Connor Raymond Stewart
101041125

MATH 3802 Project

Introduction

This project's topic assignment is topic K. Topic K is a write-up topic on the K-Opt algorithm for the travelling salesman problem (TSP). As per the topic's specifications, for computation experiments, only 2-Opt will be implemented. Test cases will be in the form of a symmetric matrix. The project includes test cases of up to one thousand cities for the computational experiments on the 2-Opt algorithm. For this project, the 2-Opt algorithm is implemented and tested with Python.

Background

In algorithmic theory, classes of problems are challenging to solve precisely within a reasonable timeframe. In computational complexity theory, researchers classify algorithmic problems by difficulty by determining if their outputs differ by at least one input (Arora & Barak, 2010, p. 68). One such problem is called the Travelling salesman problem (TSP). The TSP problem is a notoriously tricky problem wherein a list of nodes (or cities) needs to all be connected and returned to the origin city with the shortest route possible (Arora & Barak, 2010, p. 61).

The TSP problem exists in a specific class of computationally tricky problems called the NP-complete problems (Arora & Barak, 2010, p. 38). To understand this, we must explore a specific set of computational complexity classes known as the P, NP, and NP-hard classes. Computational problems in P are problems that a deterministic Turing machine can solve using a polynomial time complexity (Arora & Barak, 2010, p. 365). NP problems are a complex class of provable problems in polynomial time by a deterministic Turing machine (Arora & Barak, 2010, p. 41). NP-hard problems are a complex class of problems that are as hard to solve as the most complex problems in NP (Arora & Barak, 2010, p. 43). Finally, we can summarize the NP-complete complexity class as the set of problems that can be solved (and verified) by brute-force searches and can simulate all other NP problems (Arora & Barak, 2010, p. 57). As it is described above, it becomes clear that the NP-complete problems are an intersection between the NP and NP-hard complexity classes.

There exist numerous heuristic-based strategies for solving the TSP in polynomial time with near-Optimal solutions. These methods can be classified into various categories, such as the tour construction, the tour improvement, the ant colony optimization, and the Held-Kerp lower bound methods (Ma *et al*., 2016, 6537). Included within the tour improvement framework are a set of methods are, namely the Lin-Kernighan, 2-Opt, 3-Opt, tabu search approaches, and simulated annealing and genetic algorithm-based methods (Ma *et al*., 2016, 6537). The k-Opt method belongs to the tour improvement category, meaning it contains a set of operations used to convert one tour to another (Ma *et al*., 2016, 6537).

The paper contains the Pythonic implementation of the 2-Opt algorithm, a simple iterative algorithm where each iteration of the algorithmic loop results in two edges from the tour getting replaced with two new edges such that the tour remains valid and now has a lower cost (Ma *et al*., 2016, 6537). As an analogy, a higher complexity model for the k-Opt algorithm would be the 3-Opt method, which is like the 2-Opt method. The primary changes the 3-Opt method has from the 2-Opt method is that three edges get changed in each iteration of the algorithmic loop (Ma *et al*., 2016, 6537). Naturally, the number of edges replaced in the 3-Opt method increases the time complexity from the 2-Opt method due to the algorithm having more permutations it can iterate through. The TSP can be represented by the following linear programming formulation (Ma *et al*., 2016, 6537):

Equation 1: Linear Programming Formulation for the TSP

$$\begin{aligned}
&\min \sum_{i=1}^{n} \sum_{j\neq i, j=1}^{n} c_{ij} x_{ij}: && \\
&x_{ij} \in \{0,1\} && i,j = 1,\ldots,n; \\
&u_i \in \mathbf{Z} && i = 2,\ldots,n; \\
&\sum_{i=1, i\neq j}^{n} x_{ij} = 1 && j = 1,\ldots,n; \\
&\sum_{j=1, j\neq i}^{n} x_{ij} = 1 && i = 1,\ldots,n; \\
&u_i - u_j + n x_{ij} \leq n-1 && 2 \leq i \neq j \leq n; \\
&1 \leq u_i \leq n-1 && 2 \leq i \leq n.
\end{aligned}$$

Where cities are labelled with numbers one up to n, and we define x_{ij} as (Ma *et al*., 2016, 6537):

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

Problem Motivation

The primary motivation for the K-Opt algorithm is to solve the TSP in a reasonable amount of time. The TSP has many applications in real-world problems in domains such as network information trafficking, manufacturing, and search optimization. Since the TSP problem shows itself in many domains in the real world, having a way of finding reasonable solutions to the TSP is essential. If many nodes are within a graph, the TSP problem may take very long to solve computationally, and many applications like air-traffic control, robotic manufacturing, and search optimization require fast results from a computer. Applications for the TSP can include the drilling of printed circuit boards with the minimum number of drill movements, computer wiring with the minimum number of wires needed, X-ray crystallography pattern optimizations, and optimal job sequencing

The K-Opt algorithm for the TSP provides solutions for the travelling salesman problem within a reasonable timeframe. The K-Opt algorithm allows obtaining accurate solutions for TSP-type problems in a short timeframe. The TSP problem is NP-complete, meaning that to solve the TSP with a large set of cities quickly, we would need to use a specially designed method. One of the most successful techniques for obtaining accurate TSP solutions is the local search method (Korte & Vygen, 2012, p. 569). The local search method is a heuristic-based method used for solving complex optimization problems (Korte & Vygen, 2012, p. 569). Local search problems are - in essence - an algorithmic principle that can solve problems convertible to a maximization problem (Korte & Vygen, 2012, p. 569). Widespread usage of the local search method is the K-Opt algorithm discussed earlier.

The K-Opt algorithm being a tour improvement algorithm, works by taking an initially suboptimal tour and repeatedly attempts to lower the tour's cost until no improvements can be made (Ma *et al*., 2016, 6537). The k-Opt method is the most popular heuristic-based method for the TSP and works by setting a parameter k representing the number of edge exchanges/re-permutation within specific neighborhoods (Korte & Vygen, 2012, p. 569). The most popular values for the k parameter within the k-Opt method are values two and three because these values are easy to implement and allow the K-Opt algorithm to terminate at any point in the execution of the algorithm (Ma *et al*., 2016, 6537). The time complexity of the K-Opt algorithm increases exponentially with the value of k (specifically, O(n*k) for n many nodes in a graph), meaning that although the accuracy can improve in higher k-models, there is a sizeable computational tradeoff (Korte & Vygen, 2012, p. 570). Thus, the K-Opt algorithm is helpful as it provides an algorithm with an adjustable time complexity based on how high the precision and accuracy of the solution needs to be.

Examples with Illustrations and Diagrams

The TSPs complexity

An initial example will examine the complexity of the TSP before discussing the methods behind the K-Opt algorithm. Since the travelling salesman problem requires finding the shortest possible loop to connect every node in a graph, we can see the following graph can be complex to solve. If we are using Euclidian distance between nodes in a cartesian plain to represent the weight values for the edges between the nodes, then we have (n choose 2) = ½(n)(n-1) edges in the graph. The number of edges is because the edges of the graph are defined as the Euclidian distance between the points in the graph, meaning the final graph resulting from the points on the cartesian plane must be a clique graph. In a graph with many points, we will end up with an even more significant number of edges that can make brute force difficult. Take the example of a cartesian plane with 12400000 points (the number of houses in Canada), with distances representing the distance between each home in Canada projected onto a cartesian plain (Statistics Canada, 2018). Since 12400000 points would result in ½(12 400 000)(12 400 000 – 1) = 76 879 993 800 000 edges, it can take quite a while to find a solution to this specific TSP. Finding an exact solution to the TSP for this dataset would result in high time complexity. Using a recursive method of trying all possible permutations would result in time complexity of O(n!), whereas dynamic programming algorithms for an exact solution to the TSP result in time complexity of $O(n^2*2^n)$ (Datta, 2020). As n represents the number of cities, we can see that 12400000! It is an excessive number of permutations, and that 124000002*212400000 is better, but not very helpful in terms of being reasonable. As discussed earlier, the k-Opt algorithm has a time complexity of O(n*k), meaning or a 2-Opt problem with the above dataset, we have a big-oh of O(124000002) = 153760000000000. So, we can see that the 2-Opt algorithm can be a manageable solution for the above TSP when speed is a concern.

The 2-Opt algorithm

The 2-Opt problem is the most straightforward k-Opt algorithm, and it only requires exchanging two edges per iteration of the algorithmic loop (Ma *et al*., 2016, 6537). Therefore, we end up producing the following 2-Opt diagram in a cartesian plane, with the Euclidian distance between each point representing the edges between each node:

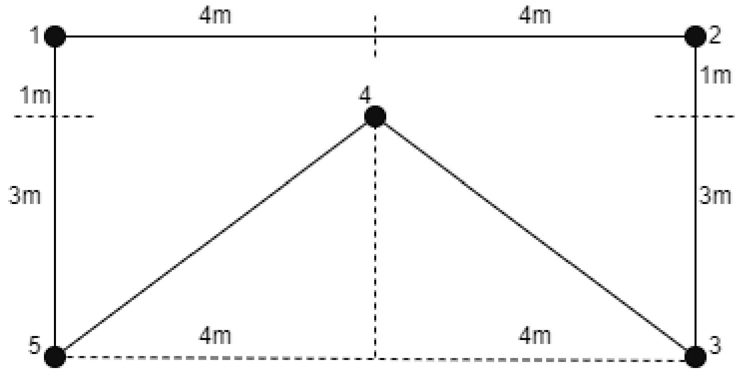

Figure 1: 2-Opt Optimization Diagram

Note that the above diagram - despite being 2-Opt - is not 3-Opt. A 3-Opt would require making a trip between points 1, 4, 2, 3, 5, 1 in the above diagram.

Another simplified example of the 2-Opt is as follows:

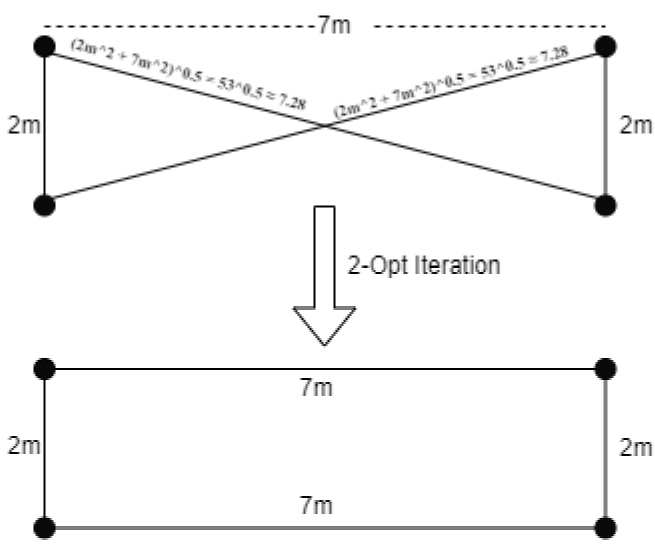

Figure 2: 2-Opt Optimization after Iteration

As can be seen, the 2-Opt algorithm shortens the trip length by deleting the previous edges – the ones connecting the dots in quadrants one and three and quadrants two and four together – and replacing them with edges connecting the dots into the rectangular image above. We can formulate what happened above by defining the two edges as (x1, x3) and (x2, x4), where the elements of xi are distinct elements with the element-i representing the quadrant that the dot is in in the corresponding diagram. We can replace these edges with the edges formulated as (x1, x2) and (x3, x4) since the edges' replacement reduced the length of the tour. We can represent the final cost for the 2-Opt trip as:

Table 1: 2-Opt Cost Update per Iteration

$\Delta C = c(x_1, x_3) + c(x_2, x_4) - [c(x_1, x_2) + c(x_3, x_4)]$ $C \rightarrow C + \Delta C$ Where: C represents the current cost of the tour ΔC represents the change of the cost of the tour after a 2-Opt iteration changes a pair of edges in the tour $c(x_i, x_j)$ represents the length of the edge in the tour

The 3-Opt algorithm

The 3-Opt algorithm works by removing three edges and replacing them in the graph. Looking at the previous graph we created for the 2-Opt algorithm above, we can see the following:

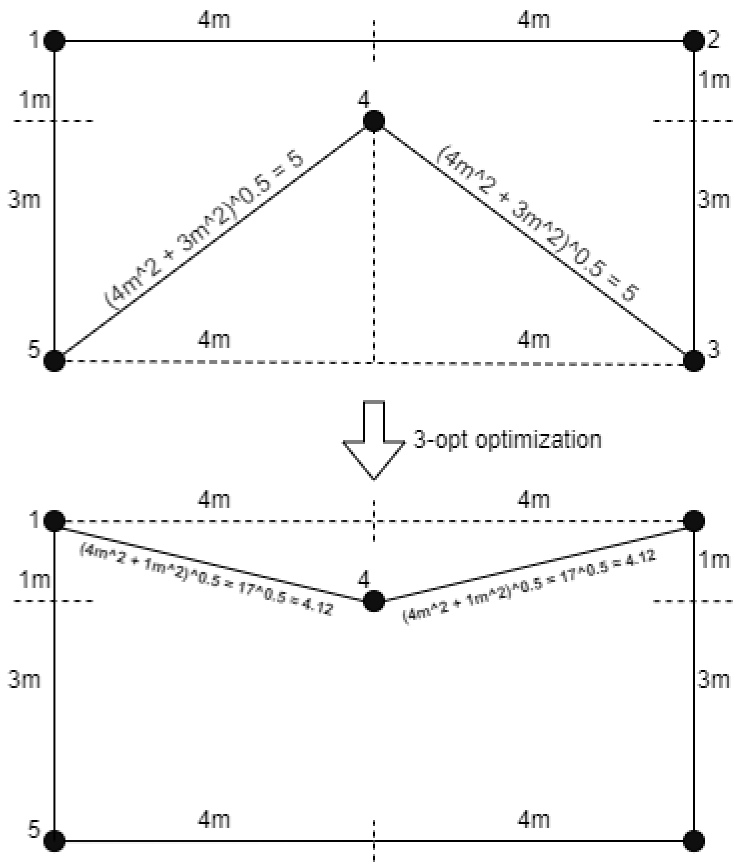

Figure 3: 3-Opt Optimization after Iteration

As can be seen, this shortens the path when compared to the previous diagram. Previously, the 2-Opt placed the tour cost at 26m and the 3-Opt path on the exact diagram costs (16+2(17)0.5)m ≈ 24.2462m which shows a reduction in the total distance tour.

A second example from the course textbook *Combinatorial Optimization: Theory and Algorithms* by Korte & Vygen (2012) is excised in this paper as well. The diagram on the right-hand side is 3-Opt concerning the diagram on the left-hand side (Korte & Vygen, 2012, p. 570). The diagram on the left-hand side has its weights listed as numerical values along the edges connecting the nodes (Korte & Vygen, 2012, p. 570). All edges not shown in the diagram weight four (Korte & Vygen, 2012, p. 570):

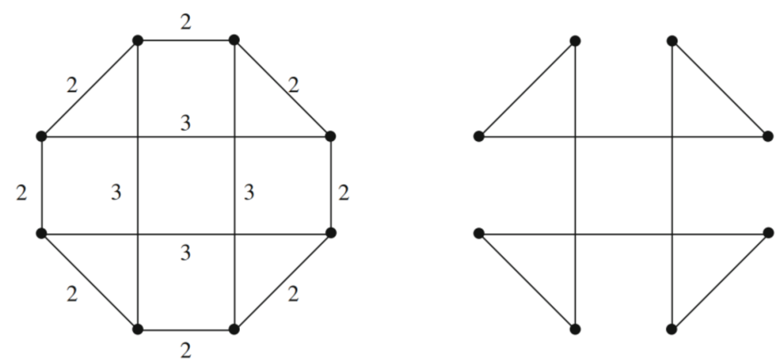

Figure 4: 3-Opt Optimization Results

The K-Opt Algorithm

The K-Opt algorithm can have the parameter k set to any number greater or equal to two; this means that any graph modified by the K-Opt algorithm has steps dependent on the size of k, resulting in different tour outcomes. Therefore, it is not easy to display the K-Opt algorithm graphically in the general sense in the same way that the 2-Opt and 3-Opt algorithms were displayed above. However, we can rigorously define the K-Opt algorithm for the TSP as follows:

Table 2: The K-Opt Algorithm (Korte & Vygen, 2012, p. 569)

K-Opt Algorithm Input: An instance (K_n, c) of the TSP Output: A tour T
[1] Let T be any valid tour [2] Let S be the family of k-element subsets of E(T) [3] For all s ∈ S and all tours T' with E(T') ⊇ E(T) \ s do: If c(E(T')) < c(E(T)) then set T := T' and go to [2]

Therefore, we can conclude that for any constant value k is set to, there are TSP instances and K-Opt tours that are not (k+1)-Opt (Korte & Vygen, 2012, p. 569). Choosing a value for k in advance is one of the main concerns when using this algorithm. For problems with fixed complexity and a predefined shape, using a fixed value for k is sufficient as we can predict any boundary cases where the algorithm may find a non-Optimal tour. The problem is when we are not sure what constraints the input graph will have in it. In this case, using a heuristic where k is not a fixed constant can be effective. The value of k can be determined by an algorithm as is demonstrated in *Combinatorial Optimization: Theory and Algorithms* by Korte & Vygen (2012); see below:

Equation 2: Heuristic Method for K-Opt Optimization

Definition 21.15. *Given an instance* (K_n, c) *of the* TSP *and a tour* T. *An* **alternating walk** *is a sequence of vertices (cities)* $P = (x_0, x_1, \dots, x_{2m})$ *such that* $\{x_i, x_{i+1}\} \neq \{x_j, x_{j+1}\}$ *for all* $0 \leq i < j < 2m$, *and for* $i = 0, \dots, 2m-1$ *we have* $\{x_i, x_{i+1}\} \in E(T)$ *if and only if* i *is even.* P *is* **closed** *if in addition* $x_0 = x_{2m}$.

The **gain** *of* P *is defined by*

$$g(P) := \sum_{i=0}^{m-1} (c(\{x_{2i}, x_{2i+1}\}) - c(\{x_{2i+1}, x_{2i+2}\})).$$

P *is called* **proper** *if* $g((x_0, \dots, x_{2i})) > 0$ *for all* $i \in \{1, \dots, m\}$. *We use the abbreviation* $E(P) = \{\{x_i, x_{i+1}\} : i = 0, \dots, 2m-1\}$.

Without going into too much detail, we can see that methods for creating a heuristic can occur to help the K-Opt method. We notice a second example of a heuristic method in *An Adaptive K-Opt Method for Solving Traveling Salesman Problem* by Ma *et al.* (2016, pp. 6638-6642) where an adaptive K-Opt method for the TSP is proven and has a time complexity of O(N*n3), where N is the number of stages that depend on a stop criterion.

Citations

Arora, S., & Barak, B. (2010). *Computational complexity: a modern approach*. Cambridge Univ. Press.

Datta, S. (2020, November 1). *Traveling Salesman Problem – Dynamic Programming Approach*. Baeldung on Computer Science. https://www.baeldung.com/cs/tsp-dynamic-programming.

Korte, B., & Vygen, J. (2012). In *Combinatorial optimization: theory and algorithms* (5th ed., Vol. 21, pp. 569–572). essay, Springer.

Ma, Z., Liu, L., & Sukhatme, G. S. (2016). An adaptive k-Opt method for solving traveling salesman problem. *2016 IEEE 55th Conference on Decision and Control (CDC)*, 6537–6543. https://doi.org/10.1109/cdc.2016.7799275

Statistics Canada. (2018, January 17). *Families, households and housing*. https://www150.statcan.gc.ca/n1/pub/11-402-x/2011000/chap/fam/fam-eng.htm.

Experiments

The project specified a computation experiment to be performed with the 2-Opt algorithm using Python, Java, C/C++, or Julia. The chosen language for the 2-Opt algorithm implemented is Python. The results of the experiment turned positive, symmetric matrixes were generated randomly with a size specified as per a user's specifications. Test cases of sizes ranging from 5 cities, 25 cities, 100 cities, 500 cities, and 1000 cities are included. All tests turned back with results within a reasonable margin of error for a 2-Opt algorithm. All code in the computational experiments was based off researched contained within the report herein. Since test cases are randomly generated, please run the attached code below in a Python compiler to see specific examples if needed:

```
#imported modules
import random

"""
Name:
   getSymmetricMatrix
Input:
   size - this is the length and
width of the matrix
   start - this is the starting value
for city distances
   end - this is the ending value for
city distances
Output:
   Outputs a symmetric matrix
representing the distances
between cities
Description:
   This function takes a size to
represent the size of a symmetric
matrix
   It also takes a range (start, end)
to represent random number
generation values
   It returns a symmetric matrix
representation of a Graph = G(V,
E) for a 2-Opt algorithm
Algorithm:
   This is based on the assumption
that:
      A is symmetric if and only if A
= A-transpose
"""
def getSymmetricMatrix(size,
start=0, end=1) :
   #create a square matrix based
on the size specified
   matrix = [[None for column in
range(size)] for row in range(size)]

   #for each row in the matrix
   for row in range(len(matrix)) :
      #for each column in the
matrix
      for column in
range(len(matrix[row])) :
         #if the current row equals
the column index, it is a diagonal
entry
         if column == row :
            #assign a value of zero
since a city has no distance from
itself
            matrix[row][column] = 0
         elif row > column : #if we
are looking at the left side of the
diagonal
            #assign the left side of
the diagonal a random distance
value
            matrix[row][column] =
int(round(random.uniform(start,
end) * 100, 0))
            #assign the right side the
same value such that the matrix is
symmetric
            matrix[column][row] =
int(matrix[row][column])
         #END IF
      #END FOR
   #END FOR

   #return the final matrix to the
caller
   return matrix
#END getSymmetricMatrix

"""
Name:
   getDistance
Input:
   matrix - this is the symmetric
matrix representing the cities
   tour - this is the current loop
through all the cities
Output:
   totalDistance - this is the total
distance between all the cities in
the loop
Description:
   gets the distance for a given
tour between all the cities
"""
def getDistance(matrix, tour) :
   #tracks the current distance
between travelled
   totalDistance = 0

   #for each edge in the loop of
cities
   for trip in range(len(tour) - 1) :
      #tally the distance between
the two points
      totalDistance +=
matrix[tour[trip]][tour[trip + 1]]
   #END FOR

   #returns the total distance
between all the traversed cities
   return totalDistance
#END getDistance

"""
Name:
   twoOptSwap
Input:
   tour - a set of cities connected
as a linked list (i.e. 1->2->etc.)
   i - the first edge being swaped
   k - the second edge being
swapped
Output:
   newTour - this is the new set of
edges for the TSP
Description:
   changes two edges in a tour for
a 2-Opt TSP algorithm and returns
a the new tour
"""
def twoOptSwap(tour, i, k) :
   #contains the new tour
   reverse = []
   newTour = []
   firstPass = True

   #for each edge in the list of all
edges
   for edge in range(len(tour)) :
      #if the edge is less then the
current value i
      if edge < i :
```

```
                newTour.append(tour[edge])
            #add these values in reverse order to change the path
            elif edge >= i and edge <= k :
                reverse.append(tour[edge])
            #add the ramining values in order
            elif edge >= k + 1 :
                #add the reverse path if it has not been added yet
                if firstPass :
                    newTour = newTour + reverse[::-1]
                    firstPass = False
                #END IF
                newTour.append(tour[edge])
            #END IF
    #END FOR

    #add the reverse path if it has not been added yet
    if firstPass :
        newTour = newTour + reverse[::-1]
        firstPass = False
    #END IF

    #returns the newly generated tour
    return newTour
#END twoOptSwap

"""
Name:
    twoOptLoop
Input:
    matrix - symmetric matrix representing the city
Output:
    bestTour - this is the shortest path which connect all the cities in the loop
    bestDistance - this is the distanced traversed by the bestTour path above
Description:
    Runs a 2-Opt Algorithm on the dataset in order to determine the appoximate solution for the TSP Algorithm:
    This is an implementation of the K-Opt algorithm for the TSP, with K equal to two
"""
def twoOptLoop(matrix) :
    #holds the current tours
    bestTour = []
    bestDistance = 0

    #create a default path between the nodes in order
    for row in range(len(matrix)) :
        bestTour.append(row)
    #END FOR
    bestTour.append(0)

    #append the last connection between the ending and starting node

    #checks the distance of the current tour
    bestDistance = getDistance(matrix, bestTour)

    #checks if an improvement was made this iteration
    improvement = True

    while improvement : #run the loop for as long as an improvement is made
        improvement = False #no improvement noted this iteration
        for i in range(len(bestTour)) : #checks each row in the matrix
            for k in range(i + 1, len(bestTour)) : #checks each column in the matrix
                #gets a new tour pointer
                newTour = list(bestTour)

                #gets a new 2-Opt path
                newTour = twoOptSwap(newTour, i, k)

                #checks if the new path is valid
                error = False
                for element in range(len(newTour) - 1) :
                    if newTour[element] == newTour[element + 1] :
                        #we cannot move nowhere
                        error = True
                    #END IF
                #END FOR

                #move to the next iteration of the loop
                if error : continue

                #gets new tour distance
                newDistance = getDistance(matrix, newTour)

                #finds if the new path is better then the old one
                if bestDistance > newDistance :
                    #replaces the old path with the new path
                    bestDistance = newDistance
                    bestTour = newTour
                    improvement = True
                    #leave the current loop
                    break
                #END IF
            #END FOR

            #checks if an improvement was made this iteration
            if improvement == True :
                #go to the topmost loop
                break
            #END IF
        #END FOR
    #END WHILE
    return [bestTour, bestDistance]
```

```
#END twoOptLoop


#---Main (Entry Point Driver)---#

#gets a random symmetric matrix (of length five) and runs the 2-Opt on it
matrix = getSymmetricMatrix(5)
print("A randomly generated matrix of five elements is as follows:")
for element in matrix : print(element)
results = twoOptLoop(matrix) #returns the results of the 2-Opt search
print("We get the following results for the 2-Opt algorithm")
print("Optimal path:")
print(results[0])
print("Optimal distance:")
print(results[1])

#gets a random symmetric matrix (of length five) and runs the 2-Opt on it
matrix = getSymmetricMatrix(25)
print("A randomly generated matrix of 25 elements is as follows:")
#--uncomment the code to get the matrix generated
#for element in matrix : print(element)
results = twoOptLoop(matrix) #returns the results of the 2-Opt search
print("We get the following results for the 2-Opt algorithm")
print("Optimal path:")
print(results[0])
print("Optimal distance:")
print(results[1])

#gets a random symmetric matrix (of length five) and runs the 2-Opt on it
matrix = getSymmetricMatrix(100)
print("A randomly generated matrix of 100 elements is as follows:")
#--uncomment the code to get the matrix generated
#for element in matrix : print(element)
results = twoOptLoop(matrix) #returns the results of the 2-Opt search
print("We get the following results for the 2-Opt algorithm")
print("Optimal path:")
print(results[0])
print("Optimal distance:")
print(results[1])

#gets a random symmetric matrix (of length five) and runs the 2-Opt on it
matrix = getSymmetricMatrix(500)
print("A randomly generated matrix of 500 elements is as follows:")
#--uncomment the code to get the matrix generated
#for element in matrix : print(element)
results = twoOptLoop(matrix) #returns the results of the 2-Opt search
print("We get the following results for the 2-Opt algorithm")
print("Optimal path:")
print(results[0])
print("Optimal distance:")
print(results[1])

#gets a random symmetric matrix (of length one thousand) and runs the 2-Opt on it
matrix = getSymmetricMatrix(1000)
#--uncomment the code to get the matrix generated
#print("A randomly generated matrix of one thousand elements is as follows:")
#output = ""
#for element in matrix : output += str(element) + "\n"
#print(output)
results = twoOptLoop(matrix) #returns the results of the 2-Opt search
print("We get the following results for the 2-Opt algorithm")
print("Optimal path:")
print(results[0])
print("Optimal distance:")
print(results[1])
```