Group 65 Project CS 325 Project Report

Mindy Jones, Mathew Kagel, Benjamin Fridkis

Introduction

The objective of this group project was to research algorithms for solving the travelling salesman problem and implement an optimal algorithm that emits correct tours with time and optimality constraints. Our group used https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf as a starting point for our research. We each chose a tour construction algorithm to research and a tour optimization algorithm to research. We each implemented some code to test a subset of the algorithms we researched, in order to sanity check them.

Team member	**Algorithm Phase**	**Algorithm**	**Implementation**
Mindy Jones	Tour construction	Nearest Insertion	None
Mathew Kagel	Tour construction	Nearest Neighbor	Benjamin implemented Nearest Neighbor accidentally, Mindy implemented explicitly
Benjamin Fridkis	Tour construction	Greedy heuristic	Benjamin nominally implemented Greedy initially, but it turned out to be Nearest Neighbor. He then implemented a true Greedy heuristic, as described below.
Mindy Jones	Tour optimization	Or-opt	Mindy implemented
Mathew Kagel	Tour optimization	Lin-Kernighan	None
Benjamin Fridkis	Tour optimization	2-opt	Benjamin implemented

TSP algorithms and pseudocode

Tour Construction

Nearest Insertion

The htsp.pdf doc states that Nearest Insertion is O(n^2). Unfortunately, the description of the algorithm in the htsp.pdf doc and the description of it in the original paper “Approximate algorithms for the traveling salesperson problem” (Rosenkrantz, Stearns, Lewis, 1974) didn’t give enough detail for Mindy to write pseudocode with that runtime. After some analysis, the fastest concrete algorithm Mindy was able to write for Nearest Insertion is O(n^2 * log n), which is described below.

As part of trying to find a fast algorithm, Mindy researched algorithms to compute “all nearest neighbors” among a set of points. Almost all of the options used either quad trees or k-d trees. The performance of these algorithms depend on characteristics of the input, but most were on the order of O(n log n) to form the quad tree, and O(log n) or O(1) for looking up the nearest neighbor of a single point in the quad tree.

Using a quad tree with O(log n) for nearest neighbor search and O(log n) for removal, the Nearest Insertion algorithm can execute in O(n^2 log n) time (per Mindy’s analysis).

The general approach for the Nearest Insertion algorithm is the following (described in htsp.pdf):

1. Select the shortest edge, and make a subtour of it.
2. Select a city not in the subtour, having the shortest distance to any one of the cities in the subtour.
3. Find an edge in the subtour such that the cost of inserting the selected city between the edge’s cities will be minimal.
4. Repeat step 2 until no more cities remain.

The algorithm for Nearest Insertion using a quad tree is below.

```
nearestNeighborTour(cities)
  unusedQuadTree <- new QuadTree
  for city in cities
    unusedQuadTree.insert(city)
  tourQuadTree <- new QuadTree
  tour <- new list
  closestDist = infinity
  minFromCity = null
  minToCity = null
```

```
for city in cities
  nearestCity = unusedQuadTree.nearestTo(city)
  if city.distTo(nearestCity) < closestDist
    closestDist = city.distTo(nearestCity)
    minFromCity = city
    minToCity = nearestCity
tour.first = minFromCity
minFromCity.next = minToCity
minToCity.next = minFromCity
tourQuadTree.insert(minFromCity)
tourQuadTree.insert(minToCity)
unusedQuadTree.remove(minFromCity)
unusedQuadTree.remove(minToCity)
while size(unusedQuadTree) > 0
  closestDist = infinity
  minUnusedCity = null
  minTourCity = null
  for city in unusedQuadTrees
    nearestCity = tourQuadTree.nearestTo(city)
    if city.distTo(nearestCity) < closestDist
      closestDist = city.distTo(nearestCity)
      minUnusedCity = city
      minTourCity = nearestCity
  minAddedDist = infinity
  minAddedTourCity = null
  for tourCity in tour
    curDist = tourCity.distTo(tourCity.next)
    newDist = tourCity.distTo(minUnusedCity) + minUnusedCity.distTo(tourCity.next)
    if newDist < minAddedDist
      minAddedDist = newDist
      minAddedTourCity = tourCity
  prevNext = minAddedTourCity.next
  minAddedTourCity.next = minUnusedCity
  minUnusedCity.next = prevNext
  tourQuadTree.insert(minUnusedCity)
  unusedQuadTree.remove(minUnusedCity)
```

Nearest Neighbor

The following summary of nearest neighbor comes from [1]:

Algorithm:

1. Select a random city
2. Find the nearest unvisited city and go there
3. Are there any unvisited cities left? If yes keep going.
4. Return to the first city

This algorithm takes $O(n^2)$ time to run, where n is the number of points to build a tour from.

Pseudocode:

```
E -> Hash map holding collection of points to make TSP tour, use point number as
search key.
T -> Dynamic array holding final collection of points in the TSP tour. Initialized to
be the length of E + 1, which is necessary to hold all of the points and the tour
distance so no resizing occurs.
First element of T is the tour distance, so T[0] = 0
Add all points to E
R = first point in hash map
while (E not empty) {
             Get R from E and assign to P
             Add P to T
             Remove P from E
             Iterate over the points left in E to find the closest point to P, and
             assign to R
             D += distance from P to R
             P = R
}
// At the end of the loop P is the point at the end of the tour
T[0] += distance from P to S           // Add distance from end point to start point,
completing tour
```

Greedy Heuristic

Steps[1]:

1. Sort all edges.
2. Add the next shortest edge to the tour that does not create a cycle within any subgraphs thereof or result in a vertex with a degree greater than 2 for any vertex already added thereto.
3. Repeat until N edges have been added.

Summary

This technique has a runtime complexity of $O(n^2 \log_2(n))$. A edge list is generated in which each edge is placed into a priority queue (or min-heap, with a heap order property of edge distance). The tour construction following this (priority queue generation) step is simple: n (where n = the number of cities) edges are added in min heap order (i.e. shortest to longest), so long as adding a given edge does not result in a degree of three for any vertex (city) already added to the tour, nor create a cycle within any subgraph already established within the tour (except when the last edge is added, in which case a cycle including all the cities is allowed.) Edges are “popped” (discarded) from the priority queue without being added to the tour if their addition would create cycles or result in a vertex (already in the tour) having a degree of three as just described. The algorithm is bound above by $f(n) = n^2 \log_2(n)$ (and below, for that matter, rendering it $\Theta(n^2$

$\log_2(n)$) because the dominating process is the generation of the priority queue edge list with n^2 vertices.

Runtime Analysis:

To construct a priority queue for every edge of the complete graph (provided initially as city coordinates), lgn operations are required for each of n^2 vertices (with constant work for each operation). This is the dominating runtime factor, and hence the algorithm is $\Theta(n^2 \log_2(n))$. To construct the tour after generating the priority queue, edges which create a degree of three on some vertex already added to the tour and self-referential edges are removed from the priority queue until an edge is identified that will not create said third degree and is not self-referential. This edge is then tested to determine if its potential addition to the tour will result in a cycle (unless it is the last edge to be added).

The number of edges that could possibly create a cycle (after passing the first check that prevents degrees of three for existing tour cities and self-referential edges from consideration) is equal to the number of disjoint subgraphs existing in the tour, with subgraph size proportional thereto. For instance, if the tour established thus far is connected (i.e. exists as a single subgraph), there is only one edge that can pass the previous check (i.e. will not create a third degree for any vertex already added) and still create a cycle (i.e. the edge that would create a Hamiltonian cycle on the subgraph). To determine this edge does in fact create a cycle, at worst an operation count equal to the number of vertices already added to the tour is required. In other words, at worst n operations are required (n -1 actually, because the last edge does not perform this check). As another example, if the pre-established tour consists of two disjoint subgraphs of equal size, there are two edges that can pass the previous check and still create cycles (one to create a Hamiltonian cycle for each subgraph). To identify these as such, at worst each of these two edges requires n/2 operations (to traverse through each subgraph in total, respectively) . In other words, at worst n operations are required to vet out all edges that can possibly create a cycle in the tour prematurely, given its current state. This principle holds for any combination of subgraphs in the existing tour: the number of edges possibly creating cycles multiplied by the number of vertices to traverse for identifying said cycles is equal to n. So for each of n edges added, n operations are required to rule out cycle-creating edges. This equates to a runtime complexity of $O(n^2)$ for cycle checks. Finally, the number of edges to remove from the priority queue is at worst is n^2, but this is only additive in respect to the cycle checks described above. Hence, the overall runtime is bound by the priority queue generation ($\Theta(n^2 \log_2(n))$.

Pseudocode:

```
loadGraphOfMapAsPriorityQueue(dataInput)
        let pq be an empty priority queue
        cityCount = 0
        bool cityCountEstablished = false
        do for each line of dataInput
```

```
        get x and y coordinates of city from dataInput
        startingCity = get city number from dataInput
        startingCityXCoord = get X from dataInput
        startingCityYCoord = get Y from dataInput
        for each line of dataInput
                get x and y coordinates of city from dataInput
                cityXCoord = get X from dataInput
                cityYCoord = get Y from dataInput
                distanceToCity = sqrt((cityXCoord –
                                        (startingCityXCoord)^2)) +
                                        (cityYCoord –
                                        (startingCityYCoord)^2))
                Add starting city, city, and associated distance to pq
    return pq
```

loadTour(pq) //pq is a priority queue containing all graph edges

```
    let tspTour be an empty array
    //cityTourPositionTracker (below) saves each cities //previous city, next city,
    and status as a city already //added to the tour. All values are initialized to
    false to //indicate no assignment has been made.
    let cityTourPositionTracker be an array of size city count, with all values
    initialized to false
    cityCount = sqrt(pq.size)
    distance = 0
    i = 0
    for i to cityCount
        edgeAdded = false
        while !edgeAdded
                while edge city is in tour or edge next city is in tour or edge
                city = edge next city
                        pq.pop //remove edge
                downstreamCity = pq.top.nextCity
                edgeCreatesCycle = false
                while cityTourPositionTracker[downstreamCity] = true (i.e.
                nextCity of the current edge is established as a city in tour) and
                its next city respectively is established in the tour and there is
                more than one edge left to add
                        downstreamCity = the next city of the current edge’s next
                        city
                        if downstreamCity == the current edge’s city
                                pq.pop
                                edgeCreatesCycle = true
                                break
                if !edgeCreatesCycle
                        cityTourPositionTracker[edge’s city.citystatus] = true;
                        cityTourPositionTracker[edge’s city.nextCity] = edge’s next
                        city
```

```
                        cityTourPositionTracker[edge's nextCity.previousCity] =
                        edge's city

                        distance += edge's distance

                        pq.pop

                        edgeAdded = true

    i = 0, j = 0

    for i to cityCount

        tspTour[i] = j

        j = cityTourPositionTracker[j.nextCity]

    return tspTour and distance
```

Tour optimization

Or-opt

Or opt is an algorithm that moves 1, 2, or 3 consecutive cities to another position in the tour. Since it can move up to 3 cities, it is technically a restricted version of 3-opt. The information on Or-opt is available at http://tsp-basics.blogspot.com/2017/03/or-opt.html .

The shift operation forms the basis of the changes made to the tour (more information at http://tsp-basics.blogspot.com/2017/03/shifting-segment.html):

```
proc Shift_Segment(tour: var Tour_Array; i, j, k: Tour_Index) =
  ## Shifts the segment of tour:
  # cities from t[i+1] to t[j] from their current position to position
  # after current city t[k], that is between cities t[k] and t[k+1].
  # Assumes:  k, k+1 are not within the segment [i+1..j]
  let
    segmentSize = (j - i + N) mod N
    shiftSize = ((k - i + N)  - segmentSize + N) mod N
    offset = i + 1 + shiftSize
  var
    pos: Tour_Index
    segment: seq[City_Number] = newSeq[City_Number](segmentSize)

  # make a copy of the segment before shift
  for counter in 0 .. segmentSize-1:
    segment[pos] = tour[(pos + i+1) mod N]

  # shift to the left by segmentSize all cities between old position
  # of right end of the segment and new position of its left end
```

```
  pos  = (i + 1) mod N
  for counter in 1 .. shiftSize:
    tour[pos] = tour[(pos + segmentSize) mod N]
    pos  = (pos + 1) mod N

  # put the copy of the segment into its new place in the tour
  for pos in 0 .. segmentSize-1:
    tour[(pos + offset) mod N] = segment[pos]
```

Now in order to figure out how much a segment shift improves the tour length, it is necessary to have a function which computes the gain.

```
proc Gain_From_Segment_Shift(X1, X2, Y1, Y2, Z1, Z2: City_Number):
                             Length_Gain =
  ## Gain of tour length which can be obtained by performing Segment
  ## Shift
  # Cities from X2 to Y1 would be moved from its current position,
  # between X1 and Y2, to position between cities Z1 and Z2.
  # Assumes: X1!=Z1
  #          X2==successor(X1); Y2==successor(Y1); Z2==successor(Z1)
  let del_Length = distance(X1, X2) + distance(Y1, Y2) + distance(Z1, Z2)
  let add_Length = distance(X1, Y2) + distance(Z1, X2) + distance(Y1, Z2)
  result = del_Length - add_Length
```

The overall Or-opt algorithm is below.

```
proc LS_Or_opt_Take_First(tour: var Tour_Array) =
  ## Optimizes the given tour using Or-opt
  # Shortens the tour by repeating Segment Shift moves for segment
  # length equal 3, 2, 1 until no improvement can by done: in every
  # iteration immediately makes permanent the first move found that
  # gives any length gain.
  var
    locallyOptimal: bool = false
    i, j, k: Tour_Index
    X1, X2, Y1, Y2, Z1, Z2: City_Number

  while not locallyOptimal:
    locallyOptimal = true

    for segmentLen in countdown(3, 1):

      block two_loops:
        for pos in 0 .. N-1:
          i = pos
          X1 = tour[i]
          X2 = tour[(i + 1) mod N]

          j = (i + segmentLen) mod N
```

```
Y1 = tour[j]
Y2 = tour[(j + 1) mod N]

for shift in segmentLen+1 .. N-1:
  k  = (i + shift) mod N
  Z1 = tour[k]
  Z2 = tour[(k + 1) mod N]

  if Gain_From_Segment_Shift(X1, X2, Y1, Y2, Z1, Z2) > 0:
    Shift_Segment(tour, i, j, k)
    locallyOptimal = false
    break two_loops
```

Lin-Kernighan

The Lin-Kernighan heuristic is an adaptive version of the 2-Opt swapping procedure. 2-Opt finds two edges to replace with two other edges in a tour that will make the tour more optimal, and then it makes the exchange. The Lin-Kernighan heuristic does this same procedure, except that it does so for a variable k-Opt, searching through multiple vertices until the gain function that it uses begins to decrease. Once the gain function begins to decrease, the Lin-Kernighan heuristic does the k-Opt swap with the largest gain. There is a small allowance for backtracking in Step 6 of the algorithm among the first four edges that the algorithm selects. There is also a small allowance for total gain to decrease, as long as it's at the 2-Opt step in step 6a. The ability to have a temporarily decreasing gain function sets Lin-Kernighan apart significantly from other optimism TSP heuristics. After 2-Opt, either the gain needs to be monotonically nondecreasing, or the algorithm stops. According to [3], the Lin-Kernighan heuristic will run at approximately n^2 time, if implemented according to the paper. According to [1], the Lin-Kernighan heuristic will run at $n^{2.2}$ time.

Algorithm:

The algorithm outline below comes from the original paper on Lin-Kernighan [3]. There is a more cursory outline of the Lin-Kernighan algorithm in the Lin-Kernighan-Helsgaun heuristic detailed in [2].

1. Start with a tour T [3]. This will have a length and an order of points to visit.

2. G_star, or optimal gain, gets set to zero. G_star is the best improvement so far. Choose any node t_1 and let x_1 be one of the edges of T adjacent to t_1. Let i = 1 [3].

t_1 gets selected at random. t_1 needs to get marked as having been visited with this tour T so that it does not get run again. Other vertices can still use t_1 in their calculations, but it should only be the center of the k-Opt move one time. This means mark t_1, but don't remove it completely. All of the point numbers need to be sequential in the point list that the tour is generated from, in order to make finding a random point to start from easier. The x edges are edges that are already in the tour. The y edges are edges that could be swapped into the tour for the x edges.

3. From the other endpoint t_2 of x_1 choose y_1 to t_3 with gain $g_1 > 0$. If there is no y_1 that exists, go to Step 6(d) [3]. The shortest y edge that meets the criteria gets picked.

4. Let i = i + 1. Choose x_i [which currently joins t_{2i-1} to t_{2i}] and y_i as follows:
   a) x_i is chosen so that, if t_{2i} is joined to t_1, the resulting configuration is a tour. To do this, add x_i temporarily and add y_i temporarily to be the edge that closes the tour. Then call CheckTour() (see pseudocode) to make sure that the swap can form a tour. If it can't, pick another x_i [Thus, for a given y_{i-1}, x_i is uniquely determined. This is the application of the feasibility criterion; it guarantees that we can always 'close up' to a tour if we wish, simply by joining t_{2i} to t_1, for any i >= 2. The choice of y_{i-1} Step 4(e), ensures that there is always such an x_i] [3]
   b) In the event that immediate tour closure is not optimal, y_i is some available link at the endpoint i_{2i} shared with x_i subject to (c), (d), and (e). If no y_i exists, go to Step 5. [Clearly, to make a large cost reduction at the ith step, y_i distance should be small, and so in general we choose nearest neighbors preferentially.] [3]
   c) x's and y's must be disjoint, at least for this round of the algorithm. x_i cannot be a link previously joined, and y_i cannot be a link previously broken [3].
   d) Gain criterion, total gain must be positive [3].
   e) Sequence criterion, y_i choice must permit the breaking of an x_{i+1} [3].
   f) Before y_i is constructed, check if closing up by joining t_{2i} to t_1 will give a better gain. If so k = i, and the exchange occurs [3].

5. Terminate the construction of x_i and y_i in Steps 2 through 4 when either no further links x_i and y_i satisfy 4(c) through 4(e) or when g_i <= G_star. At this stage find the k value that yields maximum gain, and make the exchange using that k value. Use ExchangeEdges() to do this (see pseudocode). Now there is an optimized tour, all previously marked vertices become unmarked and can have a k-Opt move run on them again. The entire algorithm resets each time a new tour is generated. There is no need to generate a new T and make a new data structure, since the new T is just a modified old T using the old T's data structure [3].

6. If G_star = 0, a limited backtracking facility is invoked, as follows:
   a) Repeat Steps 4 and 5, choosing y_2's in order of increasing length, as long as they satisfy the gain criterion $g_1 + g_2 > 0$. [If an improvement is found at any time, of course, this causes a return to Step 2] [3]
   b) If all choices of y_2 in Step 4(b) are exhausted without profit, return to Step 4(a) and try the alternate choice for x_2. [3]
   c) If this also fails to give improvement, a further backup is performed to Step 3, where the y_1's are examined in order of increasing length [3].
   d) If the y_1's are also exhausted without profit, we try the alternate x_1 in Step 2 [3].
   e) If this fails, a new t_1 is selected, and we repeat at Step 2 [3].

7. This procedure terminates when all n values of t_1 have been examined without profit. At this time, either you're done, or a different initial tour can be run through the algorithm to drive closer to the global minimum for the tour [3].

Pseudocode:

This is an improvement heuristic, so it needs to operate on an already constructed tour. The list of point numbers, x-coordinates, and y-coordinates gets loaded into a hash map called E. The hash map is searched using point number. It's possible that this has already been done in order to construct the tour. Anything in this pseudocode that needs x or y coordinates will get them from E. The tour T gets loaded into a dynamic array called T, which also may have already been done. n refers to the length of the tour.

Use an n^2 size array called D to memoize distances. Make an n^2 space and store distances from vertices in this space, and attempt to use it each time a distance needs to get calculated. If the entry is empty or has a sentinel value, then calculate the distance.

Additional arrays are needed as follows:
Q -> array to hold the starting vertices that have already been visited by the algorithm
R -> array to hold all of the potential y edges from any vertex t that gets picked by FindNewEdges()
X -> array to hold pairs of vertices for the x edges, the order of the pairs at each index doesn't matter
Y -> array to hold pairs of vertices for the y edges, the order of the pairs at each index doesn't matter
U -> array to hold a list of vertices at each index that have been used to construct that numbered edge in x. For example, once a selection for t_2 is made for edge x_1, the vertex t_2 would get added to index 1 of array U so that this does not get re-selected later if another choice of x_1 is necessary.
V -> array to hold a list of vertices at each index that have been used to construct that numbered edge in y.
B -> array to hold a list of gain values. The index is the k number for the k-Opt move.

From Algorithm Step 1:
Start with tour T in dynamic array T

From Algorithm Step 2:
```
Set G_star = 0
if coming from Step 3 or Step 4
       Use t1 already selected
       Clear all elements from X except x1
       // X is the list of x edges for possible k-Opt move
else
       Clear all elements from X
       Select t1 at random
       if Q has all points in T          // Nothing left to optimize
             Go to Step 7
```

```
        else if t1 is marked in Q
                Select a different t1
        Mark t1 in Q as visited.          // Mark t1 as visited.
        Add t1 to x1 within X.
Choose a t2 not in U for x1 and update x1 within X.
if there is no further choice of x1
        Clear all elements of X, Y, U, V, and B
        Go back to the beginning of Step 2
Add t2 to U at index 1
// U keeps track of the vertices that have been selected for x edges
i = 1
```

From Algorithm Step 3:

```
FindNewEdges(R, T, x1, t2)
Choose the shortest edge at the index of R holding vertices near t2 that hasn't been
used as a y1 already, so not in index 1 of V.
// Use array V to determine if an edge has been used already for a particular y edge
if there is no further choice of y1
        Clear all elements of V
        Go to Step 2 with the same t1
Else
        Add y1 to Y.
        Add t3 of y1 to V at index 1.
```

From Algorithm Step 4:

```
i++    // Increase i by 1
Choose xi from y2i - 1
xi can't be in Y
xi can't have its end vertex in common with vertices already in X and Y
if there is no further choice for xi
        if i == 2
                Clear all elements from Y, V, and B
                Go back to Step 2 with the same t1
        else
                Go to Step 5
Add xi to X
Mark choice of t2i in U.
Add yi to Y as the edge that closes the tour from the end of xi
if CheckTour(T, X, Y) returns true
        copy yi to Y[-i]    // Save this temporary yi in case of a later tour closure
        record gain from choosing yi at B[-i]    // Gain in case of tour closure
        remove yi from Y[i]  // xi is a valid choice, don't need temporary yi anymore
else
        remove xi from X           // this xi doesn't work
        Go back to beginning of Step 4 at the same i value

FindNewEdges(R, T, xi, t2i)
Add yi to Y that is not marked in V
yi can't be in X
yi can't have its end vertex in common with any vertices already in X and Y
```

```
if there are no y_i left
       if B[-i] > G_star || i == 2   // If tour closure at i is optimal or if i is 2
             copy Y[-i] to Y[i]         // Use tour closure y_i
             copy B[-i] to B[i]         // Use tour closure gain g_i
             G_star = B[i]
             Go to Step 5
       else
             Go to Step 5               // Make k-Opt move without using tour closure
else
       Mark choice of t_2i + 1 in V
       Calculate B[i] using X and Y with newly added x_i and y_i
       if B[i] >= G_star
             G_star = B[i]
             Go back to the beginning of Step 4 with incremented i value to pick more
             edges
       else if G_star == 0 and i == 2
             Go back to the beginning of Step 4 with the same i value and same x_1
       else if B[i] < 0 and G_star < 0
             Clear all elements from Y, V and B
             Go back to Step 2 with the same t_1
       else
             Go to Step 5               // Make k-Opt move without using tour closure
```

From Algorithm Step 5:

```
Find the maximum gain g in B. Record the index k in B.
Call ExchangeEdges(T, X, Y, k)
Clear all elements in X, Y, U, V, B and R
Go to Step 2 with an updated tour T
```

From Algorithm Step 7:

```
If no further improvement can be found, stop.
```

Helper Functions:

```
FindNewEdges(R, T, x, t) {
        if R already has a list at this vertex t
              return
        Else
             Take the t vertex, and use T to get the eight preceding and eight
             following vertices from the t vertex.
             Calculate distances from those sixteen vertices to t. Call these
             distances Dy.
             These distances should be added to the array D and referenced from D if
             they are available.
             Generate a list of potential ending vertices for y and their distances
             and order this list by lengths Dy of y in ascending order. Starting with
             the shorter y edges is a somewhat short-sighted and greedy way to
             maximize the gain function.
             Store the list in R at the index of vertex t.
}
```

```
CheckTour(T, X, Y) {
       Copy the tour T to W
       Make another copy of tour T to K
       Replace all of the x edges from array X with y edges from array Y in tour W

       // For a sequential exchange of edges, yi (the last y selected) will run from t1
       to t2i.
       Add all vertices in Y to array P.
       Index = 1

       do
             Remove tIndex from K
             if moving forward from tIndex completes the x edge containing tIndex
                    Move backward in tour W from tIndex, removing points from tour K as
                    you go
                           if a dead end is reached // tour broken
                                  return false
                           if a point in P is reached while traversing W
                                  Call this point tnew
                                  Find the y that has tnew
                                  Remove tnew form K
                                  Index = index of vertex in y that is not tnew
                                  if Index is 1 and P is not empty
                                  // Not all vertices reached
                                         return false
                                  break from inner traversal loop to continue do-while
                           Else
                    Move forward in tour W from tIndex, removing points from tour K as
                    you go
                           if a dead end is reached
                                 return false
                           if a point in P is reached while traversing W
                                  Call this point tnew
                                  Find the y that has tnew
                                  Remove tnew form K
                                  Index = index of vertex in y that is not tnew
                                  if Index is 1 and K is not empty
                                  // Not all vertices reached
                                        return false
                                  break from inner traversal loop to continue do-while
       while (Index is not 1)

       if K still has points in it // Didn't reach all of the points in the tour
             return false
       else
             return true
}

ExchangeEdges(T, X, Y, k) {
```

```
        Make a new tour out of T, the first k edges of X, and the first k edges of Y.
        Replace the first k edges of X with the first k edges of Y in T.
        Build a new tour by doing a traversal similar to what is in CheckTour(), except
        in addition to removing points from a tour K, they would need to be added to an
        entirely new tour, and that new tour gets assigned to T.
        Update tour distance in T. Trade the x edge distances for y edge distances.
}
```

Other Considerations for Increased Efficiency of the Lin-Kernighan heuristic:

1. When backtracking, only consider five contenders for y_1 and y_2. If these five best contenders don't provide a positive gain, move on [3].
2. Record tour solutions at certain nodes. If the same local optimal tour is arrived at for a node, see if it is recorded so that it does not need to be re-checked [3].
3. During the Lin-Kernighan heuristic, keep looking at the value of $|x_{i+1}| - |y_i|$, in order to maximize gain as the algorithm is selecting edges [3]. This is referred to as lookahead [3].
4. Find edges in common between tours that are local optimums and mark them as unbreakable [3].

Have a check for facilitating non-sequential exchanges. If a non-sequential exchange that causes optimal positive gain and does not break the tour can be found, use that [3].

2-opt

This relatively straightforward technique involves "flipping" the tour at any two cities a and b such that the distances between city ((a – 1) + a) + (b + (b + 1)) is greater than the distance between ((a – 1) + b) + (a + (b + 1)). In other words, the tour order is reversed between cities a and b if the overall tour distance is decreased when doing so. This eliminates any potential "cross-overs" in the overall route that are causing a sub-optimal distance. (Note the first city cannot be changed as it is the starting city, and adjacent cities are not considered because the greedy approach already ensures the distance between them is minimized.) Each city is compared to all other cities in the tour in attempt to make this improvement at each comparison. A maximum of n cities will be swapped for each city, yielding a runtime of $O(n^2)$. This method can be repeated until no further improvement is possible.

Yet further improvement is possible if, after every swap occurs, the process is repeated entirely (i.e. execution breaks out of the current nested loop structure to repeat the process from the beginning *each time a swap occurs*). While marginal gains are achieved in this fashion, the further optimization required results in a runtime that is impractical for large data sets (with the exact bound difficult to quantify, since the tour order may be entirely restructured after each change is made). In the particular implementation given below in pseudocode form, this strategy for additional tour improvement is only employed for data sets of n <= 2500 to maintain reasonable runtimes for all data sets. (And when it is used as such, an additional graph is generated as an array of arrays instead of an array of min-heaps, so random-access is provided for all adjacency lists. See pseudocode below.)

Pseudocode:

```
twoOptImprove(tour, graphOfMapAsVectors)
      bool breakOutToOptimize, nExceeds2500, improved
      if tour.size > 2500
            nExceeds2500 = true
      else
            nExceeds2500 = false
      do
            improved = false
            breakOutToOptimize = false
            for i = 0 to i < tour.size – 2 && breakOutToOpimize == false
                  for j = i,k = i + 1 to k < tour.size && breakOutToOpimize == false
                              if k – j == 1     //Adjacent cities are not //swapped,
                                                see section //"2-Opt Solution
                                                //Improvement" above
                                j++
                                k++
                                start inner loop over
                        if distance from tour[j] to tour[k-1] +
                            distance from tour[j+1] to tour[k] <
                            distance from tour[j] to tour[j+1] +
                            distance from tour[k-1] to tour[k]
                            reverse the order of all cities between (but not
                            including) tour[j] and tour[k] and recalculate distance
                            improved = true
                            if nExceeds2500 == false
                              breakOutToOptimize = true
                        j++
                        k++
                  i++
      while improved
```

References:

[1] Christian Nilsson. 2003. Heuristics for the Traveling Salesman Problem. Linköping University, Linköping, Sweden. (Informal reference is the following link: https://web.tuke.sk/fei-cit/butka/hop/htsp.pdf)

[2] Keld Helsgaun. 2000. An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic. Department of Computer Science Roskilde University, Roskilde, Denmark. (Informal reference is the following link: http://akira.ruc.dk/~keld/research/LKH/LKH-1.3/DOC/LKH_REPORT.pdf (pg. 8-16))

[3] S. Lin and B. W. Kernighan. 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. J. Operations Research Volume 21, Issue 2 (April 1973), 498-516. (Informal reference is the following link: http://160592857366.free.fr/joe/ebooks/ShareData/An%20Effective%20Heuristic%20Algorithm%20for%20the%20Traveling-Salesman%20Problem.pdf (1973 paper on Lin-Kernighan algorithm))

Implemented algorithms

For tour construction, our team implemented nearest neighbor and the greedy heuristic as described above. For tour optimization, our team implemented 2-opt and or-opt as described above. The pseudocode for these algorithms is listed in the sections above after their verbal descriptions.

Final algorithm selection

Our team ran the algorithms and checked the runtime for them. For tour lengths <= 1000 cities, all combinations of tour construction and optimization ran under 3 minutes, so we decided to actually run multiple algorithms and just pick the one with the best result, because different algorithms performed differently for different tours. Run 1 is nearest neighbor + 2-opt + or-opt, and run 2 is the greedy heuristic + 2-opt + or-opt. For tour lengths > 1000 cities, only run 1 is used due to the 3 minute time limit. Also, or-opt will bail out if the program has run for 3 minutes, so that it doesn't run for too long.

Measurements

(running time and tour lengths)

Example test instances

Example tour number	distance	ratio	time
1	110948	1.0258	0.031371 seconds
2	2702	1.0477	1.23551 seconds
3	1669192	1.0611	178.706 seconds

Competition test instances

Competition test number	distance	time
1	5461	0.010106 seconds
2	7533	0.077711 seconds
3	12679	1.10494 seconds
4	17440	7.98397 seconds
5	24209	64.6112 seconds
6	33845	57.1511 seconds
7	53512	178.147 seconds