Travelling Salesman Problem (TSP)

Louise Harding
Robin Khatri
Laetitia Couge
Mohammed Paul Doust
Mohammed Nassar

December 11, 2018

Contents

1 Introduction

Consider a salesman who has to visit a set of cities and return to the city he started from provided he visitseach city only once.

The problem is to minimize the total cost or distance of the route. This is known as the Travelling Salesman Problem.

The problem can be summarised as follows:

TSP = $\{(G, f, t)$: $G = (V, E)$ is a complete graph,
f is a function $V \times V \to Z$,
$t \in Z$,
G is a graph that contains a travelling salesman tour with costs or distances that
does not exceed $t\}$.

1.1 Symmetric Travelling Salesman Problem

If in a travelling salesman problem, cost or distance of travelling from city i to city j is equal to the cost or distance of travelling from city j, *i.e.* the cost matrix is symmetric, then the problem is said to be *Symmetric Travelling Salesman Problem.*

1.2 Asymmetric Travelling Salesman Problem

If in a travelling salesman problem, cost or distance of travelling from city i to city j can differ from the cost or distance of travelling from city j, *i.e.* the cost matrix is asymmetric, then the problem is said to be *Asymmetric Travelling Salesman Problem.* A real world example could be routes consisting of some one-way roads.

1.3 Sparsity of TSP

A graph with only a few edges is referred to as a sparse graph. In our testing with different exact and approximate algorithms, we have tested on problems of varying sparsity levels.

2 Brute Force Algorithm

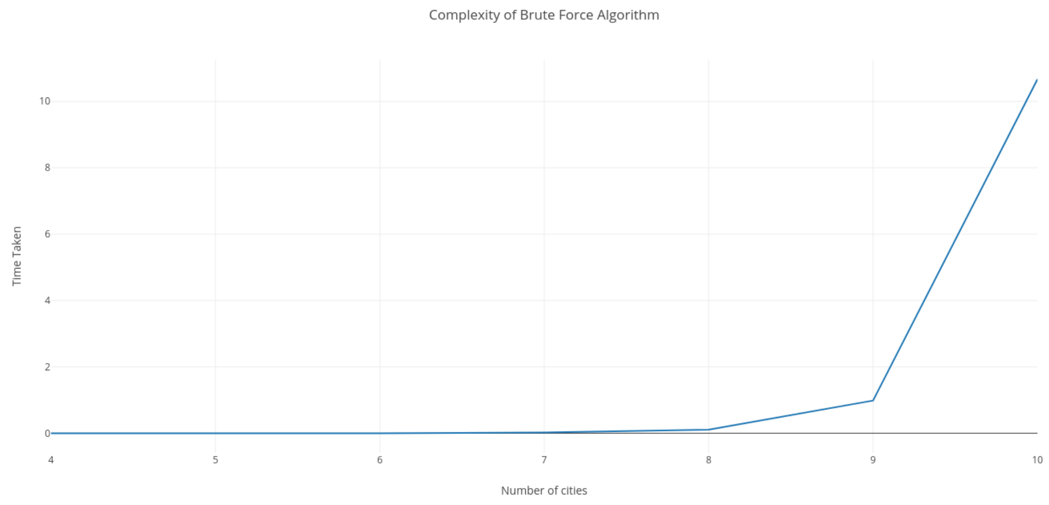
2.1 Introduction

Bruteforce algorithm runs through all possible solutions and selects the best one. It is not an optimal algorithm to use for TSP with a large number of nodes.

2.2 Experiment and complexity

We tested the code on both symmetric and asymmetric problems. Since it provides exacts solutions, the solutions were optimal.

The complexity of the algorithm is $\mathcal{O}(n!)$. We tested it on tsp's up to 10 cities, as the complexity increased rapidly after that.



In case of Symmetric TSPs, the computational time can be reduced by computing distances only for permutations which begin with our choice of first city. However, the complexity still remains $\mathcal{O}(n!)$, but number of permutations to cover decrease.

Improvements on Bruteforce Algorithm are provided with Branch and Bound Algorithms presented in the next section.

3 Branch and Bound - Matrix Reduction method

3.1 Introduction

A branch-and-bound algorithm consists of a systematically looking for the shortest path by searching a state space tree : the approach tries to minimise the initial depth first search by selecting the lest cost path option at each decision point (node) in the tree. Once this first cost (upper bound) is found the cost at each node is evalated in future searches of the tree and if it exceeds the value, the tree is pruned and the lower branches are not searched.

3.2 Approach

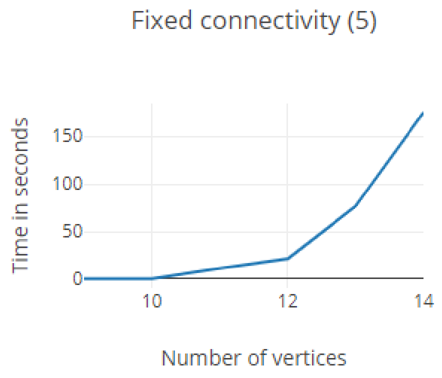
The approach used for this algorithm was the one presented in class. Starting with an adjecency matrix, subtract the minimum edge weight for all values in the same row and then do the same for the columns, the sum of the values subtracted equals the lower bound for this tsp problem. After the initial matrix reduction, using a state space tree, map the possible edges from the starting node (root) and choose the one with the least weight. With this node, using the vertex it corresponds to, set the represented row values in the matrix to infinity, and the matrix values that represents the reverse of this edge (eg if going for 1 to 2, we set 2 to 1) to infinity. Repeat this process until the first leaf is reached. This provides the first upper bound. Continue this process by backtracking up and down the tree, each time checking for the upper bound. If the upper bound is smaller, update the upper bound. If it is larger, then prune the branch.

3.3 Complexity

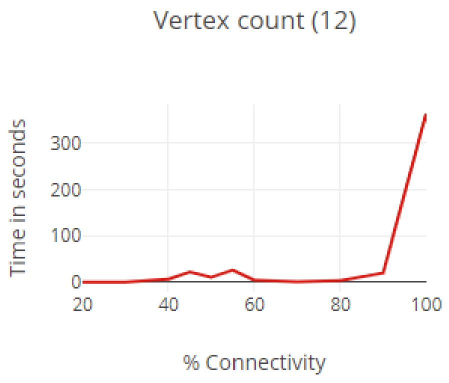
The complexity of this algorithm is at worst $\mathcal{O}(n!)$, which is the same as Brute Force, because this happens in the case where no pruning of the braches occurs. In practice, however, it will be better than this as the entire space will not need to be searched.

3.4 Testing

Due to the complexity Brand and Bound takes too long to run against the TSP library, so we tested it against a variety of generated tsp problems. We tested the code on both symmetric and asymmetric problems, with differing levels of sparcity. As expected the number of vertices dramatically increased the time, up until about 13 vertices fully connected to each other taking greater than 1 hour.
The sparcity of the connections between vertices affected the time taken due to less connections results in a smaller search area. There were mixed results with symmetric vs asymetric with neither showing to be consistently easier to process than the other, most likely the pruning between the different problems resulted in less clear results. We also tested it against Brute force and it did in reality produce faster results.



(a) Example of increase in vertex count

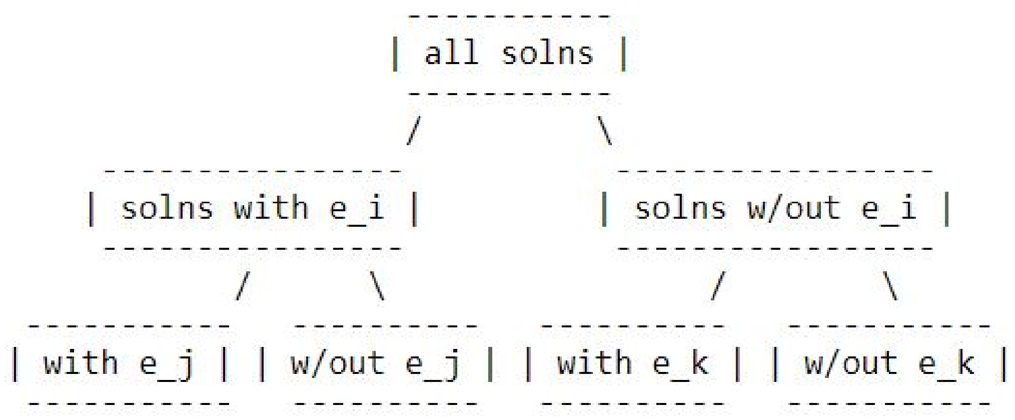


(b) Example of increase in connectivity

4 Branch and Bound with Adding and Removing Edges

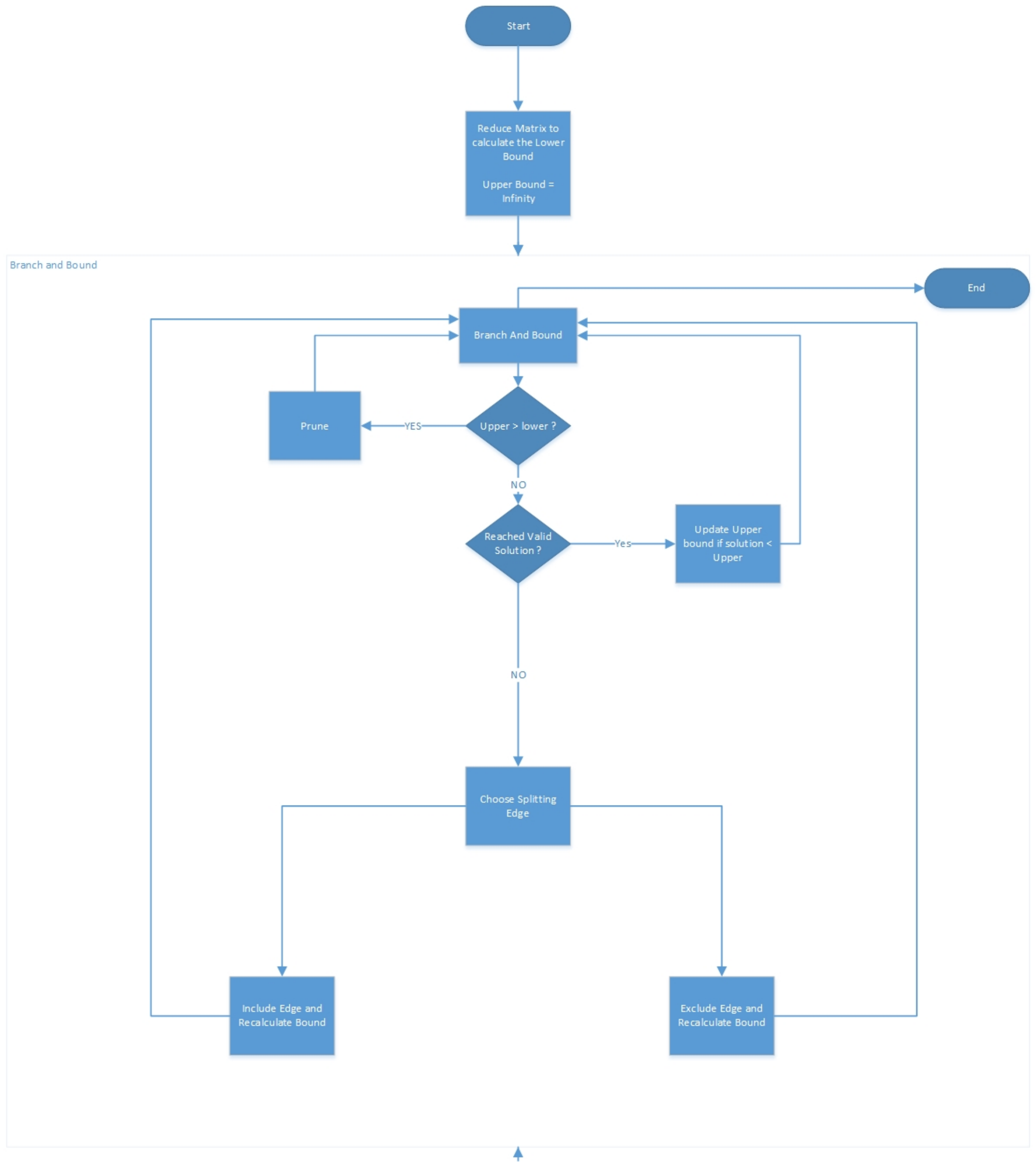
4.1 Introduction

One of the strategies used for searching the solution space is Branch and Bound which keeps dividing the space into branches. one for solutions containing a given edge and the other for those excluding the given edge. forming a binary tree as follows:



The main parts of these algorithm are:

1. Bounding Function
2. Choosing Splitting Edge
3. How to Include Edge
4. How to Exclude Edge


Start
Reduce Matrix to calculate the Lower Bound
Upper Bound = Infinity
Branch and Bound
End
Branch And Bound
Prune
YES
Upper > lower ?
NO
Reached Valid Solution ?
Yes
Update Upper bound if solution < Upper
NO
Choose Splitting Edge
Include Edge and Recalculate Bound
Exclude Edge and Recalculate Bound

4.2 Bounding Function (Reduction)

The solution is bounded by normalizing the solution matrix. this is done by reducing the rows first and the columns after. by reducing the Rows/Columns we mean normalizing them. we subtract the minimum element from each row from each element at that row. and the same for columns. at the end we will have a matrix with at least one zero in each column and each row. Our lower bound will be the sum of all minimum values with used to reduce the matrix.

4.3 Choosing Splitting Edge

We are looking to maximize the right part by trying to raising the lower bound of the right sub-tree. In order to do that, we choose to split on the edge that best maximize the lower bound. We look for the zero weight edges that maximize the increasing in the lower bound.

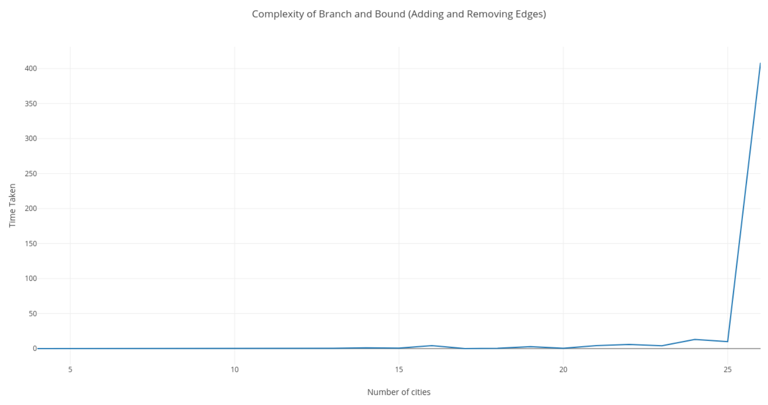
4.4 How to Include Edge

Including an edge (ie, I -> J) is done by first, forbidding the going back from J -> I by setting the weight of edge J -> to INFINITY (we also forbid the going back to any sub-path in our partial solution). Moreover, since we have used this edge, we cannot go from node i to any other node, and similarly, we cannot reach node J. Consequently, we delete the I th row and the J th column from our solutions matrix. in the end, we reduce the new matrix after including the edge.

4.5 How to Exclude Edge

To exclude edge (ie, I-> J), we start by setting the cost of the edge I->J to INFINITY. and we reduce the new matrix afterwards.

4.6 Complexity



5 Randomized algorithm

The purpose of the algorithm is to get the cost of a path generated randomly from a set of node $S = \{0, ..., n\}$, starting from node 0, with each node visited exactly once. The cost is the sum of the distance d_{ij} between two successive nodes i and j. The distance for nodes that are not connected is set to -1.
For a subset $S' \subseteq S$ of nodes not yet visited, starting from node j, we compute a probability distribution P_k, $k \in S' - \{j\}$, for the next possible nodes k based on the distance d_{jk} such that the lower, the higher the probability. From the distribution P, the cumulative distribution P' is constructed using increasing probabilities P_k. A random number r between 0 and 1 is picked from a uniform distribution. For r such that $P'_{k-1} < r < P'_k$, the node k is picked.

$$sum = \sum_{k \in S' - \{j\}} d_{jk}, \quad where \quad d_{jk}! = -1$$

$P_k = (sum - d_{jk})/(sum * (n-1))$, *where n number of node in S'-{j} connected to node j*

The division by (n-1) allows normalization so that $\sum_k P_k = 1$,

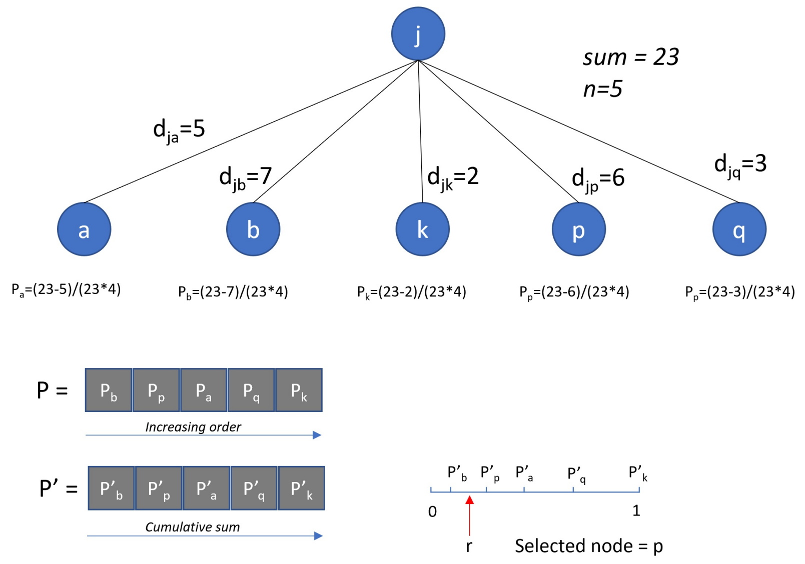


Figure 2: Example of node selection

5.1 Pseudo code

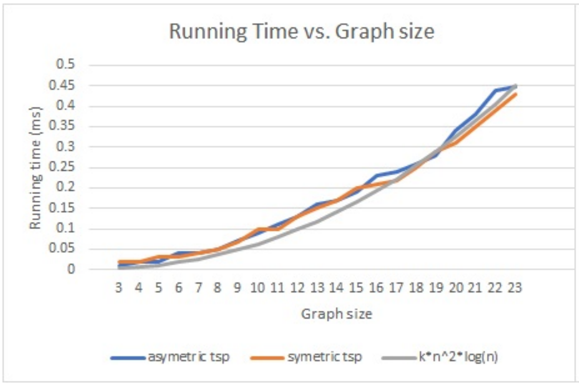
```
**Require:**
  $d[nbNode][nbNode]$
  $S = \{1, ..., n\}$
**Ensure:**
  sourcenode := 0
  S' := S
  path := []
  $cost = 0$
  **while** $|S'| > 1$ **do**
    sum := 0
    nbnode := 0
    **for** node in S'-{sourcenode} with d[sourcenode][node] != -1 **do**
      sum = sum + d[sourcenode][node]
      nbnode= nbnode + 1
    **end for**
    **for** k in S'-{sourcenode} with d[sourcenode][node] != -1 **do**
      $P(k) = (sum - d[sourcenode][k])/sum/(nbnode - 1)$
    **end for**
    sort(P)
    P' := cumulative sum of elements of P
    r := random(0,1)
    search k s.t. $P(k-1) < r < P(k)$
    $cost := cost + d[sourcenode][k]$
    add k to path
    remove k from S'
    $startnode := k$
  **end while**
  $cost := cost + d[startnode][0]$
  **return** $cost, path$
```

Algorithm 1: Randomized Algorithm

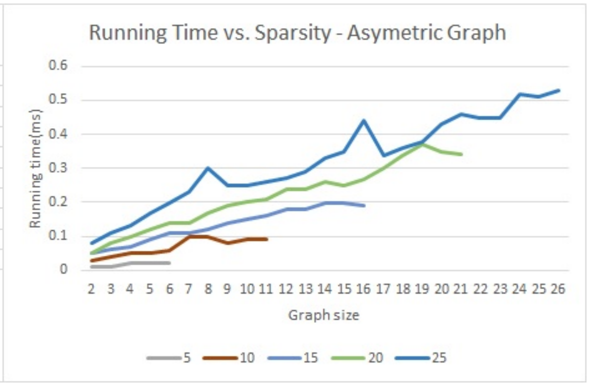
5.2 Complexity

The algorithm is proceeding from top to down. For each node source (n loops) we compute the probability table of the possible next nodes using operations of the order $O(1)$. The table is sorted in increasing order using Python sort function whose complexity is $O(nlogn)$. The search of the node whose probability matches the random number is O(log n) using divide and conquer algorithm. Consequently, the time complexity for the proposed version of randomized algorithm is $O(n^2logn)$. For each source node, we compute a probability table of at most n elements. The space complexity is O(n).

5.3 Performances



(a) Example of node selection



(b) Example of node selection

Figure 3: Example of node selection

The algorithm is tested on a set of problems generated randomly with different city numbers and different sparsity. The algorithm is tested on a set of problems generated randomly with different city numbers and different sparsity. Figure 3a shows the evolution of the algorithm running time with the number of cities. We can observe that the curves for symetric and asymetric problems follow the theoritical complexity of $O(n^2 logn)$.
Figure 3bshows the evolution of the algorithm running time with the number of cities. We can observe that the curves for symetric and asymetric problems fo.

The algorithm is tested on a set of problems from the TSLIB. The alg

6 Dynamic Programming algorithm

The dynamic programming implementation is based on the Bellman-Held-Karp algorithm proposed in 1962 independently by Bellman[1] and by Held and Karp.
Dynamic programming express a solution for a problem through the solution of smaller problem.
Let's consider a set of node $S = \{0, 1, ..., n\}$. We want to compute the minimum cost for a path starting at node 0 and visiting all nodes exactly once.

Optimal solution
Let's consider a set S' $\subseteq$ S and $C(S', j)$ the minimum cost for a path between node 0 and node j containing all nodes from S'. Then the cost $C(S', j)$ can be decomposed in the sum of the minimum cost of the path from node 0 to k including all nodes from $S' - \{j\}$, and the distance d_{kj}.

$$C(S', j) = min_{k \in S'-\{j\}, k \neq j}\{C(S', k) + d_{kj}\}$$

Base case
If $S' = \{0\}$,

$$C(S', 0) = 0$$

.

Pseudo code

```
Initialization
for j := 2 to n do do
   $C(\{j\}, j) = 0$
end for
for $subsetsize$ := 2 to n-1 do
   for all $S' \subseteq \{2, ..., n\}$ with $|S'| = subsetsize$ do
      for all j in S' do
         $C(S', j) = min_{k \in S'-\{j\}}\{C(S' - \{j\}, k) + d_{kj}\}$
      end for
   end for
end for
$mincost := min_{1<j\leq n}\{C(\{2, ..., n\}, j) + d_{j0}\}$
return $mincost$
```
Algorithm 2: Dynamic Programming Algorithm

Implementation
We use a bit field to code the subset of nodes selected in the set {1,...n}. Each node j is code by the value 2^j when selected and 0 when not selected.

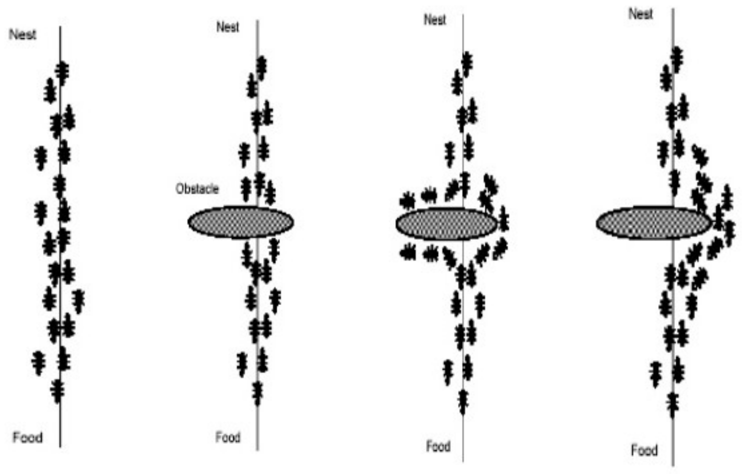
Complexity
We need to build all the subsets of {1,...,n} i.e. 2^n subsets. For all nodes j of each subset (at most n nodes) and for all nodes k distinct from node j(at most n-1 nodes), we compute the cost of the sub paths terminated by node k for the subset deprived of node j. Thus the time complexity is $2^n n^2$.
In term of space complexity, if we use a bit field to code all subsets for each node $j \subseteq \{0, ..., n\}$, the required space is $2^n(n + 1)$.
Nevertheless, we can note that to compute the cost for the subsets of size s, we only need the cost of the subsets of size (s-1). Thus the space complexity can be reduced to

7 Ant Colony Algorithm

7.1 Introduction

Ant Colony Algorithm is a probabilistic algorithm that takes inspiration from the behavior of ants to Travelling Salesman Problems. The first algorithm that took inspiration from ants was described in 1992 by Marco Dorigo in his PhD thesis [1]. Later, there has been various optimization techniques building upon this research. In 2004, book titled Ant Colony Optimzation was published. We took inspiration from this book for our implementation of Ant Colony Algorithm.

7.2 Behavior of Ants

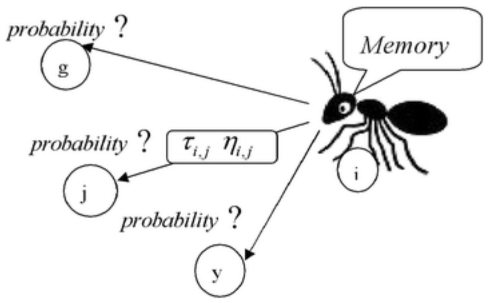


Ants live together in colonies and they form a highly structured societies. Ants do not have a developed visual skills and some of the species of ants are blind. They, however, make use of pheromones which they leave behind while naviagting. This is a form of indirect communication with other ants in the colony. Due to high concentration of pheromones on a path, ants are influenced to take the path previously travelled by the former ants. This behavior leads to the convergence of establishing a path, *i.e.* after a certain time, all the ants which are travelling together follow the same path (and find the food!)

7.3 Theory

The behavior of ants described above can be utilised to form an algorithm able to provide approximate solutions to TSP problems.

Consider m ants and they are placed in n cities chosen randomly from the list of cities in our problem.



Ant number k moves from city i to city j with probability p_{ij}^{m} given below:

$$p_{ij}^{k} = \frac{(\tau_{ij}^{\alpha})(\eta_{ij}^{\beta})}{\sum_{ij \to allowed} (\tau_{ij}^{\alpha})(\theta_{ij}^{\beta})}$$

Where, τ_{ij} is the amount of pheromone deposited for transition from city i to j.
η_{ij} is the desirability of going to city j from city i. It is given by $1/d_{ij}$.
α and β are parameters that respectively control the influence of τ_{ij} and η_{xy}.

Many special cases of Ant Colony Optimzation has been proposed. In this project, we have considered Ant Colony System (ACS) [2]. Other popular approaches are Ant System (Earliest Ant Colony Algorithm)[3] and MAX-MIN Ant System (MMAS)[4]. These Optimization methods differ in their approach of updating pheromones and calculating transition probabilities p_{ij}.

Ant System (AS):

In AS,
Updates of pheromones:
When all ants complete their tours, trails of pheromones are updated as follows:

$$\tau_{ij} = (1 - \rho).\tau_{ij} + \sum_{k} \Delta\tau_{ij}^{k}$$

where $\Delta\tau_{ij}^{k}$ can be calculated depending upon the strategy we choose, and ρ is the evaporation rate. Higher the ρ, quicker is the evaporation rate of pheromones.

$$\Delta\tau_{ij}^{k} = \begin{cases} Q/L_k & \text{if ant } k \text{ uses transition city } i \to j \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

Where L_k is the cost of ant k's tour, and Q is a constant.

Ant Colony System (ACS):

In ACS,
The local pheromone update is done by each ant k after every transition $i \to j$. Each ant applies this update to the last edge it transversed.

$$\tau_{ij} = (1 - \phi).\tau_{ij} + \phi.\tau_0$$

Where τ_0 is the initial value of the phreromone (Usually kept small) and ϕ is the pheromone decay coefficient. After a complete travel, the local updates are deleted.[2]

Secondly, after all ants have completed the paths, the global update of pheromone is done only taking into account the best tour so far. In this case,

$$\tau_{ij} = (1 - \rho).\tau_{ij} + \Delta\tau_{ij}^{best}$$

Where,

$$\Delta\tau_{ij}^{best} = \begin{cases} Q/L_{Best} & \text{if best ant uses transition city } i \to j \text{ in its tour} \\ 0 & \text{otherwise} \end{cases}$$

L_{Best} is the total length of the best path so far, and Q is a constant.

7.4 Algorithm

1. Initialize parameters
2. Randomly place m ants in n cities
3. Choose transitions
    (a) Calculate p_{ij} using equation given above.
    (b) trasit from city i to j randomly based on probabilities p_{ij} and locally update pheromones
4. When all ants have completed a solution, choose best solution and globally update pheromones using aforementioned expressions and remove local updates
5. Iterate the process

7.5 Results on TSPLIB Problems

We chose α to be 1 and β to be 10. We have not experimented with other values of these parameters, however, there is literature suggesting choice of these hueristics but solutions were close to optimal. Choice of number of ants depend on the number of cities in our problem. In general we considered number of ants to be close to the number of cities.

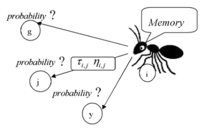
In our experiments, number of ants were chosen to be in a range of [10,50], and number of iterations we chose to were in a range of (100-200).

Since TSPLIB contains repository of TSPs which vary in their sparsity, range of elements in cost matrix, as a benchmark of the algorithm, we tested the algorithm on TSPLIB problems. For this purpose, we considered both Symmetric and Assymetric TSP Problems from TSPLIB database, and solutions are presented below:

Table 1: Observed vs Optimal solutions

Dataset	**No. of cities**	**No. of ants**	**Iterations**	**Solution found**	**Opt. solution**
burma14.tsp	14	10	100	3381	3323
ulysses16.tsp	16	10	100	6987	6859
att48.tsp	48	40	200	10725	11153
berlin52.tsp	52	40	200	7664	7542
ch150.tsp	150	50	200	6761	6528
st70.tsp	70	40	100	698	675
gr17.tsp	17	15	100	2149	2085
gr24.tsp	24	20	100	1278	1272
gr48.tsp	48	40	200	5280	5046
eil101.tsp	101	50	200	661	629
gr96.tsp	96	40	200	56752	55209
rat195.tsp	195	40	100	2454	2323
ft70.atsp	70	50	100	40729	38673
ftv47.atsp	47	30	100	1842	1776
ftv64.atsp	64	50	200	1839	1839
ftv33.atsp	33	30	100	1321	1286
ftv38.atsp	38	30	100	1699	1530
kro124p.atsp	134	50	100	40709	36230
ry48p.atsp	48	30	100	15636	14422

Graph below shows the evolution of computational time with increasing size of TSP problem:



This can be improved using further optimisation methods *e.g* nearest neighbour search and 3-OPT methods. These optimizations are suitable for TSPs of large sizes.

7.6 Complexity Analysis of Ant Colony System

Complexity of ant colony system (ACS) method is quite high, especially for problems with hundereds of cities. The reason being essentially two loops in every pheromone update. And so the complexity is $\Omega(n^2)$. Computational Complexity of Ant Colony has been studied by Sudholt et. al. (2012) [5]. One idea of an improvment can be selecting a certain number of neighbouring cities which are yet to be visited in every transition, and thus reducing the time complexity as not all cities have to be searched in every update. [6]. Other optimization methods such as MAX-MIN Ant System provide for a faster implementation of ant colony optimization. In MMAS, maximum and minimum pheromone amounts ($\tau_m ax$, $\tau_m in$) are added. Only global best or iteration best tour deposited pheromones. All edges are initialized to $\tau_m in$ and reinitialized to $\tau_m ax$ when nearing stagnation.[4]

8 Genetic Algorithm

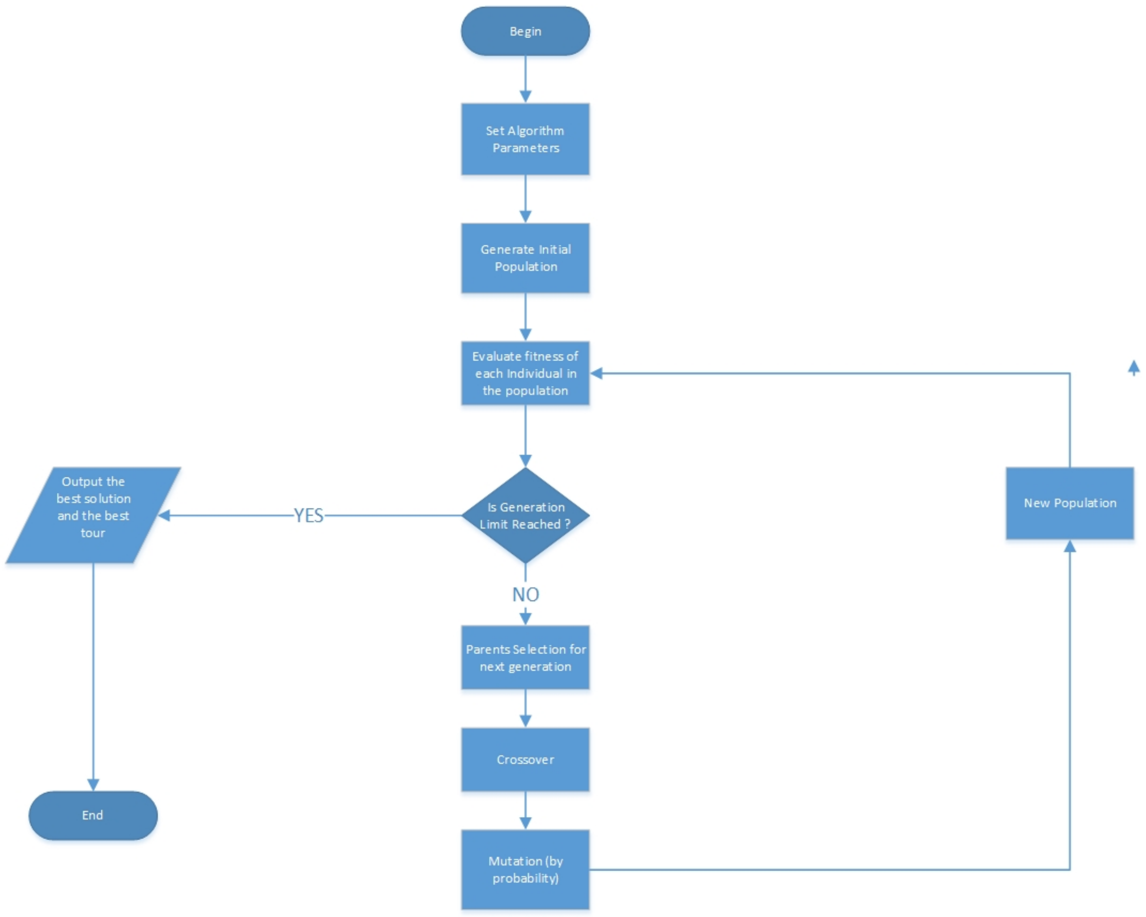
8.1 Introduction

We present in this report the main description for the attached source code to solve Travelling salesman Problem.

As illustrated in the figure below, the main algorithm pipeline contains several main components:

1. Initializing Algorithm Parameters
2. Generate Initial Population
3. Fitness Evaluation
4. Parents Selection
5. Crossover
6. Mutation
7. Generate New Population

The following sections will explain briefly each components.


Begin
Set Algorithm Parameters
Generate Initial Population
Evaluate fitness of each Individual in the population
Is Generation Limit Reached ?
YES
Output the best solution and the best tour
End
NO
Parents Selection for next generation
Crossover
Mutation (by probability)
New Population

8.2 Initializing Algorithm Parameters:

In this step, several parameters should be initialized by the user. Specifically,

1. Max Population Size: to specify the maximum number of individuals in any population
2. Max Generation Numbers: usually the termination condition
3. Mutation Rate: a real number in [0,1] specify the likelihood of mutation to happen

8.3 Generate Initial Population:

In this step an initial generation is created randomly by creating a first random individual (Tour), and use shuffling on this individual until we reach the max population size.

8.4 Fitness Evaluation for Elitism:

in this step, the fitness function for each individual should be calculated. For the TSP problem. the fitness function could be define as the total sum of distances for each tour. This fitness function is used to evaluate how good each individual (Solution). Consequently, the fittest individual is elited to the next generation directly.

8.5 Parents Selection:

In this step, parents are chosen to mate together (crossover) and have a new individual. the selection process could be done in several ways. we applied Roulette Selection (with flexibility in the code to add new other selection algorithms without changing the code). the basic of Roulette Selection is to pick a parent randomly according to weighted probability for each individual. the weights here represented by the fitness of the individual. In other words, individuals with high fitness function, have higher probabilities to be selected.

8.6 Crossover:

The crossover is the process of merging two individuals to have a new ones. there is several ways to implement the crossover. in our case, we implemented the Single Point Crossover. where a random point is generated bounded by the length of the solution. Afterwards, a new individuals are generated by taking the first part (before the picked point) from the first parent and the rest from the other taking into account the satisfaction of TSP constraints (no duplication)

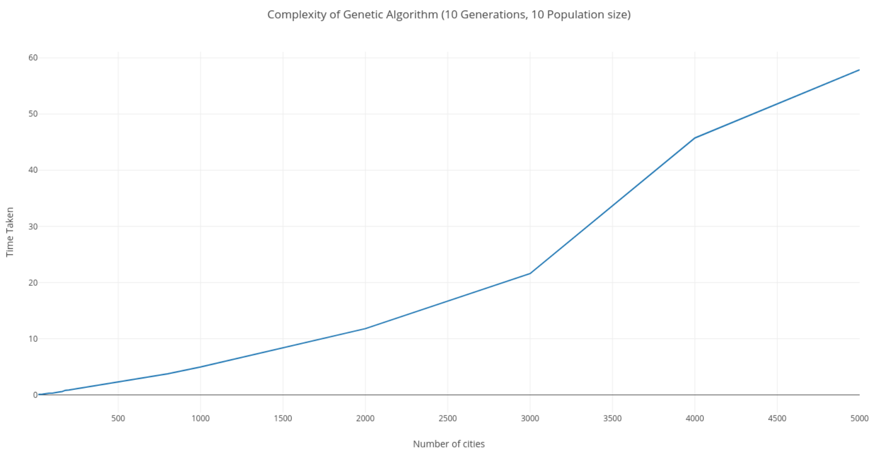
8.7 Mutation:

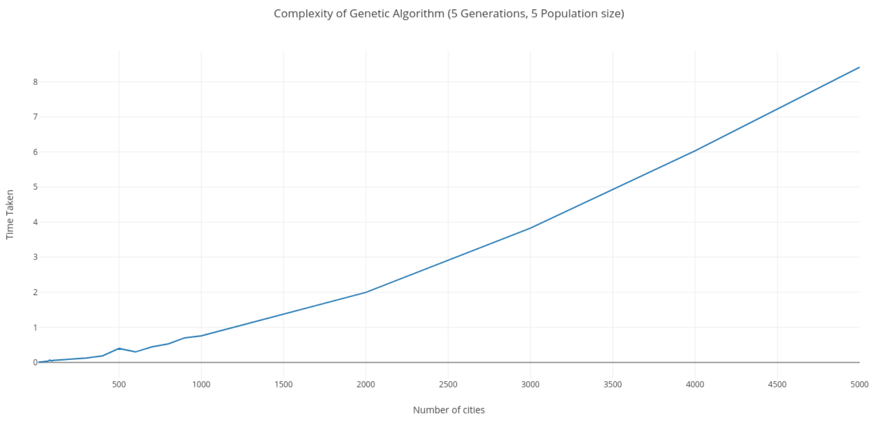
Each individual might undergo a mutation with a probability (specified in the parameters). the mutation basically is done by scanning the individual genes and picking a random number (bounded by the length of tour) and swap the current gene with the randomly picked one.

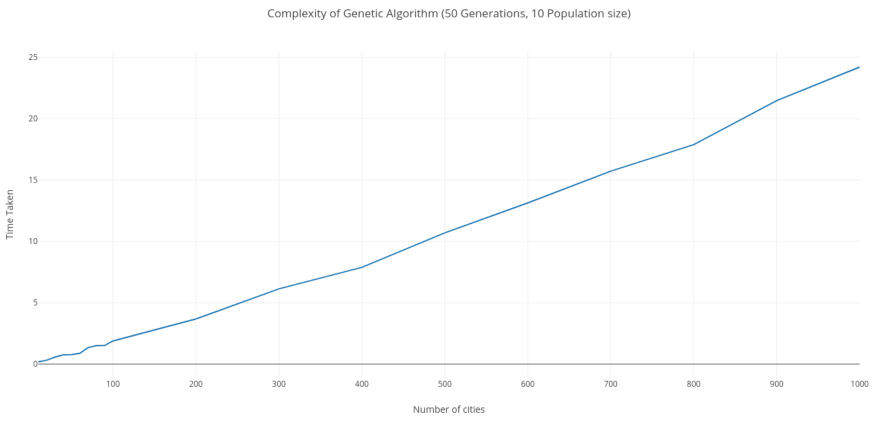
8.8 Generate New Population:

We keep repeating steps 5,6,7 until we have a full new population (enhanced one). and we go back to step 4 while we have not reached the generation limit.

8.9 Complexity







9 Greedy Algorithm

The greedy algorithm proceeds by applying a local optimal choice at each node. This strategy can sometimes provide a global optimal but it's not always the case. In most cases it provides an approximation of the optimal solution to the problem.

9.1 Details of implementation

```
Data: m x m Adjacency matrix
Result: Tour of length (m+1) and total cost of the
        tour

tour ← [0]
totatl_cost ← 0
while not all nodes visited do
    current_node ← last element in tour
    next_node ← node with min weight (list of
      adjacent nodes)
    append next_node to the end of tour
    total_cost += distance(current_node,
      next_node)
end
append 0 to the end of tour
total_cost += distance( next_node, first_node)
return tour , total_cost
```

Algorithm 3: Greedy algorithm for the TSP

9.2 Results

9.3 Discussion

From the implementation of the algorithm we can see it has a complexity of $O(n)$. This makes the algorithm efficient in approximating even the biggest problem in a very reasonable time. And this can be seen in the results section, as even the *rl11849.tsp* problem with 11849 vertexes took less than a minute to run.
However, the weak side of this algorithm is that it provides only an approximation rather than the optimal solution. This makes it suitable for problem with large n values, where the optimal solution is not necessarily required and an approximation would be enough.

The algorithm was run on the complete set of TSP-LIB problems (both symmetric and asymmetric). Following is a sample of these results to showcase strong and weak points of this algorithm.

Problem Name	No of vertexes	Execution Time(s)	Min Path Cost	Library
att48.tsp	48	12861	<1 ms	Symmetric TSP
a280.tsp	280	3213	8.808 ms	Symmetric TSP
fl1577.tsp	1577	29178	328.72 ms	Symmetric TSP
fnl4461.tsp	4461	223855	3525.013 ms	Symmetric TSP
rl11849.tsp	11849	1118490	52218.616 ms	Symmetric TSP
ftv33.atsp	34	1683	<1 ms	Asymmetric TSP
ftv170.atsp	171	3923	6.13 ms	Asymmetric TSP
rbg443.atsp	443	6183	42.4 ms	Asymmetric TSP

A set of .csv files are included in our github repository with the full set of results.

Table 2: Summary of greedy results on the TSP-LIB

10 MST Algorithm

The Minimum Spanning Tree (MST) approach in the TSP problem starts by generating an MST for the problem's graph (which is defined by a tree that connects all the vertexes in a graph with the least possible overall weight of edges). Then a Depth First Search (DFS) is applied to acquire the tour path for the TSP.

10.1 Details of implementation

```
Data: m x m Adjacency matrix
Result: Tour of length (m+1) and total cost of the tour

$T \leftarrow \phi$
$U \leftarrow [0]$
while $U \neq V$ do
    let (u, v) be the lowest cost edge such that u ∈ U and v ∈ V - U
    T = T ∪ {(u, v)}
    U = U ∪ {v}
end
tour = DFS (T, u)
return tour , total_cost
```

Algorithm 4: MST algorithm for the TSP

10.2 Results

The algorithm was run on the complete set of TSP-LIB problems (both symmetric and asymmetric). Following is a sample of these results to showcase strong and weak points of this algorithm.

Problem Name	No of vertexes	Execution Time(s)	Min Path Cost	Library
att48.tsp	48	13926	<1 ms	Symmetric TSP
a280.tsp	280	3436	26.90 ms	Symmetric TSP
fl1577.tsp	1577	29411	1012.26 ms	Symmetric TSP
fnl4461.tsp	4461	248612	7920.18 ms	Symmetric TSP
rl11849.tsp	11849	1384790	76464.248 ms	Symmetric TSP
ftv33.atsp	34	1712	1.02 ms	Asymmetric TSP
ftv170.atsp	171	5032	9.95 ms	Asymmetric TSP
rbg443.atsp	443	7561	73.76 ms	Asymmetric TSP

A set of .csv files are included in our github repository with the full set of results.

Table 3: Summary of greedy results on the TSP-LIB

10.3 Discussion

The MST algorithm consists of two main parts; creating the minimum spanning tree and then searching the tree in a depth first manner, by analyzing the implementation: extracting the edge with the lowest cost is $O(log(V))$. This means creating the MST's complexity is $O(Vlog(V))$. That makes the total complexity for the algorithm $O(Vlog(V) + V)$.
As for the strong and weak points of the algorithm we can draw a lot of similarities to the greedy algorithm. It's very efficient in approximating huge problems as shown in the results table. But also it only approximates a solution and doesn't usually provide an optimal one.

References

[1] M. Dorigo, *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italy, 1992.

[2] M. Dorigo and L. Gambardella, "Ant colony system: a cooperative learning approach to the traveling salesman problem," *IEEE Transactions on Evolutionary Computation*, vol. 1, pp. 53–66, apr 1997.

[3] M. Dorigo, M. Birattari, and T. Stützle, "Ant colony optimization – artificial ants as a computational intelligence technique," *IEEE Comput. Intell. Mag*, vol. 1, pp. 28–39, 2006.

[4] T. Stützle and H. Hoos, “Improvements on the ant-system: Introducing the MAX-MIN ant system,” in *Artificial Neural Nets and Genetic Algorithms*, pp. 245–249, Springer Vienna, 1998.

[5] D. Sudholt and C. Thyssen, “Running time analysis of ant colony optimization for shortest path problems,” *Journal of Discrete Algorithms*, vol. 10, pp. 165–180, jan 2012.

[6] F. Neumann, D. Sudholt, and C. Witt, “Computational complexity of ant colony optimization and its hybridization with local search,” in *Innovations in Swarm Intelligence*, pp. 91–120, Springer, 2009.