1 Traveling Salesman Problem

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

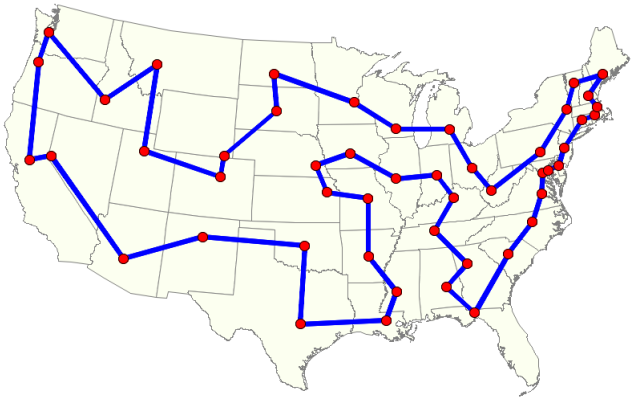


Figure 1: TSP illustration. Source: Robert Allison `http://robslink.com/SAS/democd62/new_94_sas.htm`

For many decades now, the TSP has been receiving a lot of attention from mathematicians due to its difficulty to solve, despite being so easy to describe. In fact, the TSP has a tremendous importance in the operational research area as it is a representative of many combinatorial optimization problems, meaning that advances in solving the TSP have a huge impact in many others real problems.

The biggest challenge though is that TSP belongs in the class of such problems known as NP-complete, in which there is no polynomial-time algorithm able to solve it to optimality. As matter of fact, as shown in Table 1, an algorithm that enumerates all possible routes and chooses the best is completely impracticable as the numbers of solutions explodes in $O(n!)$ when the number of cities scales.

n	**Solutions** $(n-1!)$	**CPU time**
5	24	0.000
10	362.880	0.003 sec
15	8.7×10^{10}	20 min
20	1.2×10^{17}	73 years
25	6.2×10^{23}	470 millions years

Your job for this TP is to create an A* algorithm to solve the TSP. Then, instead of exploring all solutions, i.e. generating all possible routes, your A* algorithm will project a

heuristic that estimates the route's potential, prioritizing the better ones and, consequently, avoiding bad solutions to be explored.

2 Search Strategy

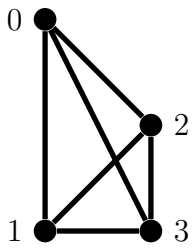
2.1 Problem Representation

For this TP we will assume the most used way to represent the TSP: a complete and symmetric graph $G(V, E)$, where V is the set of vertex (cities) and E the set of edges. As G is complete, there is an edge $(v, u) \in E$ for all $v, u \in V$ such that $v \neq u$, in other words, there is a path between every pair of cities. The cost of an edge $e_{ij} \in E$ is $c(e_{ij})$ and the symmetry ensures that $c(e_{ij}) = c(e_{ji})$.

2.2 Solutions Tree

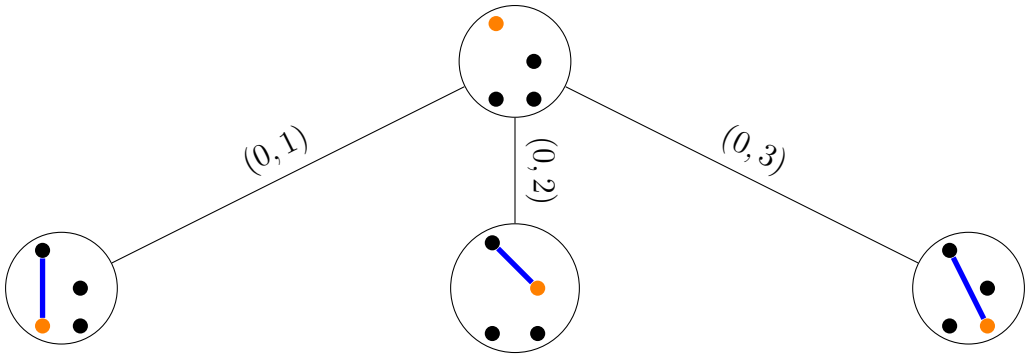
The main strategy concept of the algorithm is to perform the search over a tree of solutions. Each node of this tree will represent a solution S that is being built. A solution S has a subset of edges $E(S) \subseteq S$ and a subset $V(S) \subseteq V$ of vertices that were already **visited** (reached by an edge). When $V(S) = V$, then the solution S contains a complete route and its cost is given by $C(S) = \sum_{e \in E(S)} c(e)$.

For a more detailed way, consider the graph below with $n = 4$.

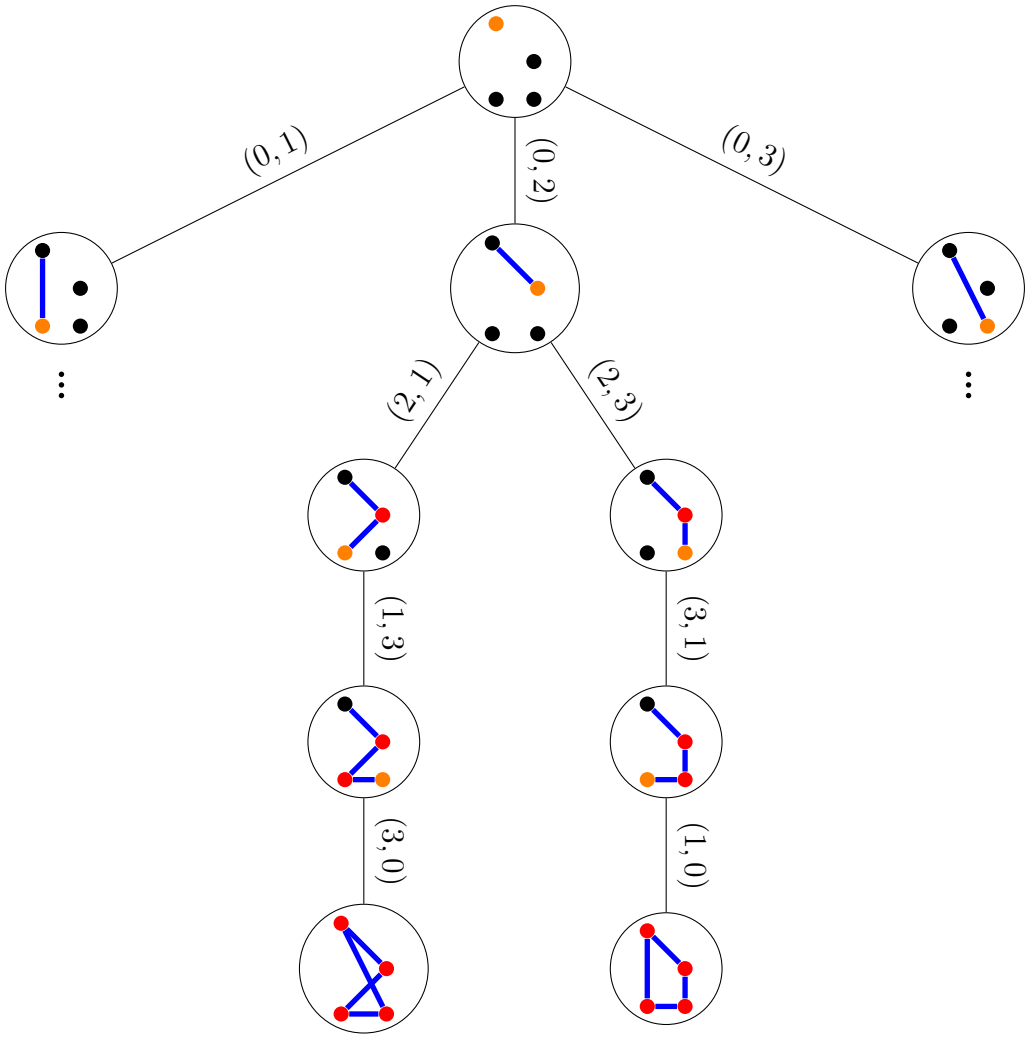


1. Start from a root *empty* solution S with $E(S) = \emptyset, V(S) = \emptyset$ and select any city as the currently point v (0 is the starting point for this example).

2. For every not visited $u \notin V(S)$, create a new solution child S' adding the edge (v, u) to $E(S')$ and setting u as the currently vertex for the new node.



3. Step 2 is repeated in every explored node in order to build completed solutions. Note that an edge returning to the start point should only be added if the start point is the only unvisited remaining vertex.



If we decide to use an ordinary queue, that works as a *first-in first-out* container, to stores the solutions and define the order in which the nodes are explored we would end up

enumerating all possible routes to find the best one. Instead, what we really want using an A* algorithm is to use a **heap** (priority queue) such that the nodes which contain the solutions with greater potential should be explored first.

3 Time to code

Note: all resources files and *skeleton* classes for this TP are provided in C++ and are available at the course web page on moodle. If you are not familiar with C++, feel free to choose any programming language that you might be more comfortable with.

- **Graph.** The `Graph.h` is a ready-to-use header file that contains the class Graph used to represent the problem. An object of this class has the following 4 useful methods:
  - `Graph(dataset_file)` the constructor method that receives a string with the dataset file name and constructs the graph itself;
  - `getN()` that returns the number of vertex n;
  - `getEdge(i,j)` that returns the object edge e_{ij} itself.
  - `getSortedEdges()` that returns a vector of edges sorted by their cost (crescent order). This method will be useful later;

  An **edge** object e has 3 attributes:
  - `e.source:` source vertex;
  - `e.destination:` destination vertex;
  - `e.cost:` cost of the edge.
- **Solution.** The Solution class (defined at *Solution.h*) has 4 variables:
  - `visited:` list of visited cities;
  - `not_visited:` list of unvisited cities;
  - `cost:` cost of the solution.
  - `graph:` the problem graph (*Graph* instance). This attribute is used by some methods of *Solution*.

  Now it is time to start coding. You must implement the 4 methods of class Solution in the *Solution.cpp* file.
  - `Solution(g):` the constructor of class Solution. It receives the *Graph* g as parameter and initializes all the solution attributes. This method will be used to create the solution for **root** node of the search.
  - `Solution(sol):` the copy constructor. This method receives another *Solution* instance, *sol*, and copies all its value to the solution being created. This method will be used to create **child** node as copy of its solutions's father and then new edges can be added;

- `addEdge(v,u)`: adds the edge (v,u) to the solution and updates its cost. v is an already visited vertex (index of the current city from the node being explored). u is an unvisited vertex being added to the route. Instead of being just an index, u is an element (*list* :: *iterator* `http://www.cplusplus.com/reference/list/`) of the `not_visited` list, this will help to removed u from it.
- `print()`: prints the solution's edges and its cost. This method should go through the `visited` list printing the exactly order in how the cities were visited. The starting point should appears twice, as the first and last element printed. Example of output:

0
2
3
1
0
`COST` = 10

- **A_star.cpp** This is the main file where the A* algorithm is implemented.

  The first thing to note here is the definition of class ***Node*** that wraps 3 attributes:

  - `solution`: node's *Solution* itself;
  - `v`: the currently vertex v (last visited city of the route);
  - `heuristic_cost`: the *heuristic cost* for that node.

  Recalling the A* algorithm, we want to choose the node i that minimizes the function

$$f(i) = g(i) + h(i)$$

  where $g(i)$ is the solution cost and $h(i)$ is the heuristic cost. Consequently, the `heuristic_cost` plays a very important role in A* algorithm, because it is used to help decide which is the next node to explore. The lower the `heuristic_cost`, the greater the node's potential. Based on that, implement the following method of *Node*:

  - `isN2betterThanN1(N1,N2)`: that receives two *Node*s, N1 and N2, and must return *true* only if the N2 should be explored before N1, or return *false* otherwise. This function will be invoked by the **heap** in order to place the best node at the top position.

  At this first stage though, only the solution cost will contribute to the A*, so you can consider the $h(i)$ term to be 0. Later on, in Section 4, we will improve this value and our search.

  Now, using the search strategy presented in Section 2, implement the two methods below:

- `main()`: creates the initial solution and *root* node of the heap and starts the search. During the search, for each node, checks if the solution is completed and if so, print the solution and finish the execution, otherwise invoke the method to explore the node.
- `explore_node(node)`: explores the node, creating its children nodes and adding them to the heap.

The documentation for heap can be accessed here: `http://www.cplusplus.com/reference/queue/priority_queue/`.

Execution. Now run your A* algorithm to the **N10.data.txt** and **N12.data** datasets and report your experiments! (All the results are provided in the Appendix A.)

- Printed Solution;
- Number of explored nodes;
- Number of created nodes;
- CPU time of execution.

Now try to do the same for **N15.data**. Can you solve it? Probably it will take a long time to finish. This is because using the solution cost as the unique cost of $f(i)$ is almost as bad as doing a complete enumeration, once only really bad solution would be avoided.

A better way though is to try to look forward at the solution and estimate how good it could be in the future (after visiting all cities). We are going to appeal to a heuristic for doing that.

Note. Before proceeding to the heuristic itself, note that it is possible to enhance the node selection by taking into account some other solution's attribute instead of only looking at the A* function $f =$ `solution.cost` + `heuristic_cost`. For example, is both nodes of function `isN2betterThanN1(N1,N2)` have the same f value, what could be used as tiebreaker? Apply the changes and report your improvements.

4 A* Heuristic

It is important highlight that the heuristic to the A* must be ***admissible***. This means that it never *overestimates* the cost of reaching the goal, i.e. the cost it estimates to reach the goal (in our case, visit all cities) is not higher than the lowest possible cost from the current partial solution.

4.1 Minimum Spanning Tree

For a given graph $G(V, E)$, a minimum spanning tree (MST) is the subset of the edges that connects all the vertices together, without any cycles, and with the minimum possible total edge cost. In other words, find the subset $E' \subseteq E$ that visited all vertices and the sum $\sum_{e \in E'} c(e)$ is minimal.

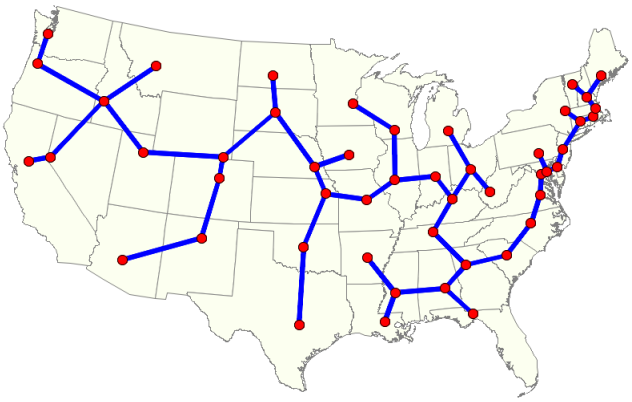
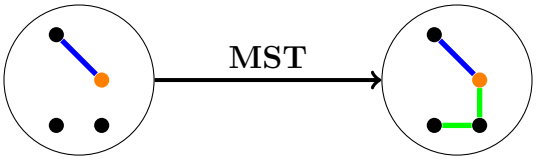


Figure 2: MST illustration. Source: Robert Allison `http://robslink.com/SAS/democd62/new_94_sas.htm`

Now imagine that you have a partial solution S being explored that has the set $V(S)$ of visited vertex and $E(S)$ of edges. If we apply an algorithm to find the MST over S, but firstly fixing the already added edges $E(S)$, we will obtain the **best** (minimum cost) set of edges $E'(S)$ that must be added to S in order to visit all unvisited vertex $u \notin V(S)$. In other words, if we obtain a MST from a partial solution S, we are obtaining a *lower bound* cost representing how good my solution could be.



In summary, the MST cost will represent the solution cost (edges of $E(S)$, all fixed) plus the minimum cost to reach every other no visited vertex (edges of $E'(S)$). This cost is a very good prediction of the solution's potential and we are going to use it as the node's heuristic cost.

One of the best-known algorithm for finding MST is Kruskal [1], which takes $O(mlog(n))$ time, with m being the number of edges. The pseudo-code below shows how the Kruskal's algorithm works.

Kruskal's algorithm

1: $MST \leftarrow \emptyset$ // initializes the tree as empty
2: Sort the edges
3: **for each** edge e **in** the sorted order **do**
4: **if** e does not create a cycle in MST **then**
5: $MST \leftarrow T \cup \{e\}$
6: **end if**

[1] Kruskal's algorithm animation: `https://en.wikipedia.org/wiki/Kruskal%27s_algorithm`

7: **end for**
8: **return** MST // return the MST

- **Kruskal** You are already provided with a *UnionFind* class able to identify a cycle in the tree. An object of this class, uf, has two major important methods:
  - `isMakeCycle(e):` that returns true only if a cycle is created after inserting the edge e.
  - `add(e):` that adds the edge e to the structure that handles the cycles. This method should be called after every insertion of an edge to the tree T (line 6 of the algorithm) to keep the structure updated.

  Now it is your turn to implement this MST algorithm by completing the following method at the *Kruskal.cpp* file:

  - `getMSTCost(sol,sp):` returns the cost of the new edges, $E'(S)'$, added to the MST obtained from *sol*. At the beginning of this procedure, remember to always include the edges of *sol* (fix $E(S)$ first). Then, implement the algorithm as presented until all cities are visited. The parameter *sp* is the ID of the start point, used to fix the first edge.

Execution. Now run your A* algorithm using the MST as heuristic to the **N10.data**, **N12.data** and **N15.data**. Report all your results and improvements.

4.2 Tightening the lower bound

Along side with the MST heuristic we can add two values to enhance the node's heuristic cost:

1. **Distance from the current city v to the nearest unvisited city**
   This cost will help the bound because it evaluates the next possible *best* edge to be added to the solution.
2. **Nearest distance from an unvisited city v to the start point**
   This cost will help the bound because it evaluates the best possible edge available to return to the start point. **Question:** If you add this value to the heuristic cost, can you assume that the first completed route found is the optimal solution? Why?

Now implement both these values and incorporate them to the node's heuristic cost during the `explore_node` method.

Execution. Rerun your experiments and reports your gains.

5 Bonus

How could you improve the lower bound even more? Are you capable of solving the instance **N17.data**?

6 Directives de remise

Le travail sera réalisé en équipe de deux ou trois. Vous remettrez un fichiers .zip par personne, nommés TD1_nom_prenom_matricule.zip. Vous devez également remettre un fichier pdf contenant une explication de vos implémentations ainsi que les réponses des questions. Tout devra être remis avant le **1 Octobre à 23h55**. Tout travail en retard sera pénalisé d'une valeur de 10% par jour de retard.

Barème :

implémentations:	50pts
résultats:	35 pts
rapport :	15pts

*Ce TP a été imaginé par Daniel Aloise et développé par Rodrigo Randel

A Results

Dataset	Optimal Cost
`N10.data`	135
`N12.data`	1733
`N15.data`	291
`N17.data`	2085