ARTIFICIAL INTELLIGENCE CSC 662

1441/42 H, 2019/20 G, Spring 2020

Solving Travelling Salesman Problem (TSP)

using A star, RBFS, and Hill-climbing algorithms

Detailed Report

Hamdi Altaheri
hamdi.altahery@gmail.com

Contents:

A. Algorithms: pseudocode and performance analysis

1. A* algorithm.

<u>Pseudocode of A* algorithm to solve TSP problem (tree search)</u>

```
function solution = solveTSP_Astar_treesearch(TSPGraph)  % returns a solution (path and cost)
    firstNode ← TSPGraph.getFirstNode                    % Get the start node (randomly).
    open ← []                                            % create an empty fringe list.
    startNode ← initializeStartNode(TSPGraph, firstNode) % initialize the start node.
    open.add(startNode)                      % add the start node to the open list (fringe list).
    solution ← Astar(TSPGraph, startNode, open)          % call A* algorithm to solve TSP.
    return solution           % if there is no solution (failure), the solution will be empty.
```

```
function solution = Astar(TSPGraph, startNode, open)     % returns a solution (path and cost)
    solution ← []                                  % create an empty solution (failure).
    while(open list is not empty)
        currentNode ← minCost(open)                % get the minimum cost node in the fringe.
        solution ← isGoal(currentNode, TSPGraph)   % check if the current node is the goal.
        if(solution is not empty) then return solution
        successors ← expandNode(TSPGraph, startNode, currentNode); % expand the current node.
        open.remove(currentNode)     % remove the current node form the open list (fringe).
        open.add(successors)         % add the successors (if there are) to the open list.
    return failure                   % return failure (empty solution) if no solution.
```

```
function successors = expandNode(TSPGraph, startNode, currentNode) % returns successors of node
    successors ← []         % create an empty successors list.
    % For TSP problem (each node is visited only once), so the successors of currentNode are
    % the unvisited nodes in the current path (path from the first node to currentNode).
    unvisited ← getUnvisitedNodes(TSPGraph, currentNode)
    % estimate cost for each successor
    for all unvisited do:
        % Estimate heuristic of successor h[n], which equal:
        node.h_n ← cost of the MST of the unvisited subgraph +
                   minimum cost to connect MST to unvisited subgraph +
                   minimum cost to connect MST to start node;
        % Calculate the distance from the start node to the successor g[n], which equal:
        node.g_n ← distance from the stat node to the current node +
                   the distance between the current node and successor;
        % Calculate the estimated cost from the start node to goal node throw the successor
        node.cost ← node.h_n + node.g_n
        node.parent ← currentNode
        node.depth ← currentNode.depth+1
        Add node to successors
    return successors
```

Time and space complexities of A* algorithm

The minimum spanning tree (MST) is a heuristic function when it is used for traveling salesman problem (TSP). MST finds the tree with minimum cost that visit all nodes at least once with zero cost for revisiting the node again. Therefore, the MST for node A will be always less than the actual cost form node A to the goal node, which make MST a heuristic function.

Since minimum spanning tree (MST) heuristic is admissible, A$*$ search (tree search) is complete and optimal. However, it has limitations in computation time and storage space. The time complexity of A$*$ is exponential based on the solution depth (d) and it keeps all generated nodes in memory, so the space complexity is also exponential based on (d). For the traveling salesman problem (TSP) the solution depth is at the maximum depth $(n-1)$, so it is impractical to use A* to solve TSP if the number of nodes is high. The detailed of A* complexity analysis is given below.

The complexity analysis of A$*$ depends on the state space of the problem. For a problem with a single reversible goal, the time complexity of A* is exponential in the maximum absolute error, as follow.

$$\boldsymbol{O}\left(\boldsymbol{b}^{\Delta}\right), \quad where\ b\ is\ the\ branching\ factor\ and\ \Delta\ is\ the\ maximum\ absolute\ error$$

$$\Delta \equiv h^{*}-h, \quad where\ h^{*}\ the\ actual\ path\ cost\ from\ the\ start\ node\ to\ the\ goal$$

$$if\ the\ relative\ error\ is\ defined\ as:\ \epsilon \equiv \frac{\Delta}{h^{*}}, \quad then\ (for\ constant\ step\ costs)$$

$$\boldsymbol{O}\left(\boldsymbol{b}^{\boldsymbol{\epsilon d}}\right), \quad where\ d\ is\ the\ solution\ depth$$

For MST heuristic, as for almost all heuristics [1], the absolute error is at least proportional to the path cost h^{*}, so ϵ is constant or growing and the time complexity is exponential in d.

Moreover, for the traveling salesman problem (TSP), which has $(n-1)!$ goal states, the search process could follow a non-optimal path and there is an additional cost that is proportional to the number of goals that have cost within a factor ϵ of the optimal cost.

We can improve the process of A* (in time and space) by using a graph search in which we create a list of visited cities (closed list) in addition to the list of unvisited cities (fringe list or open). The experiment results of this report show that the graph search of A* was able to give optimal or very near-optimal solutions in reasonable computation time and usage space.

A* algorithm using (graph search)

```
function solution = solveTSP_Astar_graphsearch(TSPGraph)              % returns a solution
    firstNode ← TSPGraph.getFirstNode                      % Get the start node (randomly).
    open ← []                                              % create an empty fringe list.
    closed ← []                                             % create an empty closed list.
    startNode ← initializeStartNode(TSPGraph, firstNode) % initialize the start node.
    open.add(startNode)                  % add the start node to the open list (fringe list).
    solution ← Astar (TSPGraph, startNode, open)           % call A* algorithm to solve TSP.
    return solution           % if there is no solution (failure), the solution will be empty.

function solution = Astar (TSPGraph, startNode, open)    % returns a solution (path and cost)
    solution ← []                                 % create an empty solution (failure).
    while(open list is not empty)
        currentNode ← minCost(open)               % get the minimum cost node in the fringe.
        Remove the current Node from open list
        add the current Node to the closed list
        if the current node is the goal node then return solution(currentNode, TSPGraph)
        successors ← expandNode(TSPGraph, startNode, currentNode); % expand the current node.

        % add/update successors in the open and closed lists based on the node cost and depth
        -------------------------------------------------------------------------------------
        For each node s in successors
           if [s is in closed and (s.cost <= s_inClosed.cost)] or
              [s is in closed and (s.cost > s_inClosed.cost) and (s.depth>s_inClosed.depth)]
              then move s from closed to open
           else if [s is not already in open] then add s to open
           else if (s.cost<s_inOpen.cost) and (s.depth>s_inOpen.depth) then remove s from open
                else if [s.cost <= s_inOpen.cost] or
                        [(s.cost > s_inOpen.cost) and (s.depth > s_inOpen.depth)]
                     then add s to open
        -------------------------------------------------------------------------------------
    return failure                  % return failure (empty solution) if no solution.

function successors = expandNode(TSPGraph, startNode, currentNode) % returns successors of node
   ….
   % Same as the 'expandNode' function mentioned before
   ….
```

2. Hill-climbing approach

Pseudocode of hill-climbing algorithm to solve TSP problem

```
function solution = solveTSP_hillclimbing(TSPGraph)        % returns a solution (path and cost)
    curTour ← createInitialState(TSPGraph)          % create an initial state (initial tour).
    do loop
        % get the next local solution
        newTour ← getNextSolution(TSPGraph, curTour)
        if the current solution does not change, return the current tour as a potential maxima
        % (minimum cost).
        if(newState.distance == curState.distance) return the curState as solution
        curTour ← newTour

function solution = getNextSolution (TSPGraph, curTour)   % return the next best local
    do loop for all the permutations of the current state (current tour)
        % get next nearest tour by changing the positions of two nodes in the current tour
        % (or deleting two edges and adding two).
        % The new and current tours are called 2-opt neighbours.
        newTour ← Get next nearest tour
        % compare the cost of the current and new tours. If the new tour has lower cost,
        % then we change the current tour by the new tour.
        if( newTour.distance < curTour.distance ) then
             curTour ← newTour
    return curTour % return the current tour as solution
```

Time and space complexities of hill-climbing algorithm

Hill climbing is neither complete nor optimal. Hill climbing performs local search that starts with a random initial solution, and then tries to find a better solution by gradually changing a single element. In travelling salesman problem TSP, we try to minimize the traveling distance by changing the positions of two nodes in the path. If the change produces a lower distance, the new solution becomes the current solution. We repeat until no further improvements can be found. Hill climbing has O(∞) time complexity because it may fail to reach the global maximum. However, it can give a good solution in reasonable time. The space complexity of Hill climbing is O(b), because it only move one single branch level, or b. Here in the TSP problem, the branching is equal to b= nC2=n(n-1)/2.

3. RBFS (Recursive Best First Search) algorithm

Pseudocode of RBFS algorithm to solve TSP problem

```
function solution = solveTSP_RBFS(TSPGraph)                  % returns a solution (path and cost)
    firstNode = TSPGraph.getFirstNode                            % Get the start node (randomly)
    startNode ← initializeStartNode(TSPGraph, firstNode)         % initialize the start node.
    solution ← TSP_RBFS(TSPGraph, startNode, startNode, ∞) % call RBFS to solve TSP.
    return solution           % if there is no solution (failure), the solution will be empty.
```

```
function solution = TSP_RBFS(TSPGraph, startNode, currentNode, f_limit) % returns a solution
    solution ← isGoal(currentNode, TSPGraph)    % check if the current node is the goal.
    if (solution is not empty) return solution  % return the solution if node is the goal.
    successors ← expandNode(TSPGraph, startNode, currentNode); % expand the current node.

    if there are no successors (successors is empty), then return empty solution [failure, ∞]
    for each successors do:
        suc.cost ← max( suc.g_n+suc.h_n, currentNode.cost);

    while(true)
        best ← get the minimum cost node in the successors;
        if (best.cost>f_limit) then return [failure, best.cost]
        secMinCost ← get the second minimum cost in all successors;
        [solution, best.cost] ← TSP_RBFS(TSPGraph, startNode, best, min(f_limit, secMinCost))
        if we reach the goal (solution is not empty), then return the solution
```

```
function successors = expandNode(TSPGraph, startNode, currentNode) % returns successors of node
   ….
   % Same as the 'expandNode' function described in A* algorithm
   ….
```

Time and space complexities of RBFS algorithm

Since we are using the minimum spanning tree (MST) as a heuristic function, which is admissible, RBFS algorithm is complete and optimal.

RBFS is characterized by a linear space complexity with the depth of the optimal solution $O(bd)$. The time complexity of RBFS depends on the accuracy of the heuristic function and on the number of times the best path changes when nodes are expanded, which is difficult to characterize. Since RBFS uses limited memory space, it is required to regenerate (reexpand) the forgotten states, which cause overheating in time. This overheating may cause exponential increase in time complexity associated with redundant paths in graphs.

B. Experiments and discussion

Result of running A* (tree search) on dataset att48.xml,

After around 5 hours, A* algorithm reached **depth 24 with fringe size = 570138**. From the results, it is clear that the fringe is growing exponentially and the time required to reach the next depth is growing exponentially.

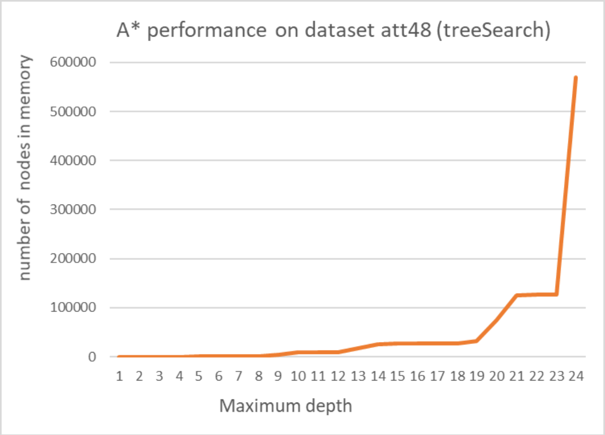


Fig.1 The relationship between the maximum depth so far and the number of nodes in memory. (exponential space complexity)

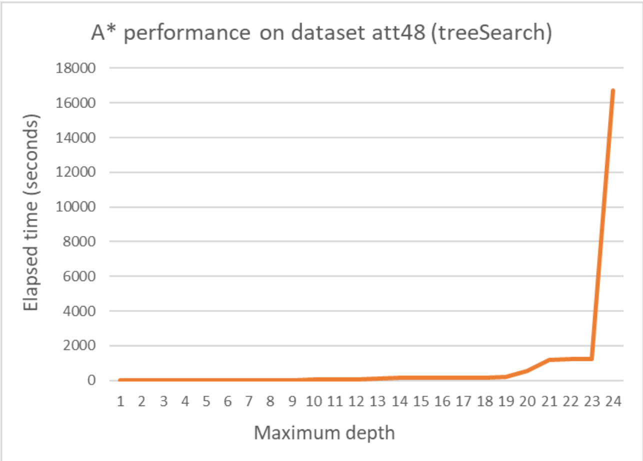


Fig.2 The relationship between the maximum depth so far and the running time. (exponential time complexity)

Maximum depth	Elapsed time sec	number of nodes in memory
1	0.232686	47
2	0.454157	92
3	0.661945	136
4	0.87335	179
5	1.071374	221
6	4.443945	919
7	5.047392	1046
8	5.4378	1127
9	17.36652	3584
10	44.56735	8850
11	48.62703	9606
12	48.84067	9641
13	87.59452	16603
14	147.07	26282
15	148.1451	26435
16	148.9439	26543
17	149.4813	26612
18	149.7739	26641
19	191.0603	32690
20	569.13	75161
21	1208.611	125780
22	1217.592	126378
23	1223.262	126742
24	16708.22	570138

Result of running A* (tree search) on dataset a280.xml,

After around 3 hours, A* algorithm reached **depth 81 with fringe size = 858565**. From the results, it is clear that the fringe is growing exponentially and the time required to reach the next depth is growing exponentially.

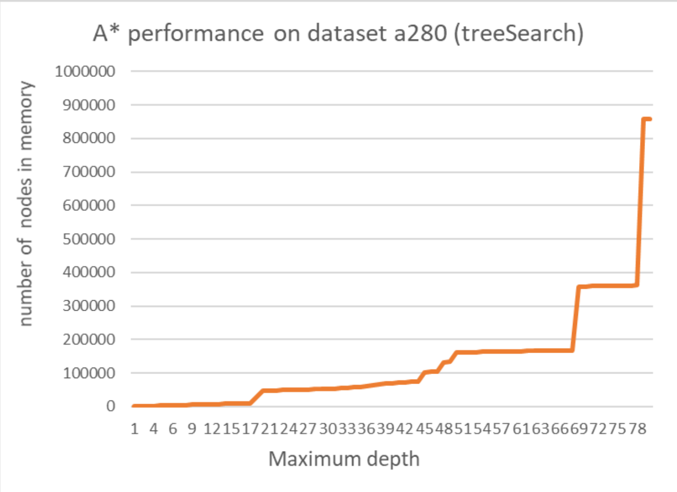


Fig.3 The relationship between the maximum depth so far and the number of nodes in memory. (exponential space complexity)

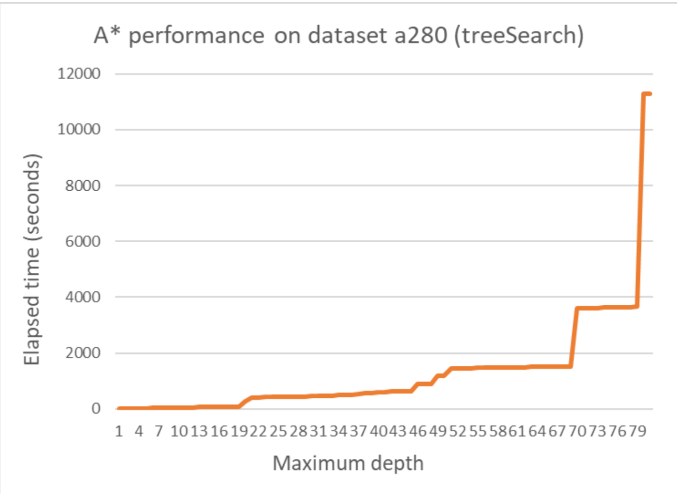


Fig.4 The relationship between the maximum depth so far and the running time. (exponential time complexity)

Maximum depth	Elapsed time sec	number of nodes in memory
1	8.277604	279
2	10.60587	556
3	12.81818	832
4	21.86556	1936
5	24.05733	2210
6	28.90807	2757
7	35.79692	3575
8	40.22534	4118
9	44.52867	4659
10	48.8032	5198
11	53.10047	5735
12	57.47386	6270
13	61.76254	6803
14	66.01332	7334
15	70.24641	7863
16	74.47455	8390
17	78.60887	8915
18	82.87361	9438
19	84.97203	9698
20	263.7075	29030
21	414.029	47181
22	418.4258	47696
23	422.9172	48209
24	427.327	48720
25	431.6232	49229
26	435.9229	49736
27	440.2342	50241
28	444.5721	50744
29	448.8602	51245
30	453.1304	51744
31	457.3953	52241
32	465.8293	53232
33	474.1821	54219
34	486.74	55694
35	499.2591	57163
36	516.2369	59114
37	533.1969	61057
38	554.9028	63476
39	571.5808	65403
40	592.6236	67802
41	605.1617	69235
42	622.1338	71138
43	630.4789	72085
44	642.9046	73500
45	647.1528	73969
46	899.7549	102433
47	904.2195	102898
48	906.6055	103129
49	1175.483	132470
50	1180.098	132929
51	1445.146	160884
52	1454.773	161795
53	1459.5	162248
54	1464.372	162699
55	1469.193	163148
56	1473.909	163595
57	1478.678	164040
58	1483.491	164483
59	1488.132	164924
60	1490.641	165143
61	1493.148	165361
62	1495.621	165578
63	1498.152	165794
64	1500.614	166009
65	1505.244	166438
66	1509.852	166865
67	1514.476	167290
68	1519.124	167713
69	1521.594	167923
70	3601.276	358114
71	3607.496	358531
72	3613.66	358946
73	3619.816	359359
74	3625.966	359770
75	3632.55	360179
76	3638.651	360586
77	3644.789	360991
78	3650.869	361394
79	3656.983	361795
80	11284.74	858168
81	11294.64	858565

Result of running A* (graph search) on dataset att48.xml,

In less than 3 minutes, A* algorithm reached the optimal solution, with path cost equals to 10628 [2], starting from a random initial node named '2'. The complete solution tour is given below:

'2' '26' '4' '35' '45' '10' '24' '42' '5' '48' '39' '32' '21' '47' '20' '33' '46' '36' '30' '43' '17' '27' '19' '37' '6' '28' '7' '18' '44' '31' '38' '8' '1' 9' '40' '15' '12' '11' '13' '25' '14' '23' '3' '22' '16' '41' '34' '29' **'2'**

In order to obtain this optimal solution, A* spend 161 second and stored a maximum of 589 nodes in memory. Employing graph search improved the process of A* algorithm in both time and space. However, by using different initial states, A* does not always give the optimal solution. In fact, among the different 48 initial nodes on dataset 'att48', A* was able to provide the optimal solution for 8 initial nodes, '2', '8', '19', '29', '31', '38', '41', and '44'. For the other nodes, it gives very near optimal solutions, which range between 10648 to 10795.

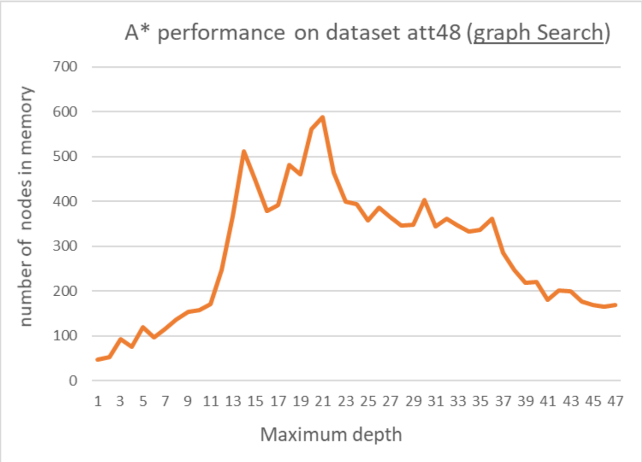


Fig.5 The relationship between the maximum depth so far and the number of nodes in memory.

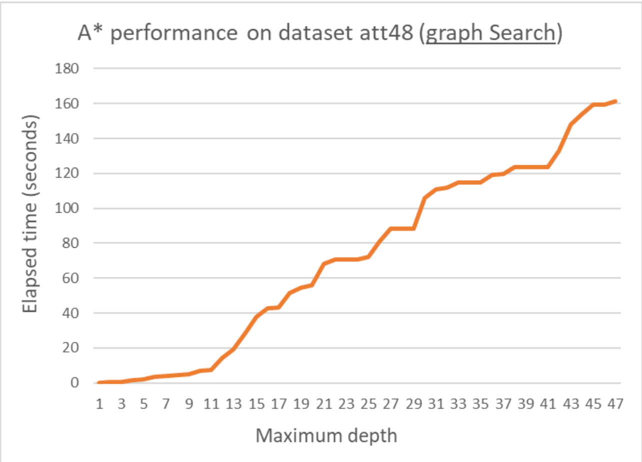


Fig.6 The relationship between the maximum depth so far and the running time. (linear time complexity)

Maximum depth	Elapsed time sec	number of nodes in memory
1	0.2889	47
2	0.522	53
3	0.7516	92
4	1.6678	76
5	2.0961	119
6	3.5882	96
7	3.7787	116
8	4.5887	137
9	4.7767	154
10	6.9816	158
11	7.3735	171
12	14.5273	247
13	19.1757	367
14	28.6775	511
15	37.7771	449
16	42.9005	379
17	43.0897	391
18	51.4263	482
19	54.3806	461
20	55.9979	562
21	68.4523	589
22	70.4215	464
23	70.5842	399
24	70.7411	394
25	72.1444	357
26	81.0084	386
27	88.211	365
28	88.3562	346
29	88.5012	348
30	106.0336	404
31	110.8847	344
32	111.6966	362
33	114.552	347
34	114.7153	332
35	114.8471	336
36	119.3022	362
37	119.426	286
38	123.4243	247
39	123.5231	219
40	123.6241	221
41	123.7185	181
42	132.9091	202
43	148.0806	200
44	153.8212	177
45	159.3322	168
46	159.4104	166
47	161.1567	168

Result of running A* (graph search) on dataset a280.xml,

From the results, for a280 dataset, A* with graph search showed enhancement to the performance compared with A* with tree search. For the same a280 dataset, A* tree search reached depth 81 with fringe size = 858565 after 3.1 hours, while A* graph search reached the same depth with fringe size = 5993 after 0.4 hours. A* graph search enhanced the space and make the usage of memory bounded instead of exponential growing in the A* with tree search. However, the time complexity started to grow exponentially after reaching depth 86, which may cause improvement in time is ineffective for the problem with a large number of nodes.

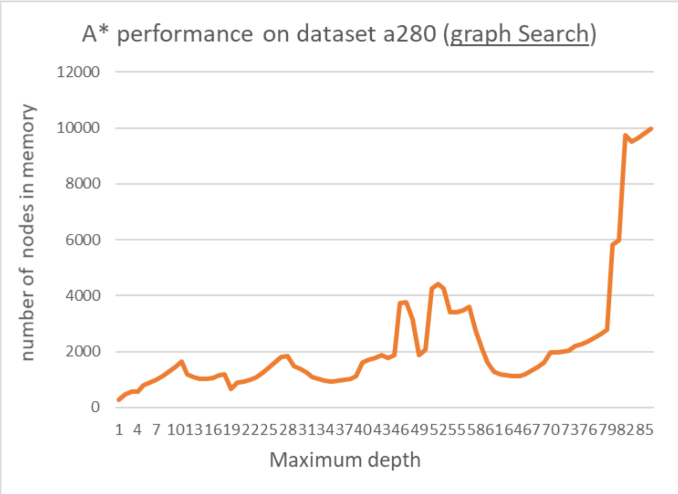


Fig.7 The relationship between the maximum depth so far and the number of nodes in memory.

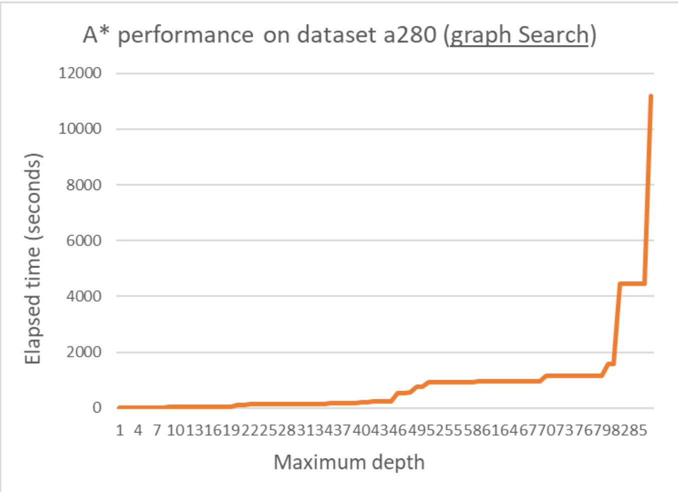
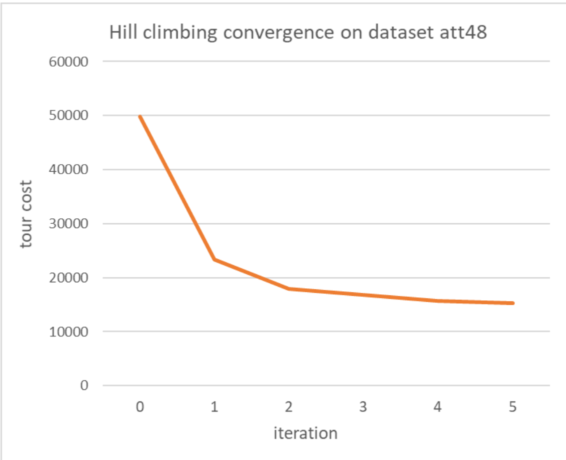


Fig.8 The relationship between the maximum depth so far and the running time.

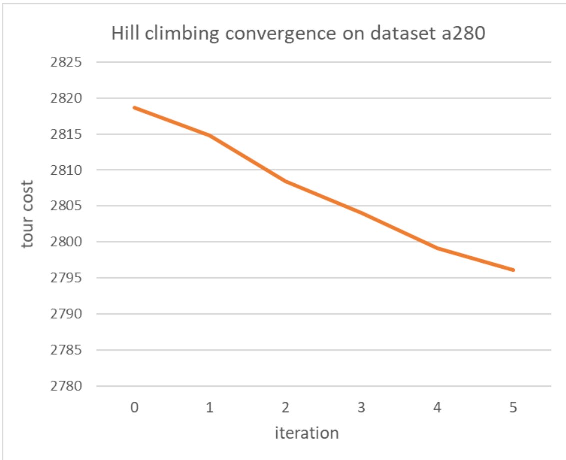
Maximum depth	Elapsed time sec	number of nodes in memory
1	6.1	279
2	8.562	479
3	11.1779	563
4	16.3096	573
5	18.7379	802
6	21.2158	877
7	23.6813	981
8	26.1655	1115
9	28.6924	1272
10	31.1803	1442
11	33.7256	1629
12	36.4637	1173
13	38.9354	1091
14	41.396	1038
15	43.9529	1024
16	46.8137	1058
17	49.5221	1141
18	52.9003	1172
19	55.8355	649
20	122.7096	894
21	125.2991	911
22	127.6978	989
23	130.0322	1103
24	132.3276	1246
25	134.6737	1414
26	136.9832	1600
27	139.421	1805
28	141.905	1841
29	144.5888	1479
30	147.0882	1397
31	149.5527	1239
32	151.8938	1102
33	154.1945	1018
34	156.4234	963
35	158.7008	940
36	160.9124	946
37	163.1213	981
38	165.2998	1025
39	167.4952	1107
40	201.2238	1595
41	203.4488	1700
42	229.0295	1772
43	231.2909	1866
44	250.2978	1788
45	252.5619	1874
46	544.5825	3725
47	547.4176	3755
48	551.8045	3134
49	774.8214	1866
50	777.5865	2075
51	919.1752	4272
52	921.9501	4429
53	925.1753	4265
54	929.0758	3416
55	931.6561	3391
56	934.1372	3478
57	936.6189	3591
58	940.1074	2750
59	943.0702	2107
60	945.5857	1595
61	947.8678	1267
62	949.9464	1174
63	951.9644	1140
64	953.9801	1108
65	955.9555	1135
66	957.8971	1193
67	959.8095	1315
68	961.7546	1444
69	963.6955	1604
70	1144.5999	1974
71	1146.7745	1965
72	1148.9281	1994
73	1151.0003	2045
74	1153.0208	2193
75	1155.0959	2279
76	1157.508	2377
77	1159.5677	2490
78	1161.5909	2621
79	1163.6717	2770
80	1564.9255	5834
81	1567.6337	5993
82	4443.2014	9736
83	4448.188	9522
84	4451.6072	9652
85	4454.9754	9804
86	4458.2951	9965
86	11177.9996	29094

Result of running Hill-climbing algorithm,

Fig.9 show the convergence of TSP tour cost for att48 and a280 datasets from initial solution to the minimum cost for each iteration using Hill-climbing algorithm. Att48 dataset has 48 nodes, so it has branching factor b= 48C2 = 1128. This means that the objective function will search for the lowest cost between these 1128 permutations. In this experiment, hill climbing took 166 seconds and 5 iterations to give a solution for Att48 dataset with traveling distance equals to 15335. This solution is not optimal, the optimal solution equals 10628[2], however, the search algorithm give us a fast solution which performed in little amount of time and took constant memory space (b=1128). For a280 dataset, which has branching factor b=280C2 = 39060, the objective function will search for the lowest cost between 39060 permutations for each iteration. This may take more time but the space will still be constant. If we need very fast solution, we can interrupt the search process according to our time requirements and took the best solution so far. In this experiment, we interrupted the search process for a280 database after 2 hour, and the result of convergence is shown in fig.9 b.



(a) att48 dataset



(a) a280 dataset (interrupted after 2 hours)

Fig.9 The convergence of tour cost for each iteration until it reach the minimum cost (local or hopefully optimal)

Result of running RBFS on dataset att-48.xml

After around 2 hours, the RBFS algorithm reached **depth 9 with a total number of nodes in memory 267**. After 3 hours, the algorithm did not go beyond depth 9 and the number of nodes became 267 (same as in depth 6), this means that the algorithm returned to lower depths (depth six in this case). From the results, we notice that the relationship between the maximum depth so far and the number of nodes in memory is leaner. However, the relationship between the maximum depth and the execution time started linearly and then became exponential. This due to the overhead by regenerating the nodes.

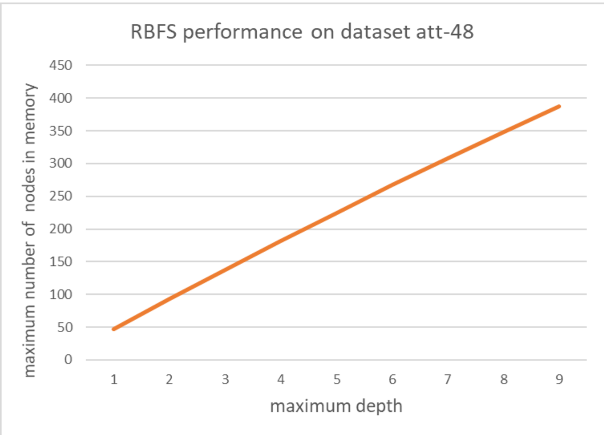


Fig.10 The relationship between the maximum depth so far and the number of nodes in memory. (Linear).

Maximum depth	Elapsed time sec	number of nodes in memory
1	5.056	47
2	5.2632	93
3	5.4822	138
4	5.6815	182
5	14.9825	225
6	18.9175	267
7	26.0411	308
8	282.1837	348
9	6712.904	387
9	11373.07	267

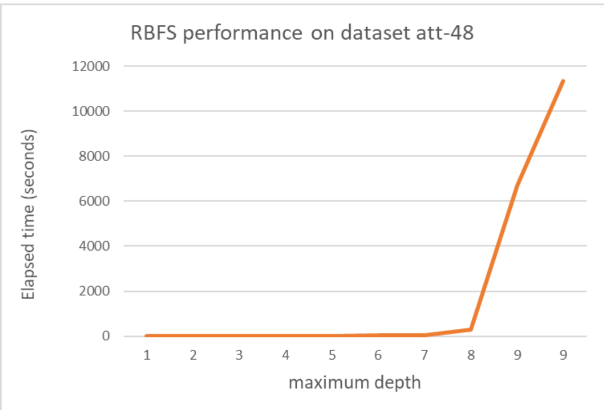


Fig.11 The relationship between the maximum depth so far and the running time.

Result of running RBFS on dataset a280.xml,

After around 4 hours, the RBFS algorithm reached <u>depth 83 with total number of visited nodes was 19160</u>. From the results, we notice that the relationship between the maximum depth so far and the number of nodes in memory is leaner. However, the relationship between the maximum depth and the execution time started linearly and then became exponential. This due to the overhead by regenerating the nodes.

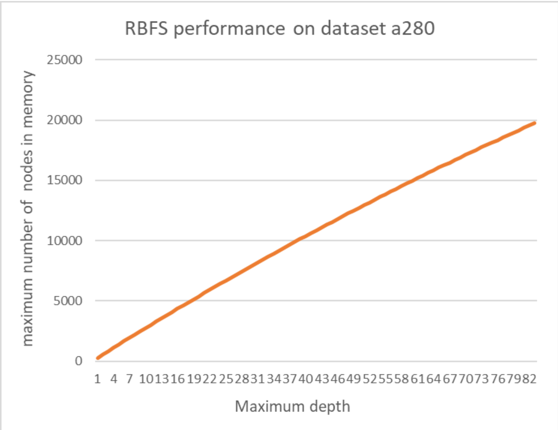


Fig.12 The relationship between the maximum depth so far and the number of nodes in memory. (Linear).

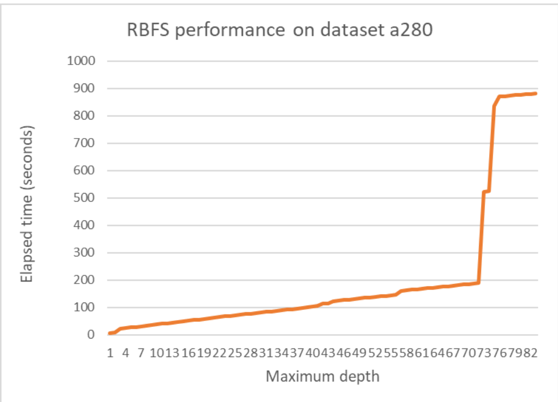


Fig.13 The relationship between the maximum depth so far and the running time (0 to around 15 minutes).

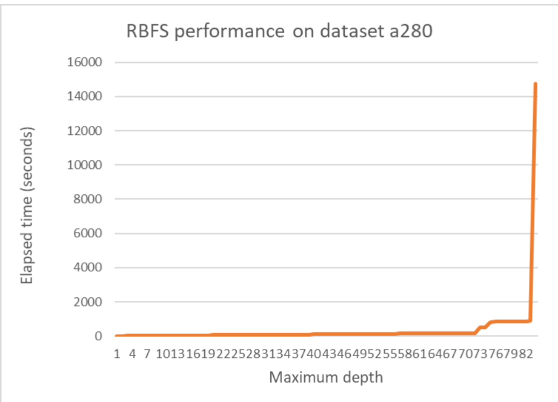


Fig.14 The relationship between the maximum depth so far and the running time (0 to around 4 hours).

Maximum depth	Elapsed time sec	number of nodes in memory
1	5.597	279
2	7.9434	557
3	21.7373	834
4	24.0705	1110
5	26.4012	1385
6	28.6401	1659
7	30.8731	1932
8	33.0844	2204
9	35.3647	2475
10	37.6231	2745
11	39.8792	3014
12	42.1684	3282
13	44.3937	3549
14	46.6487	3815
15	48.842	4080
16	51.0723	4344
17	53.3907	4607
18	55.7147	4869
19	58.0689	5130
20	60.2911	5390
21	62.3982	5649
22	64.5581	5907
23	66.6564	6164
24	68.7688	6420
25	70.8363	6675
26	72.9938	6929
27	75.0895	7182
28	77.1275	7434
29	79.1552	7685
30	81.2429	7935
31	83.2828	8184
32	85.327	8432
33	87.3584	8679
34	89.3816	8925
35	91.4043	9170
36	93.4128	9414
37	95.421	9657
38	97.3965	9899
39	99.3716	10140
40	103.3219	10380
41	105.3528	10619
42	113.2504	10857
43	115.2441	11094
44	123.035	11330
45	124.9417	11565
46	126.8491	11799
47	128.7683	12032
48	130.6846	12264
49	132.5971	12495
50	134.4954	12725
51	136.4066	12954
52	138.2899	13182
53	140.1022	13409
54	141.9866	13635
55	143.8413	13860
56	145.6539	14084
57	160.5231	14307
58	162.3888	14529
59	164.2592	14750
60	166.4229	14970
61	168.3027	15189
62	170.0877	15407
63	171.9381	15624
64	173.7746	15840
65	175.5897	16055
66	177.3721	16269
67	179.1951	16482
68	181.2339	16694
69	183.1446	16905
70	184.8845	17115
71	186.6514	17324
72	188.4424	17532
73	521.9428	17739
74	525.3982	17945
75	836.8644	18150
76	869.8419	18354
77	871.5712	18557
78	873.32	18759
79	875.1801	18960
80	877.0093	19160
81	878.6845	19359
82	880.3652	19557
83	882.0438	19754
83	14743.8736	19160

C. Conclusion

A* search algorithm is complete and optimal with admissible heuristic function. However, it has exponential complexity problems in computation time and memory usage. The previous experiments showed that fringe size and the time required to reach the next depth grows exponentially. A* search was not able to reach a solution and after running for a long time it run out of memory. Therefore, A∗ is not practical for many problems including TSP.

Implementing a graph search for A* improves the performance of A* in time and space. However, we need to deal carefully with the open and closed list to guarantee an optimal solution. In addition, the time complexity could grow exponentially after reaching a certain level of depth as shown in previous experiments

Hill climbing is neither complete nor optimal. However, it gives a quick sub-optimal solution, which performed in a little amount of time and took constant memory space. Hill climbing, unlike A*, always provides a solution, which preferred in some application that requires a quick approximate (suboptimal) solution.

RBFS like A* is complete and optimal. It is characterized by a linear space complexity with the depth of the optimal solution $O(bd)$. RBFS overcome the space problem of A*, however it suffers from excessive node regeneration, which may cause exponential time complexity. Generally, RBFS with enough time can solve problems that A∗ cannot solve because it runs out of memory.

D. Appendix (the codes of the algorithms)

All codes in this assignment are implemented and run using MATLAB 2018b. The code for each algorithm is in the appendix at the end of this report.

Common function to read the tsp problem form xml file

```
function [G, Gnodes]= loadTSPGraph(fileName)
    % read the TSP graph from xml file and convert it to matlab structure
    tsp = xml2struct(fileName);
    % convert graph structure to graph matrix
    nodeNo = size(tsp.graph.vertex, 2);
    graphMat = zeros(nodeNo, nodeNo);
    names = zeros(nodeNo,1);
    for v=1 : nodeNo
        names(v) = v;
        vertex = tsp.graph.vertex(v);
        diag = 0;
        for e=1 : nodeNo-1
            edge = vertex{1}.edge(e);
            cost = str2double(edge{1}.Attributes.cost);
            if(v==e)
                diag = 1;
            end
            graphMat(v, e+diag) = cost;
        end
    end

    % convert graph matrix to matlab graph object
    Gnodes = cellstr(string(names));
    G = graph(graphMat, Gnodes);
end
```

1. A* algorithm (tree search)

```
% read the TSP graph from xml file and convert it to matlab graph object
[G, Gnodes] = loadTSPGraph("att48_.xml");
%[G, Gnodes] = loadTSPGraph("data2.xml");
%[G, Gnodes] = loadTSPGraph("burma14_.xml");
%[G, Gnodes] = loadTSPGraph("a280_.xml");

solutions =[];
time = [];
tic
Max_depth=0;
no_expanded_nodes = 0;

for idx=1 : size(Gnodes,1)   % select a node to be as start node
    % initialize the startt node
    strnode = initializeStartNode(G, Gnodes{idx});
    solution = RBFS(G, strnode, strnode, inf, 0, 0);
    if (~isempty(solution))
        solution.cost
        solution.path
        solutions = [solutions; solution];
    end
end

function [solution, f_limit, Max_depth] = RBFS(G, strnode, currentNode, f_limit, Max_depth,
no_expanded_nodes)
    % display the state of searching
        if( Max_depth < currentNode.depth)
            Max_depth = currentNode.depth;
            disp( ['Node: ', currentNode.name, ...
                ' ,  Depth: ', num2str(currentNode.depth), ...
                ' ,  Max-depth: ', num2str(Max_depth), ...
                ' ,  current_nodes: ', num2str(no_expanded_nodes), ...
                ' ,  Elapsed time is ', num2str(toc) ,' seconds.'] );
        end

    % check if the node is the goal.
```

```
    % if we don't reach the goal, solution will be empty
    solution = isGoal(currentNode, G);
    % if we reach the goal (solution is not empty), return the solution
    if (~isempty(solution))
        return
    end

    % expand the node
    nodelist = expandNode(G, strnode.name, currentNode);
    no_expanded_nodes = no_expanded_nodes+size(nodelist,1);

    % if there are no successors (nodelist is empty), return empty solution
    if (isempty(nodelist))
        f_limit = inf;
        return
    end

    for i=1:size(nodelist,1)
        suc = nodelist(i);
        suc.cost = max( suc.g_n+suc.h_n, currentNode.cost);
    end
    while(true)
        [minCost, minidx] = min(cell2mat({nodelist.cost}));
        best = nodelist(minidx);
        if (best.cost>f_limit)
            solution = [];
            f_limit = best.cost;
            return
        end
        % calculate the second minimum cost in all successors
        if (size(nodelist,1) == 1)
            % if only one successor, the second minimum cost = f_limit
            secMinCost = f_limit;
        else
            nodelist2 = nodelist;
            nodelist2(minidx) = [];
            secMinCost = min(cell2mat({nodelist2.cost}));
        end

        [solution, nodelist(minidx).cost, Max_depth] = RBFS(G, strnode, best,
min(f_limit,secMinCost), Max_depth, no_expanded_nodes);
        % if we reach the goal (solution is not empty), return the solution
        if (~isempty(solution))
            return
        end
    end
end

function node = initializeStartNode(G, startNode)
    % remove the first node from the the graph and add it to the open list
    % for the first node we have to

    unvisitedGraph = rmnode(G, startNode);
    h_n = estimateDist(G, unvisitedGraph, startNode, startNode);
    node.name = startNode;
    node.h_n = h_n;
    node.g_n = 0;
    node.depth = 0;
    node.cost = node.h_n + node.g_n - node.depth;
    node.parent = [];
end

% estimateDist function
% estimate the distance from a current Node 'currentNode' to the end node 'startNode'
function cost = estimateDist(G, unvisitedGraph, startNode, currentNode)
    % caculate Minimum spanning tree using Prim's algorithm
    T = minspantree(unvisitedGraph);
    % estimated the distance to visit all unvisited nodes using MST heuristic
    h_n_MST = sum(T.Edges.Weight);

    unvisitedNs = table2cell(unvisitedGraph.Nodes);
    % caculate the minimum distance from an unvisited node to the current node
    idx_to_curr_node = findedge(G, currentNode, unvisitedNs);
    mindist_to_curr_node = min(G.Edges.Weight(idx_to_curr_node));

    % caculate the minimum distance from an unvisited node to the start node
    idx_to_str_node = findedge(G, startNode, unvisitedNs);
```

```
    mindist_to_str_node = min(G.Edges.Weight(idx_to_str_node));

    cost = h_n_MST + mindist_to_str_node + mindist_to_curr_node;
end

% expandNode function
% returns all the successors of a node
function nodelist = expandNode(G, startNode, currentNode)
    % For TSP problem (each node is visited only once)
    % here we just get the nodes that are not already in the current node path
    unvisitedGraph = G;
    node1 = currentNode;
    while(~isempty(node1))
        unvisitedGraph = rmnode(unvisitedGraph, node1.name);
        node1 = node1.parent;
    end
    % the expanded nodes
    unvisitedNs = table2cell(unvisitedGraph.Nodes);
    % the expanded nodes as structure (to be returned)
    nodelist=[];

    % estimate cost for the expanded nodes
    for i=1:size(unvisitedNs,1)
        expandedNode = unvisitedNs{i}; % expanded node name as a string (char array)
        % if the node is leaf (the last node in the path), the heuristic is
        % the cost of connectin this node to the start node
        if (size(unvisitedNs,1)==1)
            idx_to_str_node = findedge(G, startNode, expandedNode);
            h_n = G.Edges.Weight(idx_to_str_node);
        else
            unvistg = rmnode(unvisitedGraph, expandedNode);
            % h_n = cost of the MST of the subgraph of expandedNode +
            % minimum cost to connecte MST to expandedNode +
            % minimum cost to connecte MST to start node
            h_n = estimateDist(G, unvistg, startNode, expandedNode);
        end

        % create the successor node as structure
        node.name = expandedNode; % the successor node
        node.h_n = h_n; % the heuristic of the successor node
        % the cost (edge value) of connecting the current node with its
        % successor (the expanded node)
        g_n = G.Edges.Weight(findedge(G, node.name ,currentNode.name));
        % the cost of connecting the current node to the first node
        % (currentNode.g_n). node.g_n is the cost of the successor node to
        % the start node
        node.g_n = currentNode.g_n + g_n;

        node.parent = currentNode;
        node.depth = currentNode.depth+1;
        % the estimated cost from the current from start node to the goal
        % node throw the successor node
        node.cost = node.h_n + node.g_n - node.depth;
        nodelist = [nodelist; node];
    end
end

% isGoal function
% Test if the node is the goal node (if the goal is satisfied)
function solution = isGoal(node, G)
    path=[];
    snode=[];
    cost=node.cost;
    while(~isempty(node))
        path = [path; {node.name}];
        G = rmnode(G, node.name);
        snode = {node.name};
        node = node.parent;
    end
    if (isempty(G.Nodes))
        path = [snode; path];
        solution.path=path;
        solution.cost=cost;
    else
        solution = [];
    end
end
```

2. A* algorithm (graph search)

```
% read the TSP graph from xml file and convert it to matlab graph object
[G, Gnodes] = loadTSPGraph("att48_.xml");
%[G, Gnodes] = loadTSPGraph("a280_.xml");
%[G, Gnodes] = loadTSPGraph("data2.xml");
%[G, Gnodes] = loadTSPGraph("burma14_.xml");

solutions =[];
time = [];
tic

for idx=1 : size(Gnodes,1)  % select a node to be as start node
    open=[];
    closed=[];
    % initialize the startt node
    strnode = initializeStartNode(G, Gnodes{idx});
    % add the start node to the open list ( finge list)
    open = [open; strnode];
    solution = [];  % the goal strarting from node i
    Max_depth=0;
    no_expanded_nodes = 0;

    while(~isempty(open))
        no_expanded_nodes=no_expanded_nodes+1;
        % get the minimum cost in the fringe
        costs = cell2mat({open.cost});
        depths = cell2mat({open.depth});
        % get the nodes that have the minimum cost
        minCNodes = find(costs==min(costs));
        % if there are more than one node with the same cost, get deepest
        % one
        [val, index] = max(depths(minCNodes));
        minidx = minCNodes(index);
        currentNode = open(minidx);

        % display the state of searching
        if( Max_depth < max(cell2mat({open.depth})) )
            Max_depth = max(cell2mat({open.depth}));
            disp( ['Node: ', currentNode.name, ...
                ',  Depth: ', num2str(currentNode.depth), ...
                ',  Max-depth: ', num2str(max(cell2mat({open.depth}))), ...
                ',  no_expanded_nodes: ', num2str(no_expanded_nodes), ...
                ',  Fringe-size: ', num2str(size(open,1)), ...
                ' ,  Elapsed time is ', num2str(toc) ,' seconds.'] );
        end

        open(minidx)=[];
        if(~ismember(currentNode.name, {open.name}))
            closed = [closed; currentNode];
        end


        % check if the node is the goal.
        solution = isGoal(currentNode, G);
        if (~isempty(solution))
            break;
        end

        % expand the node
        nodelist = expandNode(G, strnode.name, currentNode);

        % For graph search, we expand the node only if it was not expanded before
        for i=1:size(nodelist, 1)
            if (isempty(closed))
                val2 = 0;
            else
                [val2, id2] = ismember(nodelist(i).name,{closed.name});
            end

            if(val2 && (nodelist(i).cost<=closed(id2).cost))
                closed(id2)=[];
                open = [open; nodelist(i)];
            elseif(val2 && (nodelist(i).cost>closed(id2).cost))
```

```
                if((nodelist(i).depth > closed(id2).depth))
                    closed(id2)=[];
                    open = [open; nodelist(i)];
                end
            else
                id1 = find(ismember({open.name}, nodelist(i).name));
                if (isempty(id1))
                    open = [open; nodelist(i)];
                else
                    addnode = false;
                    removeidxs = [];
                    for s=1:size(id1,2)
                        if((nodelist(i).cost <= open(id1(s)).cost))
                            addnode = true;
                            if( (nodelist(i).cost < open(id1(s)).cost) && ...
                                (nodelist(i).depth >= open(id1(s)).depth))
                                removeidxs = [removeidxs; id1(s)];
                            end
                        elseif( (nodelist(i).cost > open(id1(s)).cost) && ...
                                (nodelist(i).depth > open(id1(s)).depth) )
                            addnode = true;
                         end
                    end
                    if(~isempty(removeidxs))
                        removeidxs = sort(removeidxs, 'descend');
                    end
                    for s=1:size(removeidxs,1)
                        open(removeidxs(s))=[];
                    end
                    if(addnode)
                        open = [open; nodelist(i)];
                    end
                end
            end
        end
    end
    if (~isempty(solution))
        solution.cost
        solution.path(1)
        solutions = [solutions; solution];
        time = [time; toc];
    end
end
sum(cell2mat({solutions.cost}))/size(solutions,1)
toc
function node = initializeStartNode(G, startNode)
    % remove the first node from the the graph and add it to the open list
    % for the first node we have to

    unvisitedGraph = rmnode(G, startNode);
    h_n = estimateDist(G, unvisitedGraph, startNode, startNode);
    node.name = startNode;
    node.h_n = h_n;
    node.g_n = 0;
    node.depth = 0;
    node.cost = node.h_n + node.g_n;
    node.parent = [];
end


% estimateDist function
% estimate the distance from a current Node 'currentNode' to the end node 'startNode'
function cost = estimateDist(G, unvisitedGraph, startNode, currentNode)
    % caculate Minimum spanning tree using Prim's algorithm
    T = minspantree(unvisitedGraph);
    % estimated the distance to visit all unvisited nodes using MST heuristic
    h_n_MST = sum(T.Edges.Weight);

    unvisitedNs = table2cell(unvisitedGraph.Nodes);
    % caculate the minimum distance from an unvisited node to the current node
    idx_to_curr_node = findedge(G, currentNode, unvisitedNs);
    mindist_to_curr_node = min(G.Edges.Weight(idx_to_curr_node));

    % caculate the minimum distance from an unvisited node to the start node
    idx_to_str_node = findedge(G, startNode, unvisitedNs);
    mindist_to_str_node = min(G.Edges.Weight(idx_to_str_node));
```

```
    cost = h_n_MST + mindist_to_str_node + mindist_to_curr_node;
end

% expandNode function
% returns all the successors of a node
function nodelist = expandNode(G, startNode, currentNode)
    % For TSP problem (each node is visited only once)
    % here we just get the nodes that are not already in the current node path
    unvisitedGraph = G;
    node1 = currentNode;
    while(~isempty(node1))
        unvisitedGraph = rmnode(unvisitedGraph, node1.name);
        node1 = node1.parent;
    end
    % the expanded nodes
    unvisitedNs = table2cell(unvisitedGraph.Nodes);
    % the expanded nodes as structure (to be returned)
    nodelist=[];

    % estimate cost for the expanded nodes
    for i=1:size(unvisitedNs,1)
        expandedNode = unvisitedNs{i}; % expanded node name as a string (char array)
        % if the node is leaf (the last node in the path), the heuristic is
        % the cost of connectin this node to the start node
        if (size(unvisitedNs,1)==1)
            idx_to_str_node = findedge(G, startNode, expandedNode);
            h_n = G.Edges.Weight(idx_to_str_node);
        else
            unvistg = rmnode(unvisitedGraph, expandedNode);
            % h_n = cost of the MST of the subgraph of expandedNode +
            % minimum cost to connecte MST to expandedNode +
            % minimum cost to connecte MST to start node
            h_n = estimateDist(G, unvistg, startNode, expandedNode);
        end

        % create the successor node as structure
        node.name = expandedNode; % the successor node
        node.h_n = h_n; % the heuristic of the successor node
        % the cost (edge value) of connecting the current node with its
        % successor (the expanded node)
        g_n = G.Edges.Weight(findedge(G, node.name ,currentNode.name));
        % the cost of connecting the current node to the first node
        % (currentNode.g_n). node.g_n is the cost of the successor node to
        % the start node
        node.g_n = currentNode.g_n + g_n;
        % the estimated cost from the current from start node to the goal
        % node throw the successor node
        node.depth = currentNode.depth+1;
        node.cost = node.h_n + node.g_n;
        node.parent = currentNode;
        nodelist = [nodelist; node];
    end
end

% isGoal function
% Test if the node is the goal node (if the goal is satisfied)
function solution = isGoal(node, G)
    path=[];
    snode=[];
    cost=node.cost;
    while(~isempty(node))
        path = [path; {node.name}];
        G = rmnode(G, node.name);
        snode = {node.name};
        node = node.parent;
    end
    if (isempty(G.Nodes))
        path = [snode; path];
        solution.path=path;
        solution.cost=cost;
    else
        solution = [];
    end
end
```

3. Hill-climbing algorithm

```
% read the TSP graph from xml file and convert it to matlab graph object
[G, Gnodes] = loadTSPGraph("att48_.xml");
%[G, Gnodes] = loadTSPGraph("a280_.xml");
%[G, Gnodes] = loadTSPGraph("data2.xml");
tic

% create an initial state (initial tour)
curState = createInitialState(G);

% navigate through the permutations of the current state (current tour).
% one permutation is performed by changing the positions of two nodes
% in the current tour (or deleting two edges and adding two).
% The new and current tours are called 2-opt neighbours.
% For each new tour, we compare the cost of the current and new tours.
% If the new tour has lower cost, we change the current tour by the new tour and loop again

% disp( [curState.node , sum(curState.distance)]);
iterations = 0;
while(true)
    disp( [num2str(iterations) , '    ', num2str(sum(curState.distance))]);

    newState = getNextSolution(G, curState);
    if(newState.distance == curState.distance)
        break;
    end
    curState = newState;
    iterations = iterations + 1;
end

toc

function curState = getNextSolution(G, curState)
    iterations = 1;
    for p1 = 1 : size(curState.node,2)-1
        for p2 = p1+1 : size(curState.node,2)
            newTour = curState.node;
            tempNode = newTour(p1);
            newTour(p1)= curState.node(p2);
            newTour(p2)= tempNode;
            newState = getTourState(G, newTour');
            if( sum(newState.distance) < sum(curState.distance) )
                curState = newState;
            end
            iterations = iterations + 1;
        end
    end
end

function initState = createInitialState(G)
    tour = table2cell(G.Nodes);
    initState = getTourState(G, tour);
end

function state = getTourState(G, tour)
    for idx=1 : size(tour,1)
        curNode = tour(idx);
        if(idx == size(tour,1))
            nextNode = cell2mat(tour(1));
        else
            nextNode = cell2mat(tour(idx+1));
        end
        state.node(idx) = curNode;
        state.distance(idx) = G.Edges.Weight(findedge(G, curNode ,nextNode));
    end
end
```

4. RBFS (Recursive Best First Search) algorithm

```
% read the TSP graph from xml file and convert it to matlab graph object
%[G, Gnodes] = loadTSPGraph("att48_.xml");
%[G, Gnodes] = loadTSPGraph("data2.xml");
%[G, Gnodes] = loadTSPGraph("burma14_.xml");
[G, Gnodes] = loadTSPGraph("a280_.xml");

solutions =[];
time = [];
tic
Max_depth=0;
no_expanded_nodes = 0;


for idx=1 : size(Gnodes,1)  % select a node to be as start node
    % initialize the startt node
    strnode = initializeStartNode(G, Gnodes{idx});
    solution = RBFS(G, strnode, strnode, inf, 0, 0);
    if (~isempty(solution))
        solution.cost
        solution.path
        solutions = [solutions; solution];
    end
end


function [solution, f_limit, Max_depth] = RBFS(G, strnode, currentNode, f_limit, Max_depth,
no_expanded_nodes)
    % display the state of searching
        if( Max_depth < currentNode.depth)
            Max_depth = currentNode.depth;
            disp( ['Node: ', currentNode.name, ...
                 ' ,  Depth: ', num2str(currentNode.depth), ...
                 ' ,  Max-depth: ', num2str(Max_depth), ...
                 ' ,  current_nodes: ', num2str(no_expanded_nodes), ...
                 ' ,  Elapsed time is ', num2str(toc) ,'seconds'] );
        end

    % check if the node is the goal.
    % if we don't reach the goal, solution will be empty
    solution = isGoal(currentNode, G);
    % if we reach the goal (solution is not empty), return the solution
    if (~isempty(solution))
        return
    end

    % expand the node
    nodelist = expandNode(G, strnode.name, currentNode);
    no_expanded_nodes = no_expanded_nodes+size(nodelist,1);

    % if there are no successors (nodelist is empty), return empty solution
    if (isempty(nodelist))
        f_limit = inf;
        return
    end

    for i=1:size(nodelist,1)
        suc = nodelist(i);
        suc.cost = max( suc.g_n+suc.h_n, currentNode.cost);
    end
    while(true)
        [minCost, minidx] = min(cell2mat({nodelist.cost}));
        best = nodelist(minidx);
        if (best.cost>f_limit)
            solution = [];
            f_limit = best.cost;
            return
        end
        % calculate the second minimum cost in all successors
        if (size(nodelist,1) == 1)
            % if only one successor, the second minimum cost = f_limit
            secMinCost = f_limit;
        else
            nodelist2 = nodelist;
```

```
            nodelist2(minidx) = [];
            secMinCost = min(cell2mat({nodelist2.cost}));
        end

        [solution, nodelist(minidx).cost, Max_depth] = RBFS(G, strnode, best,
min(f_limit,secMinCost), Max_depth, no_expanded_nodes);
        % if we reach the goal (solution is not empty), return the solution
        if (~isempty(solution))
            return
        end
    end
end


function node = initializeStartNode(G, startNode)
    % remove the first node from the the graph and add it to the open list
    % for the first node we have to

    unvisitedGraph = rmnode(G, startNode);
    h_n = estimateDist(G, unvisitedGraph, startNode, startNode);
    node.name = startNode;
    node.h_n = h_n;
    node.g_n = 0;
    node.depth = 0;
    node.cost = node.h_n + node.g_n - node.depth;
    node.parent = [];
end


% estimateDist function
% estimate the distance from a current Node 'currentNode' to the end node 'startNode'
function cost = estimateDist(G, unvisitedGraph, startNode, currentNode)
    % caculate Minimum spanning tree using Prim's algorithm
    T = minspantree(unvisitedGraph);
    % estimated the distance to visit all unvisited nodes using MST heuristic
    h_n_MST = sum(T.Edges.Weight);

    unvisitedNs = table2cell(unvisitedGraph.Nodes);
    % caculate the minimum distance from an unvisited node to the current node
    idx_to_curr_node = findedge(G, currentNode, unvisitedNs);
    mindist_to_curr_node = min(G.Edges.Weight(idx_to_curr_node));

    % caculate the minimum distance from an unvisited node to the start node
    idx_to_str_node = findedge(G, startNode, unvisitedNs);
    mindist_to_str_node = min(G.Edges.Weight(idx_to_str_node));

    cost = h_n_MST + mindist_to_str_node + mindist_to_curr_node;
end

% expandNode function
% returns all the successors of a node
function nodelist = expandNode(G, startNode, currentNode)
    % For TSP problem (each node is visited only once)
    % here we just get the nodes that are not already in the current node path
    unvisitedGraph = G;
    node1 = currentNode;
    while(~isempty(node1))
        unvisitedGraph = rmnode(unvisitedGraph, node1.name);
        node1 = node1.parent;
    end
    % the expanded nodes
    unvisitedNs = table2cell(unvisitedGraph.Nodes);
    % the expanded nodes as structure (to be returned)
    nodelist=[];

    % estimate cost for the expanded nodes
    for i=1:size(unvisitedNs,1)
        expandedNode = unvisitedNs{i}; % expanded node name as a string (char array)
        % if the node is leaf (the last node in the path), the heuristic is
        % the cost of connectin this node to the start node
        if (size(unvisitedNs,1)==1)
            idx_to_str_node = findedge(G, startNode, expandedNode);
            h_n = G.Edges.Weight(idx_to_str_node);
        else
            unvistg = rmnode(unvisitedGraph, expandedNode);
            % h_n = cost of the MST of the subgraph of expandedNode +
            % minimum cost to connecte MST to expandedNode +
```

```
            % minimum cost to connecte MST to start node
            h_n = estimateDist(G, unvistg, startNode, expandedNode);
        end

        % create the successor node as structure
        node.name = expandedNode; % the successor node
        node.h_n = h_n; % the heuristic of the successor node
        % the cost (edge value) of connecting the current node with its
        % successor (the expanded node)
        g_n = G.Edges.Weight(findedge(G, node.name ,currentNode.name));
        % the cost of connecting the current node to the first node
        % (currentNode.g_n). node.g_n is the cost of the successor node to
        % the start node
        node.g_n = currentNode.g_n + g_n;

        node.parent = currentNode;
        node.depth = currentNode.depth+1;
        % the estimated cost from the current from start node to the goal
        % node throw the successor node
        node.cost = node.h_n + node.g_n - node.depth;
        nodelist = [nodelist; node];
    end
end

% isGoal function
% Test if the node is the goal node (if the goal is satisfied)
function solution = isGoal(node, G)
    path=[];
    snode=[];
    cost=node.cost;
    while(~isempty(node))
        path = [path; {node.name}];
        G = rmnode(G, node.name);
        snode = {node.name};
        node = node.parent;
    end
    if (isempty(G.Nodes))
        path = [snode; path];
        solution.path=path;
        solution.cost=cost;
    else
        solution = [];
    end
end
```

References

[1] S. Rassel and P. Norwig, “Artificial intelligence: a modern approach (AIMA).” Moscow: Wiliams, 2007.

[2] “Symmetric TSPs.” [Online]. Available: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/STSP.html. [Accessed: 08-Apr-2020].