracy and speed.

Exact algorithms only perform well with small N. Heuristic algorithms can only find the good local optima in an acceptable computing time. With metaheuristic algorithms, the solution could tend to the global optimal but also nothing is guaranteed.

In terms of solution optimality:

$$\text{Nearest Neighbor} < \text{LS}^2 \text{ heuristic} < \text{LS-based metaheuristic} < \text{CP} = \text{BnB}\ ^3$$

The project has given us a lot of precious insights and knowledge about Operations Research, about algorithms and professional tools. We had also studied valuable techniques that can be applied to our future problems. We have expanded our knowledge thanks to this mini-project.

We also would like to thank our supervisor, Dr. Bui Quoc Trung, for being one of the most supportive teacher, and has given us valuable knowledge about the problem and some mistakes in our presentation that help us getting better.

[2] Local Search

[3] Originally in the presentation, we generalized Greedy and mistook OR-Tools as algorithms. Our supervisor had pointed out the mistakes in this conclusion and what we now report is the corrected conclusion.