Efficient Algorithms for Pokémon Go based on traditional TSP Problem

Wenxiang Lei
5245 Whiteoak Ave
Smyrna,GA 30080
1(404)579-7207
lwxstc@gatech.edu

Hao Wu
3223E Post Woods Dr
Atlanta, GA 30339
1(470)775-8647
hwu353@gatech.edu

Zhaohan Sun
100 10th Street NW
Atlanta, GA 30309
1(404)2329475
szhnkumath@gatech.edu

Hao Yan
765 Ferst Dr NE
Atlanta, GA 30319
1(678)907-3998
yanhao@gatech.edu

1. Introduction

Actually, our pokemon go problem is travelling salesman problem, which is that given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city.

TSP is a NP-hard problem, and we apply five algorithms to solve the problem: branch and bound, simulated annealing, hill climbing, approximation with minimum spanning tree and nearest neighbor. We will discuss our algorithms and experiment results on pokemon go data later in the article.

2. Problem definition

Given a collection of cities and the cost of travel between each pair of them, the traveling salesman problem, or TSP for short, is to find the cheapest way of visiting all of the cities and returning to your starting point. In the standard version we study, the travel costs are symmetric in the sense that traveling from city X to city Y costs just as much as traveling from Y to X. In the theory of computational complexity, the decision version of the TSP (where, given a length L, the task is to decide whether the graph has any tour shorter than L) belongs to the class of NP-complete problems.

TSP can be modelled as an undirected weighted graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's weight. It is a minimization problem starting and finishing at a specified vertex after having visited each other vertex exactly once.

3. Related work

3.1 Popular heuristics and import results

The nearest neighbor algorithm lets the salesman choose the nearest unvisited city as his next move. For N cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest possible path [4];

The Minimum Spanning Tree algorithm follows a similar outline but combines the minimum spanning tree with a solution of another problem, minimum-weight perfect matching. This gives a TSP tour which is at most 2 times the optimal [5].

3.2 Using Branch and Bound to solve tsp problem

The quality of Branch and Bound relies heavily on the calculation of the lower bound. The famous Herd-Karp lower bound is an even tighter lower bound.[2] We can alter graph in a special way so that we could generate different minimum-1-trees than the original. These different modifications result in different lower bounds. The tightest of these is the Held-Karp lower bound[3].

3.3 Using Local Search to solve tsp problem

The first local search algorithm is simulated annealing (SA), which has been successfully adapted to give approximate solutions for the TSP. SA is basically a randomized local search algorithm allowing moves with negative gain. A baseline implementation of SA for the TSP is presented in [4]. They use 2-opt moves to find neighboring solutions. Thus the resulting tours are comparable to those of a normal 2-opt algorithm. The second local search method is hill climbing, it will always try to find a better alternative to reach a solution [6]. In HC, the method will generate several states from an initial state, and then it will analyze all the alternatives, and find the best one. After that, it will move to that state.

3.4 Using Simulated Annealing to solve Traveling Salesman Problem

An efficient simulated algorithm for TSP problem is implemented referencing the paper [1]. Compared with other algorithms of local search.Simulated Annealing's advantage over

other methods is the ability to obviate being trapped in local minima. Eventually, SA produces a better solution than the original iterative improvement algorithms.

4. Algorithms

4.1 Branch and bound

4.1.1 Description

Our upper bound is best solution found in the search progress, we compare two lower bound methods in the literature. i) One is based on the matrix reduction method in the lecture. ii) The other one is based on MST bounding mentioned in the lecture. We traverse the solution in a depthward motion by recursion if its lowerbound smaller than the upper bound. In the searching process we compare two types of methods: i) DFS with choosing the most promising child to explore first. (DFSsmart) ii) DFS with random child to explore first

4.1.2 Pseudo code

Algorithm 1 Branch and Bound(DFSsmart)

```
graphData ← readData(inputFile)
best ← inf
partialSol ← [starting node]
remainNode ← [All nodes except starting node]
dfs(partialSol, remainNode, edges, distanceMatrix)
return best

def dfs (partialSol, remainNode, graphData ):
        global best
        if time > cuttime: return
        tempheap = []
        for i in range(0, len(remainNode)):
                partialSol_new = partialSol
                remainNode_new = remainNode
                partialSol_new.add(remainNode(i))
                remainNode_new.delete(remainNode(i))
                LB = computeLowerBound(partialSol, remianNode)
                tempheap.push((LB,partialSo_new, remainSol_new))
        while tempheap is not empty:
                lb, partialSol, remainSol = tempheap.pop
                if lb > best: return
                if len(remainNode) == 0:
                        best = lb
                        return
                dfs(partialSol, remainNode, graphData)
```

4.1.3 Data structure used

- Union-find
- List
- Set

4.1.4 Time and space complexity

- Time complexity: depends on how we compute the lowerbound
  - Matrix Lowerbound method: O(n* 2^n), n is the number of cities, and because we have 2^n subproblems and each subproblem is O(n).
  -
- Space complexity: O(n * 2^n), n is the number of cities, and because we have 2^n subproblems and each subproblem is O(n).

4.2 Local search

4.2.1 Simulated annealing

4.2.1.1 Description

First we start from a random solution and choose initial parameters, such as temperature, cooling rate and Tmin; Then use 2-opt to find its neighbours and update our solution if we find a better solution or its probability satisfies; Finally we terminate our program if the temperature is below Tmin.

4.2.1.2 Pseudo code

Algorithm 3 Simulated annealing

```
graphData ← readData(inputFile)
initialize T, Tmin, coolingRate
sol ← generate one solution randomly

while T > Tmin:
        find 2-opt neighbor
        if it is better than current sol: update sol
        elif possibility satisfies: update sol
        update p
        T = T*(1- coolingRate)
return sol
```

4.2.1.3 Data structure used

- List

4.2.1.4 Time and space complexity

- Time complexity: O(n*k), n is the number of cities, and k is the number of iterations. For each iteration, we calculate its 2-opt neighbor which cost O(n).
- Space complexity: O(n), n is the number of cities, and because we do not use distance matrix so space is not O(n^2), we just use O(n) space to store the nodes.

4.2.2 Hill climbing

4.2.2.1 Description

First we start from a random solution; Then use 2-opt to find its neighbors and update our solution if we find a better solution; Finally, we terminate our program if we could not find a better solution.

4.2.2.2 Pseudo code

Algorithm 4 Hill climbing

```
graphData ← readData(inputFile)
```

```
sol ← generate one solution randomly
findSmall ← True

while findSmall == True:
        findSmall = false
        find all 2-opt neighbor and if there is a better sol:
                update sol
                findSmall = True
return sol
```

4.2.2.3 Data structure used

- List

4.2.2.4 Time and space complexity

- Time complexity: O(n*k), n is the number of cities, and k is the number of iterations. For each iteration, we calculate its 2-opt neighbor which cost O(n).
- Space complexity: O(n), n is the number of cities, and because we do not use distance matrix so space is not O(n^2), we just use O(n) space to store the nodes.

4.3 Minimum spanning tree

4.3.1 Description

We first find an MST T of G, then double every edge of MST to obtain an Eulerian graph. Pick a root, find an Eulerian tour, T*, on this graph. Then Output the tour that visits vertices of G in the order of their first appearance in T*, let C be this tour.

4.3.2 Pseudo Code

Algorithm 4 Minimum spanning tree

```
G = graph of tsp
mst = computeMST(G)
mst_directed = convert mst to directed graph
eul = Eulerian circle of mst_directed
eul_list = all nodes in eul
eul_list = remove duplicate nodes in eul_list
return tour in eul_list
```

4.3.3 Data structure used

• Union-find

4.3.4 Approximation guarantee

1. $cost(T) \leq OPT$ (because mst is a lower bound of tsp)

2. cost(T*) = 2cost(T) (because every edge appears twice)

3. $cost(C) \leq cost(T^*)$ (because of triangle inequalities)

4. So, $cost(C) \leq cost(T^*) = 2cost(T) \leq 2OPT$

4.3.5 Time and space complexity

In Kruskal's algorithm step, assume there are m edges and n nodes, for each iteration, take the minimum distance edge that creates no cycles in the tree, thus total running time is m*log(n). In finding Eulerian cycle step, we can find it in O(V+E) time. Thus total time complexity of tsp-mst is m*log(n).

4.4 Nearest neighbor

4.4.1 Description

We first start with a beginning node, then find the node not yet on the path which is closest to the node last added and add to the path the edge connecting these two nodes. When all nodes have been added to the path, add an edge connecting the starting node and the last node added.

4.4.2 Pseudo Code

Algorithm 5 Nearest Neighbor

```
u=1
S1= V - {u}
S2= empty
while S1 is not empty:
    find closest node v in S1 to last added node u in S2
    append this edge (v, u) into Traveled set
remove v from S1
sum(edges in S2)
```

4.4.3 Data structure used

• No special data structure used

4.4.4 Approximation guarantee

• $NN(I)/OPT(I) \leq 0.5 * ([\log N] + 1)$

4.4.5 Time and space complexity

Since in each step, we need to find the nearest two nodes in the graph, this step cost O(n) time complexity, and there are n nodes, hence we need to repeat at most n times, thus, total time complexity is O(n^2). We need to store the graph, it cost O(n^2) space, and cost O(n) space for purpose of storing tour edges.

5. EVALUATION AND DISCUSSION

5.1 Platform

We use device 1 for producing trace files, solution file, result table of local search algorithm: Hill Climbing and Simulated Annealing.

	Device 1	**Device 2**

Processor	**2.5 GHz Intel Core i5**	**2.3 GHz Intel Core i7**
Ram	**8 G**	**8 G**
Language	**Python 2.7.10**	**Python 2.7.10**
IDE	**Pycharm**	**Vim**
System	**macOS Sierra**	**macOS Sierra**

5.2 Experimental Procedure

Since the repeated experiments will not change the result of Approximation and Branch and Bound (BnB) algorithm, these algorithms only have one result for each, the Branch and Bound algorithm is implemented by Device 2.

For the results table, HC and SA are run 10 times for each date on device 1. For plots, HC is run 50 times for Roanoke and Toronto on Device 1; SA is run 10 times for Roanoke and Toronto on Device 1. BnB runs only once since it is a deterministic algorithm.

Cutoff time: For the HC, SA and BnB algorithms, the user could choose the cutoff time as they want. If the user does not set the cut-off time manually, the algorithm will automatically set the cut-off time as 600 seconds.

5.3 Result and Discussion

5.3.1 Comprehensive Table

In order to compare the performance of different algorithm provided above, the author provided a Comprehensive Table. The table contains the result, relative error and running time of all the algorithm mentioned above.

Branch and Bound

Our branch and bound algorithm explore the most promising child first when using depth first search algorithms ('DFSsmart' in Figure 7). If we set the cut time as 600s, the algorithm find the optimal path (0% error) of Cincinnati (0.95s), UkansasState (0.06s), and Atlanta (600s). For all other city except Roanoke, it can reduce the error to less than 10%. For Roanoke, we extend the running time to 860s which is required for the algorithm output the first searched path.

We compare 'DFSsmart-MST', 'DFS-MST' and 'DFSsmart-Matrix' in Figure 7. The 'DFS' and 'DFSsmart' corresponds to different search methods. The 'MST' and 'Matrix' corresponds to different lower bound calculation metohds.

Effect of search method:

We compare the result with original dfs ('DFS') without choosing the most promising child first. We also plot the relative error (RelErr) in Figure 7. For small city such as 'Atlanta', choosing the most promising child may not help much. However, for mid or large city such as 'Champaign', choosing the most promising child helps a lot since it helps to find a much better starting points. For other types of search methods such as BFS, it fails to find any paths within 600s for both 'Atlanta' and 'Champaign'. Therefore, BFS method is not included in the comparison. This implies DFS with the most promising child ('DFSsmart') works best in general.

Effect of lower bound calculation:

We compare the MST based lower-bound ('MST') method and the matrix reduction based lower bound ('Matrix') method. We plot the error for 'Atlanta' and 'Champaign'. For small city, such as Atlanta, Cincinnati, UKansasState, 'Matrix' works better. However, for all other cities, 'MST' works better. The reason is that though MST requires more time to compute, however, it provides a better lower bound than the 'Matrix' method. Therefore, it works better for large graph.

Hill Climbing

Our hill climbing algorithm can make the final solution reach optimal solution very closely in 10% in a very short time, even though for large dataset such as Roanoke and Toronto, time of less than 2s can be still accepted.

Simulated Annealing

For small datasets, the error can be reduced to close to 7 in a short time. And for large datasets, the error can be reduced below 14% in 20s. For UKansasState data, the error is only 4.7%.

Nearest Neighbor

Our nearest neighbor search is straightforward, and simple, thus has very small time complexity, however, it has an acceptable result with 2-opt approximation. In case of Atlanta, relative error can reach 5%

Minimum Spanning Tree

This algorithm uses minimum spanning tree and Eulerian cycle to construct tsp tour, in our experiment, the best relative error can reach 11% in the case of UkansasState.

5.3.2 QRTDs

To compare the performance of two local search algorithms, three types of figures have been showed below, Qualified Runtime for various solution qualities (QRTDs), Solution Quality Distributions for various run-times (SQDs) and Box plots for running times. The seeds of those range from 1 to 50 to run 50 times for each dataset in Hill Climbing algorithm and run 10 times for each dataset in Simulated Annealing algorithm.

Both the HC and SA algorithm have demonstrated that our algorithms can reach the standard earlier at a lower number of q.Compare figure 1 and figure 2, for Roanoke and Toronto datasets, HC can get a better solution in a shorter time than SA.

The figure 1 and figure 2 shows that for a given run-time, the higher the solution quality, the smaller the probability we will get the desired solution. On the other hand, the probability of getting a solution of certain quality increases as the run-time goes up.

5.3.3 SQDs

SQDs has also been used for the evaluation. Instead of fixing the relative solution quality and varying the time, we fix the time and vary the solution quality. The HC algorithm will get to a

local optimal solution in short time, and the Simulated Annealing will keep improving the result in a relative longer time. See figure 3 and 4 for the results.

5.3.4 Box Plots

To compare the variance of the result, we also generated the box plots for them. See figure 5 and 6 for the results. Since the local search algorithm are randomized, there will be some variation in the running times. Boxplots show this variation. For a given quality, the running time needed to achieve that quality varies in each run. From figure 5 and 6, we can see that there exists one or two outliers in terms of the running time, but the variation is not too much.

6. Conclusion

In conclusion, we compare different methods of solving the travelling salesman problem with different graph size.

BnB works best for small graph such as CinCinnati and UkansasState (with 10 nodes) since it is able to find the optimal solution exactly. For Atlanta (with 20 nodes), it takes around 400s to find the optimal solution. However, as the graph size becomes larger (more than 50 nodes), due to the limited time to search all the branches, the RelErr increases faster than other approximation methods and it takes much longer time to compute. Furthermore, for BnB, we compare different search methods (DFS, DFSsmart) and different lower-bound computation methods (MST, Matrix). We conclude that search method DFSsmart and lower-bound method MST works the best in general..

For the local search algorithm, the result would be useful as it could yield better result compared with approximation and run in less time compared with the Branch and Bound. Through the experiment of the two local search algorithm we could say that Simulated Annealing and Hill Climbing are stable with efficiency. Also, for small and mediate datasets, Simulated Annealing can achieve a better solution in a shorter time than Hill Climbing. But for a very large dataset, such as Roanoke, HC can obtain a better solution than SA in a shorter time.

For approximation algorithm, if we don't care much about the accuracy, we can sacrifice accuracy for time by using this algorithm. In our experiment, most of the relative error results fall in the interval [10%, 20%], which means our algorithm won't lose much accuracy in most cases.

7. References

[1] Hüsamettin Bayram, and Ramazan Şahin(2013), A NEW SIMULATED ANNEALING APPROACH FOR TRAVELLING SALESMAN PROBLEM, Mathematical and Computational Applications, Vol. 18, No. 3, pp. 313-322

[2] Held, M.; Karp, R. M. (1962), "A Dynamic Programming Approach to Sequencing Problems", Journal of the Society for Industrial and Applied Mathematics, 10 (1): 196–210, doi:10.1137/0110015.

[3] Volgenant, Ton & Jonker, Roy, 1982. "A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation," European Journal of Operational Research, Elsevier, vol. 9(1), pages 83-89, January.

[4] Johnson, D. S.; McGeoch, L. A. (1997). *"The Traveling Salesman Problem: A Case Study in Local Optimization"(PDF).* In Aarts, E. H. L.; Lenstra, J. K. Local Search in Combinatorial Optimisation. London: John Wiley and Sons Ltd. pp. 215–310.

[5] Hoffman, A. J., et al. *The traveling salesman problem: a guided tour of combinatorial optimization*. J. Wiley & Sons, 1986.

[6] Riera-Ledesma, Jorge, and Juan José Salazar-González. "*A heuristic approach for the travelling purchaser problem.*" *European Journal of Operational Research* 162.1 (2005): 142-152.

APPENDIX

Dataset	**Branch & Bound**	**Minimum Spanning tree**	**Hill Climbing**	**Simulated Annealing**	**Nearest Neighbor**

	Time	Result	RelErr	Time	Result	RelErr	Time	Result	RelErr	Time	Result	RelErr	Time	Result	RelErr
Roanoke	850	835198	0.274	1.3119	788046	0.20	1.7083	726219	0.108	15.4079	747328.5	0.140	0.636	840996	0.283
Toronto	600	1239726	0.054	0.3086	1708026	0.45	0.3746	1372076	0.167	7.9459	1309997.2	0.114	0.1478	1386622	0.1789
Atlanta	468.9	2003763	0.000	0.0141	2401697	0.19	0.0065	2197163.5	0.097	1.9742	2188998.3	0.092	0.0058	2117963	0.057
Boston	600	946109	0.059	0.454	1160719	0.29	0.0355	990351.7	0.108	3.4202	944295.6	0.057	0.0214	1115479	0.2484
Cincinnati	0.94	277952	0.000	0.0052	316626	0.13	0.0014	289968.7	0.043	1.2938	285562.7	0.027	0.0024	333791	0.2009
Denver	600	116183	0.157	0.1769	125879	0.25	0.1868	110607.4	0.101	6.6055	110784.1	0.103	0.0873	135430	0.3485
NYC	600	1743858	0.121	0.1246	2052894	0.32	0.1097	1763191.9	0.134	5.2534	1680015.8	0.080	0.0585	2008450	0.2916
Philadelphia	600	1421930	0.019	0.0264	1663662	0.19	0.0204	1455598.8	0.043	2.6915	1490173.7	0.067	0.0121	1691226	0.2115
Champaign	600	56520	0.074	0.0814	60490	0.14	0.0740	58312.7	0.108	4.5525	56143.8	0.067	0.0392	62920	0.1952
UKansas State	0.06	62962	0.000	0.005	70143	0.11	0.0014	66350.7	0.054	1.3205	65943.3	0.047	0.0024	74967	0.1907
UMissouri	600	156249	0.177	0.2843	171245	0.29	0.3156	150480.2	0.134	7.7201	145718.4	0.098	0.1431	164590	0.2402
SanFrancisco	600	946072	0.168	0.2748	1103804	0.36	0.2966	930292.3	0.148	7.0992	914320.3	0.129	0.1257	897275	0.1075

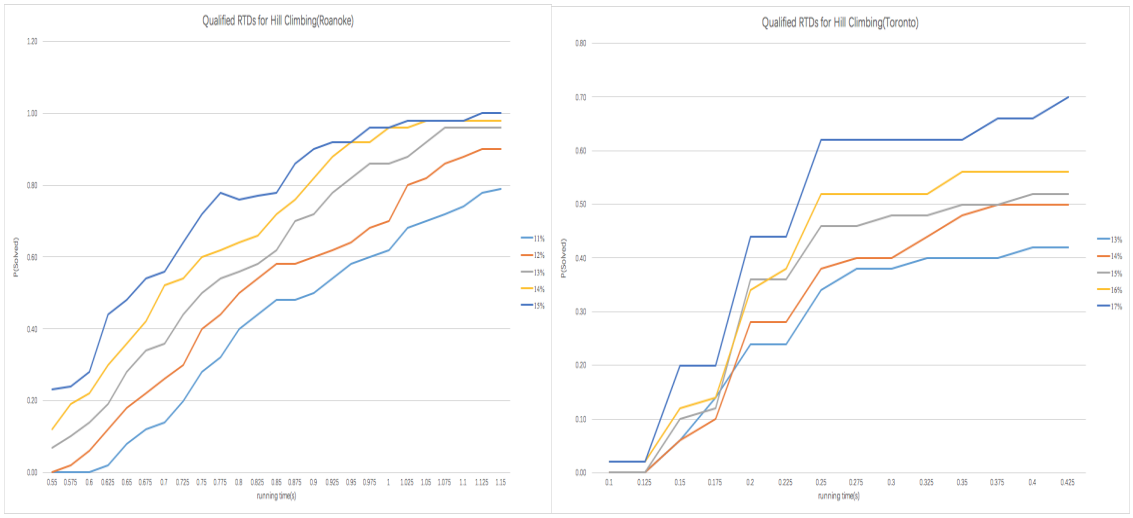


Figure 1. QRTDs for Hill Climbing

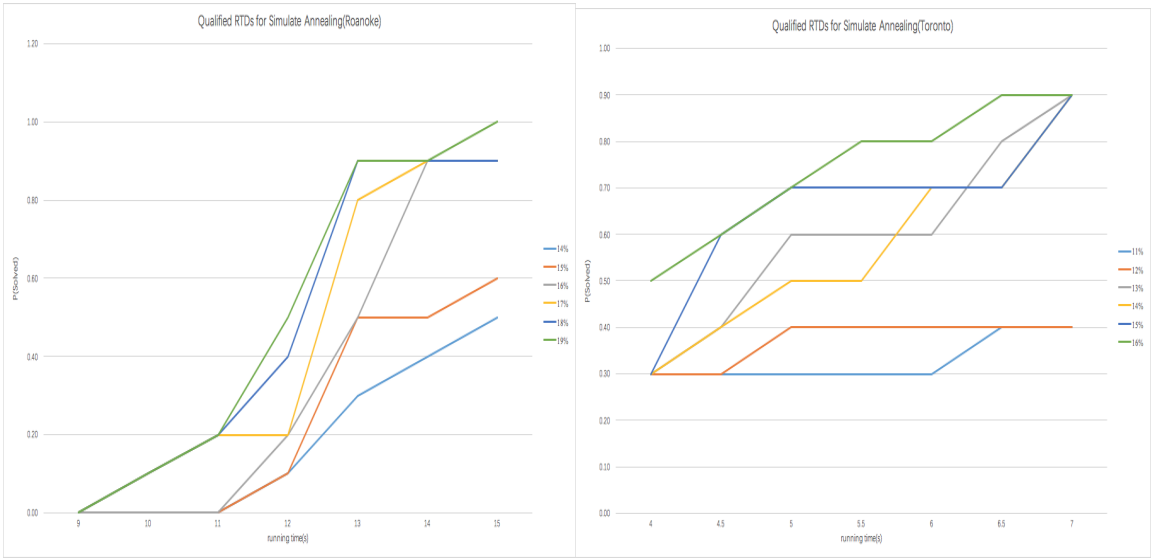


Figure 2. QRTDs for Simulated Annealing

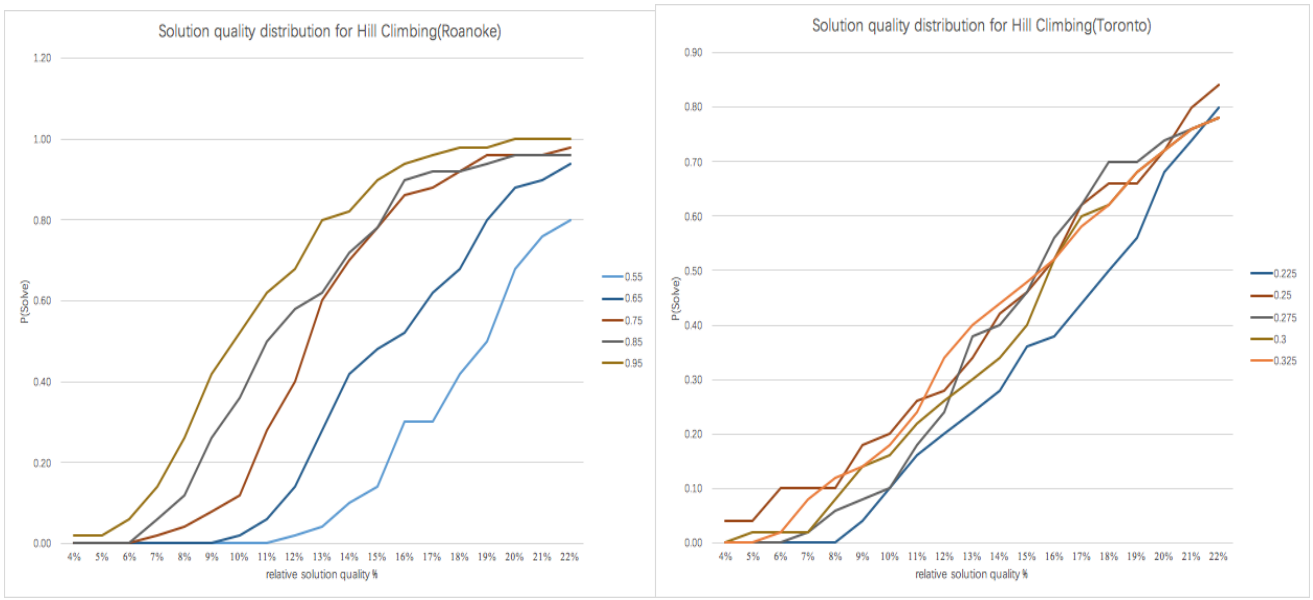


Figure 3. SQDs for Hill Climbing

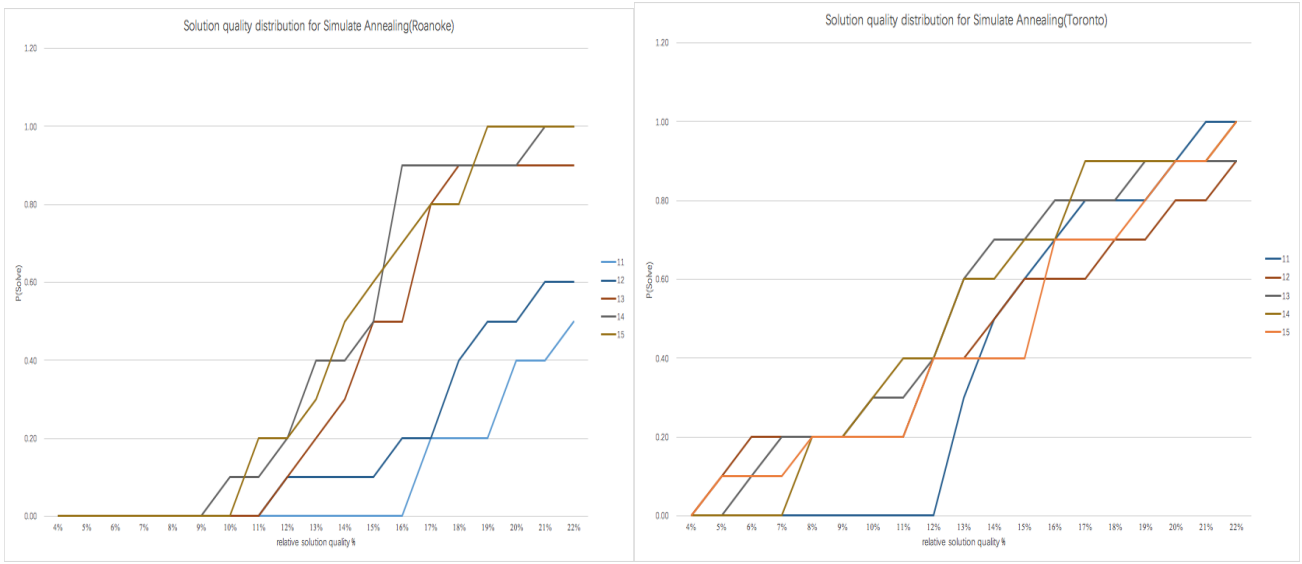
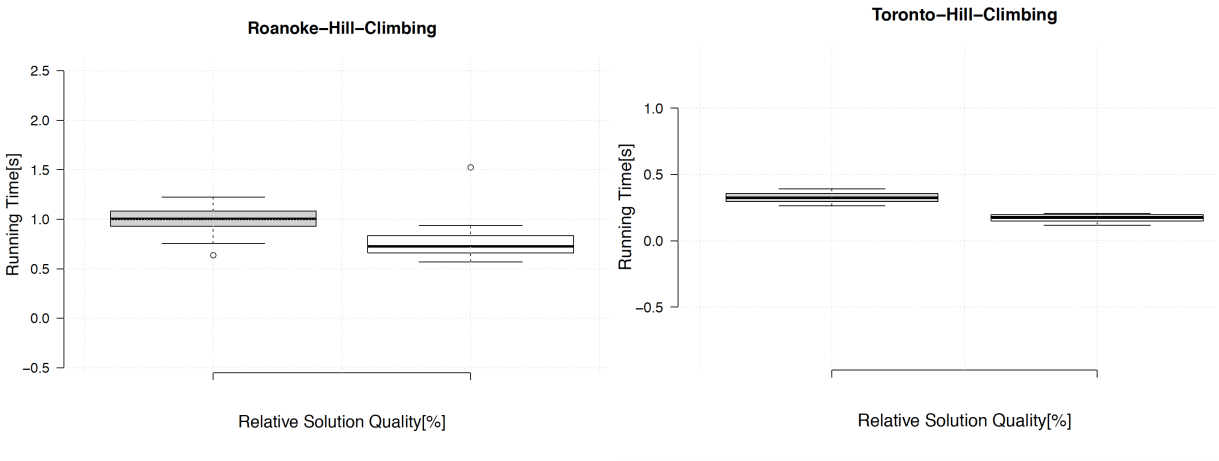
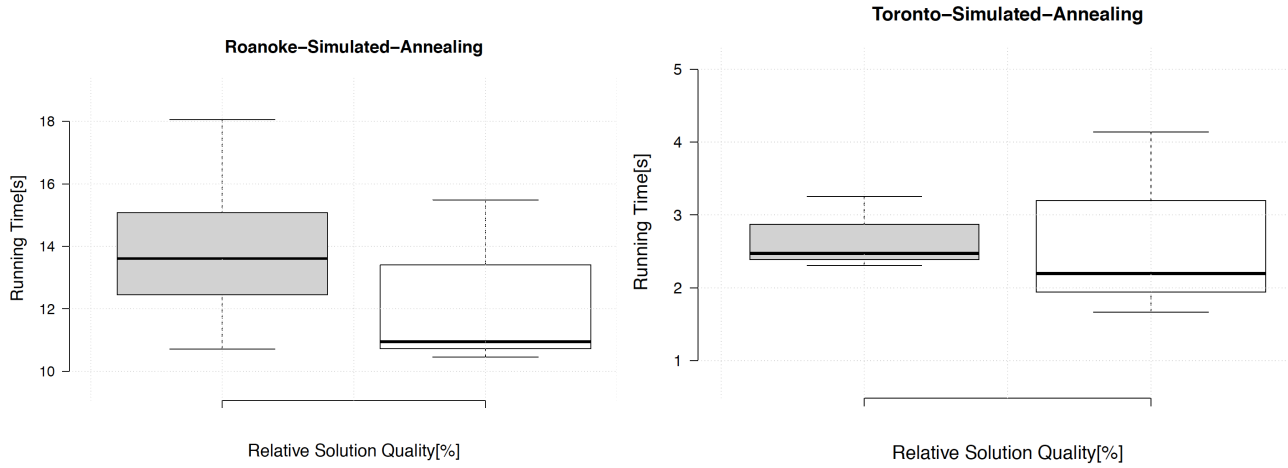


Figure 4. SQDs for Simulated Annealing



err.12 err.19 err.14 err.18

Figure 5. Boxplot for Hill Climbing



err.14 err.17 err.12 err.18

Figure 6. Boxplot for Simulated Annealing

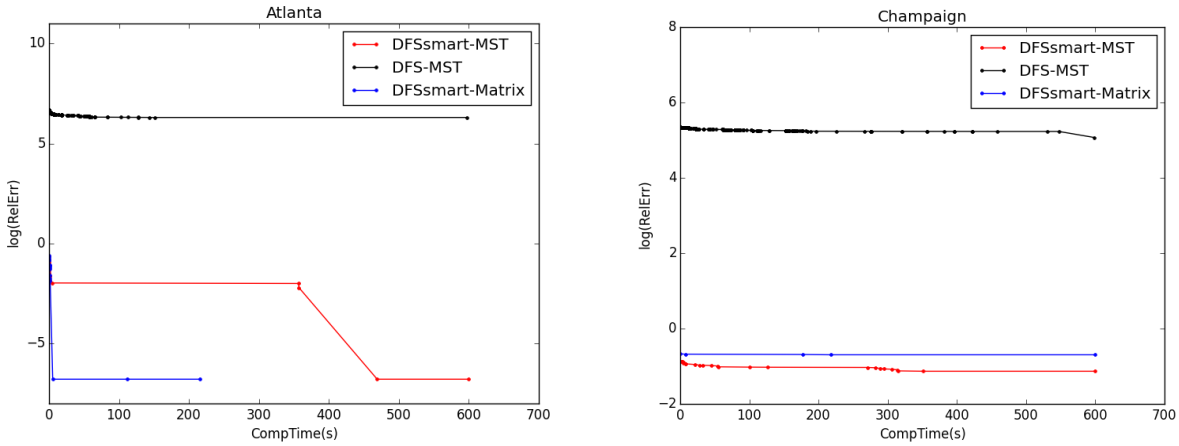


Figure 7. log(RelErr) vs Computation Time for Different Search Methods (DFS, or DFSsmart) and Different lower-bound computation (MST or Matri)