**AI Mini Project report**

Submitted by

Group Information

**Group Leader**  **:** Rishabh Sahu 19ucs129

Member2 : Manit Mehta 19ucs147;

Member3 : Aditya Khandelwal 19ucs163;

Member4 : Abhishek Singhal 19ucs127;

Working on option1 of the list of topics

**Project overview**

Travelling Salesman Problem (TSP)

a) Study and solve it using minimum-spanning-tree (MST) heuristic.

b) Find and implement an efficient algorithm in the literature for constructing the

MST, and use it with A\* graph search to solve TSP.

c) Study and report current state of the art methods which show (significant)

improvement over MST heuristic.

**Task Definition**

The travelling salesman problem (TSP) is a problem in computational mathematics and combinatorial optimization that has received a lot of attention. It's also known as the NP-complete class of combinatorial optimization problems. Many methods and methodologies have been proposed in the literature to solve the TSP. However, **there are currently no algorithms that can provide a precise optimal solution to the TSP issue.**

Many methods solved real-world problem with help of AI

Various heuristics and approximation techniques have been designed to quickly produce good solutions. Modern approaches may identify solutions to extremely huge issues (millions of cities) in a reasonable amount of time that are, on average, only 2–3% from the best solution.

**Applications of TSP**

A) The TSP naturally arises as a subproblem in many transportation and logistics applications, such as determining school bus routes to pick up students in a school district (Vehicle Routine Problem).

B) Another classic example is the scheduling of a machine to drill holes in a circuit board or other object, the cost of travel is the time it takes to move the drill head from one hole to the next.

**Formulation and modelling the TSP**

Let G= (V, E) denote a complete undirected graph with vertices V, |V|= n, and edges E, and dij denote the edge length (i, j). The objective is to find a tour that visits each vertex exactly once and whose total length is as small as possible.

**We model the task as a state space search problem where the states represent the cities and the edge costs are the distances between the cities.**

**Approaches**

**Infrastructure**

To execute our programs, we have used Ubuntu 20.04.2 LTS 64-bit virtual machine on VirtualBox which is working on 3GB allocated RAM and 4 processor cores of intel core i5 8250U CPU. The name of the device used is HP Pavillion x360 convertible and the host operating system is windows 10 which is also 64 bit.

**Method 1 Using A \* search with MST heuristic**

**Theory**

The A\* (or A star) algorithm finds the shortest path between two nodes. It is considered an extension of the Dijkstra algorithm, however instead of using a heuristic to discover the best solution, it uses a heuristic to find the best solution. It functions on the following parameters

f'(i)=g(i)+h'(i)}

F(i)represents the estimated total costs of a path from the start node, through the node i to the endnode(i) is the distance from the start node to node,h(i) is the estimated distance from ith node to the end node, and is the the heuristic function. The function h(i) has to full fill the criteria that it doesn't overestimate the real distance to the endnode.

**Astar search**

## Generic Steps of the A\* algorithm

1. Add the start node s to the OPEN list.
2. If the OPEN list is empty, end the algorithm with no result. In this case a solution cannot be found.
3. Chose node n from the OPEN list which has the minimal *{\displaystyle f'(i)}* and move it from the OPEN to the CLOSED list.
4. If node *n* is the destination node the algorithm has found the optimal solution and terminates. To get the shortest path from the start node to the end node travel back from n to s.
5. Otherwise expand n. That mean you create the successor n' from n with a reference from n' back to n and determine for every successor n' the following values:
   1. If n' doesn't exist in the OPEN- or CLOSED list, estimate *f'(n')=g(n')+h'(n')}* and put Node n' on the OPEN list.
   2. If n' already exist in the OPEN- or CLOSED list and you have found a better connection, then you have to update the back reference of n' and its value for *{\displaystyle f'(n')}*.
   3. If n' is on the CLOSED list and its reference was modified then put n' back on the OPEN list. (This only applies for none monoton estimaters. In our case this step can be left out because our estimator for the TSP is monoton.)
6. Go to step 2.

**Heuristics criteria for A star:**

**Admissibility:** The admissibility criteria for heuristics states that the heuristic never overestimates the expected cost to reach the goal. For a traversed path c1->…ck among all the cities c1-> …->cn if we generate the Minimum spanning tree for all the traversed cities from ck we deduce that the path from ck to c1 has path length equal to MST path length and so it must have cost less than or equal to TSP solution.

**Consistency:** When A star search is applied using graph search the consistency criteria is considered which says that

heuristic h(n) is consistent if, for every node n and every successor n’ of n generated by any action a, the estimated cost of reaching the goal from n is no greater than the step cost of getting to n’ plus the estimated cost of reaching the goal from n’:

h(n) ≤ c (n, a, n’) + h(n’).

**Preliminary: Minimum Spanning Tree**

* Given an undirected graph G=(V, E) with arc lengths d(ij)’s.
* A spanning tree T is a subgraph of G that is a tree (a connected acyclic graph), and spans (touches) all nodes.
* Every spanning tree has (n-1) arcs.
* Length of a spanning tree T is Σ (i,j) ∈ dij.
* The minimum spanning tree problem is to find a spanning tree of minimum length.

**A Minimum Spanning Tree Based Heuristic**

We used the **Prim’s algorithm**  to get the minimum spanning tree for generating MST heuristic.

**Step 1:** Select a starting vertex

**Step 2**: Repeat Steps 3 and 4 until there are fringe vertices

**Step 3**: Select an edge e connecting the tree vertex and fringe vertex that has minimum weight

**Step 4**: Add the selected edge and the vertex to the minimum spanning tree T  
[END OF LOOP]

**Step 5**: EXIT

**Application**

The code for this part is saved in Astar.cpp and executed using the executable Astar.out .

First, we generated a graph class object which stores edges between each pair of cities after calculating the distances in the readMap() function. Then Astar search is applied starting from city 1 using the Boolean solve method .The MST heuristic is recalculated for all cities using Prim’s algorithm using the calculateHeuristic() function at each step. Finally, after all the cities are added to path the time taken and optimal path are returned as output. The Fringe list used for Astar search is implemented in the FringeList class.

bool solve()

    {

        //initial state i.e startNode

        fringeList.push( FringeListNode( startCity, 0, 0, {}, {} ) );

        // run while we have states in the fringelist or until we find the optimal solution

        while (!fringeList.empty())

        {

            // select node with least f cost

            FringeListNode curr = fringeList.top();

            fringeList.pop();

            //Goal test

            if (isGoalState(curr))

            {

                solution = curr;

                solution.path.push\_back(curr.nodeId);

                calculateCost();

                return true;

            }

            //get all adjacent nodes of current node i.e all - {already visited, itself}

            vector<int> successorsNodes = getAdjacentNodes(curr);

            // for each successor node insert corresponding state in the fringeList

            for (int successor : successorsNodes)

            {

                FringeListNode successorState;

                successorState.nodeId = successor;

                successorState.gValue = curr.gValue + dist[curr.nodeId][successor];

                successorState.path = curr.path;

                successorState.path.push\_back(curr.nodeId); //add current node id to the path of successor

                successorState.visitedNodes = curr.visitedNodes;

                successorState.visitedNodes.insert(curr.nodeId);   //similarly add current node id to the visited set

                int h = calculateHeuristic(successorState);        // MST heuristic

                successorState.fValue = successorState.gValue + h; // f = g + h

                fringeList.push(successorState);

                }

        }

        return false;

    }

Code snippet for Astar search solve function

**Method 2 Using state of the art method: Genetic algorithm with Edge Assembly Crossover**

**Theory:**

**Genetic algorithms**

Genetic algorithms are inspired from Charles Darwin's theory of natural selection. The process of natural selection starts with the selection of fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance at surviving. This process keeps on iterating and at the end, a generation with the fittest individuals will be found.

This notion can be applied for a search problem. We consider a set of solutions for a problem and select the set of best ones out of them.

Five phases are considered in a genetic algorithm.

1. Initial population: The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve. The process begins with a set of individuals which is called a Population. Each individual is a solution to the problem you want to solve.
2. Fitness function: It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.
3. Selection: Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chance to be selected for reproduction.
4. Crossover: Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes.
5. Mutation: In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.

The population has a fixed size. As new generations are formed, individuals with least fitness die, providing space for new offspring.The sequence of phases is repeated to produce individuals in each new generation which are better than the previous generation.

**Edge assembly crossover modification**

It is recombination operator that combines the edges of two parent solutions trying to add only few, short edges not found in any of the two parents. It is applied in the crossover step.

**Application** The code for this part is saved in GA\_EAX.cpp and executed using the executable GAEAX.out. First the coordinates data is converted into an appropriate form that stores city wise travelling distances using the methods of TEnvironment class.

void Define();                         /\* Define the variables \*/

  void DoIt();                           /\* Main procedure of the GA \*/

  void Init();                           /\* Initialization of the GA \*/

  bool TerminationCondition();           /\* Decide whether to proceed to next stage

              (or treminate the GA) \*/

  void SetAverageBest();                 /\* Compute average and best tour lengths of the population \*/

  void InitPop();                        /\* Create an initial population \*/

  void SelectForMating();                /\* Determine a set of pairs of parents at each generation \*/

  void SelectForSurvival( int s );       /\* Not do anything \*/

  void GenerateKids( int s );            /\* Generate offspring solutions from a selected pair of

                                            parents. Selection for survival is also performed here. \*/

  void GetEdgeFreq();                    /\* Compute the frequency of the edges of the population \*/

code snippet 1 from genetic algorithm, a brief description about the methods of TEnvironment class are described which call all other functions to apply all parts of genetic algorithm.

void TEnvironment::GenerateKids( int s )

{

  tCross->SetParents( tCurPop[fIndexForMating[s]], tCurPop[fIndexForMating[s+1]], fFlagC, fNumOfKids );

  tCross->DoIt( tCurPop[fIndexForMating[s]], tCurPop[fIndexForMating[s+1]], fNumOfKids, 1, fFlagC, fEdgeFreq );

  fAccumurateNumCh += tCross->fNumOfGeneratedCh;

}

Code snippet 2 of genetic algorithm, showing how the offsprings are selected using the stored indexes from previous step.

**Challenges faced**

The main challenges we faced were understanding the algorithms to be implemented and selection of appropriate test cases for comparison. A very large number of cities will be infeasible to solve since Travelling salesman problem is an NP hard problem. Also, very small test data would create the possibility of not representing the proper time comparison among them. So, we tested the execution on multiple conditions to get better idea.

Initially we thought that the Genetic Algorithm will outperform A star search as well as give optimal solution as well because it has been described as a state of art algorithm and also described better than few other heuristic algorithms. But as it is visible in the results the performance of Genetic algorithm turned out to be only better in case of time but it is not always displaying optimal solution.

The genetic algorithm is very complex and it required referring many sources so that the implementation code is proper.

**Test Dataset used**

The data is provided in the form of cartesian coordinates of the cities with some supporting information in the beginning.

**For 5cities**

NAME : 5cities

COMMENT : 5city sample (from 101 cities problem of Christofides/Eilon)

TYPE : TSP

DIMENSION : 5

EDGE\_WEIGHT\_TYPE : CEIL\_2D

NODE\_COORD\_SECTION

1 41 49

2 35 17

3 55 45

4 55 20

5 15 30

EOF

**For 15cities**

NAME : 15cities

COMMENT : 15city sample (from 101 cities problem of Christofides/Eilon)

TYPE : TSP

DIMENSION : 15

EDGE\_WEIGHT\_TYPE : CEIL\_2D

NODE\_COORD\_SECTION

1 41 49

2 35 17

3 55 45

4 55 20

5 15 30

6 25 30

7 20 50

8 10 43

9 55 60

10 30 60

11 20 65

12 50 35

13 30 25

14 15 10

15 30 5

EOF

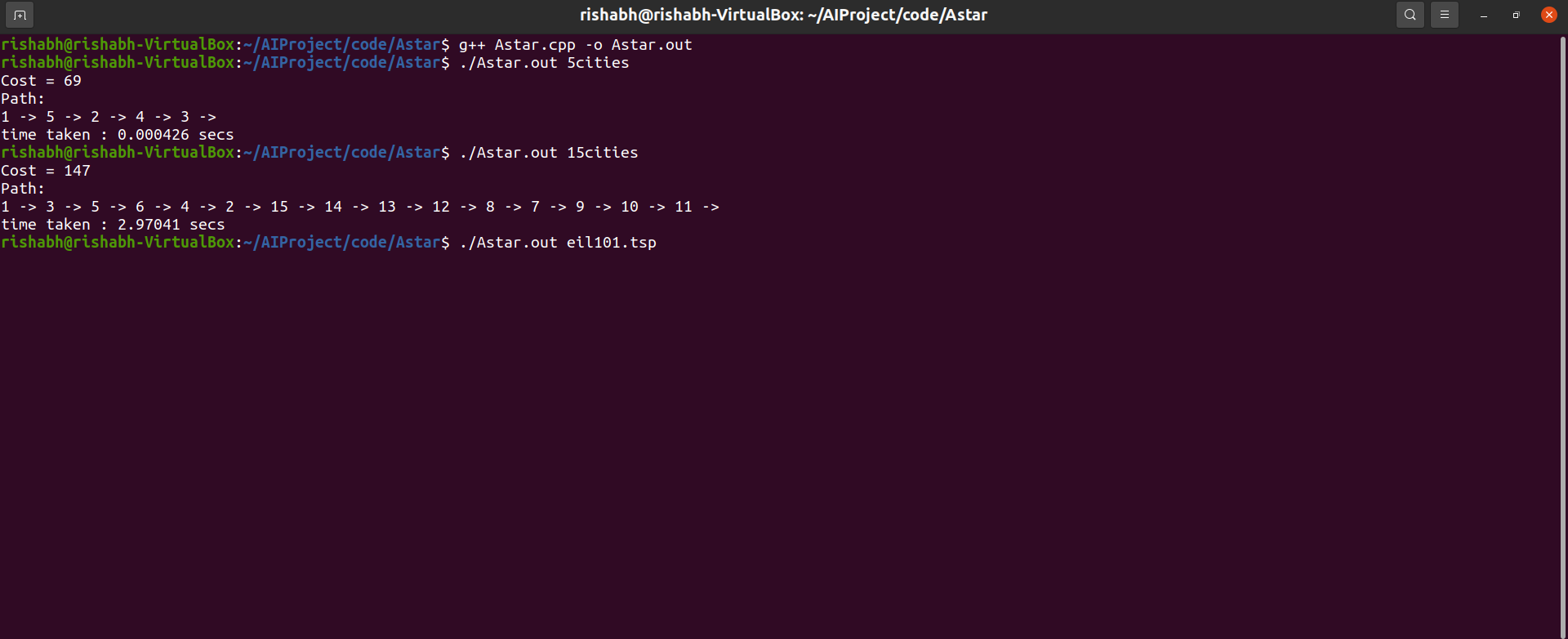
**For 101 cities**

As the data is large it is not directly shared in the report but it is available in the submission zip file the name of file is eil101.tsp

**Results**

1. **Astar search**

The execution requires single parameter, which is the name of the file from which the city coordinates are being read.

****

1. **Genetic Algorithm with EAX crossover**

The execution requires 5 parameters which are(provided in the following sequence)

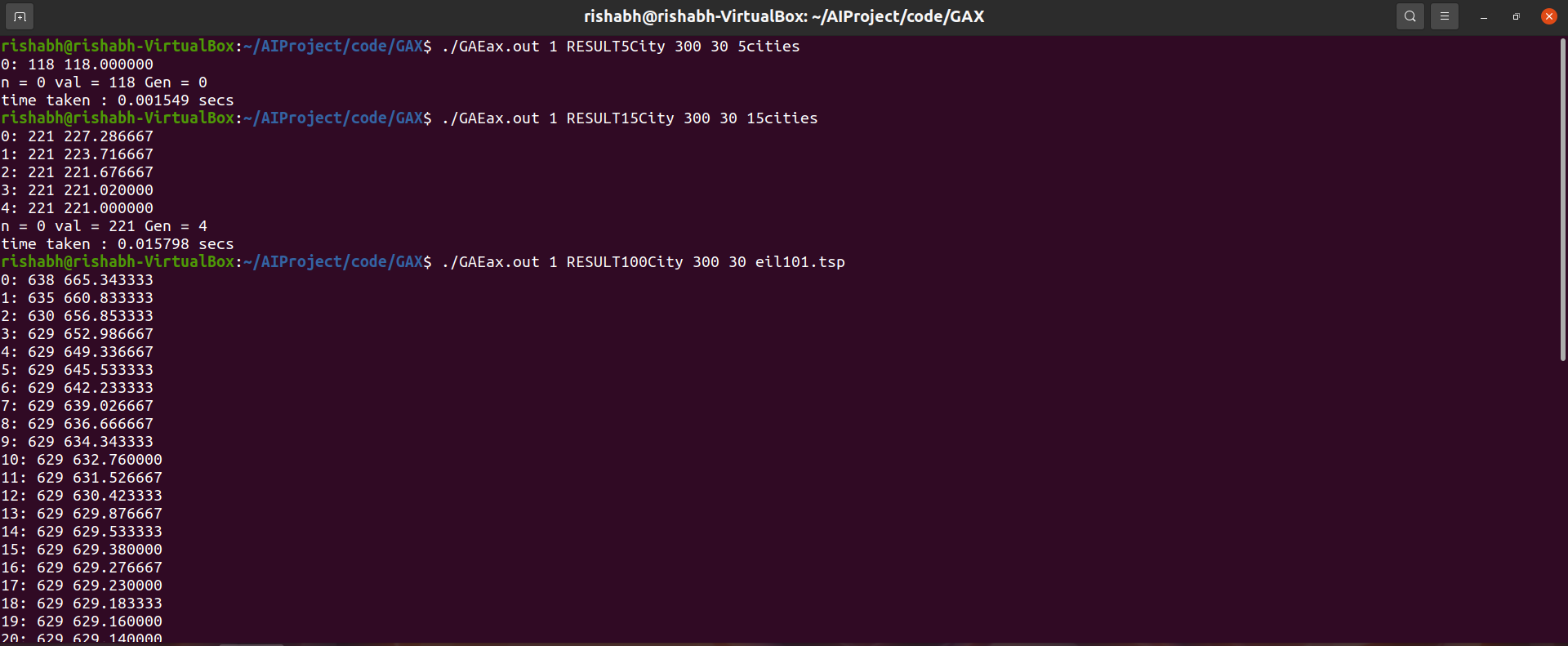
Number of trials,

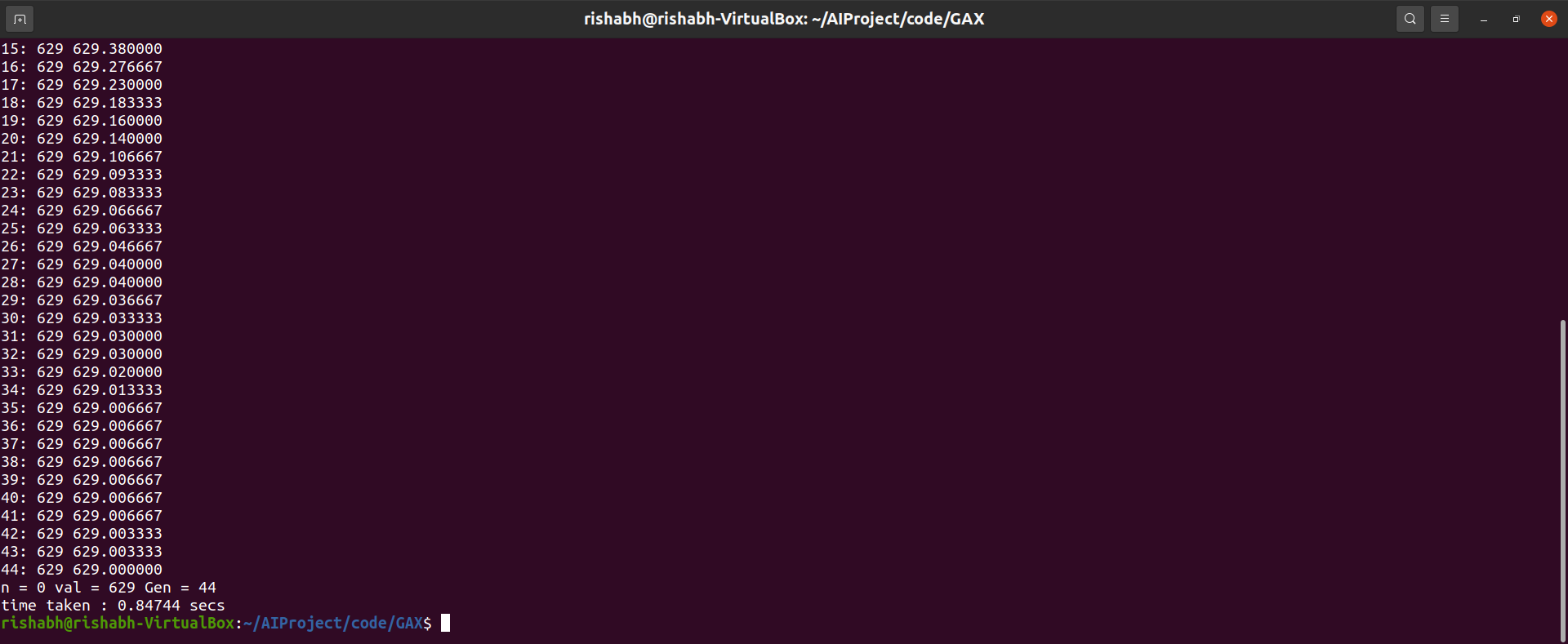
Result file name (to store the deduced optimal path)

The number of populations

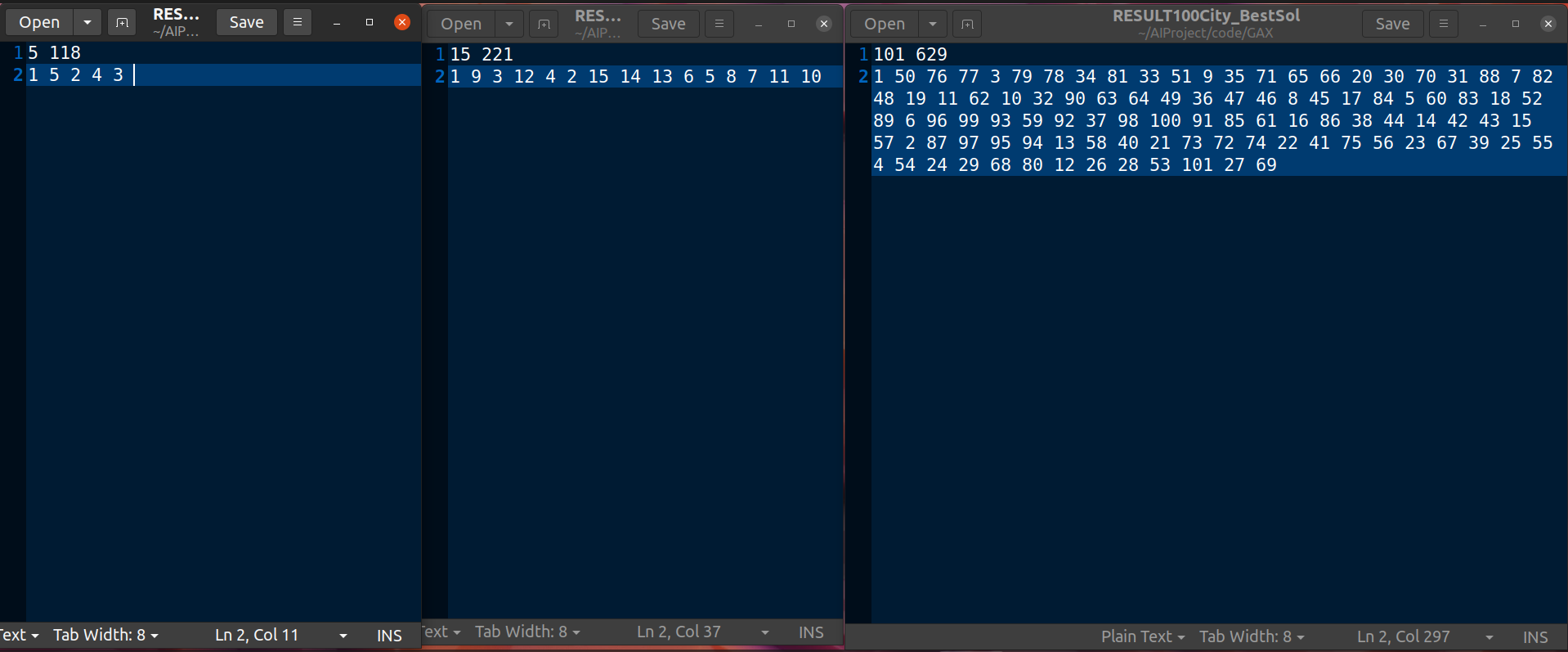
The number of offspring solutions in a step

Cities file name: to read city coordinates from



****

Paths and their costs stored in files for the above 3 cases are



Left most is for 5 cities case (5cities file), Middle one is for 15 cities (15cities file), Last one is for 101 cities (eil101.tsp file) .

The first line shows number of cities followed by the cost of the path found , the next line shows the sequence of cities in the path so found.

**Inferences**

The Astar search with MST heuristic is able to find the optimal path for 5 cities and 15 cities in a reasonable amount of time, but is not feasible for 100 cities. The Genetic algorithm with Edge Assembly crossover gives an estimated optimal path taking lesser time than A star for 5 cities and 15 cities and is able to provide an estimate for 100 cities within few seconds as compared to A star search with MST heuristic.

From this we can infer that the Astar search using MST heuristic gives an optimal solution but takes a lot of time for large number of cities, but the Genetic algorithm which has been deemed as a state-of-the-art method gives an approximate solution in lesser time so has significantly better performance in terms of time of execution. So **whenever time constraints are more important than getting actual optimal solution, the genetic algorithm will be useful**.

**References**

Stuart Russell and Peter Norvig, Artificial Intelligence – A Modern Approach, 3rd Edition, Pearson Education, 2009

Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein Introduction to Algorithms Third Edition- The MIT Press Cambridge, Massachusetts London, England

<https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3>

[(PDF) Rigorous Performance Analysis of State-of-the-Art TSP Heuristic Solvers (researchgate.net)](https://www.researchgate.net/publication/332327761_Rigorous_Performance_Analysis_of_State-of-the-Art_TSP_Heuristic_Solvers)

<https://www.javatpoint.com/prim-algorithm>