

# Traveling Salesman Problem

June 5, 2018

Student : Buşu-Dragomir Alexandru

Artificial Intelligence, Summer 2018

Group I, Subgroup A  
CEN 2.1A

## Introduction

*The travelling salesman problem (TSP) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city and returns to the origin city?" It is an NP-hard problem in combinatorial optimization, important in operations research and theoretical computer science.*

*The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept city represents, for example, customers, soldering points, or DNA fragments, and the concept distance represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources. In many applications, additional constraints such as limited resources or time windows may be imposed. [source : Wikipedia]*

## Problem statement

### *TSP Problem*

Travelling Salesman Problem (TSP): Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Your task is to develop an application to evaluate the performance of two heuristic search algorithms. We consider A\* and Recursive best-first search – RBFS. In what follows we present the description of RBFS: “This algorithm uses the f limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the f-value of each node along the path with a backed-up value – the best f-value of its children. In this way, RBFS remembers the f-value of the best leaf in the forgotten subtree and can therefore decide whether it’s worth reexpanding the subtree at some later time.”

## Pseudocode

### *Dijkstra's Algorithm*

```
// computes shortest path in a graph from source vertex  
// to destination vertex
```

```
function Dijkstra(Graph, source, destination):
```

```
    create vertex priority queue Queue
```

```
    for each vertex v in Graph:  
        distance[v] ← INFINITY  
        previous[v] ← UNDEFINED  
        add v to Queue
```

```
    distance[source] ← 0
```

```
    while Queue is not empty:  
        u ← vertex in Queue with min distance[u]
```

```
        remove u from Queue
```

```
        for each neighbor v of u  
            auxiliary ← distance[u] + length(u, v)  
            if auxiliary < distance[v]  
                distance[v] ← auxiliary  
                previous[v] ← u
```

```
    get shortest path from dist[] and prev[]  
    calculate shortest path cost
```

```
    return cost
```

### *Prim's Algorithm*

```
// the code is very similar to Dijkstra
// there is only a difference in the final if

function Prim(Graph):

    create vertex priority queue Queue

    for each vertex v in Graph:
        distance[v] <- INFINITY
        previous[v] <- UNDEFINED
        add v to Queue

    distance[source] <- 0

    while Queue is not empty
        u <- vertex in Queue with min distance[u]

        remove u from Queue

        for each neighbor v of u
            auxiliary <- length(u, v)

            // differences here

            if auxiliary < distance[v]
                distance[v] <- auxiliary
                previous[v] <- u

    get minimum spanning tree from dist[] and prev[]
    calculate minimum spanning tree cost

    return cost
```

## Application outline

### *A\* class review*

We will use in this class the following data structures and members:

- a priority queue in which we have the nodes sorted in ascending order by their cost
- a list that informs us if we have or have not already visited a certain node
- a list of cities that have been visited in the visiting order to recreate the path
- a total traveling cost variable to store the total cost of the chosen way
- we also have a cost matrix which tells us what is the cost from vertex i to vertex j (matrix[i][j] == cost from i to j)
- and a starting node which is set in the constructor of the class
- this class is practically very similar with the Dijkstra class, but instead of sorting the priority queue after the edge costs, we use the total cost (i.e. the cost of the edges + heuristic cost)
- implementation is using a minimum spanning tree cost from Prim's algorithm as a heuristic
- implementation is using a shortest path from Dijkstra's algorithm as a heuristic

### *A\* solve method*

- we are in the initial town at the start, so we mark it as visited and we add it to the visitedNodes list
- we save in a variable the total number of visited nodes, initially 1
- we visit nodes until all nodes (cities) have been visited
- we take each node, except the current one, and check if it is visited; we don't have to check if there is an edge from one to another because the graph is complete
- we take all neighbours (we do not have self loops); we check if the neighbours are visited or not
- for each neighbour which is not visited, we calculate the totalCost with one of the two heuristics, using a Minimum Spanning Tree (Prim's Algorithm) or a shortest path (Dijkstra's Algorithm)
- for each neighbour which is not visited, we calculate the totalCost we get the total cost on the edges of the spanning tree and save it; this is the heuristic value for the current node
- we calculate the total cost by adding the heuristic cost and the cost on the edge to this node
- now we create the node and we save it in the priorityQueue
- in other words, like in the case of Dijkstra algorithm, we add in the priority queue all the neighbours of the current node which are not visited
- now we update the currentNode and mark it as visited and then we add it to the visitedNodes list; increase visitedNodesNumber
- now we update the currentNode and mark it as visited and then we add it to the visitedNodes list; increase visitedNodesNumber
- we get the total cost of the tour for the travel through all nodes

## *RBFS class review*

We will use in this class the following data structures and members:

- implementation of recursive best first search; is using Prim's and Dijkstra's algorithms as heuristics (cost of minimum spanning tree and shortest path cost)
- a priority queue in which we have the nodes sorted in ascending order by their cost
- a list that informs us if we have or have not already visited a certain node
- a list of cities that have been visited in the visiting order to recreate the path
- a total traveling cost variable to store the total cost of the chosen way
- we also have a cost matrix which tells us what is the cost from vertex  $i$  to vertex  $j$  ( $\text{matrix}[i][j] == \text{cost from } i \text{ to } j$ )
- and a starting node which is set in the constructor of the class
- this class is practically very similar with the  $A^*$  class, but instead of sorting the priority queue after the cost of the edges + heuristic cost, we use only the heuristic cost
- implementation is using a minimum spanning tree cost from Prim's algorithm as a heuristic
- implementation is using a shortest path from Dijkstra's algorithm as a heuristic



### *RBFS solve method*

- we will save information in the ArrayLists from the position 1 in order to logically retrieve information from them because the cities have indices from 1 to "numberOfNodes"
- we initialise the isVisited list with false; the traveling cost will be 0 at the start of the tour
- function returns true if we found a way to visit nodes at a new level; returning false means that we must go back; the final path can be recreated by seeing the content of the visitedNodes
- we save in a variable the total number of visited nodes, initially 1; we have this in an if because we want to initialise it only once, in the first call of the recursive function
- if we are in the point in which we know that the tour was found, we must get out of the recursion (we went through all the nodes of the graph)
- the current node is marked as visited because we are working with it; we go through all neighbours of the current node except the node itself
- we apply Dijkstra/Prim algorithms to the graph (only the unvisited nodes)
- we retrieve the heuristic cost - the cost of all the edges of the shortest path; final cost = edge cost + heuristic cost
- we create a node object for the neighbour and we add it to the queue
- while we can go down into the recursion, we search for a path; algorithm goes back if we find a bad heuristic and we unwind to get to the second best node which is upper in the search graph; each return from the recursion gets up a level in the search, because each new recursion goes to the neighbours of a node, so "deeper"

→ we use peek method which gives us the top node from the queue without removing it from the queue; we also save the index of this node

→ if the heuristic value of the main node is worse than the fLimit, then we must go back and choose the alternative node (second minimum)

→ we update the costs while going back to all parents of the node which had a worse fLimit, until we reach the alternative; we mark the current node as unvisited because it is no longer a good option
































→ in order not to modify the original pQueue, we save it into an auxiliary one; we remove the top value and peek the other one (we get the second smallest value); this is the alternative node (we return here depending on the fLimit value)














































→ then we check if the neighbour with the minimum cost is good; if so, we go deeper in the recursion (one level down); if the neighbour represents a good option, we add it to the visited list; we mark it as visited; we continue to search through its neighbours

→ else, we can't go any further, it's a dead end, so we go back; on the way back, we update the heuristic value; we mark the current node as unvisited because it is no longer a good option;

→ we get the total cost of the tour for the travel through all nodes; we iterate through the visitedNodes list and sum the costs of the path; we get from costMatrix the cost for adjacent edges and add them

## Overview of the classes

Source File 	Total Lines
 <b>AStarDijkstra.java</b>	 134
 <b>AStarPrim.java</b>	 132
 <b>CostComp.java</b>	 24
 <b>Dijkstra.java</b>	 155
 <b>Edge.java</b>	 49
 <b>GraphCostMatrix.java</b>	 42
 <b>GraphEdgeList.java</b>	 70
 <b>IntegerPair.java</b>	 35
 <b>MSTComp.java</b>	 23
 <b>Node.java</b>	 71
 <b>NodeMST.java</b>	 36
 <b>Prim.java</b>	 157
 <b>RBFSdijkstra.java</b>	 171
 <b>RBFSprim.java</b>	 171
 <b>Total:</b>	 <b>1270</b>

Comment Lines	Comment Lines [%]	Blank Lines
 41	 31%	 16
 41	 31%	 15
 4	 17%	 4
 32	 21%	 23
 5	 10%	 8
 7	 17%	 8
 8	 11%	 14
 3	 9%	 6
 3	 13%	 4
 6	 8%	 11
 4	 11%	 6
 38	 24%	 21
 55	 32%	 19
 55	 32%	 20
 <b>302</b>	 <b>24%</b>	 <b>175</b>

## Experiments and results

In order to generate the input tests, I used a function that asks for the number of vertices (we will call it  $v$ ) in the graph and the maximum cost per one edge (we will call it  $\max$ ) and outputs in a file a matrix with  $v$  rows and  $v$  columns.

So  $\text{matrix}[x][y] = \text{cost of edge from } x \text{ to } y$  and also  $\text{matrix}[x][x] = 0$ ; we consider that we have an edge between any two nodes, so  $v(v-1)/2$  edges, the matrix being symmetric.

To sum up, we have a  $\text{matrix}[v+1][v+1]$  (storing nodes from index 1) and  $\text{matrix}[i][j]$  has a value of  $[1..\max]$  if  $i \neq j$  or 0 if  $i == j$ .

```
/// generates edges with values between 1 and maxCostPerEdge
/// they are saved in a costMatrix and then copied in an output file
/// in order not to have a stack overflow, I restricted the number of vertices to 100

void generateNodes(int numberOfNodes, int maxCostPerEdge)
{
    int costOfEdge = 0;
    int costMatrix[101][101] = {0};

    srand(time(NULL));

    for(int row = 1; row <= numberOfNodes; row++)
    {
        for(int column = 1; column <= numberOfNodes; column++)
        {
            if(row < column)
            {
                costOfEdge = rand() % maxCostPerEdge + 1;

                costMatrix[row][column] = costOfEdge;

                costMatrix[column][row] = costOfEdge;
            }
        }
    }

    for(int row = 1; row <= numberOfNodes; row++)
    {
        for(int column = 1; column <= numberOfNodes; column++)
        {
            file << costMatrix[row][column] << " ";
        }

        file << '\n';
    }

    file.close();
}
```

## Conclusions

For me, the implementation of this project has been an opportunity to test my level of preparation regarding the Artificial Intelligence object and the degree in which I can apply my Java and C++ language knowledge in solving problems which are bound to this area of study.

At the same time, it was a good way to improve my methods of processing and editing documents in LaTeX. I also understood the importance of having a well structured project documentation which does include a well organized source code, programs for testing the main application, non-trivial inputs etc.

It was a good exercise of correct code indentation, documentation, object oriented programming, design, algorithmic thinking and mathematics.

I enjoyed developing the source code and analyzing the situations that arose in the process because these have shown me once more the undeniable link that connects mathematics, algorithms and programming, creating the fundamentals of Artificial Intelligence.

I will finish by saying that the process of assembling all the project parts into one application has represented a very good practice for my long term growth associated to the domain of programming.

## References

1. <http://software.ucv.ro/~cbadica/ai/cap6.pdf>
2. [http://ieor.berkeley.edu/~aaswani/teaching/FA13/151/lecture\\_notes/ieor151-lec17.pdf](http://ieor.berkeley.edu/~aaswani/teaching/FA13/151/lecture_notes/ieor151-lec17.pdf)
3. <http://www1.cse.wustl.edu/~ychen/7102/Karp-TSP.pdf>
4. <https://courses.cs.washington.edu/courses/csep521/01au/lectures/lecture2slides.pdf>
5. <https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/>
6. <https://www.geeksforgeeks.org/greedy-algorithms-set-5-prim-minimum-spanning-tree-mst-2/>
7. <https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/>
8. [https://en.wikipedia.org/wiki/Minimum\\_spanning\\_tree](https://en.wikipedia.org/wiki/Minimum_spanning_tree)
9. <https://stackoverflow.com/questions/1909281/use-dijkstras-to-find-a-minimum-spanning-tree>
10. [http://auajournal.uab.ro/upload/48\\_611\\_Pop-Zelina.pdf](http://auajournal.uab.ro/upload/48_611_Pop-Zelina.pdf)
11. <https://stackoverflow.com/questions/4453477/using-a-to-solve-travelling-salesman>
12. [https://www.eecs.yorku.ca/course\\_archive/2013-14/F/3401/slides/15b-RBFS.pdf](https://www.eecs.yorku.ca/course_archive/2013-14/F/3401/slides/15b-RBFS.pdf)
13. <http://www.iosrjournals.org/iosr-jce/papers/Vol10-issue5/R0105105110.pdf>