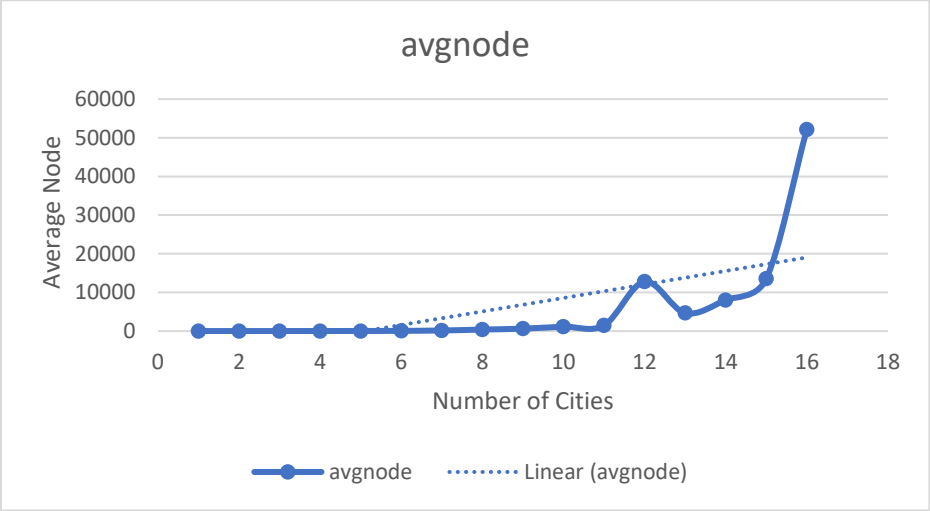
Q2 – TSP

a) Let state be a tuple ([a1, a2, a3, …, an], cost)
where [a1, a2, a3, …, an] is list of visited cities that start from a1, then a2, …., to an
and cost is the total moving cost of these cities

- Start state: ([A],0) we always starts from city A
- Goal state: ([A...A],totalcost)
  The goal state should visit all cities once, starting from A and ending at A
- Successor function: based on visited cities, we add a new city (next visit city) to the list by A* Search
- Cost: each move is the Euclidean distance between the last visited city and the new adding city

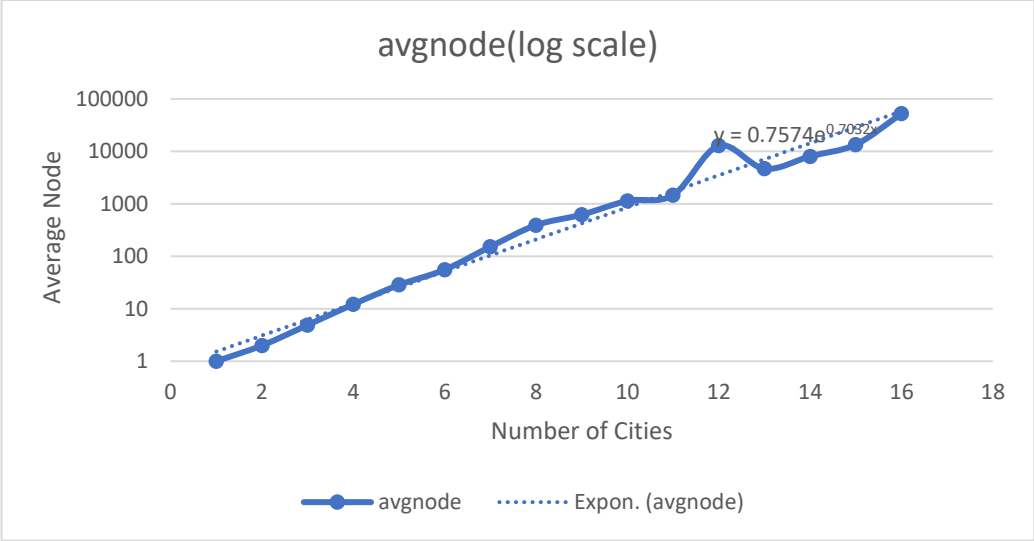
b) Let heuristic function be the cost of MST (using prim algorithm) for unvisit cities. Since we need to go back to city A, "A" is also added to the unvisit list, then MST will return the minimum cost to visit these cities. Since it will return the minimum cost, it will never overestimate the actual cost to visit the remaining cities. This heuristic function is admissible.

c)



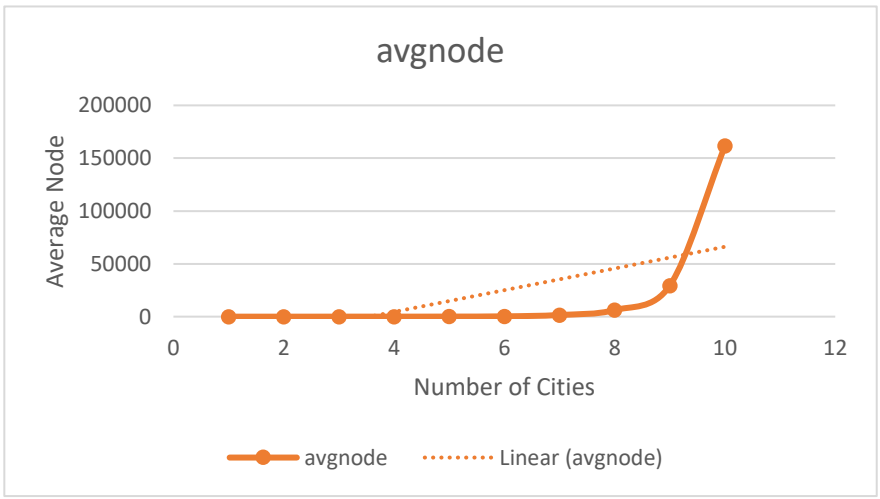
numofcity	avgnode	numofcity	avgnode
1	1	9	620.3
2	2	10	1138.7
3	4.9	11	1468.3
4	12.2	12	12796.8
5	28.5	13	4670.2
6	55.6	14	8046.7
7	152.5	15	13525.9
8	392.5	16	52122.5

d)

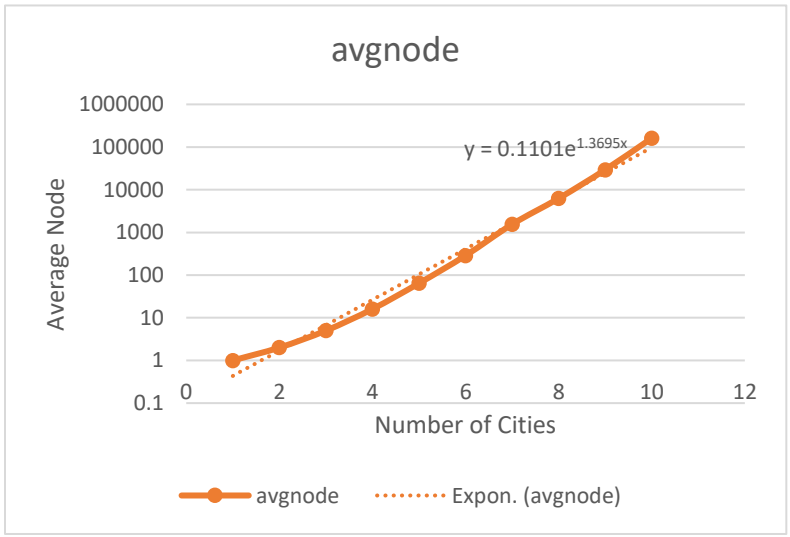


The trend line gives the estimated nodes for 36 cities is $0.7574e^{0.7032\times36} = 7.47 \times 10^{10}$, for 122366, it takes around 50 seconds, than running this instance might take one year in worst case.

e) Since for h(n) = 0, it takes too long to run for 10 cities +, the graph only shows for 1-10 cities.



f)



$0.1101e^{1.3695\times 36} = 2.84 \times 10^{20}$ for 36 city instance

It takes around 10 seconds to run around 340000 nodes, so it might take 2.32×10^{12} hours which is long long long away.

g) My heuristic functions is based on MST for unvisited cities, then A* Search will decide the next city by taking the minimum(actual cost of visited cites + MST estimated cost for unvisit cities), this will reach out a faster solution because priority queue always pop the visited list with minimum estimated total cost. However, for h(n) = 0, A* Search only consider the cost for visited cities, it might produce lots of choices that will have similar cost, then it will take longer time to get the solution.

Q3 - Sudoku

a) **Variables**: V11, V12, …, V21, V22, …, V91, …, V99
where Vij represents the value in the cell for row i, column j
Domain:

- $Vij \in \{1,2,3,\ldots,9\}$ if the corresponding cell is empty
- $Vij \in \{k\}$ if the crresponding cell is fixed

Constraints:

- Row Constraints: CR1, CR2, …, CR9
  For row i, for different columns j & k, $Vij \neq Vik$
  eg. CR1(V11, V12, V13, …, V19)
  …
  CR9(V91, V92, V93, …, V99)
- Column Constraints: CC1, CC2, …, CC9
  For column j, for different rows i & l, $Vij \neq Vlj$
  eg. CC1(V11, V21, V31, …, V91)
  …
  CC9(V19, V29, V39, …, V99)
- Sub-Square Constraints: CSS1, CSS2, …, CSS9
  eg. CSS1(V11, V12,V13, V21, V22,V23, V31,V32,V33)
  …
  CSS9(V77, V78, V79, V87, V88, V89, V97, V98, V99)

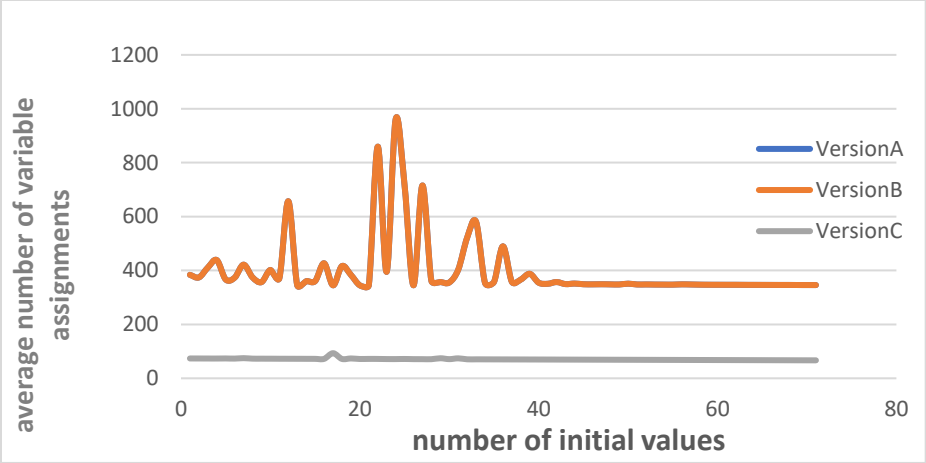
b) Please read the code instructions below to run my code.

c) Every version of the code will print out the variable assignments count for each instance.

d) See next page

e) See next page

f) See next page



Findings:

- From the graph, it is obvious that the variable assignments for **version A is the same as version B**. This might because we do backtracking search in version A, even though we try all possible values in order, we check if the value could be assigned to particular grid before we assign the value. For version B, we did not try every possible values, instead, we did reduction of the domain (filter out those values have been existed). The condition for checking a variable could be assigned is the same(one is after try, one is before try), so the average assign shows the same. **Version C has the least variable assignments**. This is because we implement heuristics based on version B. When we want to find an empty gird to assign value, we won't do it in order. Instead, we choose the one with least domain values first(**minimum-remaining-values** heuristic), if there is a tie breaker, we choose the one which involves other unassigned variables the most(**degree** heuristic). This selection method will find the solution quicker than just doing "order selection". In addition, when we assignment values to a variable, we are not trying the values in order, but choose the one influences its neighbour the least(**least-constraining-value** heuristic). These 3 heuristics makes version C much more efficient to find the solution.
- Talking about the number of initial values. For **version A and version B**, in the range of 20-40, it takes more variables assignment than others. It could be understood, when number of initial numbers are small, we have lots of grids to fill in so we have lots of possibilities; also, when we have large initial numbers, we only have few grids so we could fill them with few tries. However, it is hard to fill the grids when the number of initial values are in the middle range, there are lots of constraints. For **version C**, it is obvious that for all range 1-71, number of variable assignments keep stable around 70. The implement of heuristic makes the program do a better selection for values and filter out many unnecessary options.

Running Code Instruction

The input for all code is for one file. Please change the file path manually.

Make sure that all the files are in the same directory (the python file and the instances head folder)

Q2

- To run the MST version of A* Search, run with "./tsp.py"
- To run h(n) = 0 of A* Search, run with "./tsp2.py"
- The data for plots are in "OutputTSP.txt", "OutputTSP2.txt", "tsp.xlxs"

Q3

- To run version A of sudoku, run with "./sudoku-1.py"
- To run version B of sudoku, run with "./sudoku-2.py"
- To run version C of sudoku, run with "./sudoku-3.py"
- The data for plots are in "Output1.txt","Output2.txt","Output3.txt","sudoku.xlsx"