Performance Analysis of Travelling Salesman Solution Approaches

Devansh Messon[1], Divyam Verma[2], Mayank Rastogi[3], Amit Singh[4*]

[1, 2, 3, 4]Department of informatics (School of Computer Science), University of Petroleum and Energy Studies, Dehradun, Uttarakhand, India

devansh.messon28@gmail.com

divyam0216@gmail.com

mayikmj@gmail.com

amit.singh1@ddn.upes.ac.in

Abstract. Brute-force based solution using backtracking method of Travelling Salesman Problem (TSP) algorithm which is an NP-hard problem is not improved in terms of space and time-complexity. The algorithm is muddled to apply in graphs which contain larger number of well-connected nodes. To handle this huge time complexity, approximation/ divide and conquer methods, heuristic methods and nature inspired methods are applied. The use of a keen and numerical procedure speeds up the enhanced TSP by multiple times. In this paper, two approaches using divide and conquer technique and natured inspired algorithm like recursion with bit masking and genetic algorithm respectively are applied to minimize the running time of TSP algorithm and their performance is also analysed and compared on different number of nodes.

Keywords: TSP Algorithm, Divide and Conquer, Nature Inspired Algorithm.

1. Introduction

A wide application domain of Travelling Salesman Problem (TSP) garnered much attention among researchers to analyse the performance of the various approaches available in the literature.

1.1. Travelling Salesman Problem

Travelling Salesman Problem (TSP) states that on a given set of nodes and distance between each pair of nodes compute the shortest route that visits each city exactly once and returns to the starting point. There is no polynomial-time known solution to this date, hence it is an NP-hard problem. Algorithms with exponential time complexity are being used to solve the Travelling Salesman problem to this date [1].

1.2. Approaches to solve Travelling Salesman Problem

Brute force-based solution. Brute-force based solution of TSP computes the cost of every possible route by generating all possible permutations of the vertices. After computing the cost of all permutations, the permutation with the minimum cost is the shortest route. The time complexity of this algorithm is **O(n!)** (where n is the number of nodes) which is much high, hence there is a need to minimize the time complexity to some extent so that it can process bigger graphs in a lesser amount of time [2].

Recursion with bit-masking. The time complexity of the brute-force based solution of TSP is significantly improved by a strategy called recursion with bit-masking. This method divides the problems into sub-problems by a recurrence relation and produces at most **$n*2^n$** subsets of the absolute number of nodes and stores it in a bitmask and afterward it in the long run finds the shortest route by discovering the minimum cost among all cost [3]. Computation of each subset takes liner time. The time complexity of this algorithm is **$O(n^2 * 2^n)$**, where n is the number of nodes.

Nature-inspired genetic algorithm. The time complexity of Recursion with the bit-masking algorithm is significantly improved by a natured inspired algorithm known as the Genetic algorithm. It works by generating random valid routes and then put these random routes in a population. To improvise the fitness value of the routes in the population, the mutation is applied on it which leads to new routes and lesser fitness values of those routes. Many such routes are being generated and the route with the minimum cost is chosen. A Chromosome represents a TSP path containing cities. The fitness value of the chromosome is the cost of the whole path. A Population contains a finite number of chromosomes. This finite number is the population size. Mutation of a chromosome is a process to generate a new child chromosome by swapping two cities in the parent chromosome. When a mutation is performed, a new generation is defined [4]. The number of generations is equal to the number of populations, which is done by performing mutation on chromosomes of a population. The time-complexity of this algorithm is **O(G*P*S)**, where G is the number of generations, P is the population size and S is the size of the chromosome.

2. Novelty and Contribution

A Performance comparison based on number of iterations are carried out in this paper. In addition, the result analysis demonstrates the application of approaches like Brute-Force using Backtracking, Recursive Bit Masking, and Genetic Algorithm against the variation in the problem size.

3. Experimental Study

Various solution approaches has been proposed to solve TSP efficiently over a wide span of time. Few of them are implemented in this paper and validated rigorously through a number of iterations and demonstrated in the result section.

3.1. Solution approaches of Travelling Salesman Problem

In this paper, all three approaches are implemented in ANSI C programming and executed on GCC compiler. The flowchart, algorithm and code snippets are shown as follows:

Brute-Force based solution using Backtracking approach

Algorithm:

```
1. Input the edges(e) between the nodes(n) and starting_point=1 ,cities[ ]=2,3,…,n
2. If ( starting node = = ending node )
3.     Calculate cost of the path by adding edge_weights
4.     Optimal_cost = minimum (Optimal_cost, calculated_cost) and update optimal_path
5. LOOP from i=0 till n-1:
6.     swap(cities[starting_point], cities[i])
7.     starting_point = starting_point + 1 and goto point 3.
8.     swap(cities[starting_point], cities[i])
9. Return optimal_cost , optimal_path.
```

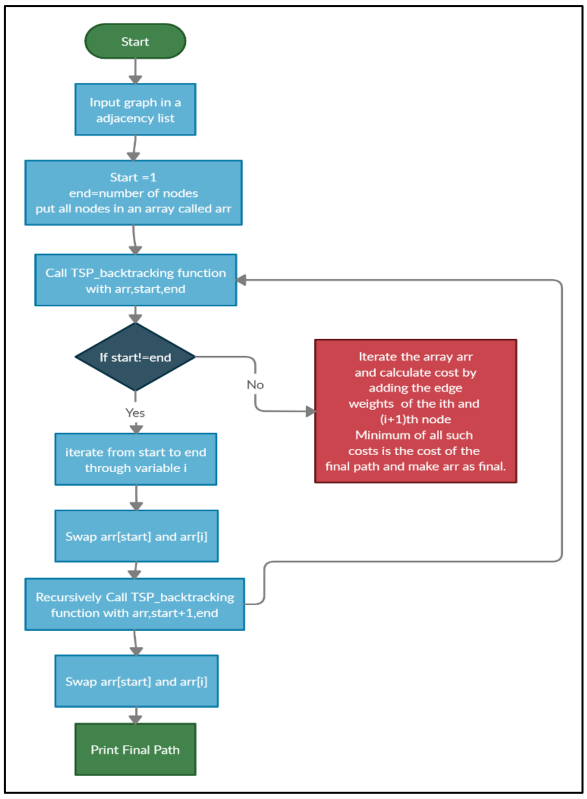
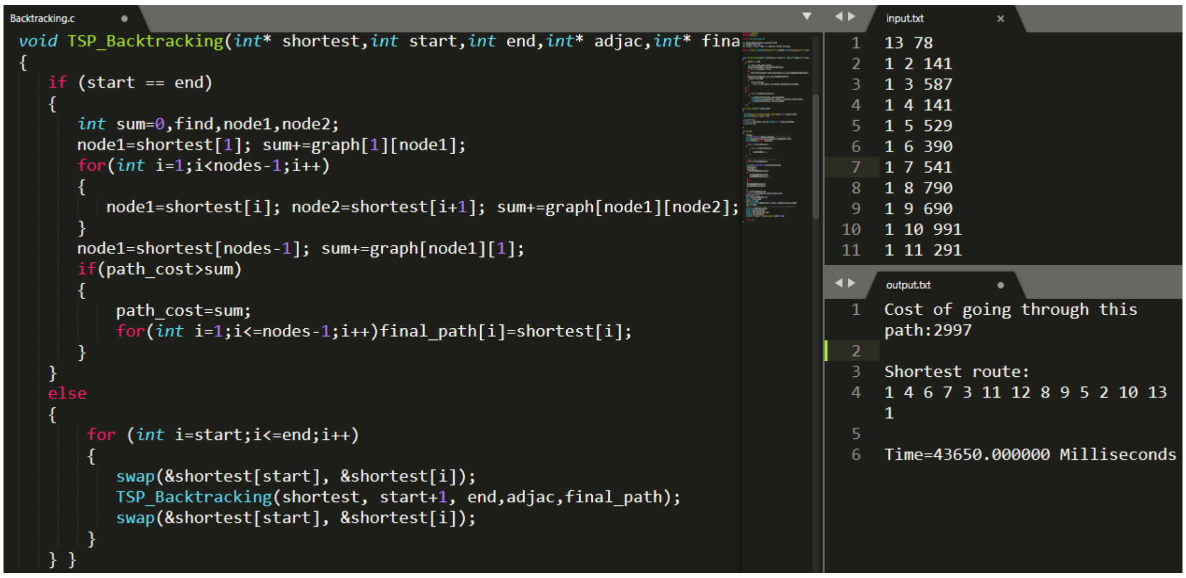

Start
Input graph in a adjacency list
Start =1
end=number of nodes
put all nodes in an array called arr
Call TSP_backtracking function with arr,start,end
If start!=end
No
Yes
Iterate the array arr and calculate cost by adding the edge weights of the ith and (i+1)th node
Minimum of all such costs is the cost of the final path and make arr as final.
iterate from start to end through variable i
Swap arr[start] and arr[i]
Recursively Call TSP_backtracking function with arr,start+1,end
Swap arr[start] and arr[i]
Print Final Path

Fig. 1. Flowchart for Brute-force based solution using Backtrack method



```
void TSP_Backtracking(int* shortest,int start,int end,int* adjac,int* fina
{
    if (start == end)
    {
        int sum=0,find,node1,node2;
        node1=shortest[1]; sum+=graph[1][node1];
        for(int i=1;i<nodes-1;i++)
        {
            node1=shortest[i]; node2=shortest[i+1]; sum+=graph[node1][node2];
        }
        node1=shortest[nodes-1]; sum+=graph[node1][1];
        if(path_cost>sum)
        {
            path_cost=sum;
            for(int i=1;i<=nodes-1;i++)final_path[i]=shortest[i];
        }
    }
    else
    {
        for (int i=start;i<=end;i++)
        {
            swap(&shortest[start], &shortest[i]);
            TSP_Backtracking(shortest, start+1, end,adjac,final_path);
            swap(&shortest[start], &shortest[i]);
        }
    } }
```

```
13 78
1 2 141
1 3 587
1 4 141
1 5 529
1 6 390
1 7 541
1 8 790
1 9 690
1 10 991
1 11 291
```

```
Cost of going through this
path:2997

Shortest route:
1 4 6 7 3 11 12 8 9 5 2 10 13
1

Time=43650.000000 Milliseconds
```

Fig. 2. Implementation of Brute-forced based solution using Backtracking method

Recursion with Bit Masking

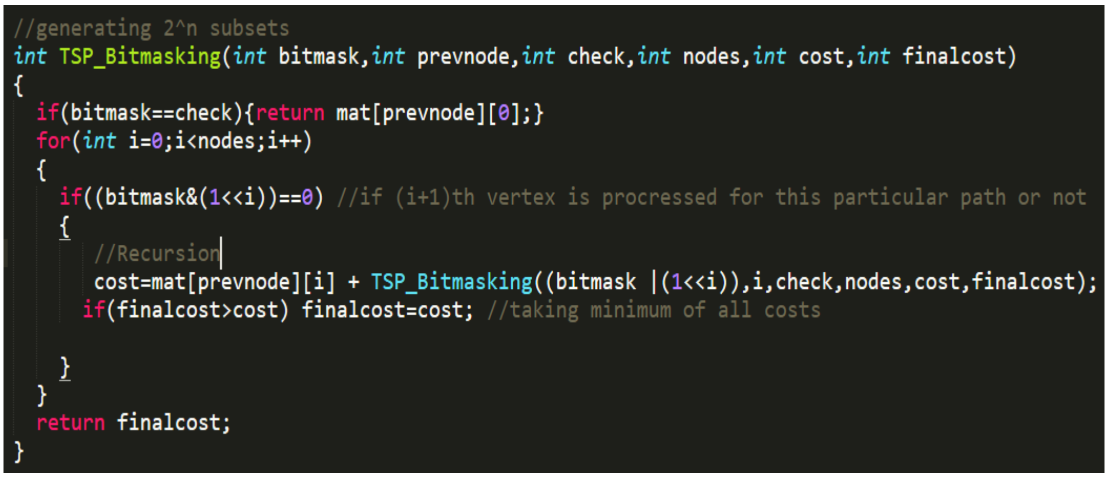
In this approach a bitmask of 0-based indexed is used for the naming of graph nodes ranges from 0 to n-1. Bitmask represents a visited array, whereas bits of the bitmask represents nodes/cities. For example bitmask 1010 represents the 1st and 3rd cities are not visited yet whereas 2nd and 4th cities are visited. A recursive algorithm based on the following equation is presented below to the cost of Salesman while travelling the cities:

$$cost(S,i) = dist[i][j] + cost(\{S-i-1\},j)$$

where S represents set of all vertices and j belongs to S

Algorithm:

```
1.  temp_bitmask = 0 , ideal_bitmask = $2^n - 1$ , previous_node = 0
2.  If (temp_bitmask = = ideal_bitmask) return edge_weight(previous_node , 0)
3.  LOOP from i=0 till nodes:
4.    If (temp_bitmask BITWISE AND $2^i$ = = 0 ):
5.        temp_bitmask = temp_bitmask BITWISE OR $2^i$
6.        cost = edge_weight(previous node , i) + returning value (goto step 6)
7.        Optimal_cost = minimum (Optimal_cost , cost)
8.  Return Optimal_cost.
```



```
//generating 2^n subsets
int TSP_Bitmasking(int bitmask,int prevnode,int check,int nodes,int cost,int finalcost)
{
  if(bitmask==check){return mat[prevnode][0];}
  for(int i=0;i<nodes;i++)
  {
    if((bitmask&(1<<i))==0) //if (i+1)th vertex is procressed for this particular path or not
    {
        //Recursion
        cost=mat[prevnode][i] + TSP_Bitmasking((bitmask |(1<<i)),i,check,nodes,cost,finalcost);
      if(finalcost>cost) finalcost=cost; //taking minimum of all costs

    }
  }
  return finalcost;
}
```

Fig. 3. Implementation of Recursion with Bit-Masking

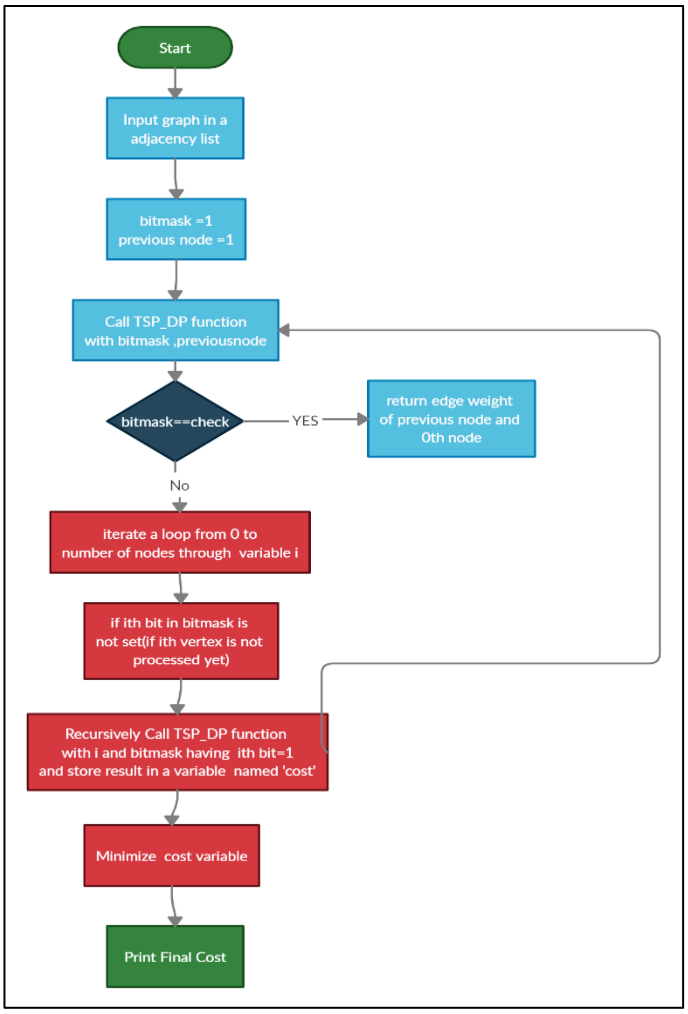

Start
Input graph in a adjacency list
bitmask =1
previous node =1
Call TSP_DP function with bitmask ,previousnode
bitmask==check
YES
return edge weight of previous node and 0th node
No
iterate a loop from 0 to number of nodes through variable i
if ith bit in bitmask is not set(if ith vertex is not processed yet)
Recursively Call TSP_DP function with i and bitmask having ith bit=1 and store result in a variable named 'cost'
Minimize cost variable
Print Final Cost

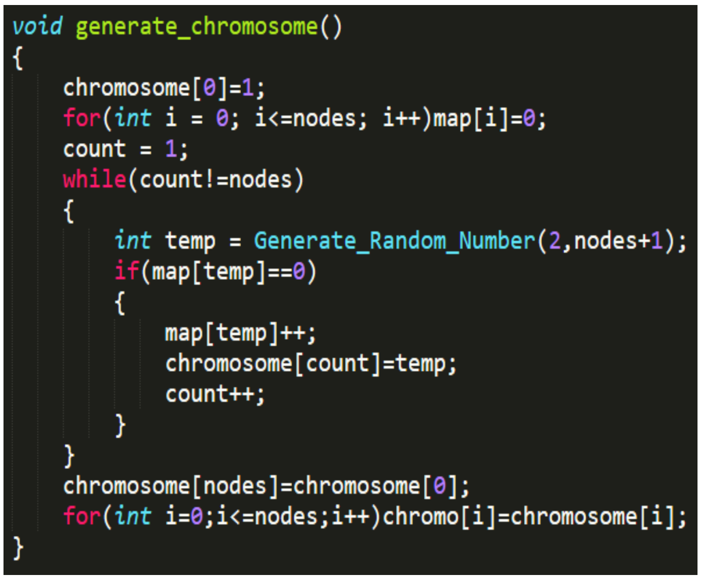
Fig. 4. Flowchart of Recursion with Bit- Masking

Genetic Algorithm

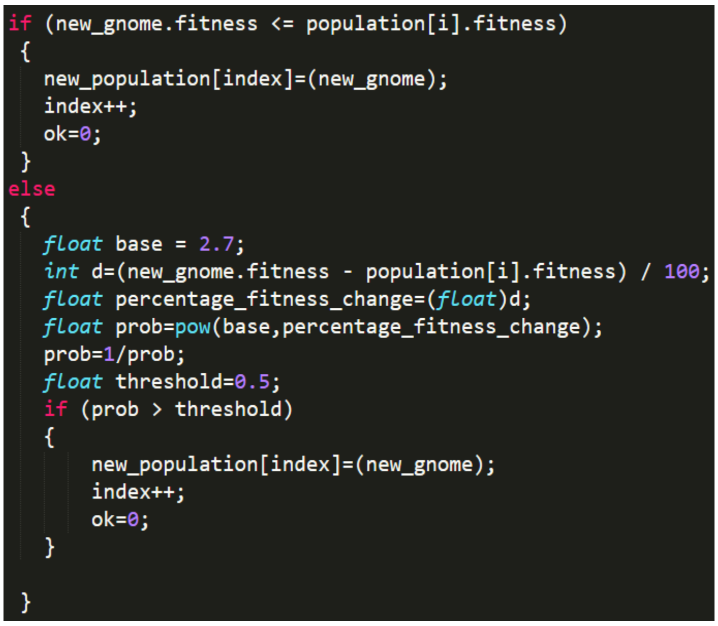
Algorithm:

```
1.  input number_of_cities, population_size, number_of_generations.
2.  LOOP till population_size:
3.      Parent_Chromosome = Generate_chromosome()
4.      Parent_Fitness = Calculate_fitness_of_chromosome()
5.      If Parent_Fitness < Optimal_cost:
6.          Optimal_cost=Parent_Fitness
7.          Optimal_path=Parent_Chromosome
8.      Append Parent_Chromosome, Parent_Fitness in Initial_population
9.  LOOP till number_of_generations:
10.     LOOP till population_size(iterator i):
11.         Parent_Chromosome=Initial_populationchromosome[i]
12.         While true:
13.             Child_Chromosome = mutated_chromosome(Parent_Chromosome)
14.             Child_Fitness = Calculate_fitness_of_chromosome()
15.             If Child_Fitness < Optimal_cost:
16.                 Optimal_cost=Child_Fitness
17.                 Optimal_path=Child_chromosome
18.             If Child_Fitness < Parent_Fitness:
19.                 Append Child_Chromosome in New_population
```

```
20.              Append Child_Fitness in New_population
21.              Break
22.            Else:
23.               Percentage_change_fitness=(Child_Fitness-Parent_Fitness)/100
24.               Probability=1/ $e^{Percentage\_change\_fitness}$
25.               If Probability > 0.50:
26.                  Append Child_chromosome in New_population
27.                  Append Child_Fitness in New_population
28.                  Break
29.      Copy contents of New_population to Initial_population
```



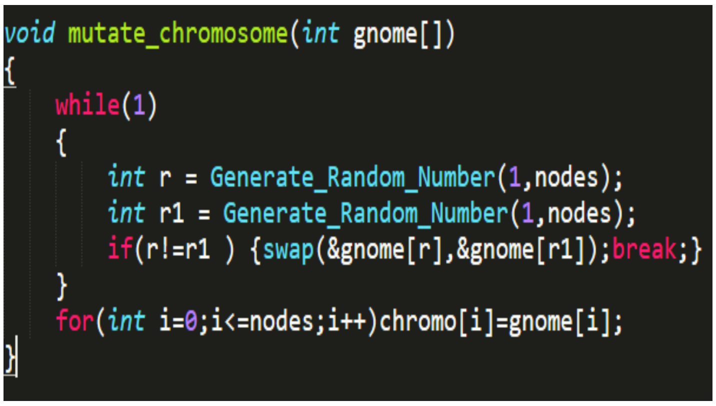
```
void generate_chromosome()
{
    chromosome[0]=1;
    for(int i = 0; i<=nodes; i++)map[i]=0;
    count = 1;
    while(count!=nodes)
    {
        int temp = Generate_Random_Number(2,nodes+1);
        if(map[temp]==0)
        {
            map[temp]++;
            chromosome[count]=temp;
            count++;
        }
    }
    chromosome[nodes]=chromosome[0];
    for(int i=0;i<=nodes;i++)chromo[i]=chromosome[i];
}
```



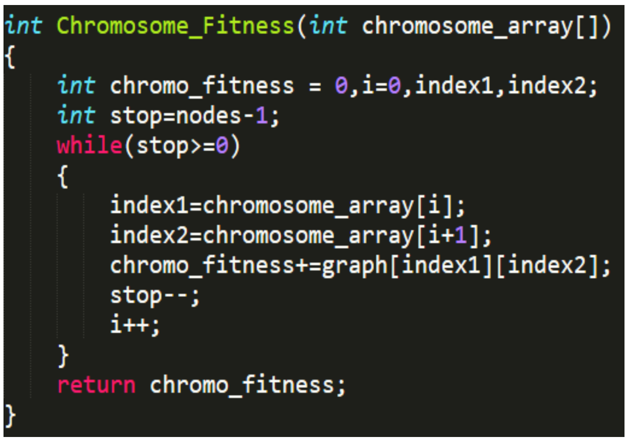
```
if (new_gnome.fitness <= population[i].fitness)
 {
   new_population[index]=(new_gnome);
   index++;
   ok=0;
 }
else
 {
   float base = 2.7;
   int d=(new_gnome.fitness - population[i].fitness) / 100;
   float percentage_fitness_change=(float)d;
   float prob=pow(base,percentage_fitness_change);
   prob=1/prob;
   float threshold=0.5;
   if (prob > threshold)
   {
       new_population[index]=(new_gnome);
       index++;
       ok=0;
   }

 }
```

Fig. 5. Implementation of Generating Chromosomes and Accepting the Mutated Chromosomes

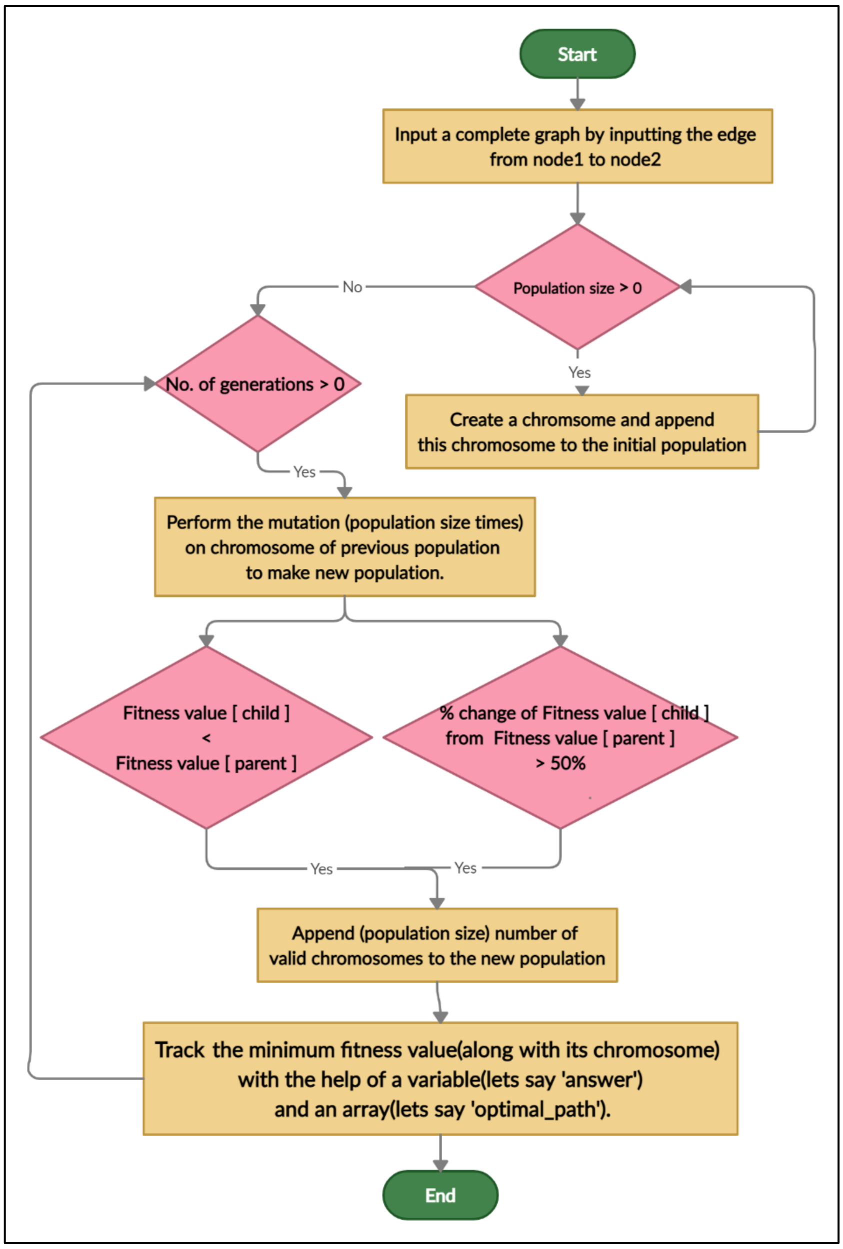


```
void mutate_chromosome(int gnome[])
{
    while(1)
    {
        int r = Generate_Random_Number(1,nodes);
        int r1 = Generate_Random_Number(1,nodes);
        if(r!=r1 ) {swap(&gnome[r],&gnome[r1]);break;}
    }
    for(int i=0;i<=nodes;i++)chromo[i]=gnome[i];
}
```



```
int Chromosome_Fitness(int chromosome_array[])
{
    int chromo_fitness = 0,i=0,index1,index2;
    int stop=nodes-1;
    while(stop>=0)
    {
        index1=chromosome_array[i];
        index2=chromosome_array[i+1];
        chromo_fitness+=graph[index1][index2];
        stop--;
        i++;
    }
    return chromo_fitness;
}
```

Fig. 6. Implementation of Mutation and Fitness Function That Optimize TSP



Start
Input a complete graph by inputting the edge from node1 to node2
Population size > 0
No
Yes
Create a chromsome and append this chromosome to the initial population
No. of generations > 0
Yes
Perform the mutation (population size times) on chromosome of previous population to make new population.
Fitness value [child] < Fitness value [parent]
% change of Fitness value [child] from Fitness value [parent] > 50%
Yes
Yes
Append (population size) number of valid chromosomes to the new population
Track the minimum fitness value(along with its chromosome) with the help of a variable(lets say 'answer') and an array(lets say 'optimal_path').
End

Fig. 7. Flowchart of Natured-Inspired Genetic Algorithm

4. Result Analysis

In this project, result analysis is done based on the execution time (in milliseconds) of all proposed algorithms to implement TSP which are examined on different number of nodes as shown in Table 1, that belong in the fully connected graph in which there are N numbers of nodes, at that point the number of edges would be equivalent to:

$$\{N * (N - 1)\}/2$$

There are 5 test cases in the experiment, which are used for all three proposed implementations (Brute-force based solution (BFB), Recursion with Bit-Masking (RBM), and Genetic algorithm (GA)) on the same processor (i3, 4 GB RAM). The decision point is the algorithm with the best performance among all other algorithms on the basis of execution time.

Table 1. Execution time (in milliseconds) of all proposed methods to implement TSP

Number of Nodes	Case no	BFB	RBM	GA	D-point
10	Case 1	48	30	1	GA
11	Case 2	328	311	16	GA
12	Case 3	4050	3613	156	GA
13	Case 4	58510	45619	6136	GA
14	Case 5	Fail	1462645	764393	GA

4.1. Performance Analysis

In Table 1 as per the outcomes, GA has shown impressive performance as compared to that of RBM and BFB. GA is continually increasing in normal pattern as the number of nodes all the while which can also be concluded from Figure 9. While on account of BFB, it has shown the worst performance, and as the number of nodes incremented from 13 to 14 the BFB got failed to compute it which can be concluded from Table 1 and Figure 9. In the case of RBM, it is optimizing the TSP but it is not that efficient as compared to GA. RBM has shown improvement over BFB in terms of execution time but the execution time difference between these two strategies is not that impressive yet RBM is not failed for case 5(number of nodes equal to 14) which can be concluded from Table 1 and Figure 9

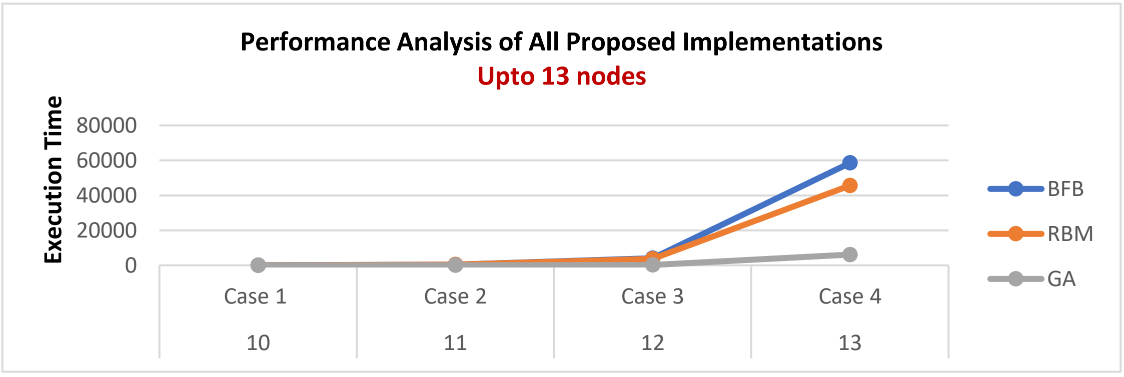
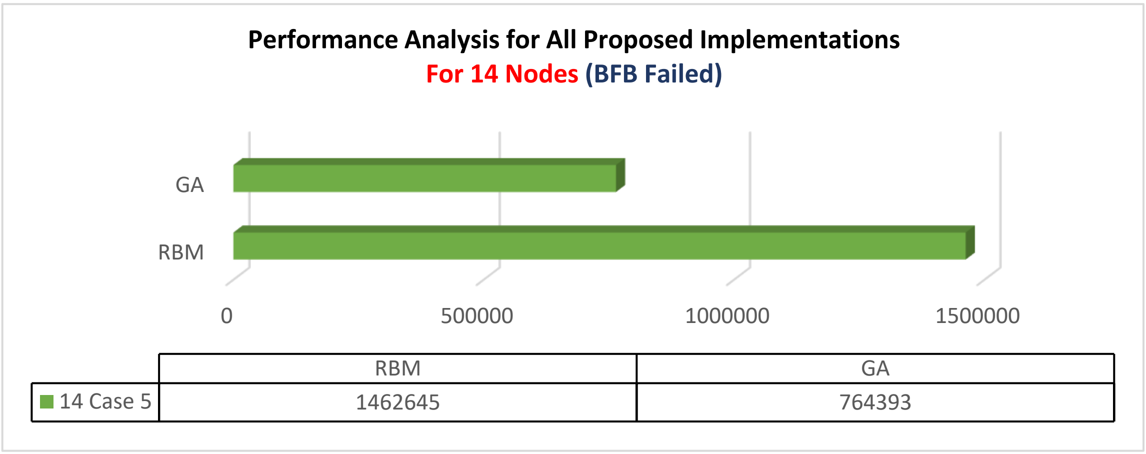


Fig. 8. Performance Analysis for BFB, RBM and GA on the Basis of Execution Time



	RBM	GA
🟩 14 Case 5	1462645	764393

Fig. 9. Performance Analysis of BFB, RMB and GA for 14 Nodes on the Basis of Execution Time Where BFB Failed

5. Conclusion and Future Scope

After analysing the execution time of three different algorithms on exactly same test cases for the different number of nodes in a graph, it is clearly visible that the genetic algorithm which is a natured-inspired algorithm is performing much better than the other two methods on the basis of time complexity. Since the Travelling Salesman problem is an NP-hard problem, a graph containing 14 nodes is the highest number of nodes that the genetic algorithm processed easily unlike the brute-force method which failed in processing the same graph, but genetic algorithm broke the user's patience to process a graph containing the number of nodes greater than 14. The two major deciding factors for the genetic algorithm's time complexity are the number of generations and the population size. There is no rule to wisely decide these two factors, instead the hit and trial method is a technique that can decide these two factors and it may lead to a lesser execution time and this is the main reason for the better performance of the genetic algorithm as compared to other algorithms.

As a future scope of the present research, the genetic algorithm may be further optimized by using the cross-over technique over mutation. Travelling Salesman Problem may be further optimized by another nature-inspired technique named Ant Colony Method. In this method, artificial ants travel from the origin city and then take different orders to visit all cities and they return back to the origin city. While traveling through the cities, they release a fluid called pheromone. The path which has the largest amount of pheromone is the shortest path [6]. To date, there is no algorithm that solves the Travelling salesman problem in polynomial time. Improvement done by other algorithms in terms of the running time is very little to this date.

References

1. Johnson, D. S., McGeoch, L. A. : The Traveling Salesman Problem: A Case Study in Local Optimization. In:, in Aarts, E. H. L.; Lenstra, J. K. (eds.) Local Search in Combinatorial Optimization (PDF), John Wiley and Sons Ltd., pp. 215–310 (1997).

2. Ryu, H.: A Revisiting Method Using a Covariance Traveling Salesman Problem Algorithm for Landmark-Based Simultaneous Localization and Mapping. Sensors 19, 4910(2019).

3. Bellman, R.: Dynamic Programming Treatment of the Travelling Salesman Problem. J. Assoc. Comput. Mach.9, 61–63 (1962).

4. Kralev, V.: Different Applications of the Genetic Mutation Operator for Symmetric Travelling Salesman Problem. Int. J. on Advanced Science Engineering Information Technology 8(3), 762-770 (2018)

5. Dorigo M., Stützle T.: Ant Colony Optimization: Overview and Recent Advances. In: Gendreau M., Potvin JY. (eds) Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol 272. Springer, Cham(2019).