Investigation of the Travelling Salesman Problem and its Heuristic Methods

Cedric Anover
Department of Mathematics and Statistics
Curtin Univeristy

Industrial Project Report

Advisor
Qun Lin

Abstract

In this paper, I will give a survey and overview of the travelling salesman problem and its heuristic methods, which is a popular combinatorial optimization problem. The travelling salesman problem (TSP) asks to find, in a given network or graph, the shortest distance cycle passing through each city exactly once. Understanding TSP is very easy but deriving an algorithm that gives 'good' solution is the main challenge and still an active area of research. I will give initial definitions from graph theory that I will use to discuss the TSP itself and its existing methods. I will introduce the integer programming formulation of TSP and the two common to constraints to eliminate sub-tour. I will also discuss different special cases and variations of the TSP. This paper will also give a survey of the current & standard heuristics and metaheuristics for solving TSP and will be illustrated with concrete examples. I will also present the simulation experiment where I compared different heuristic methods in different distance matrix sizes and see which heuristic methods gives the most accurate solution.

1 Introduction

Travelling Salesman Problem (TSP) is a well-known combinatorial optimization problem. TSP acts as a testbed for any newly developed algorithms and methods for solving other combinatorial optimization problem. So after deriving an algorithm for solving a optimization problem, people usually test their derived algorithm to TSP and see is their algorithm is 'better' than other existing algorithms. By 'better' meaning the algorithm gives more accurate solution and solves the problem in less time. TSP falls in the category NP-hard problems. The TSP problem is very easy to understand but very hard to derive an algorithm that gives exact optimal solution that runs in polynomial time.

The formal definition of the travelling salesman problem:

Def(Travelling Salesman Problem):
Given a set of cities. Find the shortest distance tour passing through each city exactly once.
◊

The difficulty of TSP increases rapidly as the number of cities increases. For a given n (number of cities) cities the number of feasible solution is $\frac{(n-1)!}{2}$. So for a complete graph (see definition in next section) with n vertices/cities/nodes, the table gives the number of feasible solution.

n	(n-1)!/2
4	3
5	12
6	60
7	360
8	2520
9	20160
10	181440
…	…

So as the number of cities increase, so does the number of feasible solution. This what makes TSP hard since exact methods such as Branch-and-Cut or Brute-Force would require large amount of time to search for the optimal solution. Therefore, as an alternative, most people use approximation and heuristic algorithms for solving a large-scale TSP.

2 Theoretical Background and History

The Travelling Salesman Problem can be modelled as a weighted graph G=(V,E,w) where V is the set of all vertices representing cities and E is the set of all edges representing distance between cities with weight function $w: E \rightarrow \mathbb{R}_{>0}$. TSP is to find a Hamiltonian cycle, which is a cycle that visits each vertex exactly once, with the least weight. These graphs where first studied by William Hamilton, hence Hamiltonian Graphs. Hamilton started investigating these problems back in the 1850's. Hamilton exhibits his Icosian games. These games involve finding various paths and cycles, as well as spanning trees from a given dodecahedron (Figure 1). But the original person who posed the problem of finding Hamiltonian cycles is Thomas Kirkman: Given a graph of a polyhedron, can one always find a cycle (or circuit) that passes through each vertex exactly once. Figure shows the dodecahedron with Hamiltonian circuit (Balakrishnan & Ranganathan, 2012).

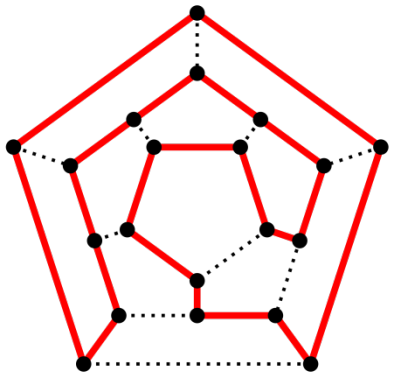


Figure 1: Hamiltonian Cycle in a Dodecahedron

There are also other problems that are related to finding a Hamiltonian cycle. One such example is the knight's tour in a chessboard where the 64 squares of the chessboard represents the vertices and the edges are represented by the knight's tour. Now the problem is to find a tour such that the knight does not visit any intermediate square more than once. This is equivalent to finding a Hamiltonian Cycle in that 64 square chessboard. This puzzle is first introduced by the great mathematician Euler. One solution could be seen in Figure 2.

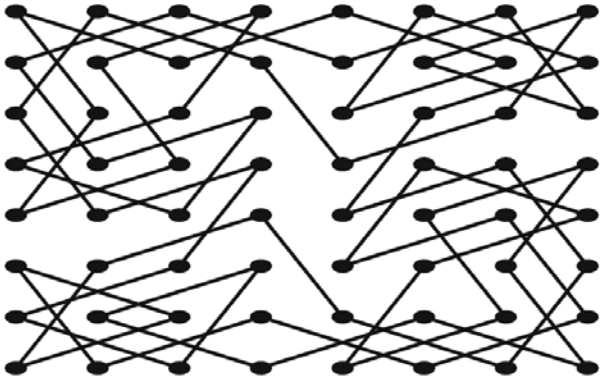


Figure 2: Hamiltonian Cycle for Knight's Tour

An analog of Hamiltonian Cycle is Eulerian Cycle (for edges). An Eulerian cycle is a cycle that uses each edge exactly once. And we say that a graph G is an Eulerian Graph if there is an Eulerian Cycle. This graph where first introduced by Euler in 1736 when he investigated the Konigberg bridge problem which asks if the seven bridges of Konigsberg can be all traversed in one trip only without going back. Figure 3 shows the visual representations of the problem. Eulerian cycle may seem different from Hamiltonian cycle, but Eulerian cycle will be a useful component to some of the heuristic methods for finding a Hamiltonian Tour (a closed tour or a cycle) in a given TSP graph.

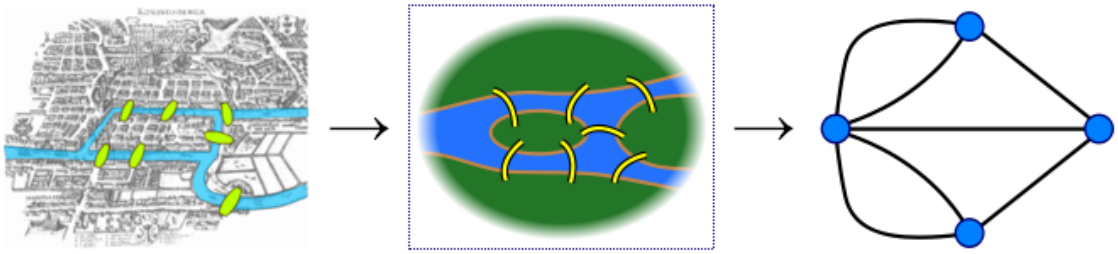
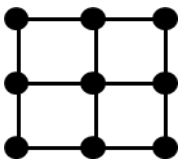


Figure 3: Konigsberg Bridge Diagram

The main difficulty in area of Hamiltonian Graphs in Graph Theory is to find simple statements for the necessary and sufficient condition for the existence of a Hamiltonian Cycle in a given graph G. There are lots of results for the necessary condition, but virtually all the assumptions in these results are 'too strong', meaning that one has to satisfy many assumptions before concluding the existence of a Hamiltonian Cycle in a given graph G. And most of these theorems assume already that a given graph already has a Hamiltonian cycle.

Theorem:
If G is Hamiltonian, then G is 2-connected. (DeLeon) ◊

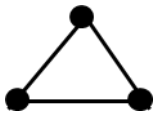
This theorem already assumes that the given graph is already Hamiltonian. 2-connected means that the minimum number of vertices to be deleted such that the resulting graph will be disconnected is 2. What about the converse of this theorem? Let's claim that if a graph is 2-connected, then this graph is Hamiltonian. This statement is false and the figure below is a counter-example.



As you can see, if you observe carefully, this graph is 2-connected but does not have a Hamiltonian cycle.

Theorem (Dirac's Condition):
If G is a graph of order $n \geq 3$ *such that* $\delta \geq \frac{n}{2}$*, then G is Hamiltonian.* (DeLeon) ◊

Example:
An extreme example for this theorem is when n=3.



The degree of each vertex in this graph is 2 ($\delta = 2$) and $\frac{n}{2} = \frac{3}{2}$. Since we satisfied the Dirac's condition then this graph is Hamiltonian. In fact it is very obvious that the graph itself is the Hamiltonian cycle. □

Theorem (Ore's Condition):
If G is a graph of order $n \geq 3$ *such that for all distinct non-adjacent pairs of vertices u & v,* $deg(u) + deg(v) \geq n$*, then G is Hamiltonian.* (DeLeon) ◊

Dirac and Ore conditions are using the intuitive fact that the more edges the graph has the more likely a Hamiltonian cycle exist.

Theorem (Nash-Williams):
Let G be a 2-connected graph of order with $\delta \geq max\{\frac{n+2}{3}, \beta\}$*. Then G is Hamiltonian.* (DeLeon) ◊

Theorem (Chvatal-Erdos):
Every graph G with $n \geq 3$ *and* $\kappa \geq \beta$ *is Hamiltonian.* (DeLeon) ◊

Note that κ is the graph connectivity. Thus, this theorem says that if G is $\beta - connected$ then G is Hamiltonian. β represent the independence number of G.

Now let's review some basic important definitions related to TSP.

Def(Graph):
A graph is an ordered-pair $G = (V, E)$ where V is the set of vertices and E is the set of edges. ◊

Def(Complete Graph):
A complete graph, denoted by K_n, with n vertices is a graph with exactly one edge between each pair of distinct vertices. ◊

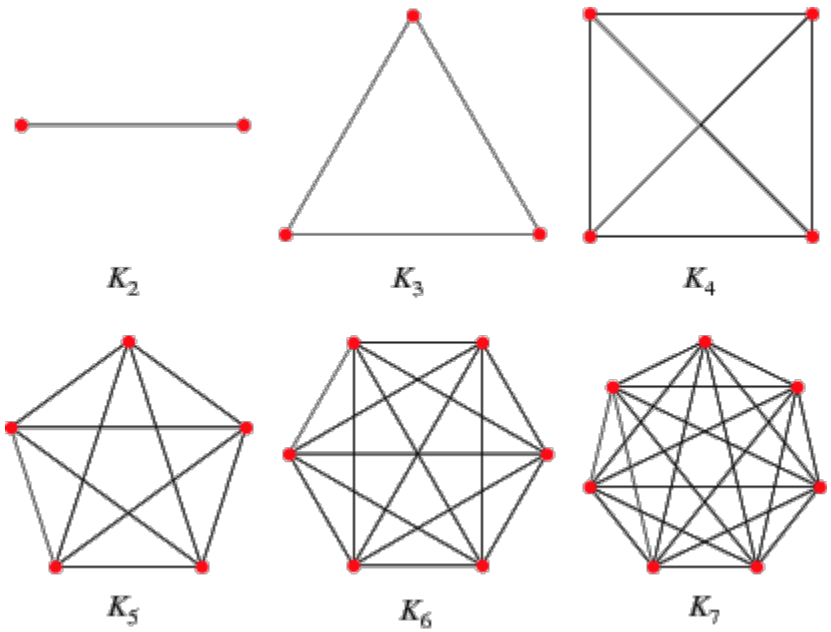


Figure 4: Initial Cases of Complete Graph

Def(Hamiltonian Cycle):
A Hamiltonian Cycle is a Cycle containing all vertices of a graph G. ◊

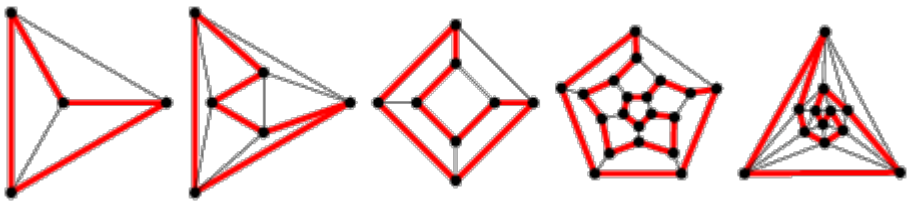


Figure 5: Examples of Hamiltonian Cycles in a Graph

Remarks: Sometimes we will use the term "Tour" but what this really meant is a 'closed' tour (which is a cycle).

Def(Hamiltonian Graph):
A graph G is called a Hamiltonian Graph is there exists a Hamiltonian Cycle. ◊

In Figure 5, the graphs are example of a Hamiltonian Graph since there exits Hamiltonian Cycles.

3 Travelling Salesman Problem Formulation and Variations

3.1 TSP as Integer Programming Problem

The Travelling Salesman Problem (TSP) is considered as an Integer Programming Problem, in particular, a knapsack problem. It's a knapsack problem since the decision variable $x_{ij} \in \{0,1\}$ represents as to whether or not an edge (i, j) will be included in the TSP tour. The elements of the distance matrix $\left[d_{ij}\right]_{n\,x\,n}$ should be a non-negative real number since it represents distance. The diagonal elements of the distance matrix could be Null or we can assign ∞ since we do not allow 'loops' in a TSP graph. Then $\sum_{i=1}^{n} \quad \sum_{j=1}^{n} d_{ij}x_{ij}$ denotes the tour length. Hence TSP can be formulated as the following minimization problem:

$$Min \sum_{i=1}^{n} \quad \sum_{j=1}^{n} d_{ij}x_{ij}$$

Subject to

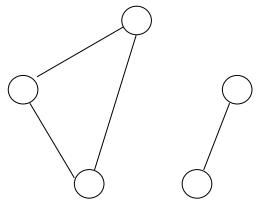
$$\sum_{j=1}^{n} x_{ij} = 1, \qquad \forall i = 1, \dots, n \quad (Tour\ leaves\ each\ city\ exactly\ once)$$

$$\sum_{i=1}^{n} x_{ij} = 1, \qquad \forall j = 1, \dots, n \quad (Tour\ enters\ each\ city\ exactly\ once)$$

$$\sum_{i \in S} \quad \sum_{j \in S} x_{ij} \leq |S| - 1, \qquad S \subset \{1, \dots, n\} \quad (Subtour\ Elimination\ Constraint)$$

$$x_{ij} \in \{0,1\}, \qquad \forall i, j \in \{1, \dots, n\}$$

The first and second constraints ensures that we can 'enter' and 'exit' every node (or city) exactly once to satisfy the problem description of TSP. The third constraint is the sub-tour elimination constraint which ensures that we will not have a sub-tour or a situation such as the following figure for a complete graph with 5 vertices (or cities).



This is clearly not a Hamiltonian cycle; therefore we have to impose the sub-tour elimination constraint. In total, there will be $2^n - 2$ sub-tour elimination constraints. As the number of cities n gets larger, so does the number of sub-tour elimination constraints. Another formulation for Sub-tour elimination constraint is the so called Miller-Tucker-Zemlin (MTZ) formulation (Pataki, 2000).

$$u_1 = 1$$
$$2 \leq u_i \leq n$$
$$u_i - u_j + 1 \leq n(1 - x_{ij}), \qquad \forall i, j \in \{1, \dots, n\} \backslash \{1\}, i \neq j$$

The last constraint of the MTZ formulation constraint is called the arc-constraint for the arc (i, j). This eliminates sub-tours because 1) the arc-constraint will be $u_j \geq u_i + 1$ when x_{ij}=1, 2) if the feasible solution contained more than one sub-tour, then at least one of these would not contain city 1 and along this sub-tour the u_i values would have increase to infinity. For MTZ there will be P_2^{n-1} (permutation) sub-tour elimination constraints, excluding $u_1 = 1$ and $2 \leq u_i \leq n$. Comparing this with the standard sub-tour elimination constraint $\sum_{i \in S} \quad \sum_{j \in S} x_{ij} \leq |S| - 1, \ S \subset \{1, \dots, n\}$, MTZ will have less number of sub-tour elimination constraints.

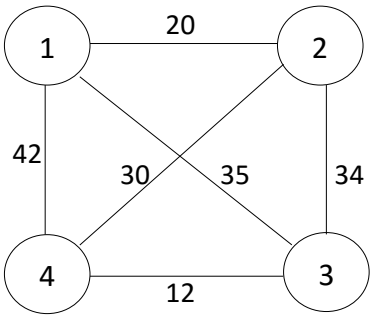
n	$2^n - 2$	P_2^{n-1}
4	14	6
5	30	12
6	62	20
7	126	30
8	254	42
9	510	56
10	1022	72

3.2 Variations of TSP

There are many special cases and variations of TSP. We shall discuss some popular examples of them such as Symmetric TSP, asymmetric TSP, metric TSP, and Euclidean TSP.

Def(Symmetric TSP):
Let $[d_{ij}]_{n\,x\,n}$ to be the distance matrix that describes the TSP graph. We say that the TSP is a symmetric TSP if $d_{ij} = d_{ji}\ for\ all\ i,j \in \{1,\dots,n\}, i \neq j.$ ◊

This definition means that the TSP graph is 'undirected' i.e. the distance going from node i to node j is the same as the distance going from node j to node i. For example,



And its corresponding distance matrix.

$$[d_{ij}] = \begin{bmatrix} - & 20 & 42 & 35 \\ 20 & - & 30 & 34 \\ 42 & 30 & - & 12 \\ 35 & 34 & 12 & - \end{bmatrix}$$

Note that if the TSP is symmetric (i.e. $d_{ij} = d_{ji}$), then the decision variable should also be symmetric (i.e. $x_{ij} = x_{ji}$).

An interesting extension to symmetric TSP is the so called Metric TSP where we impose the additional triangle inequality constraint: $d_{ik} + d_{kj} \geq d_{ij}, \forall i, j, k \in \{1, \dots, n\}, i \neq j \neq k$. Meaning that going directly from node i to node j is shorter than taking a detour through node k. The triangle inequality constraint could also be imposed to an asymmetric TSP.

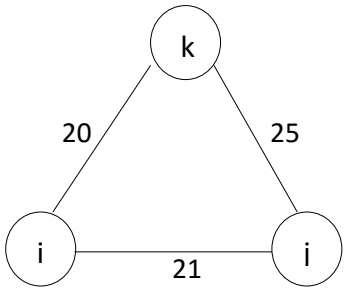


Figure gives a visual example of the Metric constraint:

The symmetric problem can be formulated with one-half the number of binary decision variables as compared to asymmetric problem (CHEN, BATSON, & DANG, 2010). As opposed to symmetric TSP we have the following definition of Asymmetric TSP.

Def(Asymmetric TSP):
Let $\left[d_{ij}\right]_{n\,x\,n}$ to be the distance matrix that describes the TSP graph. We say that the TSP is an asymmetric TSP if $d_{ij} \neq d_{ji},\ for\ some\ i, j \in \{1, \dots, n\}, i \neq j$. ◊

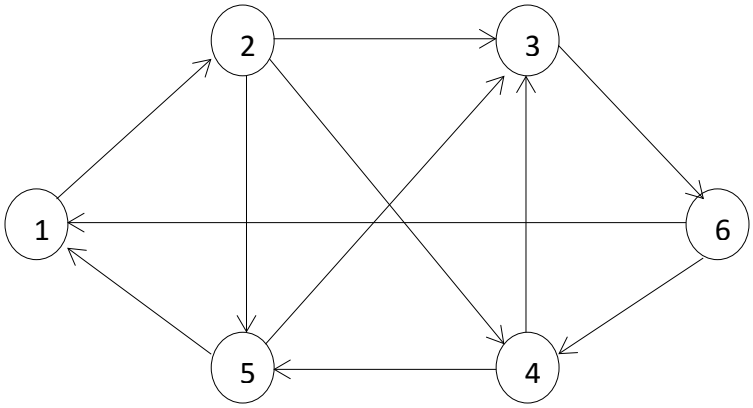
This asymmetric TSP would form a directed graph and it extends the definition of symmetric TSP. It is very likely that for some TSP graphs (especially small number of nodes or cities) there exist no Hamiltonian cycle or TSP tour this is because of the possibility that some node (or city) may only be a head (or tail) (see the definition of directed arc) and will violate the condition of the problem description TSP where each node must be visited exactly once.

Def(Directed Arc):
A directed arc is an ordered pair (a, b) of vertices where the vertex ‘a’ is called the tail and vertex ‘b’ is called the head. ◊

E.g.

Example(Asymmetric TSP graph):

Consider a graph with 6 nodes. The Distance matrix (in fact it's just an adjacency matrix) would be $\begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$.



TSP tour is 1-2-3-6-4-5-1. But suppose arc (1, 2) is change to (2, 1) (i.e. node 2 is now the tail and node 1 is the head), then there will be no Hamiltonian cycle since node 1 is a head for arcs (2, 1), (6, 1), and (5, 1). So for an asymmetric TSP graph, each node must be both a head and tail. But this condition is not completely sufficient to conclude that there exist a Hamiltonian cycle in a given asymmetric TSP graph. □

Another special case of TSP is Euclidean TSP. In Euclidean TSP the distances are on the Euclidean Plane R^2 i.e. $d(v_i, v_j) = \|v_i - v_j\|_2$ is the distances between nodes $v_i = (x_i, y_i)$ and $v_j = (x_j, y_j)$ for $i \neq j$. The following definition gives the formal description of the Euclidean TSP

Def(Euclidean TSP):

Given n points (i.e. nodes) in the plane (R^2), the Euclidean TSP is a problem that asks to find a tour that visits all points and minimizes the distance travelled, where the distance between two points is given by the Euclidean distance $d(v_i, v_j) = \|v_i - v_j\|_2$.

4 Heuristic Methods

In this section we will be discussing the description of existing methods for solving the TSP. We will only discuss heuristic (and metaheuristic) methods for solving TSP, thus we will not discuss much on exact methods for solving TSP such as Branch & Bound and Branch & cut methods.

TSP is a NP-Hard problem (i.e. as the problem size gets larger, so does the computational time to find the optimal solution). Therefore it is impractical to use exact methods for finding an optimal solution of TSP and instead we use heuristic methods. Thus we sacrifice optimality for speed. Virtually all heuristic methods run in polynomial time as opposed to exact methods which runs in exponential time.

We give the following definitions for heuristic and metaheuristic methods.

Def(Heuristic Method):
A heuristic method is a procedure that is likely to discover a very good feasible solution, but not necessarily an optimal solution for a specific problem to be considered (Hillier & Lieberman, 2015). ◊

Remarks: There is no guarantee that the feasible solution found by a heuristic method is optimal. But a well-designed heuristic can produce a near optimal solution. The procedure should also be sufficiently efficient for solving large problems. Heuristic methods are designed based on common-sense for how to search, find, or construct feasible solutions.

Def(Metaheuristic):
A metaheuristic is a general solution method that provides both a general structure and strategy guidelines for developing a specific heuristic method to fit a particular kind of problem (Hillier & Lieberman, 2015). ◊

Remarks: Metaheuristics provides the framework for developing heuristic methods for solving any specific combinatorial optimization problems.

There are two main types of Heuristics for solving TSP:

- Tour Construction Heuristics (Constructive heuristic) – The algorithm stops when a feasible solution is found and never tries to improve it. Most constructive heuristics also falls in the category of greedy algorithms because for each iteration we select the shortest weighted-edge and add it to the tour as long as it does not create a non-Hamiltonian cycle.
- Tour Improvement Heuristics – Once a tour has been generated by some constructive heuristic, we may wish to improve it. The first step of most tour improvement heuristics requires an initial feasible tour (which could be found by implementing other constructive heuristic) and then improves that feasible tour. All metaheuristics are tour improvement heuristics.

The only properties we study when we talk about heuristic methods for TSP are speed and closeness to the optimal solutions. But how does one measure the speed and the closeness to the optimal solution of a heuristic method? There are two ways to measure the speed of a heuristic method, one is by record the physical computational time and the other is by using the big O notation in computational complexity (but we are not going to discuss much of big O notation). There are also two common ways to measure the closeness to the optimal solution of a heuristic method, one is by actually calculating the optimal solution by applying exact methods and then compare this exact solution with the solution found by a heuristic methods (in practice, this is not a good choice if the problem size is too large). And another way is to calculate the lower bound of the unknown optimal solution, L_{LB}.

4.1 Nearest Neighborhood Heuristic

This is perhaps the most simplest and one of the first heuristic methods for solving the TSP. Nearest Neighborhood Heuristic (NNH) is a greedy algorithm and a constructive heuristic. NNH compares the distribution of distances that occur from data points to its nearest neighbor in a given data set with a randomly distributed data set. For a given TSP graph even if it has very large number of vertices, NNH only takes few computational time to produce a feasible solution. NNH runs in polynomial time with $O(n^2)$, where n represents the number of nodes or cities.

The key idea of this algorithm is to always visit the nearest city. It starts with an initial city, as long as there are cities that have not yet been visited, NNH visit the nearest city that has not yet appear in the TSP tour (or Hamiltonian Cycle), and finally return to the home city (Karkory & Abudalmola, 2013). To solve TSP with NNH we look at all edges coming out of the city (or node) that have not yet been visited and choose the next closest city, then return to the initial city when all other cities have already been visited (SAHALOT & SHRIMALI, 2014). Different cities usually give different feasible TSP tour, and so there is a modification to NNH called Repetitive NNH where it test all the solutions produced by stating at different initial city and then select the one with the minimum TSP tour length.

The computational time of repetitive NNH will be more than that of standard NNH, but the advantage of repetitive NNH over standard NNH is that it will give better feasible solution.

For a small sized TSP graph (by small sized meaning few number of nodes), NNH usually gives a near optimal solution. But as the problem size gets larger (e.g. n≥15), NNH gives feasible solution that is quite far to the optimal solution. Comparing NNH to other heuristics, unfortunately NNH performs the worst. Although when it comes to computational time, NNH is one of the fastest methods for solving TSP. This is because of the greedy feature of NNH.

There's a study done by (Karkory & Abudalmola, 2013) where they implement and compare NNH and Minimum spanning tree algorithm. In that paper, results shows that small size of cities, the two algorithms performed equally. But as the problem size gets larger NNH performs better than Minimum spanning tree algorithm when it comes to calculating the tour lengths. And as for computational time, NNH is always lower (i.e. faster) than Minimum spanning tree.

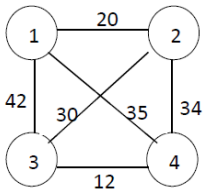
Nearest Neighborhood Heuristic Algorithm:

Step 1: Choose the home city $i_1 \in \{1, \dots, n\}$.
Step 2: Set k=1 & $i = i_1$. Where k is the iteration number and i is the current city.
Step 3: Define set $\Gamma_k = \{i_1, .., i_k\}$
Step 4: Calculate $i_{k+1} = argmin_{j \in \{1,\dots n\} \setminus \Gamma_k} \{d_{ij}\}$
Step 5: Set k=k+1 & $i = i_k$
Step 6: If k=n, define $i_{n+1} = i_1$

Remark(s): This algorithm constructs a feasible TSP tour $\Gamma_k = \{i_1, i_2.., i_n, i_1\}$. Different choice of home city usually gives different feasible solution. It would be better to define a count-controlled loop to test different initial city and select the city which has the least objective value as a candidate for optimality.

Example (Applying NNH):

Consider a TSP graph



This has a distance matrix of $[d_{ij}] = \begin{bmatrix} - & 20 & 42 & 35 \\ 20 & - & 30 & 34 \\ 42 & 30 & - & 12 \\ 35 & 34 & 12 & - \end{bmatrix}$ which is symmetric. Let V= {1,2,3,4} the set of vertices.

Choose $i_1 = 1$
k=1; $i = i_1 = 1$ & $\Gamma_1 = \{i_1\} = \{1\}$.
$i_2 = argmin_{j \in V \setminus \Gamma_1}\{d_{12}, d_{13}, d_{14}\}$ = argmin{20,42,35} = 2
k=1+1=2 & $i = i_2 = 2$

k=2; $i = i_2 = 2$ & $\Gamma_1 = \{i_1, i_2\} = \{1, 2\}$.
$i_3 = argmin_{j \in V \setminus \Gamma_2}\{d_{23}, d_{24}\}$ = argmin{30,34} = 3
k=2+1=3 & $i = i_3 = 3$

k=3; $i = i_3 = 3$ & $\Gamma_1 = \{i_1, i_2\} = \{1, 2\}$.
$i_4 = argmin_{j \in V \setminus \Gamma_3}\{d_{34}\}$ = argmin{12} = 4
k=3+1=4 & $i = i_4 = 4$

k=n=4
We stop here and let $i_5 = i_1 = 1$ and the path is $\Gamma_4 = \{1, 2,3,4,1\}$ with length of 97. For this example, it turns out that this is the optimal length. Although in the algorithm it will not stop here, it will test all different home cities and then compare which gives the best candidate for optimality. □

4.2 Insertion Heuristics & Nearest Insertion Heuristic

Similar to NNH is the Nearest Insertion Heuristic (NIH). Just like NNH, NIH is a greedy algorithm and a constructive heuristic. Insertion heuristics has many variants to choose from, one of them is NIH (there are other variants such as farthest insertion and cheapest Insertion Heuristics). The basics of NIH's is to start with a tour of a subset of all cities, and then inserting the rest by some heuristic. The idea is to build a tour starting with one initial node and insert other nodes one by one, always insert the node that is closest to a vertex already in the tour. Insertion algorithms iteratively add a city to a partial tour, initially made of one city chosen at random (e.g. $\Gamma = \{s, s\}$ where s is the selected initial city), until all cities have been inserted. Just like NNH, NIH has computational complexity $O(n^2)$ (Nilsson). NIH belongs to the class of algorithms that runs in polynomial time. Just like NNH, the procedure of NIH can be repeated with each city as the initial city and the best of

the tours obtained taken as the output of the algorithm. With the same reason that different starting city gives different feasible tours.

Variants of Insertion Heuristic:

- Nearest Insertion Heuristic (NIH)
- Farthest Insertion Heuristic (FIH)
- Cheapest Insertion Heuristic (CIH)

The other variant of Insertion Heuristic is the Furthest Insertion Heuristic (FIH). In FIH, we insert node whose minimum distance to a node on the cycle is maximum i.e. $u = argmax_{i \in V \setminus r}\{d(i)\}$. So the difference with NIH is that the selection part of FIH is to find nodes j & k (j belonging to the partial sub-tour r and k does not belong in r) for which $min_{kj}\{d_{kj}\}$ is minimized.

Another variant of Insertion Heuristic is the Cheapest Insertion Heuristic (CIH). This procedure grows a sub-tour (starting with the initial sub-tour $r = \{s, s\}$), until it become a tour (i.e. a Hamiltonian Cycle). CIH is initiated the same way as NIH. The only difference in this algorithm is the selection part. Where we find cities k, i, and j (i and j being the extremes of an edge belonging to the partial tour and k not belonging to that tour) for which $d_{ik} + d_{kj} - d_{ij}$ is minimized. The triangle inequality is satisfied.

According to some experimental results, FIH usually gives better average feasible tour length than NIH and CIH.

<u>Nearest Insertion Heuristic Algorithm:</u>

Step 1: Choose a home city $s \in \{1, \dots, n\}$.

Step 2: Set k=1 and $r = \{s, s\}$ where k is the iteration number and r is the current sub-tour.

Step 3: For each i=1,…,n. set $d(i) = d_{si}$ where $d(i)$ is the distance of i to r.

Step 4: Define $= argmin_{i \in V \setminus r}\{d(i)\} = vertex\ to\ be\ inserted\ into$ r .

Step 5: Insert u between consecutive vertices i & j in r, where i & j are chosen such that $d_{iu} + d_{uj} - d_{ij}$ is minimized (distance added to the subtour).

Step 6: If k=n-1, then stop. Otherwise, for each $i \in V \setminus r$, set $d(i) = \min\{d(i), d_{ui}\}$.

Step 7: Set k=k+1 and return to step 4.

4.3 k-Opt

The k-Opt algorithm basically removes k edges from the tour and reconnects the two paths created. This is often called k-opt move. The k-opt algorithm is a tour improvement algorithm. Continue removing and reconnecting the tours until no k-opt improvements can be found. This tour is now called k-Optimal. Usually, we only use k=2 or k=3 since larger k would increase the computational time.

Outline of k-Opt Algorithm:

Step 1: Construct a Hamiltonian cycle, denote by X_1

Step 2: Execute k-change, resulting in cycle X_2

Step 3: If $f(X_2) < f(X_1)$, set $X_1 := X_2$

Step 4: Go to step 2 until local optimum is found (the k-optimal)

The experimental result would show that this heuristic will give better solution than other heuristics (excluding tree and Christofide's algorithms)

4.4 Tree Algorithm & Christofide's Algorithm

Most heuristics can only guarantee a worse-case ration of 2 (ie a tour with twice the length of the optimal tour) (Nilsson). Tree algorithm (TA) and Christofide's algorithm (CA) is one of the best heuristics for solving TSP. Both TA & CA construct minimal spanning trees. TA & CA uses other procedures such as

- Fleury's Algorithm – For finding an Eulerian Tour
- Prim's Algorithm or Kruskal's Algorithm – For finding minimal spanning tree
- Matching Algorithms – Finding a Perfect Match

Take note that CA is an extension of TA. A minimum weight perfect matching in a complete graph of n vertices K_n can be found in $O(n^3)$.

Tree Algorithm:

Step 1: Construct a complete graph K_n in which the vertices represent cities and the weights are distances.

Step 2: Find a minimal spanning tree (denote by T) for the complete graph constructed in step 1 (implement Kruskal's algorithm or Prim's Algorithm).

Step 3: Duplicate the edges of the tree T to form a new graph (V, E)

Step 4: Find an Euler Tour for (V, E) (Implement Fleury's Algorithm).

Step 5: Form a TSP tour (i.e. Hamiltonian cycle) by taking a vertex sequence of the Euler tour and then deleting any repeated vertices (i.e. skip vertices that have already been visited).

Christofide's Algorithm:

Step 1: Construct a complete graph K_n in which the vertices represent cities and the weights are distances.

Step 2: Find a minimal spanning tree (denote by T) for the complete graph constructed in step 1 (implement Kruskal's algorithm or Prim's Algorithm).

Step 3: Define $\Omega := Set\ of\ odd - degree\ vectices\ in\ tree\ T$

Step 4: Construct a complete graph K_{2m} in which each vertex corresponds to a vertex in Ω. The weight of edge (i, j) in K_{2m} is d_{ij}.

Step 5: Find a minimum weight perfect matching for K_{2m} (implement matching algorithm). Let M^* denote this matching.

Step 6: Form a new graph (V, E) by adding edges of M^* to tree T.

Step 7: Find an Eulerian tour for (V, E) (implement Fleury's Algorithm). Note that each vertex of (V, E) has even degree.

Step 8: Form a TSP tour (i.e. Hamiltonian cycle) by taking a vertex sequence of the Euler tour and then deleting any repeated vertices (i.e. skip vertices that have already been visited).

4.5 Tabu Search

Tabu search is an example of a metaheuristic that uses obvious decisions (common sense) that enable the search procedure to escape the local optima (Ross, p. 5). This is why Tabu search is an example of a local search algorithm. There is a class of strategies that uses Tabu search called "Ejection Chains", and offers a high quality TSP solutions. When an algorithm is developed that is based from TS (Tabu Search), the travelling salesman problem is usually the best example to test that developed method. The key idea of the process is to search for local optima(s) and continue searching by allowing moves that may give a worse solution or non-improving solution in the neighborhood of the local optima. One of the essential features of Tabu search is to avoid getting stuck in local optima by keeping track of your moves ('moves' simply means subtour reversal or just reverse) and save it in a so called 'Tabu list'. This is also one of the reasons why we are allowed to have worse solution for the next iteration of the process because all it does it to save all the local optimal solutions.

Since we allow bad solutions (assuming it's bad), we may end up with cycles or Hamiltonian path that is already searched, as one moved may counter act the previous solution from previous iteration. To avoid we use Tabu list that will record illegal moves (or 'Tabu moves'). One could see Tabu list as a 'memory' to guide the search procedure by recording the previous searches so that we will not explore that solutions that is already been explored. These Tabu moves (or edges in the context of TSP) from the list could then be used to perform reversal (e.g. the reverse of path or edge 4-5 is 5-4).

A local search procedure behaves like local improvement procedure but local search procedure may not require that each new trial solution is better than the previous (i.e. it can get worse). So Tabu search is usually treated as a subroutine for other heuristic methods for solving combinatorial optimization problem such as TSP (Travelling Salesman Problem). Tabu search is analogous to another heuristic called the Hill Climbing; the process is sometimes called steepest ascent/mildest descent approach because each iteration selects the available move(s) when upward move is not available, select a move that falls down the 'hill' (Hillier & Lieberman, 2015). One could visualize the search process as the following Figure 6:

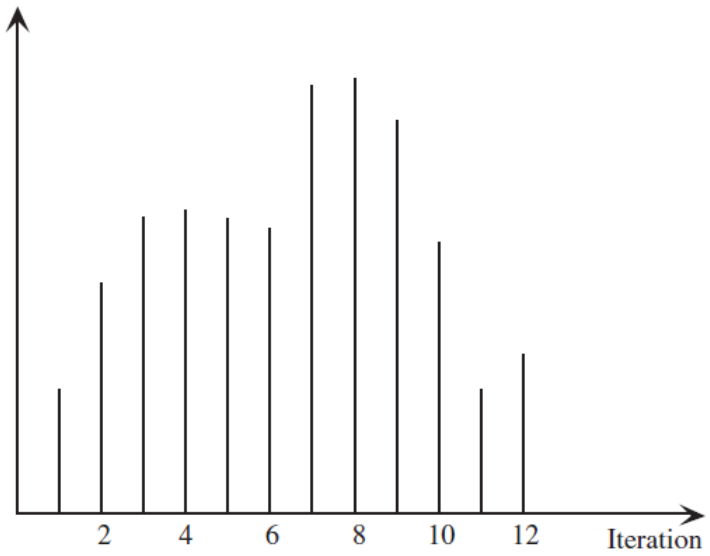


Figure 6: Graph of Objective value v.s. Iteration

Notice in this figure that in each iteration it is possible to get a worse solution. Tabu List is a distinctive feature of TS because this memory-like feature guides the search in such way that it records all the moves that may give a bad solution (Hillier & Lieberman, 2015, p. 625).

Before presenting the algorithm on has to remember the following terminologies

Def(Intensification): To intensify the search meaning searching a portion of the feasible region more thoroughly after it has been identified that the current solution will yield a better solution (Hillier & Lieberman, 2015, p. 625). ◊

Def(Diversification): To diversify the search means that to force the search away from the current solution to explore unexplored parts of the feasible region. ◊

Outline of the Tabu Search Algorithm:

Stage 1: (Initialization)

- Start with a feasible solution. This could be found by implementing other heuristic such as Nearest Neighborhood Heuristic or Nearest Insertion Heuristic. Note that this may (or may not) yield an optimal solution.

Stage 2: (Iteration):

- Use a local search procedure (e.g. Sub tour Reversal) to define the feasible moves into a local neighborhood of the current trial solution.
- Eliminate from consideration any sub tour reversal on the current tabu list, unless would yield a better solution than the trial solution searched so far.
- Determine which edge gives the best solution when reversed (e.g. the reverse of path or edge 4-5 is 5-4). This requires enumeration if there is two or more edges from the previous tabu list. We try to reverse both cases and compare which gives the minimum length.
- The one that has the minimum length shall be the next trial solution.
- If the Tabu list if full, delete the oldest edge and use it for future move (by 'move' means subtour reversal or just reversing).

Stage 3: (Stopping Criterions)

- Finite number of iterations and/or if all possible feasible solutions are already searched.
- Fixed amount of CPU time.

The diversification makes Tabu search perform brute force. If no stopping criterion imposed, the algorithm will get stuck in an infinite loop. Therefore, we need to set a finite number of iterations to prevent this situation. If we would select the finite number of iterations as our stopping criterion, it would be better to give a reasonable and large value for the number of iterations because of diversification. Figure 7 gives the flow chart of the

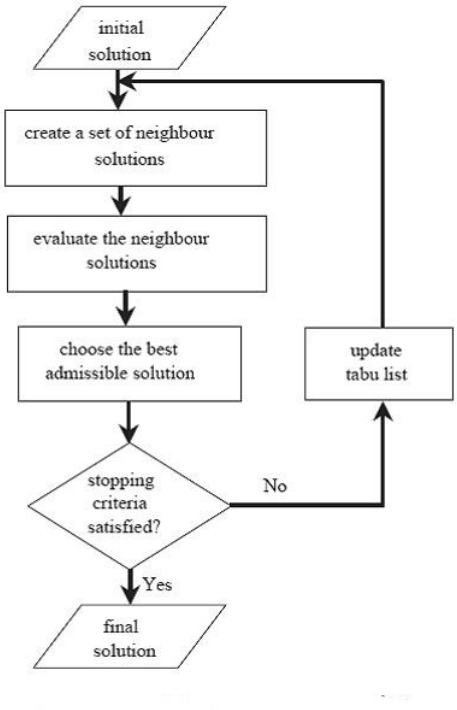


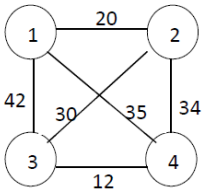
Figure 7: Flow Chart of Tabu-Search

Tabu Search Heuristic (in general, not just for TSP).

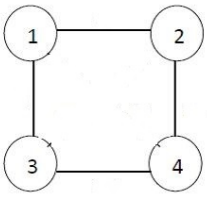
The following example illustrates how the Tabu Search Method works.

Example (Applying Tabu Search Heuristic for solving TSP):

Consider the graph

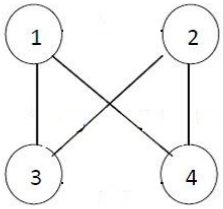


Initialization:
1-2-4-3-1
Length 108
List = Null



Iteration 1:

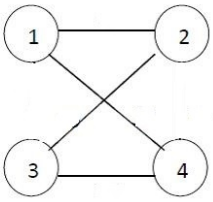
Reverse	2-4
Delete	1-2 , 4-3
Add	1-4 , 2-3
Tabu List	1-4 , 2-3
New solution	1-4-2-3-1
Length	141



Iteration 2:
There are two possibilities: Either reversing 1-4 or 2-3. But for this trivial case, both would give the optimal solution

Case 1: Reversing 2-3

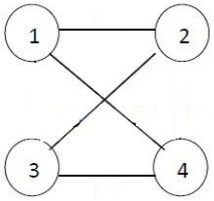
Reverse	2-3
Delete	4-2 , 3-1
Add	4-3 , 2-1
Tabu List	1-4 , 2-3 , 4-3 , 2-1
New solution	1-4-3-2-1



Length	97

Case 2: Reversing 1-4

Reverse	1-4
Delete	1-3 , 2-4
Add	1-2 , 3-4
Tabu List	1-4 , 2-3 , 1-2 , 3-4
New solution	1-4-3-2-1
Length	97



Either Case 1 or Case 2 could be selected for the next iteration's Solution. But for this case, we need to stop searching after iteration 2 since we have exhausted all possible feasible solutions (ie Hamiltonian cycles). Thus the optimal length for this example is 94. For large-scale problems, one has to test all the possible subtour to reverse and select which gives the minimal length. □

4.6 Simulated Annealing

Simulated Annealing (SA) is based on the physical process in chemistry called annealing and just like any other meta-heuristic it enables the search process to escape from local optimum, allowing the search to stretch further. SA is a randomized local search allowing moves with negative gain. In chemistry, annealing involves slowly heating the metal and slowly cooling the substance by varying the temperatures until it reaches the low energy stable state resulting in reduction in energy. The SA approach is to focus mainly on searching the tallest hill, by "tallest" meaning the local optima. Since this hill could be anywhere in the feasible region, the early emphasis of SA is to take steps in random direction so as to explore much of the feasible region. Each iteration of SA search process moves from the current trial solution to its immediate neighborhood. This is similar to Tabu Search method but the difference is how the neighbor is selected.

Idea is to select an appropriate temperature T and decrease its value each iteration by multiplying it by some number, say $\alpha=0.01$. This temperature is given initially and decreases each iteration and compares it with some tolerance ε, thus we usually stop the algorithm if T, at some iteration, is less than the tolerance ε. Initially we give large value for the temperature, say 5. The temperature T acts as the tendency to accept the current candidate solution to be the next trial solution in the next iteration. Note that as $T\rightarrow\infty$, this behaves like a random walk otherwise $T\rightarrow 0$, this behaves like a hill climbing.

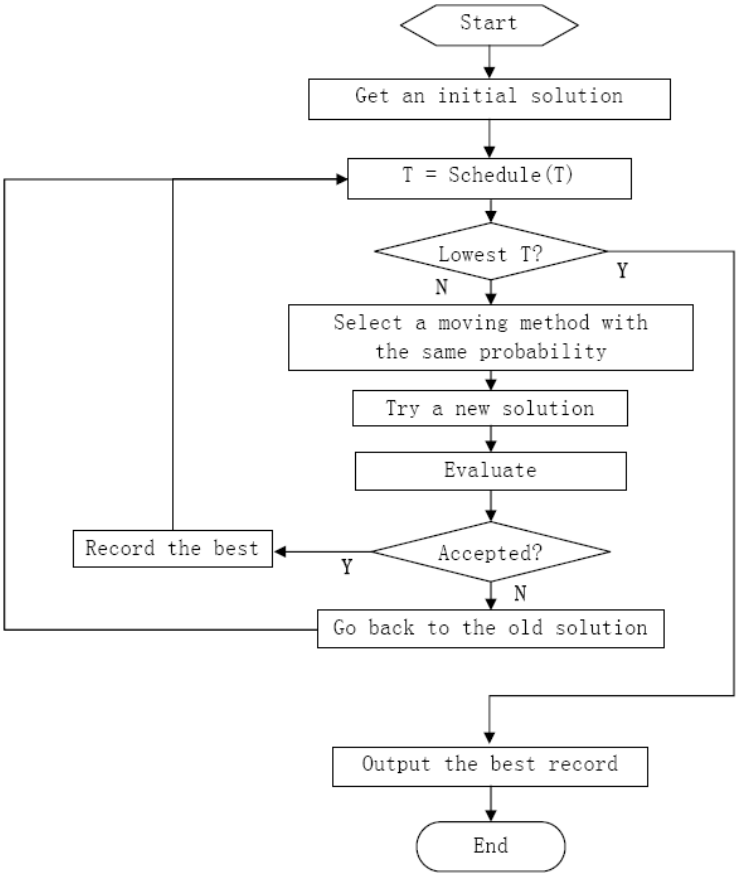


Figure 8: Flow chart of Simulated Annealing

Now let's introduce the notion of acceptance probability. Let C_{old} to be the length/objective value of the current trial solution and C_{new} to be the length/objective value of the candidate solution for the next iteration. Then we define the acceptance probability to be $P_n(C_{old}, C_{new}, T) = e^{-(\frac{C_{new}-C_{old}}{T})}$, n here represent the iteration. This is going to be calculated when $C_{new} \geq C_{old}$ since this inequality indicates that the calculated trial solution C_{new} is getting larger. What we do is to calculate the acceptance probability and then compare it with another randomly generated number from uniform distribution, call it Y. And if $P_n \geq$Y, then we accept C_{new} even though it has worse solution, otherwise go to the next iteration and generate new trial solution. This acceptance probability also acts as the moving rule for the search process. Note that if $C_{new} < C_{old}$ then we do not need to calculate the acceptance probability and just select C_{new} for the next iteration since it gives better objective value/length than the old solution. Note these set up is for minimization problem because we are dealing with TSP, if its maximization then just change the inequalities for decision making.

Simulated Annealing Algorithm:

Step 1: Generate a random initial solution. Use other methods such as Nearest Neighborhood Heuristic. This does not necessarily have to be optimal. Also initialize T $= T_0$.

Step 2: Calculate its length/Objective value.

Step 3: Generate a random neighboring solution. We generate this by randomly selecting a sub-tour from the cycle found and reversing it, obviously this will produce a new Hamiltonian cycle.

Step 4: Calculate the length/objective value of this new cycle, C_{new}.

Step 5: Compare the objective value found from previous steps. If $C_{new} < C_{old}$, then move to the next solution. This means that the algorithm gets closer to the optimal solution (candidate for optimality). It will go to that solution and save it as the basis for the next iteration. If $C_{new} \geq C_{old}$, then calculate the acceptance probability $P_n(C_{old}, C_{new}, T) = e^{-(\frac{C_{new}-C_{old}}{T})}$ and randomly generate a number Y~U[0,1] from uniform distribution and compare with the acceptance probability P_n. If $P_n \geq$Y then accept C_{new}; otherwise do not accept and go to the next iteration.

Step 6: If T$<$ ε, Stop; otherwise T$=\alpha T_0$ and go to step 3.

4.7 Genetic Algorithm

Genetic algorithm (GA) is another example of a metaheuristic method. This metaheuristic is quite different from other metaheuristics such as Tabu Search & Simulated Annealing, where Genetic Algorithm tends to be very effective at exploring various parts of the feasible region and gradually moves towards the best feasible solutions, most of the time they're near optimal solutions. GA is a class of algorithms that is based on the theory in biology called theory of evolution and the phenomenon "survival of the fittest".

The correspondence between GA & TSP:

Population ↔ Initial Feasible solution (This will be updated each Iteration)
Parent ↔ feasible solution
Child ↔ feasible solution found after performing some 'operator'
Fitness ↔ Length/Objective value
Gene ↔ Sub-tour
Generation ↔ Iteration

There will be three operators we will discuss later for generating a 'child': Mutation, Inheriting links, Sub-tour reversal from parents. Initially, we generate a random of feasible solutions (parents) and the number must be even because we will randomly pair then up,

this imitates the phenomenon called “natural selection”. In other words, initially we will have multiple feasible solutions to compare and this feature differs from other metaheuristics such as SA & TS because those metaheuristics starts with only one trial solution each iteration whereas GA starts with a population. Note that the only drawback of this feature is the computational time. After generating this population we will have to evaluate their fitness (objective function). So the population consists of members which as the trial solutions. These trial solutions are now treated as parents and are paired randomly to produce children (new trial solutions). Since the fittest members are more likely to become parents, GA tends to generate improving populations of trial solutions as it proceeds.

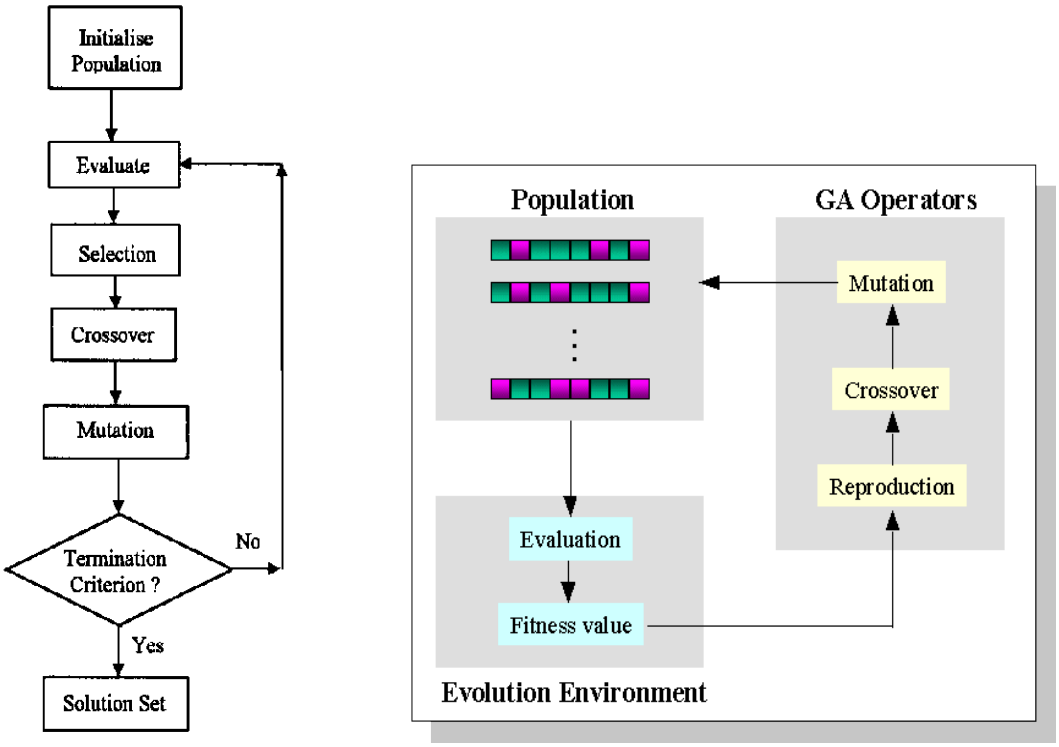


Figure 9: Flow Charts of Genetic Algorithm

Outline of Genetic Algorithm:

Step 1: Choose an initial population of individuals. Most of the time, large number of population will give a better chance of getting a near optimal solution, but the drawback is that it will increase the computational time. Therefore we have to select an appropriate size of the population.

Step 2: Evaluate the fitness of each individual members of that population.

Step 3: Repeat this Generation/Iteration until termination

- Select best fit (minimal length) individuals for mating. Only select the best members and the number of individuals for mating must be even. We're not going to let the least fit members to mate since they would be replaced by the children of the best fit parents' children.
- Breed new individuals (children) through operators such as crossover and mutation. The number of children produced by each pair must be 2 to keep the population number the same.
- Evaluate the fitness of each new members (children)
- Replace the least fit members of the population with the new members.

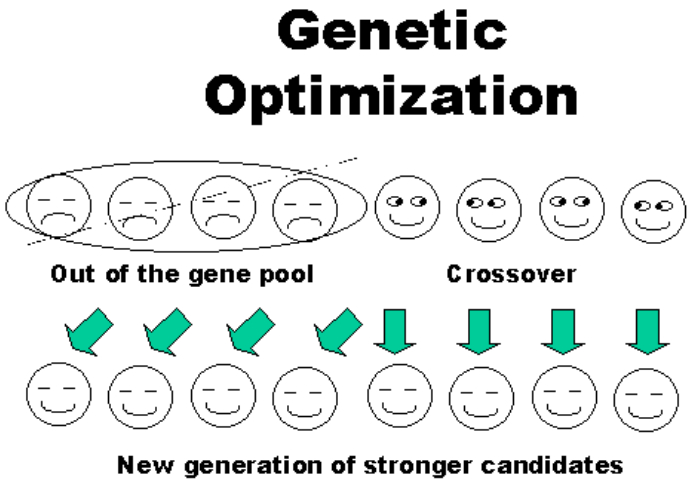


Figure 10: Visual Representation of Selection and Production of better offspring.

Generating a Child (operators):

1. Inheriting links/Crossover (Inheriting direct links from parents)

Each genes (Sub-tour) inherited by a child should come from one parent or the other or both. This genes being inherited are randomly selected one at a time until a complete child (feasible tour/Hamiltonian cycle) has been generated.

2. Sub-tour reversal from a parent (i.e. inheriting a sub-tour reversal)

One possibility of a child inheriting a gene is a link that is a sub-tour reversal from one of the parents e.g. Parent 1-2-3-4-5-6-7-1, we can reverse 3-4 thus 1-2-4-3-5-6-7-1, now gene 3-5 could be added to its child which is a gene that do not appear in both parents gene in order to ensure that the child will not be identical to both of its parents.

3. Mutation of inherited links

Whenever a particular link normally would be inherited, there is a small probability that a mutation would occur that will reject that inherited link and instead randomly select one of the other links from the current city (node) to another city not already on the tour, regardless whether that link appears to either parents.

Procedure for producing a Child:

Step 1: Initialization.

Step 2: Option for the next link.

Step 3: Selection of the next link. Assign a random variable for each link each follows a uniform distribution Uniform [0, 1].

Step 4: Check for mutation. Compare the selected 'next' link with mutation rate (a parameter), say 0.1. And mutation occurs if the probability of selected link is less than 0.1.

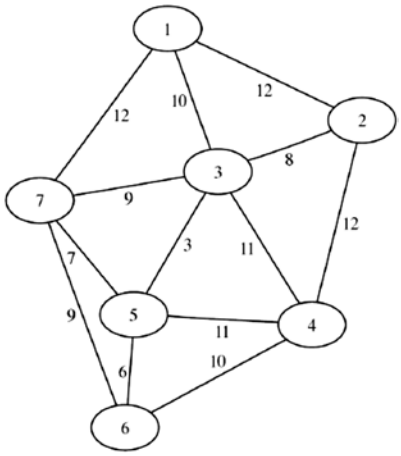
Step 5: Continuation. If there is a dead-end or there is a link that is not included in the tour then go back to step 2.

Step 6: Completion. If there is a miscarriage, go back to step 1 with new assigned probabilities.

This procedure could be treated as a subroutine when implementing in any programming language. Example could be: we have links with assigned probabilities: 1-2 (0.03), 1-3 (0.02), 1-7 (0.01). The highest probability in this case is 0.3 which corresponds to link 1-2, so this link will be inherited by a child but mutation would occur since $0.03<0.1$ (assuming the mutation rate is 0.1), thus we reject link 1-2 and instead, we randomly select either 1-3 or 1-7 without comparing their probabilities with the mutation rate.

Example (Generating a Child):

Consider the graph



Parent 1: 1-2-4-6-5-3-7-1
Parent 2: 1-7-6-4-5-3-2-1

Link	Options	Random Selection of link	Tour
1	1-2, 1-7	1-2	1-2
2	2-3, 2-4	2-4	1-2-4
3	4-6, 4-5	4-5	1-2-4-5
4	5-6, 5-3	5-6	1-2-4-5-6
5	6-7	6-7	1-2-4-5-6-7
6	7-3	7-3	1-2-4-5-6-7-3
7	3-1	3-1	1-2-4-5-6-7-3-1

In link 1, we assign random number to each option 1-2 & 1-7. Note that in this example, we treat node 1 as the home city. But since gene 1-2 appear to both parents, then the probability of this one should be high, therefore there is a chance that 1-2 is going to the selected gene. But let's assume the computer program selected this one. In this situation the crossover operator was used. Let us also assume that mutation did not occur, meaning the probabilities assigned to the options are greater than the mutation rate, say 0.1. In link 2, we cannot include 1-2 or 2-1 as part of the option since 1-2 or 2-1 already appears in the tour. Thus our only options are 2-3 & 2-4 with some assigned probabilities. Let's assume the program chooses 2-4 and no occurrence of mutation. Same thing may happen for the rest of the link. But let's discuss a more interesting situation in the last link, link 7. Notice that option in link 7 is 3-1 but this gene is not part of the parents' gene. For this situation it may be possible that the mutation occurred and the computer program tries to find a sub-tour to reverse from one of the parents. In this case the sub-tour (gene) that has been reversed is 3-7 from parent 1 which would result 1-2-4-6-5-7-3-1 and the program would select 3-1 since this is a valid sub-tour (by valid means it would result in a Hamiltonian cycle). With this gene added, the child of the parents would be 1-2-4-5-6-7-3-1. Note that in some cases it is possible for a miscarriage to happen (i.e. infeasible solution), if that's the

case then we would iterate the procedure for producing a child and only stop when we have generated a child that is feasible (i.e. Hamiltonian cycle). □

Example (Dead-end or Miscarriage situation):

Parent 1: 1-2-3-4-5-6-7-1
Parent 2: 1-2-4-6-5-7-3-1

Link	Options	Random Selection of link	Tour
1	1-2, 1-7, 1-3	1-2	1-2
2	2-3, 2-4	2-4	1-2-4
3	4-3, 4-5, 4-6	4-5	1-2-4-5
4	5-6, 5-7	5-7	1-2-4-5-7
5	7-6, 7-3	7-6	1-2-4-5-7-6
6			
7			

Notice that in link 5, there is a problem. The path 1-2-4-5-7-6 blocks node 3 which would result in a non-Hamiltonian cycle. Therefore, miscarriage occurred. If this situation happened the algorithm should go back to the parents and then assign new probabilities for the options and do the same process again until a feasible child is produced. □

5 Experimental Study

In this section I will present the result(s) of my simulation experiment study. After I present my conclusion, I will also try to compare my solution to other comparative studies in the literature. In this simulation experiment I will attempt to answer the following questions:

- Which heuristic method performs the best? i.e. Which gives the minimal tour length (by average)
- Which heuristic method performs the worst? i.e. Which gives the maximal tour length (by average).
- Are the insertion heuristics significantly different from each other? i.e. Do they give the same tour length or some of them are significantly different from other insertion heuristics.

- Which heuristics performs the best when the size of the distance matrix is small (e.g. 4-by-4 distance matrix)?
- Which heuristics performs the worst when the size of the distance matrix is small (e.g. 4-by-4 distance matrix)?
- Which heuristics performs the best when the size of the distance matrix is large (e.g. 15-by-15 distance matrix)?
- Which heuristics performs the worst when the size of the distance matrix is large (e.g. 15-by-15 distance matrix)?

In this simulation experiment I used the software R which is statistical software. But the main reason(s) I choose R is because its powerful software for manipulating & analyzing large datasets and it has a downloadable package called "TSP" which contains many functions, including many heuristics for solving TSP. The available heuristics that I will compare are the insertion heuristics (farthest insertion, cheapest insertion, arbitrary insertion, nearest insertion), Nearest Neighborhood Heuristic, and 2-Opt.

This simulation experiment has two factors with levels: Heuristic Methods (Levels: NN, 2-Opt, NI, FI, CI, AI) and Distance matrix size (Levels: 4x4, 5x5, 6x6, …,15x15). The response variable needs further explanation about how we will generate the random data.

Since this is a "simulation" experiment, the random variable of interest is the entries of the distance matrix. For simplicity, I will only restrict this simulation to symmetric TSP (i.e. symmetric TSP distance matrix). Hence the distance matrix must be square, symmetric, and the entries must be positive real numbers. Again, for simplicity I will generate natural numbers for the distance matrix and for that I burrowed geometric distribution with probability parameter of 0.6. The diagonal entries of the distance matrix must be zero (since we do not allow loops in our TSP graph).The examples below shows a sample of this random distance matrix generated in R (5x5 matrices):

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0 1272 1849 1390  385
[2,] 1272    0 1061  532  785
[3,] 1849 1061    0 3950 1994
[4,] 1390  532 3950    0  457
[5,]  385  785 1994  457    0
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0  333  799  426  551
[2,]  333    0  603 1372 4318
[3,]  799  603    0  525  976
[4,]  426 1372  525    0 2251
[5,]  551 4318  976 2251    0
```

The tables (Table 1*), (Table 2*), and (Table 3*) below show how we will perform the simulation. The table(s) means that for each iteration (i=1,…,N) we will generate random

D_{nxn} distance matrix. N here represent the number of trials, for this I chose N=1000 (i.e. we will generate 1000 random distance matrices).We will do this for each matrix sizes. My choice are n=4,5,…,15 distance matrix sizes. Theoretically, you can do simulation with a very large distance matrix size (e.g. 200x200) but large computational time will be spend for each methods. Hence, for each distance matrix size n, we will generate 1000 trials (i.e. 1000 different random distance matrices) and for each of these trials we will record the feasible solutions/lengths calculated by each heuristic methods. As a remark, these recorded lengths are 'not' the response variables; instead, for each of these heuristic methods in each matrix sizes n, we will calculate its "means". Let's just simply call these 'means' as "Responses". Denoted by $m_k^{(n)}$ for n=4,5,…,15 and for each method k=NI,CI,FI,AI,NN,2-Opt are the "Responses". In R, we will save these "Responses" as a concatenated array in (Table 4*).

(Table 1*)

i	NI	CI	FI	AI	NN	2-Opt	
1							D_{4x4}
2							D_{4x4}
…							…
N							D_{4x4}
	⇓	⇓	⇓	⇓	⇓	⇓	
	$m_1^{(4)}$	$m_2^{(4)}$	$m_3^{(4)}$	$m_4^{(4)}$	$m_5^{(4)}$	$m_6^{(4)}$	

(Table 2*)

i	NI	CI	FI	AI	NN	2-Opt	
1							D_{5x5}
2							D_{5x5}
…							…
N							D_{5x5}
	⇓	⇓	⇓	⇓	⇓	⇓	
	$m_1^{(5)}$	$m_2^{(5)}$	$m_3^{(5)}$	$m_4^{(5)}$	$m_5^{(5)}$	$m_6^{(5)}$	

(Table 3*)

i	NI	CI	FI	AI	NN	2-Opt	
1							D_{15x15}
2							D_{15x15}
…							…
N							D_{15x15}
	⇓	⇓	⇓	⇓	⇓	⇓	
	$m_1^{(15)}$	$m_2^{(15)}$	$m_3^{(15)}$	$m_4^{(15)}$	$m_5^{(15)}$	$m_6^{(15)}$	

(Table 4*)

	NI	CI	FI	AI	NN	2-Opt
4x4	$m_1^{(4)}$	$m_2^{(4)}$	$m_3^{(4)}$	$m_4^{(4)}$	$m_5^{(4)}$	$m_6^{(4)}$
5x5	$m_1^{(5)}$	$m_2^{(5)}$	$m_3^{(5)}$	$m_4^{(5)}$	$m_5^{(5)}$	$m_6^{(5)}$
…	…	…	…	…	…	…
15x15	$m_1^{(15)}$	$m_2^{(15)}$	$m_3^{(15)}$	$m_4^{(15)}$	$m_5^{(15)}$	$m_6^{(15)}$

Once we have recorded the "Responses", we will stack them in (Table 4*). I have replicated the experiment 10 times to get more empirical evidence (i.e. table will be replicated 10 times). Table 1 show one sample out of 10 replications of my experiment.

	NI	CI	FI	AI	NN	2-Opt
4	3451.948	3451.948	3451.948	3451.948	3721.884	3592.69
5	5044.759	5039.928	5041.662	5040.323	5445.193	5221.846
6	7031.913	7024.832	7065.254	7009.805	7676.699	7154.105
7	9455.256	9441.776	9500.122	9452.315	10261.417	9549.897
8	12130.701	12090.314	12192.617	12112.733	13131.579	12100.61
9	15417.951	15381.862	15513.373	15371.675	16650.347	15331.709
10	18640.122	18576.977	18778.637	18617.541	20140.733	18333.973
11	22565.65	22516.271	22728.539	22545.125	24198.992	22148.998
12	26995.896	26936.998	27127.713	26935.97	28874.559	26259.18
13	31537.957	31354.023	31693.613	31404.42	33506.263	30587.323
14	36718.056	36551.889	36916.926	36600.203	39009.346	35570.35
15	42131.547	41948.179	42394.467	42043.832	44696.007	40766.207

Table 1: A replication of the result of the experiment

The Figure 11 compares the responses for each distance matrix size of each heuristic method. From this Figure 11 we can make intuitive conclusions about the heuristic methods. Observe that for all the distance matrix sizes the 'worst' performer is the Nearest Neighborhood heuristic. It is clearly visible from this graph. Without even using any statistical methods we can predict that the feasible solution found by Nearest Neighborhood heuristic will be significantly different from other heuristic methods (by average). Table 2 gives some basic summary of the outcome. Table 2 provides evidence to

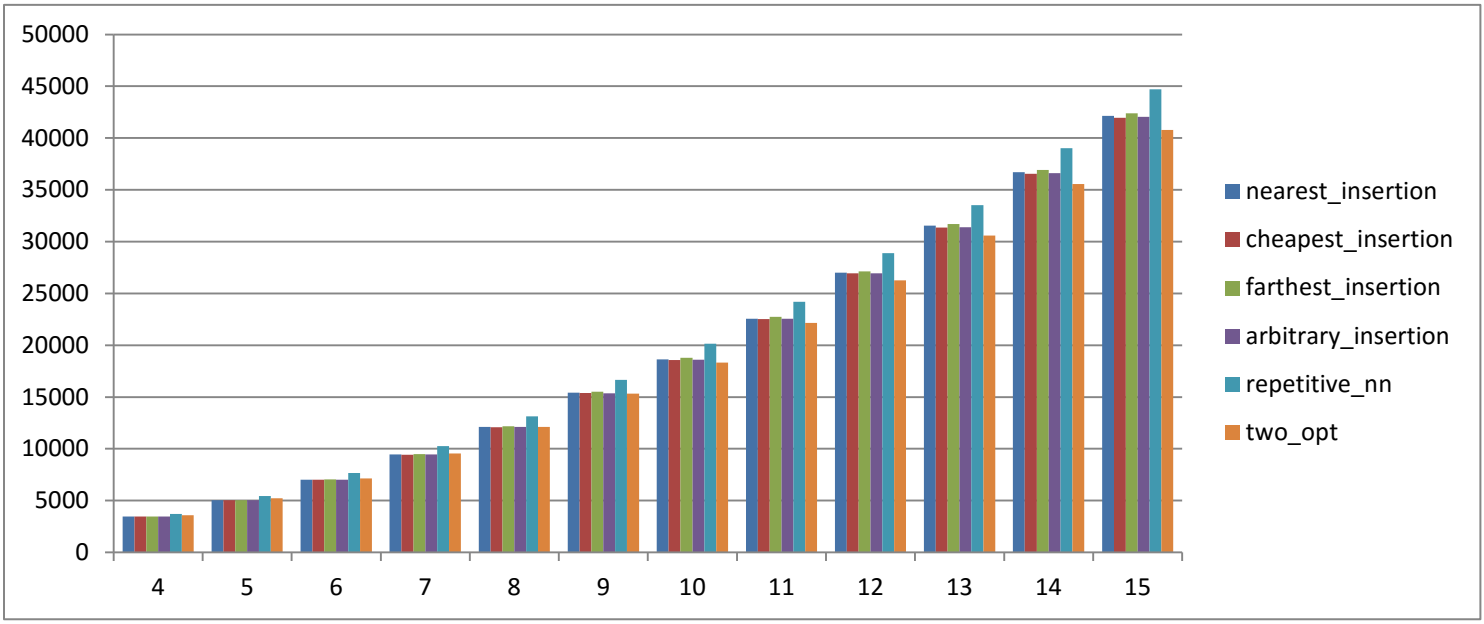


Figure 11

show that Nearest Neighborhood heuristic method is the 'worst' performer. The main reason why Nearest Neighborhood does not give an accurate feasible solution is because Nearest Neighborhood is a greedy heuristic, meaning that once it found an optimal solution it will never try to improve it.

	Min	Method	Max	Method
4	3451.948	nearest_insertion	3721.884	repetitive_nn
5	5039.928	cheapest_insertion	5445.193	repetitive_nn
6	7009.805	arbitrary_insertion	7676.699	repetitive_nn
7	9441.776	cheapest_insertion	10261.417	repetitive_nn
8	12090.314	cheapest_insertion	13131.579	repetitive_nn
9	15331.709	two_opt	16650.347	repetitive_nn
10	18333.973	two_opt	20140.733	repetitive_nn
11	22148.998	two_opt	24198.992	repetitive_nn
12	26259.18	two_opt	28874.559	repetitive_nn
13	30587.323	two_opt	33506.263	repetitive_nn

14	35570.35	two_opt	39009.346	repetitive_nn
15	40766.207	two_opt	44696.007	repetitive_nn

Table 2

Notice in Table 2, starting at 9x9 matrix size 2-Opt always gives the minimum feasible solutions (by average). In fact I also observe this for the other replications of the experiment. We can conclude that the benchmark for 2-Opt is n=9. And 2-Opt does not seem to give the best solutions when the size of matrix is relatively small (e.g. n=4,5,6). We

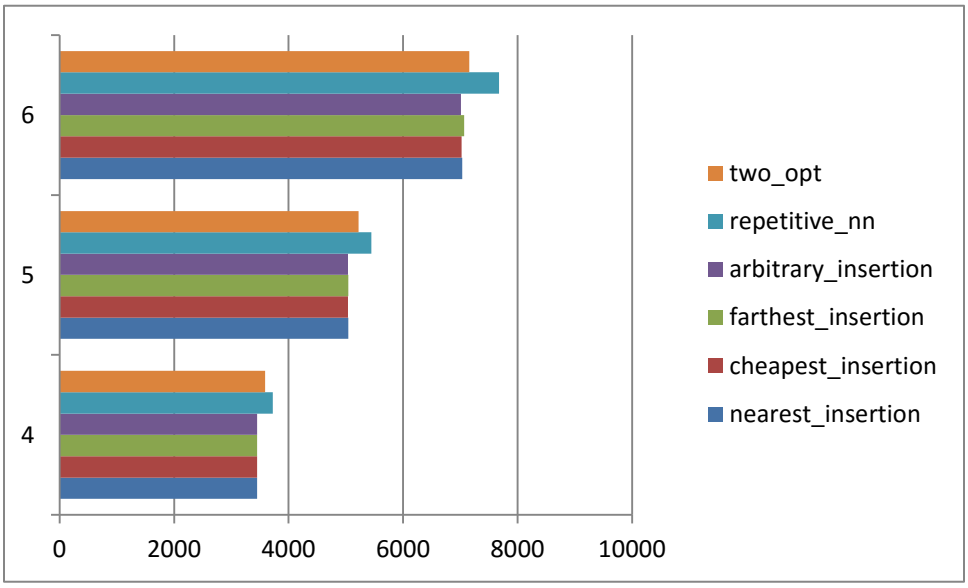


Figure 12: Restricting only to 4x4, 5x5, and 6x6 matrices

can conclude that for large distance matrices ($n \geq 9$), 2-Opt provides the most accurate feasible solution (by average). This should not come as a surprise since 2-Opt belong to the category of TSP heuristics called "Tour Improvement". Meaning that initially it calls a constructive heuristic such as Nearest Neighborhood to give an initial feasible solution and from that 2-Opt will try to improve it until there's not much to improve. Figure 12 shows that for small distance matrix sizes (n=4,5,6), 2-Opt is not the best compare to other heuristics (mostly insertion heuristics).

Now we know that the 'worst' heuristic is the Nearest Neighborhood and the 'best' heuristic (for $n \geq 9$) is 2-Opt. Now let's analyze whether or not the insertion heuristics are

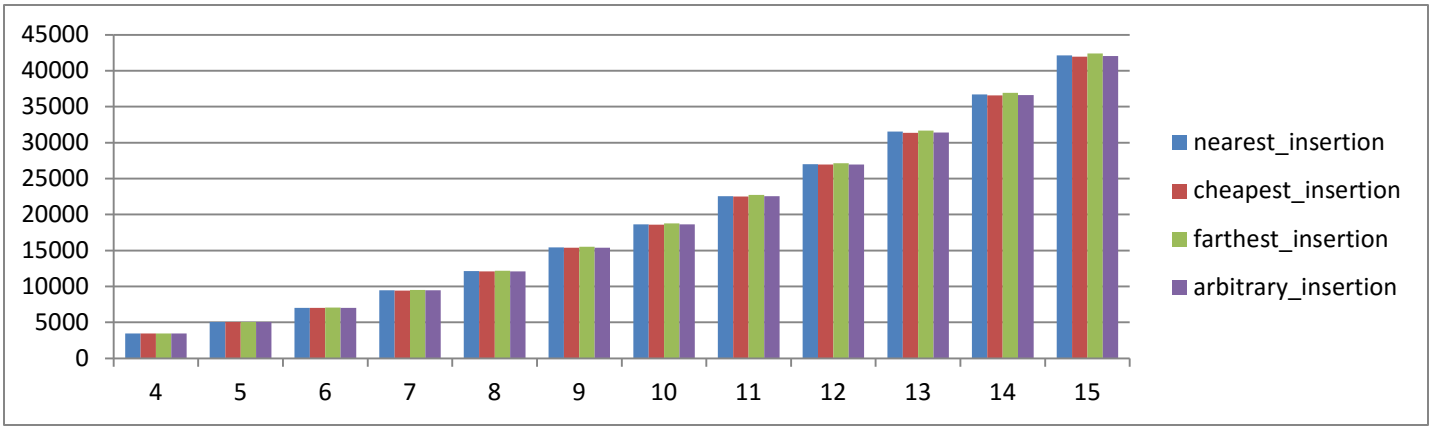


Figure 13

significantly different from each other. Figure 13 compares the responses for each distance matrix sizes of each insertion heuristics. Looking at Figure 13, it is hard to make an intuitive prediction as to, say, which insertion heuristic is the 'best' & 'worst'. To analyze this, I have made Table 3.

	Min	Method	Max	Method
4	3451.948	nearest_insertion	3451.948	nearest_insertion
5	5039.928	cheapest_insertion	5044.759	nearest_insertion
6	7009.805	arbitrary_insertion	7065.254	farthest_insertion
7	9441.776	cheapest_insertion	9500.122	farthest_insertion
8	12090.314	cheapest_insertion	12192.617	farthest_insertion
9	15371.675	arbitrary_insertion	15513.373	farthest_insertion
10	18576.977	cheapest_insertion	18778.637	farthest_insertion
11	22516.271	cheapest_insertion	22728.539	farthest_insertion
12	26935.97	arbitrary_insertion	27127.713	farthest_insertion
13	31354.023	cheapest_insertion	31693.613	farthest_insertion
14	36551.889	cheapest_insertion	36916.926	farthest_insertion
15	41948.179	cheapest_insertion	42394.467	farthest_insertion

Table 3

For a 4x4 (or n=4) distance matrix all insertion heuristics would give the same average (also see Table 1). But as the size of the distance matrix gets larger arbitrary and cheapest insertion heuristics gives smaller responses compare to other insertion heuristics. Also observe that the method that gives the maximum response is the farthest insertion heuristic. From this 'intuitive' approach of analyzing whether or not they give different responses, we cannot immediately conclude that Farthest Insertion is the 'worst' and Cheapest Insertion is the 'Best'. For this one should use MANOVA (multivariate analysis of

variances) to make this kind of conclusion. Looking at the original Table 1, one can see that the responses are not really significantly different from each insertion heuristics.

6 Conclusions

TSP is a popular combinatorial optimization problem since it is very easy to understand and difficult to find a good algorithm for solving it.

We discussed the integer formulation of the travelling salesman problem and the two ways of formulating the constraint for sub-tour elimination. But the MTZ formulation will give less number of constraints compare to the constraint formulation by Dantzig.

This report also surveys some of the special cases or variations of TSP such as symmetric TSP, asymmetric TSP, metric TSP, and Euclidean TSP.

This report surveys some of the current and standard heuristics & metaheuristics in the literature. Heuristic Methods has two main types. One type is called "Tour construction" heuristic where it starts in an arbitrary city/node and from that it will construct the TSP tour/feasible solution/Hamiltonian Cycle. Most Tour Construction Heuristics are greedy algorithms meaning that the algorithm takes the edge with minimum weight/length to the TSP tour being constructed, and thus does it does not try to improve the feasible solution found. The other type is "Tour Improvement" heuristic where is starts from an initial feasible solution founded by a tour construction heuristic, and starting from this initial feasible solution it will try to improve it until there is not much improve. Most of the metaheuristics such as simulated annealing, Tabu-Search, and Genetic Algorithm falls in the category of "Tour Improvement" Heuristics. Generally, Tour Improvement heuristics yield a more accurate feasible solution than a feasible solution found by Tour Construction heuristic.

I also conducted a simulation experiment using the statistical software called R and a downloadable package called 'TSP'. This experiment only compares the 'accuracy' of the methods not the computational time. I restricted the experiment to symmetric TSP. It turns out that Nearest Neighborhood heuristic gives less accurate feasible solution compare to other heuristics for all possible sizes of the distance matrix n. The best solver is 2-Opt, although this is not a surprise since 2-Opt is a Tour Improvement heuristic and the benchmark is $n \geq 9$. The solutions found by Insertion Heuristics are not significantly different although further analysis is required.

References

AlSalibi, B. A., Jelodar, M., & Venkat, I. (2013). A Comparative Study between the Nearest Neighbor and Genetic Algorithms: A revisit to the Traveling Salesman Problem. *International Journal of Computer Science and Electronics Engineering, 1*(1).

Balakrishnan, R., & Ranganathan, K. (2012). *A Textbook of Graph Theory* (2 ed.). Tiruchirappalli: Springer.

CHEN, D.-S., BATSON, R. G., & DANG, Y. (2010). *Applied Integer Programming: Modeling and Solution.* Hoboken: John Wiley & Sons, Inc.

DeLeon, M. (n.d.). *A Study of Sufficient Conditions for Hamiltonian Cycles.* Seton Hall University, Department of Mathematics and Computer Science.

Hillier, F. S., & Lieberman, G. J. (2015). *Introduction to Operations Research.* New York, United States of America: McGraw-Hill Education,.

Karkory, F. A., & Abudalmola, A. A. (2013). Implementation of Heuristics for Solving Travelling Salesman Problem Using Nearest Neighbour and Minimum Spanning Tree Algorithms. *International Journal of Mathematical, Computational, Statistical, Natural and Physical Engineering, 7*.

Nilsson, C. (n.d.). *Heuristics for the Traveling Salesman Problem.* Linkoping University.

Pataki, G. (2000). *The bad and the good-and-ugly: formulations for the traveling salesman problem.* Columbia University, Department of Industrial Engineering/Operations Research.

Ross, K. (n.d.). Metaheuristics.

SAHALOT, A., & SHRIMALI, S. (2014). A COMPARATIVE STUDY OF BRUTE FORCE METHOD, NEAREST NEIGHBOUR AND GREEDY ALGORITHMS TO SOLVE THE TRAVELLING SALESMAN PROBLEM. *International Journal of Research in Engineering & Technology, 2*(6).