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:

$$A \rightarrow city_1 \rightarrow city_2 \rightarrow \ldots\ldots \rightarrow city_k \rightarrow A$$

  - Where $k <= n - 1$, and
  - each of the $city_i$s , $0 <= i <= k$, 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

$$A \rightarrow city_1 \rightarrow city_2 \rightarrow \ldots\ldots \rightarrow city_{n-1} \rightarrow A$$

Where, each of the $city_i$s , $0 <= i <= n - 1$, 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 contains 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
    a. Open AstartSearch.py and edit the variable *inputFilePath* with the absolute file path of the input probem
    b. To run for h(state) = 0, set value of variable *noHeuristic* to True
    c. Run AstartSearch.py
2. To run on all the instances of the TSP problem
    a. Open tsp_runner.py and set the variable with path of the root folder of the input data
    b. Set variable *version* ‘v1’ for running with heuristic and ‘v2’ with h(n) = 0
    c. This will take long time to complete depending on the number of problems

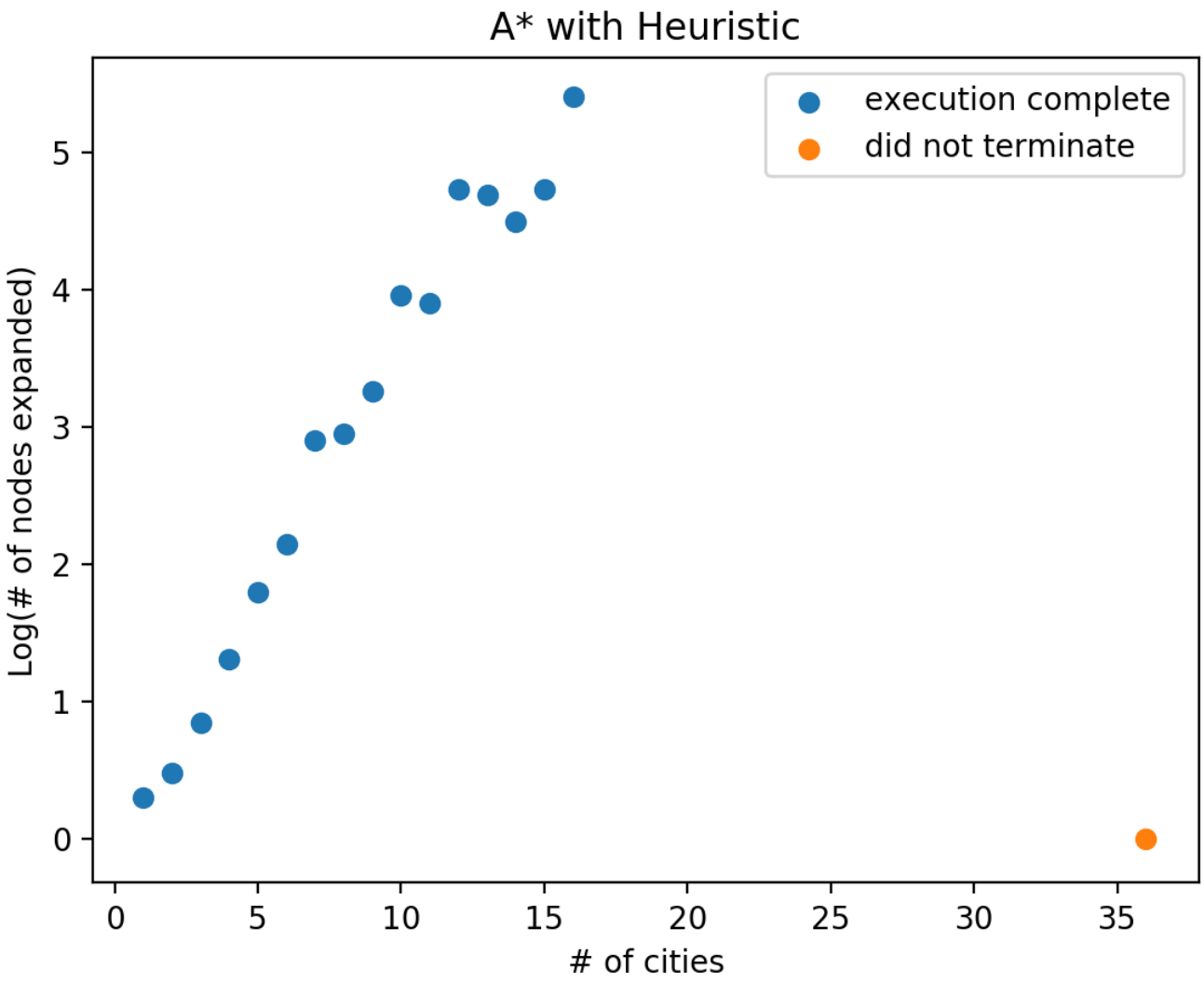
3. To plot the results
    a. Make sure you have the output-data generated by running tsp_runner.py
    b. Set the path of output folder in variable *outputRootFolder*
    c. Set variable *version* as ‘v1’ for heuristic and ‘v2’ for h(n) = 0
    d. Run result_plotter.py
    e. Save and close the output plot
    f. Save and close the predicted plot

Assumptions for the implementation:

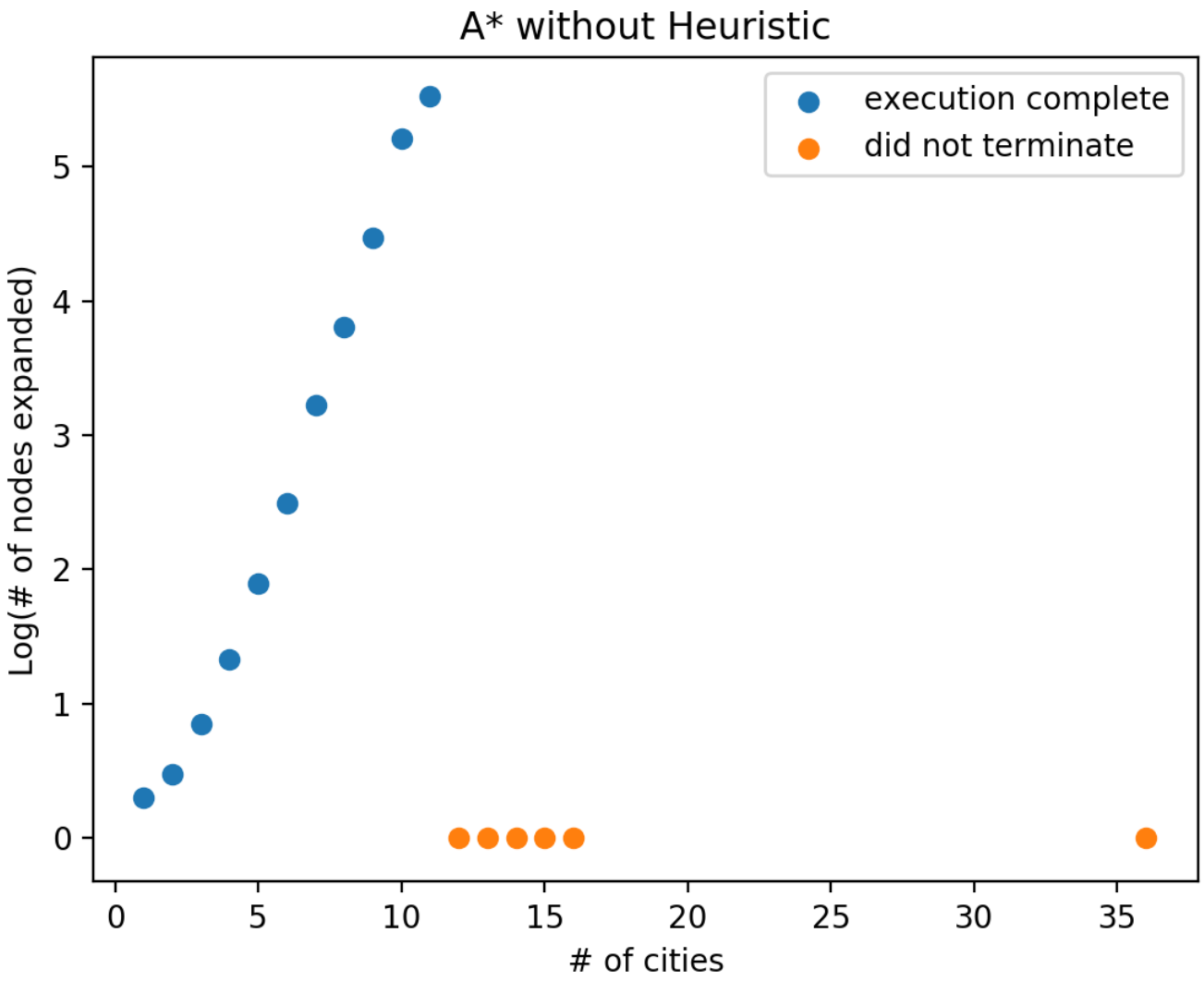
- The starting city is assumed to be always ‘A’
- For any state which is not a goal state (say s) when there are no more unvisited cities, then the successors are obtained are adding the starting city A to the end. For e.g, for a 5-city problem, the successors of A->C->D->B->E are {A->C->D->B->E->A}.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

Results

A* search with heuristic



A* search with h(n) = 0



Assumptions for the plots:

- For each data point on x-axis, the y-axis value is the base 10 logarithm of average # of assignments over the 10 instances of the TSP with same # of cities
- For problems where algorithm did not terminate, followed the below given approach:
  - The orange dot indicates that it did not terminate. There is no count obtained for such data points
  - Out of the 10 instances of the TSP problem, if at least 8 of them terminated then the count of nodes is plotted as blue dot, otherwise they are classified as 'not terminated' and plotted as an orange dot

output data

# of cities	A* with heuristic	A* with h(n) = 0
1	2	2
2	3	3
3	7	7
4	20	21
5	62	78
6	141	309
7	795	1684
8	896	6356
9	1807	29536
10	9109	162330
11	8037	336681
12	53878	Did not terminate
13	49464	Did not terminate
14	31048	Did not terminate
15	54220	Did not terminate
16	255577	Did not terminate
36	Did not terminate	Did not terminate

Comparison of Performance

When a heuristic function (such as the one described above) is used, it provides a guidance to the search algorithm on which states are better than others, without having to generate the entire state space tree.

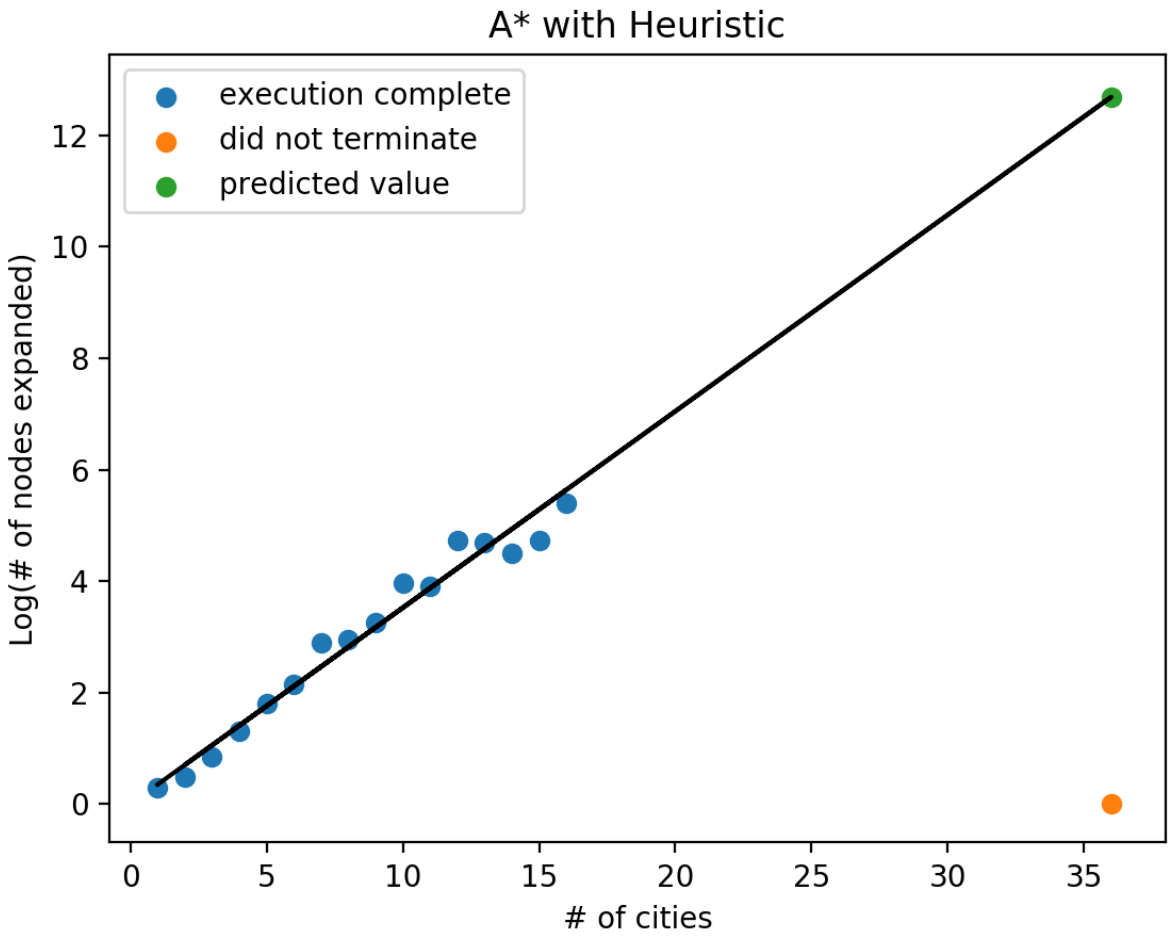
i.e. for a 5-city problem, we can compare f(A->D) and f(A->C->B->D->E) and tell which state would lead to an optimal goal state, and if f(A->D) is higher than f(A->C->B->D->E), then we can decide never to explore any paths generated from A->D. This helps us to prevent generating huge parts of the tree which do not lead to optimal goal state and we reach the solution faster.

When h(n) = 0 is used, f(n) = g(n) and the cost of a path becomes the total cost of the path represented by 'n'. Almost in all cases this does not allow us to compare and choose between A->D and A->C->B->D->E because obviously A->D will have lower cost than A->C->B->D->E. This causes the algorithm to go down A->D and generate successor state even though it may not turn out to be optimal.

Essentially when h(n) = 0, the search algorithm becomes a DFS on the state space tree and the algorithm would need to generate the entire state space tree to reach a goal state. Eventually after expanding enough nodes, the Open List will contain only the complete tour of n cities and we then choose the lowest cost tour and output this as the optimal solution.

36-city problem and extrapolating results

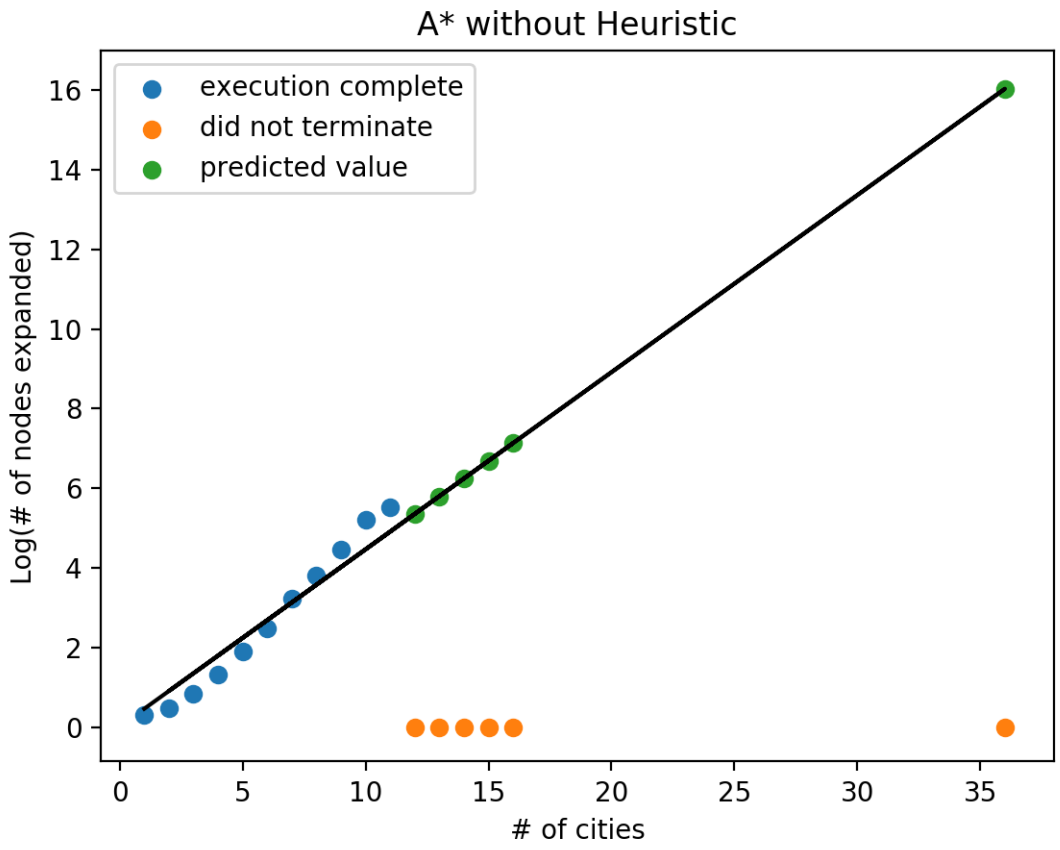
with heuristic function



As obtained from above plot the extrapolated value for 36 cities = 12.686190
This gives number of expanded nodes = 4.855 X 10^{12} nodes = **4.855 trillion nodes**

How much time will my program take for 36 cities problem?
When I ran instance_5 for 12 cities on Apple MacBook Pro with 2.7 GHz Intel Core i5 processor , it took 0.452322 minutes with 69748 nodes expanded. By simple scaling this value to 4.855 trillion nodes, it can take **59.9 years** on same machine

with h(n) = 0



As obtained from above plot the extrapolated value for 36 cities = 16.028318
This gives number of expanded nodes = 1.067 X 10^{16} nodes = **10670 trillion nodes**

How much time will my program take for 36 cities problem?
When I ran instance_1 for 10 cities on Apple MacBook Pro with 2.7 GHz Intel Core i5 processor, it took 2.900549 minutes with 163770 nodes expanded. By simple scaling this value to 10670 trillion nodes, it can take **359546 years** on same machine