CS271P Introduction to Artificial Intelligence

Project : Traveling Salesman Problem

Final Report

Team 31

Name	ID
Antonio Rodriguez	41049941
Eric Lee	49569949
Liwei Luo	20322800

I. Introduction to the Problem

The traveling salesman problem (TSP) is an optimization problem where the goal is to determine the shortest route between a set of locations that visits every location exactly once and returns to the origin location. The traveling salesman problem is an NP-hard combinatorial optimization problem with increasing complexity as you add more nodes in the problem.

II. Branch and Bound Depth First Search

Algorithm:

Heuristic:

Algorithm 1: Minimal Spanning Tree (MST)

Algorithm 2: Sum up minimal edges for each node (min_edge_sum)

Algorithm 3 (Baseline): Shortest Path with Step(SPS)

Initial Answer: greedy

Data structures:

```
class Edge (int start, end float weight)
class Disjoint_set (int rank, int father)
   find(self, x)
   union(self, x, y)
class Answer (list path, float distance)
   equal(self, other)
   last_node(self)
   first_node(self)
   add_node_copy(self, node, distance)
   dist_eqal(self,other)
```

We need class Edge and class Disjoint_set to generate the minimal spanning tree, and class Answer to hold the temporal path and the full circle path.

Pseudo-code:

```
function SORT_ALL_EDGES(n, distance)
   edge_list=[all edges whose start<end]
   edge_list.sort()
   return edge_list

function HEURISTIC_MST(end_node, node_set, distance)
```

```
    list heuristic_list
    set_number=node_set.size+1
    initialize mst_cost to 0
    Disjoint_set djs
    node_set_with_end=node_set.union({end_node})
    global all_edge_sorted
    for e in all_edge_sorted
        if both e.start and e.end are in node_set_with_end
            if djs.find(e.start) != djs.find(e.end)
                mst_cost+=e.weight
                djs.union(e.start, e.end)
                set_number-=1
                if set_number==1
                    break
    for node in node_set
        heuristic_list[node]=mst_cost
    return heuristic_list

function GET_SHORTEST_PATH_STEP(n, origin_dist):
    # dist[i][step][j]
    initialize all value in dist(n,n,n) to INF
    for i in range(n):
        for j in range(n):
            if(i!=j):
                dist[i][1][j]=origin_dist[i][j]
    for step in range(2,n):
        for i in range(n):
            for k in range(n):
                for j in range(n):
                    if dist[i][step][j] > dist[i][step-1][k] + dist[k][1][j]:
                        dist[i][step][j] = dist[i][step-1][k] + dist[k][1][j]
    return dist

function HEURISTICS_SPS(end_node, node_set, shortest_path_step):
    return shortest_path_step[end_node][len(node_set)]

function HEURISTIC_MIN_EDGE_SUM(end_node, node_set, distance)
    list heuristic_list,second_min_edge_list
    initialize edge_sum to 0
    node_set_with_end=node_set.union({end_node})
    for s_node in node_set_with_end
        min_edge1=min_edge2=INF
```

```
        for e_node in node_set_with_end
            if s_node!=e_node
                if distance[s_node][e_node]<min_edge1:
                    min_edge2=min_edge1
                    min_edge1=distance[s_node][e_node]
                elif distance[s_node][e_node]<min_edge2:
                    min_edge2=distance[s_node][e_node]
        second_min_edge_list[s_node]=min_edge2/2
        edge_sum+=(min_edge1+min_edge2)/2
    edge_sum-=second_min_edge_list[end_node]
    for node in node_set
        heuristic_list[node]=edge_sum-second_min_edge_list[node]
    return heuristic_list

function BNB-DFS(current_ans, best_ans, node_set)
    if node_set.size==1
        node=node_set.pop()
        total_distance=distance[current_ans.last_node][node]
                +distance[node][current_ans.first_node]+current_ans.distance
        if total_distance<best_ans.distance
            best_ans=Answer(current_ans.path+[node], actual_dist)
        return best_ans
    heuristic_for_node_set=HEURISTIC(current_ans.first_node, node_set, distance)
    for node in node_set
        lower_bound=heuristic_for_node_set[node]
                +distance[current_ans.last_node][node]+current_ans.distance
        if(lower_bound>=best_ans.distance)
            continue
        next_ans=Answer(current_ans.path+[node]
                ,distance[current_ans.last_node][node]+current_ans.distance)
        best_ans=BNB-DFS(next_ans, best_ans, node_set-{node})
    return best_ans

function GENERATE_INITIAL_ANSWER(n, distance)
    init_ans=Answer([],INF)
    for node in range(n)
        ans=Answer([node])
        node_set=set(range(n))-{node}
        while(node_set.size>0)
            min_dist=INF
            min_node=-1
            for node in node_set:
```

```
                if distance[ans.last_node][node]<min_dist
                    min_dist=distance[ans.last_node][node]
                    min_node=node
            ans=Answer(ans.path+[min_node],ans.distance+min_dist)
            node_set.remove(min_node)
        if ans.distance<init_ans.distance
            init_ans=ans
    return init_ans

function SOLUTION(n, distance)
    global all_edge_sorted=SORT_ALL_EDGES(n, distance)
    init_ans=GENERATE_INITIAL_ANSWER(n, distance)
    node_set=set(range(n))
    best_ans=BnB-DFS(Answer([0],0), init_ans, node_set-{0})
    return best_ans
```

Explanation:

Initial answer

We use a greedy algorithm to generate the initial answer by greedily finding the nearest unvisited node to the last node in the path. We search starting from every node and accept the answer with the minimal distance among them.

Branch and Bound DFS

First, we add an initial node to the empty path. We can start from any node since the answer is a circle, and here we use 0.
For branching, each step we choose an unvisited node(nodes in *node_set* in code) and add it to the existing path. The heuristic function estimates the distance of the shortest path starting from the new node, passing through all of the unvisited nodes once and ending with the first node in the path. If the heuristic value shows that the lower bound of this branch is no less than the current best answer, then prune this branch.
When we reach all of the nodes (node_set.size = 1) and go back to the first node (also called end_node in the code because of the circle), update the best answer if we get a better distance.
We calculate the heuristic value for all k unvisited nodes at one time before branching, to minimize the time complexity.

Minimal spanning tree heuristics

Since the TSP problem is to find the Hamiltonian Circle with shortest cost in a graph, and a Hamiltonian Path is a special spanning tree of the graph, the cost of the minimal spanning tree will be no more than the cost of a shortest Hamiltonian Path from one node to another node.

We can use the cost of the minimal spanning tree of a node set (called *node_set_with_end* in code) including nodes unvisited and the *end_node* as the lower bound of the shortest Hamiltonian Path starts from any node in node_set and ends with the *end_node.*
We first sorted all the edges ascending in the processing period, and used a disjoint set to build the minimal spanning tree for *node_set_with_end*.
Then for every remaining node:

h(node) = mst_cost

Sum up minimal edges heuristics

For a Hamiltonian Path, every internal node has two connected edges and the node at the front and the end have one. The sum of the shortest two edges of one node is not greater than the two connected edges in the shortest Hamiltonian Path.
Then for every remaining node:

h(node) = ½(shortest edge of this node + shortest edge of the *end_node* + sum of the shortest two edges of other nodes)

Shortest Path with Step heuristics (Baseline)

Suppose we have a matrix **sps** holding the shortest path's length from node u to v in s steps. The shortest Hamiltonian Path from start node to end node in the subset with k nodes is a special case of path from start node to end node in k steps, so **sps**[start][end][k] can be an admissible heuristic.

Evaluation:

Time and Space Complexity:

k=node_set.size, the number of remaining nodes not included in the path

Table 1.1: Time and Space Complexity

Algorithm	Avg time for Heuristic per node	Preprocessing time	Extra Space
MST	Worst Case O(n^2/k)	O(n^2 log n)	O(n^2)
Min_edge_sum (Edge)	O(k)	/	O(n)
Greedy(init ans)	/	O(n^3)	O(n)
Shortest Path with Step (SPS)	O(1)	O(n^4)	O(n^3)

Since in the worst case, we may need to iterate through every edge in the sorted list (size n^2) to build the minimal spanning tree in k nodes, the total time in the worst case is O(n^2). Average time in the worst case is O(n^2/k). And the preprocessing time of sorting the edges is O(n^2(log n^2)), which means O(n^2 log n). The extra space is taken by the sorted list (size O(n^2)) and the disjoint set (size O(n)).

The Min_edge_sum (Edge) algorithm takes O(k^2) time to find the shortest two edges for each node in the subgraph, so the average time is O(k). No preprocessing time needed. The extra space is taken by the list recording the shortest two edges for each node (size O(n))

For the greedy algorithm we use to generate the initial answer, it takes O(n^2) time to find a complete answer, and we try to start from every node, which is O(n^3) in total. Extra space is used by storing the current answer and the best answer, which take O(n) spaces.

The Shortest Path with Step (SPS) algorithm takes O(n^3) to get the shortest path with k+1 step between all node pairs from the k step, and we should iterate k from 1 to n. Hence the total preprocessing time is O(n^4). The heuristic function only needs to return the value in **sps** matrix, so its O(1).

Benchmark Analysis:

We examine four heuristic algorithms: Trivial, Shortest Path with Step(SPS), MST, Minimal edge sum (Edge), with the two test cases given on canvas, cases with ground truth found on the internet, and cases generated by given script in different parameters. We record total time, preprocessing time, total time executing heuristic algorithm, number of times to compute the heuristic value, and calculated heuristic execution time. The number of times to compute the heuristic value can also indicate the effect of heuristic algorithm pruning. Furthermore, to deeply understand the advantages and disadvantages of the MST and the “Min edge sum” algorithms, we compare the heuristic value generated by these 2 algorithms.

Table A.2 shows little difference in total time due to the number of cities being too small. But Table A.3 shows significant differences in total time. Our baseline SPS pruned about 70% of the branches compared to the trivial algorithm, which shows its efficiency over trivial design and is suitable to be a baseline. MST pruned 89% and Edge pruned 92% compared to the baseline.

We tested the algorithms on problem sets with known solutions to verify the correctness of our algorithm. We measure the effectiveness of pruning and time of calculating the heuristic value. Table A.4 shows one of the performance results of a 11 city case (included in source files). When the total time grows, the preprocessing time becomes only a small fraction of the total time. Although the “Min edge sum” algorithm prunes more branches than MST, its average heuristic time is greater than that of MST. The “Min edge sum” algorithm has worse total heuristic time and total time than MST in this test case.

To analyze the performance of different algorithms when increasing the number of cities, we generated 50 random cases for each n (number of cities) starting from 10 and collected the average execution time of the algorithms. Table A.5 shows that the execution time of the trivial algorithm grows exponentially and performs poorly when n grows more than 15. The execution time for our baseline (SPS) also grows exponentially, but works well until n grows larger than 17, albeit slower than the trivial algorithm. We tested and compared MST and Min edge sum when n is larger than 17. Table A.5 shows that the execution time of MST and Min edge sum grow relatively slowly. And the “Min edge sum” is able to find solutions in a reasonable time when n less than or equal to 25. Moreover, Min edge sum generally has a better average execution time than MST.

However, the heuristic value calculated by Min edge sum is not always better than MST. In Table A.7, MST+Edge calculates both heuristic algorithms and returns the minimum found between the two. The table shows that Edge is better than MST in many cases, but still can underperform compared to MST. Although MST+Edge does a better job of pruning than separate MST and Edge algorithms, it spends too much time calculating the heuristic value. The total time is worse than the Edge algorithm. So there is no benefit to combining these two algorithms.

Improvement

There are various branch-and-bound algorithms others have created that can be used to process TSPs containing 40–60 cities. If using techniques reminiscent of linear programming, it can solve up to 200 cities. Current world record is achieved using the branch-and-cut method, which is a Branch and Bound algorithm using linear programming. This algorithm is able to solve an instance with 85,900 cities.

III. Stochastic Local Search

Algorithm:

Random-Restart 2-Opt Local Search

Pseudo-code:

function RANDOM_SOLUTION(n)
    solution = list(range of n)
    Random.shuffle(solution)
    return solution

function DISTANCE_EVALUATION(tsp_matrix, n1, n2, n3, n4)
    return tsp_matrix[n1][n3] + tsp_matrix[n2][n4] - tsp_matrix[n1][n2] - tsp_matrix[n3][n4]

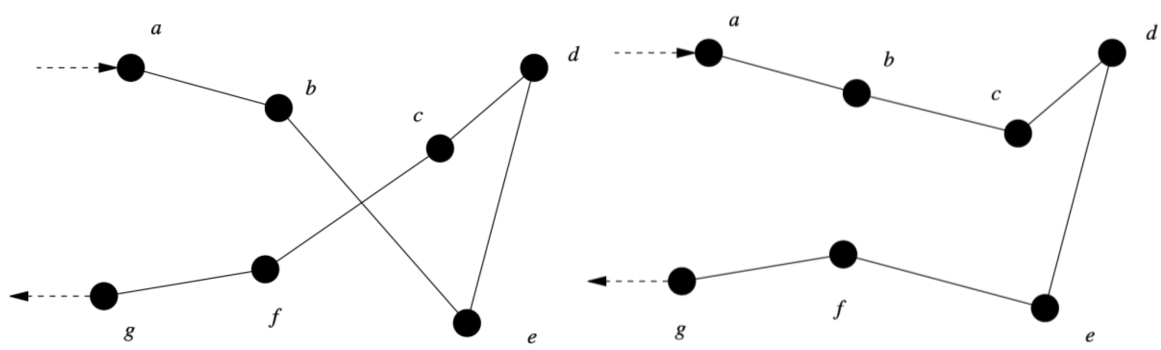
function CALCULATE_DISTANCE(tsp_matrix, solution)
    distance = 0
    for i = range(length of solution) **do**

```
        distance += tsp_matrix[solution[i-1]][solution[i]]

function 2-OPT(current_solution, tsp_matrix)
    best_solution = current_solution
    while ImrovementFound do
        ImprovementFound ← False
        for i = 1 to len(current_solution) do
            for j = i+1 to len(current_solution) do
                if j - i == 1 then continue
                if DISTANCE_EVALUATION(tsp_matrix, best_solution[i-1],
best_solution[i],  best_solution[j-1], best_solution[j]) < 0 then
                    ImprovementFound ← True
                    best_solution[i:j] = best_solution[j-1:i-1:-1]
    return best_solution

function RANDOM_RESTART(n, tsp_matrix, # of iterations)
    best_solution ← empty list
    best_distance ← INF
    for i in range(# of iterations) do
        solution ← RANDOM_SOLUTION(n)
        current_solution ← 2-OPT(solution, tsp_matrix)
        current_distance ← CALCULATE_DISTANCE(tsp_matrix, best_solution)

        if best_distance > current_distance do
            best_solution ← current_solution
            best_distance ← current_distance
    return best_solutIon, best_distance
```

Explanation:

The Stochastic Local Search algorithm used is the random-restart 2-opt local search algorithm. The algorithm will select a random initial state, traverse the neighborhood space with the 2-opt algorithm, guided by the objective function that will calculate the length of the tour. A two dimensional list will be used to store the traveling salesman problem, which will have the length of the edges. Lists will also be used to store the initial, current, and best solutions.

The 2-opt algorithm selects 2 edges from a route, swaps these edges with each other, and calculates the distance of the new route. It will update the current solution to the new route if the modified route yields a better solution. The algorithm will repeat these steps until all possible valid combinations of edges have been swapped and will return the best found results. In our rendition of the algorithm, the cost function will only consider the edges that change to make the evaluations faster than enumerating over the entire tour. If the difference between the sum of the new edges and the sum of the current edges is less than 0, we know that the new edges yield better results; therefore the current solution will be updated to the new path



The 2-opt algorithm alone may not yield great results as it may reach a local maxima. To alleviate this problem, we will utilize the random-restart wrapper function. This will perform a series of 2-opt searches from randomly generated initial states with each search running until it halts or makes no discernable improvements. In our case, we will repeat the search until the number of iterations, inputted by the user, has been reached. The random restart improves the chances of reaching a 'good' solution as we are attempting multiple trials with different initial states. The wrapper function will report the best result found across many 2-opt local search trials.

Evaluation:

Each iteration of the 2-opt algorithm has $n(n - 1) = O(n^2)$ possible moves as for each *n* edge, all the other $n - 1$ edges must be checked to explore the neighborhood of the current solution. With the restart wrapper, this will result in $O(n^3)$ time complexity. Since the only values stored are one dimensional lists, the space complexity is O(n).

Table 2.1

SLS	**Time Complexity**	**Space Complexity**
2-OPT Worst Case	O(n^3)	O(n)
2-OPT Best Case	O(n^2)	O(n)

Benchmark Analysis:

As the median of the distance found in the experiments show, one iteration of 2-opt search can easily be stuck at a local maxima. We can see that with increasing random restarts (number of iterations), the algorithm's performance increases and is able to escape some local maximas.

To test the optimality of the 2-opt local search algorithm, tests were performed on Reinelt's TSPLIB ATT48 problem set, which has 48 cities and a minimal tour length of 33523 (Appendix Table B.3). The test results show that the 2-opt algorithm fails to find the absolute best path, but it is able to find a tour with lengths that are on average at least 90% of the minimal length. At worst, the algorithm found a tour length that is 82.14% of the minimum (Appendix Tables B.6-B.9). Table below summarizes the 10 run average percentage comparing the lengths of the path found to the known minimal tour length, where performance = distance found / best distance.

Table 2.2

2-opt Iterations	**Average Performance**	**Min Performance**	**Max Performance**
1	91.87%	82.14%	97.69%
10	95.88%	92.22%	99.92%
100	97.76%	96.49%	99.01%
1000	99.12%	98.53%	99.69%

As shown in Table B.5, the 2-opt local search algorithm is able to find solutions for 1000 location tsp problem sets, but has difficulty running many random restart iterations.

The simplicity of the 2-opt local search operation makes it quick, allowing it to solve problems with a larger number of locations with good performance, but not always optimal. An improvement to the 2-opt search would be the Lin-Kerninghan algorithm, which is an adaptive algorithm choosing varying k edges to improve the tour instead of always selecting two edges. Other methods like simulated annealing are also able to find better solutions, however, with increasing complexity, they will be slower at finding solutions.

References

Contributors to Wikimedia projects. “2-Opt - Wikipedia.” *Wikipedia, the Free Encyclopedia*, Wikimedia Foundation, Inc., 8 Jan. 2007, https://en.wikipedia.org/wiki/2-opt.

Contributors to Wikimedia projects. “Travelling Salesman Problem - Wikipedia.” Wikipedia, the Free Encyclopedia, Wikimedia Foundation, Inc., 5 Dec. 2001, https://en.wikipedia.org/wiki/Travelling_salesman_problem.

Davendra, Donald, et al. “CUDA Accelerated 2-OPT Local Search for the Traveling Salesman Problem.” *Novel Trends in the Traveling Salesman Problem*, IntechOpen, 2020, http://dx.doi.org/10.5772/intechopen.93125.

geeksforgeeks.org. “Traveling Salesman Problem Using Branch And Bound - GeeksforGeeks.” GeeksforGeeks, 13 Oct. 2016, https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/.

Gerhard Reinelt, TSPLIB - A Traveling Salesman Problem Library, ORSA Journal on Computing, Volume 3, Number 4, Fall 1991, pages 376-384.

Johnson, David S., and Lyle A. McGeoch. “The Traveling Salesman Problem:” *Local Search in Combinatorial Optimization*, Princeton University Press, pp. 215–310, http://dx.doi.org/10.2307/j.ctv346t9c.13. (Johnson and McGeoch)

Venhuis, Mikko. “The Basics of Search Algorithms Explained with Intuitive Visualisations.” *Medium*, Towards Data Science, 12 Feb. 2019, https://towardsdatascience.com/around-the-world-in-90-414-kilometers-ce84c03 855.

Appendix

A. Branch and Bound Depth First Search Benchmark

Setup

Environment: Python 3.10.6, macOS Version 13.0.1, Apple M1 Pro Silicon
Concurrency: All algorithm are single-threaded

Results

Table A.2
file 5_0.0_10.0.out on canvas

Algo	Prep(s)	heuristic total(s)	total(s)	heuristic avg(s)	heuristic count
Trivial	0.000000	0.000004	0.002562	0.0000000954	40
SPS	0.000381	0.000002	0.002907	0.0000000851	28
MST	0.000010	0.000033	0.002538	0.0000023331	14
Edge	0.000000	0.000041	0.002538	0.0000029291	14

Table A.3
file 10_0.0_10.0.out on canvas

Algo	Prep(s)	heuristic total(s)	total(s)	heuristic avg(s)	heuristic count
Trivial	0.000000	0.008505	0.033032	0.0000005103	16667
SPS	0.008552	0.002538	0.018840	0.0000005091	4986
MST	0.000058	0.001337	0.004675	0.0000023919	559
Edge	0.000000	0.001306	0.004434	0.0000034545	378

Table A.4
file city-11.out on internet (include in source)

Algo	Prep(s)	heuristic total(s)	total(s)	heuristic avg(s)	heuristic count
Trivial	0.000002	0.221892	2.742390	0.0000000906	2448098
SPS	0.010904	0.271967	1.962626	0.0000001669	1629425
MST	0.000067	0.767873	0.995114	0.0000023689	324146
Edge	0.000002	0.806623	1.027532	0.0000027674	291473

Table A.5
mean=0.0, sigma=10.0, avg total time

n	10	11	12	13	14	15	16	17
Trivial	0.024122	0.094329	0.292985	0.942552	3.765129	11.326540	29.856279	/
SPS	0.017292	0.045887	0.111731	0.349114	1.192497	3.833157	10.972058	82.459250
MST	0.003457	0.007790	0.015197	0.032429	0.076116	0.190475	0.346072	1.195690
Edge	0.003683	0.007990	0.013106	0.027026	0.052694	0.120218	0.216565	0.724698

Table A.6
mean=0.0, sigma=10.0, avg total time

n	18	19	20	21	22	23	24	25
MST	1.806592	2.958599	5.087875	11.456235	35.670244	53.243729	106.623274	/
Edge	0.945622	1.579472	2.093594	5.109778	11.448007	12.419512	25.037371	41.718604

Table A.7
file saved_16_0.0_10.0.out, generated by script (include in source)

Algo	Prep(s)	heuristic total(s)	total(s)	heuristic avg(s)	heuristic count	MST better	Edge better
MST	0.000171	0.259612	0.338580	0.0000023538	110293	/	/
Edge	0.000001	0.173923	0.200222	0.0000038599	45059	/	/
MST+Edge	0.000127	0.265282	0.290375	0.0000062890	42182	6792	35390

B. SLS Benchmark

Setup

Environment: Python 3.9.6, macOS Version 13.0.1, Apple M2 Air

Results

Table B.1
File: 5_0.0_10.0.out - sample 10 runs

2-opt Iterations	Best Distance Found	Median Distance	Min Time (s)	Average Time (s)	Max Time (s)
1	33.3561	39.5631	6.79493E-05	0.000112486	0.000211954
10	33.3561	33.3561	0.000194788	0.000281572	0.000383139
100	33.3561	33.3561	0.001521826	0.002018356	0.002795935

Table B.2
File: 10_0.0_1.0.out - sample 10 runs

2-opt Iterations	Best Distance Found	Median Distance	Min Time (s)	Average Time (s)	Max Time (s)
1	2.6792	3.35845	0.000155926	0.000227332	0.000360012
10	2.6792	2.6792	0.001483917	0.001853704	0.002336025
100	2.6792	2.6792	0.009841204	0.013614178	0.017028809

Table B.3

File: ATT48.out - set of 48 cities from Reinelt's TSPLIB 10 runs

2-opt Iterations	Best Distance Found	Median Distance	Min Time (s)	Average Time (s)	Max Time (s)
1	34316	36002	0.004842997	0.007346869	0.010191917
10	33551	34863.5	0.035825014	0.045439887	0.053073883
100	33857	34264.5	0.279122829	0.298048711	0.30964613
1000	33628	33857	2.74971509	2.766633892	2.787267923

Table B.4

File: 100_0.0_1.0.out - generated 10 runs

2-opt Iterations	Best Distance Found	Median Distance	Min Time (s)	Average Time (s)	Max Time (s)
1	4.9847	6.3382	0.028286934	0.039922118	0.052544832
10	4.4149	4.7742	0.215761185	0.228683925	0.252962112
100	3.9053	4.3668	2.007338047	2.093109894	2.147711992
1000	3.7465	4.2153	20.74685192	20.98800342	21.38639808

Table B.5

File: 1000_0.0_1.0.out - generated 10 runs

2-opt Iterations	Best Distance Found	Median Distance	Min Time (s)	Average Time (s)	Max Time (s)
1	11.1265	11.6281	3.823704958	4.505968785	5.316113949
10	10.5733	10.93695	42.71259212	45.22228284	47.56056404
100	-	-	-	-	-

Table B.6

ATT48 1 iteration

Distance	Time (s)	Performance
36138	0.00725913	92.76%
35737	0.00602984	93.80%
36506	0.00520611	91.83%
35333	0.00857711	94.88%
34732	0.01019192	96.52%
40812	0.00705695	82.14%
34316	0.00870585	97.69%
36686	0.00858688	91.38%
39809	0.004843	84.21%
35866	0.00701189	93.47%

Table B.7

ATT48 10 iterations

Distance	Time (s)	Performance
35626	0.04725313	94.10%
33551	0.05307388	99.92%
34570	0.0497632	96.97%
35535	0.03582501	94.34%
36350	0.04454875	92.22%
35106	0.03820491	95.49%
34621	0.04364896	96.83%
34507	0.05012703	97.15%
34259	0.04205799	97.85%
35687	0.049896	93.94%

Table B.8

ATT48 100 iterations

Distance	Time (s)	Performance
33996	0.30964613	98.61%
34270	0.30521011	97.82%
34743	0.28936195	96.49%
34659	0.30641627	96.72%
33898	0.29329491	98.89%
33857	0.29658008	99.01%
34259	0.30820417	97.85%
34484	0.28425384	97.21%
34082	0.27912283	98.36%
34694	0.30839682	96.62%

Table B.9

ATT48 1000 iterations

Distance	Time (s)	Performance
33857	2.7744019	99.01%
33893	2.75783229	98.91%
33857	2.76358604	99.01%
33633	2.75369287	99.67%
33857	2.76416898	99.01%
33654	2.75065398	99.61%
33903	2.78726792	98.88%
34022	2.74971509	98.53%
33628	2.78089309	99.69%
33903	2.78412676	98.88%