**Problem Representation**

Travelling Salesman Problem, with ‘n’ number of cities can be defined as a search problem as follows:

* **State**: Each state is represented by a tour as follows:

A -> city1 -> city2 -> …… -> cityk -> A

* + Where k <= n – 1 , and
  + each of the cityis , 0 <= i <= k, are different, and
  + the ending A is present only if k = n-1

By this definition, these are some of valid states of a 5 city – A, B, C, D, E TSP problem:

* + A
  + A->B
  + A->C->B->D
  + A->C->D->B->E->A

And these are not states:

* + A->B->C->B (B repeating)
  + A->C->B->A (ending city can be A only if all the other cities are present in between start and end)
  + B->A->C (not starting with A)
  + A->A (ending city can be A only if all the other cities are present in between start and end)
* **Initial State**: The salesman is at city A and the state is represented by the tour A
* **Successor function**: Given a state S, the tour of which is represented by T, then the successor function generates all states represented by the tour

T->x where x is any one of the unvisited cities. All such states are the successors of the state S

For instance, for a 6-city problem if the current state is A->C->E, and (B, D, F) are unvisited, then successor function generates the states A->C->E->B, A->C->E->D and A->C->E->F

* **Goal State:** Any state represented by a tour of the form

A -> city1 -> city2 -> …… -> cityn-1 -> A

Where, each of the cityis , 0 <= i <= n - 1, are different (i.e. all cities other than A are represented exactly once)

* **Goal Test:** If the current state is a goal state
* **Solution:** Any tour represented by a goal state such that the cost(tour) is minimal
* **Cost of a state, f(state)** – defined as

f(state) = g(state) + h(state)

* **g(state)** = sum of the Euclidean distances between consecutive cities on the tour represented by the state
  + for instance, if the state is A->D->C->B, then g(A->D->C->B) = distance(A to D) + distance(D to C) + distance(C to B)
* **h(state)** – defined in next section
* **Current city of a state** – is the last city on the tour represented by that state

**Heuristic Function**

The heuristic function that is used can be defined as:

* **h(state)** = d1 + d2 + d3
  + where d1 = Distance from the current city of the state to the nearest unvisited city
  + d2 = The Cost of the Minimum Spanning Tree formed by all the remaining unvisited cities
  + d3 = Shortest distance from one of the unvisited cities back to the start city
* This function can be used as a heuristic because h(goal state) = 0
  + For a goal state,
    - d1, d2 and d3 = 0, as there are no more unvisited cities
* This is an admissible heuristic because
  + 0<=h(state)<=h\*(state) where h\* is the actual cost of the state
  + Consider an intermediate state x, and let there be 5 more remaining cities, c1, c2, c3, c4 and c5. Let the best path from this state be - go to c1, then c3, c4, c5, c2 and back to start city. There are 2 possibilities:
    - Case1: As per our heuristic, c3 was closest city from current city. So d1 = distance from current city to c3. Let d2 be x. d3 = distance from c4 back to start city because c4 was closest to start city.
    - Case2: The heuristic chose c1, because c1 was closest to current city. So d1 = distance from current city to c1. d2 will be x(same as case 1 – mst cost will remain same). d3 = distance from c5 back to start city.

In case 2, the heuristic was able to score same as the actual cost h\*. And in case 1, it scored lesser than h\* as it chose some poorer option for next city(d1) and the last city(d3) before returning back to start. In no cases will the heuristic value be greater than the actual cost. Hence, h is admissible

**Implementation Details**

**How to Run:**

* Requires Python 3, Matplotlib
* Download the code into your folder. Important files are:
  + state.py - Generic Class named State for representing a State of the search problem.
  + tspState.py – Contains subclass TSPState representing the state of TSP problem. Implements all the operations of the State superclass and additional methods specific to TSP
  + open\_list.py – generic class for Open List of a search problem
  + closed\_list - generic class for maintaining Closed List of a search problem
  + tsp\_open\_list.py and tsp\_closed\_list – which are concrete implementations of Open List and Closed List for a TSP problem
  + tsp\_data.py - The data object that stores all the input data of the TSP Problem. Loads the data from the file
  + tspHelper.py - Helper class for carrying out book keeping tasks related to TSP problem. For eg. Maintains list of unvisited cities
  + AstarSearch.py – contains the A\* search algorithm
  + tsp\_runner.py – Runs A\* search on all the input TSP problems
  + result\_plotter – plot the results from the output data

**Separating search algorithm from TSP related functions:**

The TSP related functions and search related functions are separated using runtime polymorphism and inheritance. AstarSearch.py is a generic A\* search algorithm and is programmed in a generic way. For any specific search search problem (like TSP) the classes state.py, open\_list.py and closed\_list.py only needs to subclassed and methods implemented. In this TSP example, the subclasses tspState.py, tsp\_open\_list.py and tsp\_closed\_list.py are concrete implementations

**Open List and Closed List:**

Open List keeps track of all the states that have been generated so far but not yet explored. The next state is always the lowest cost (least f value) state from the Open List. Here, tsp\_open\_list contais a Priority Queue implementation of Open List with priority given for a state with lowest f value.

The Closed List is used to keep track of all states which have been already visited and processed (means their successors generated). It ensures that we do not loop around in the same state forever. Here, tsp\_closed\_list used python sets operations for implementing Closed List

1. To run A\* search on a single TSP problem
   1. Open AstartSearch.py and edit the variable *inputFilePath* with the absolute file path of the input probem
   2. To run for h(state) = 0, set value of variable *noHeuristic* to True
   3. Run AstartSearch.py
2. To run on all the instances of the TSP problem
   1. Open tsp\_runner.py and set the variable with path of the root folder of the input data
   2. This will take long time to complete depending on the number of problems
3. To plot the results
   1. Make sure you have the output-data generated by running tsp\_runner.py
   2. Run result\_plotter.py

**Assumptions for the implementation**:

* The starting city is assumed to be always ‘A’
* For any state which is not a goal state (say s) when there are no more unvisited cities, then the successors are obtained are adding the starting city A to the end. For e.g, for a 5-city problem, the successors of A->C->D->B->E are {A->C->D->B->E->A}. This helps in forming a tour that ends back at start city and hence reach the goal state