CS 325 - Final Project

Group 45
abdullai@oregonstate.edu, adriancj@oregonstate.edu, goodmaal@oregonstate.edu

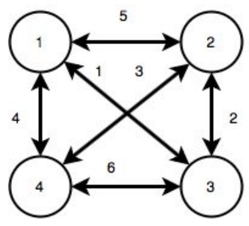
1. Algorithm Research

1.1 Held-Karp Algorithm

1.1.1 Description

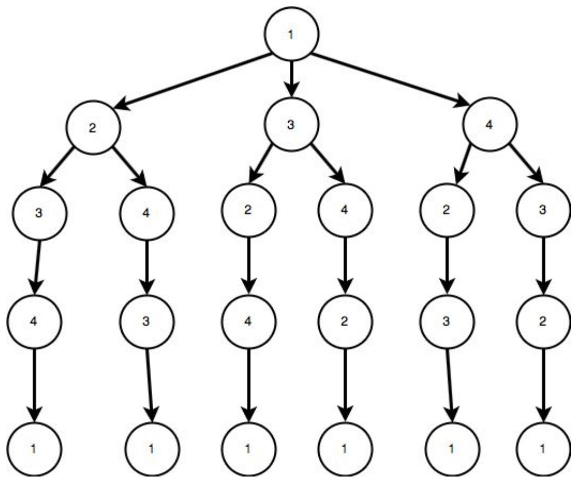
The Held-Karp algorithm is one of the exact algorithms to solve a TSP. It yields an optimal and exact solution, e.g. the shortest path possible that contains all vertices of a graph. It is using a dynamic programming approach. This algorithm is an improvement over the brute force exact algorithm in which all permutations of the graph vertices are checked in order to find the shortest path. Such an algorithm runs in O(n^2 * 2^n) as opposed to O(n!) running time of the brute force algorithm.

The algorithm based on the following optimal structure property: every subpath of the shortest path is of shortest distance itself. Consider the following symmetric graph with the weights for each edge:

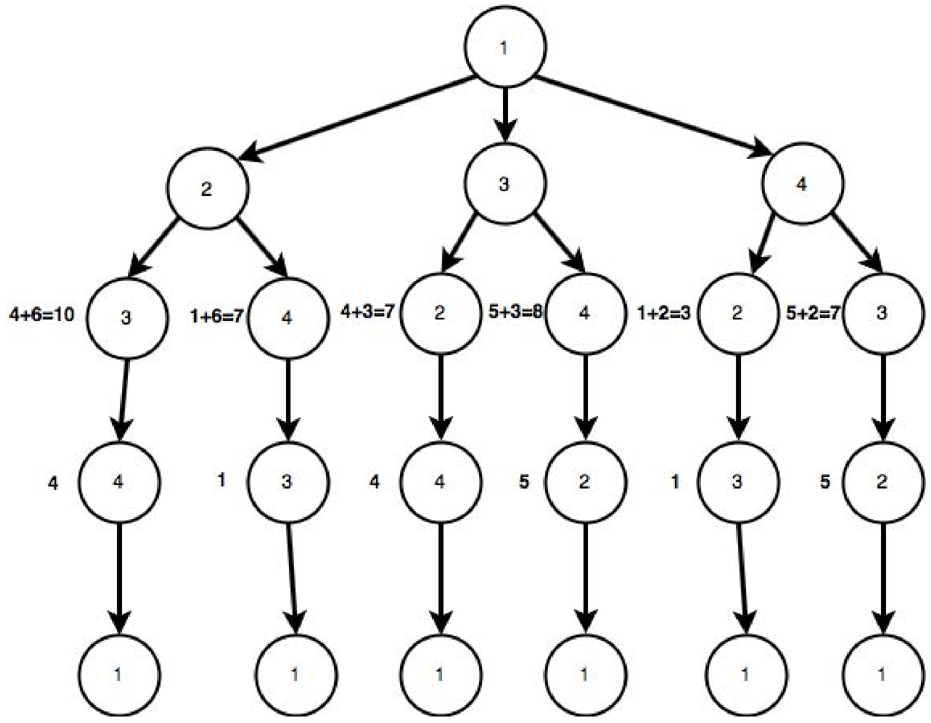


In order to calculate the shortest path, we need to construct a tree that shows possible routes, starting from vertex one and ending at vertex one, too.

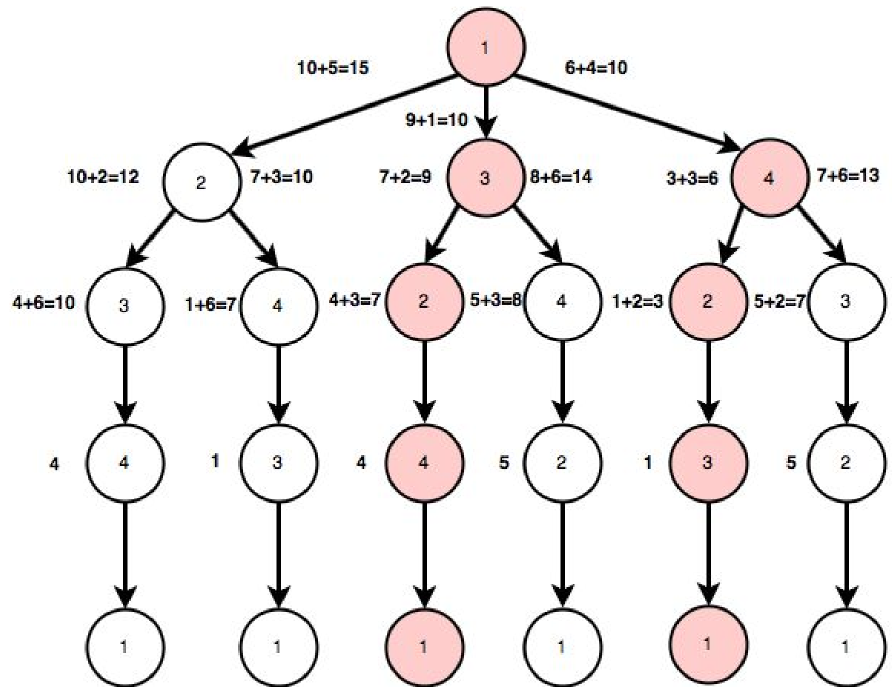
From vertex 1 there are three choices to go to: vertex 2, 3, and 4. If vertex 2 is chosen, there are two options to go to: vertex 3, and 4. Once one of them is chosen, the path must go to the other one and then finish back at vertex 1. This is best illustrated in the chart below:



The Held-Karp algorithm starts computing the distance for shortest path in the bottom up approach like so:



Once there is a decision to be made which path to take, we want to select a path with a lesser distance. For example, path 2-3-4-1 has a distance of 10+2=12 and path 2-4-3-1 has a distance of 7 + 3 = 10. Therefore, path 2-4-3-1 should be chosen. Working its way up, the algorithm would detect the following two shortest paths:



1.1.2 Pseudocode

```
HeldKarpTSP (G: graph, n: number)
    let S = new hashmap
    //base case, solve distances from one to all other vertices
    for i in range 2..n
        S[1i] = getDistance(G, 1, i)

    for i in range 2..n
        let set = G[i..n]
        for j in set
            S[1, i..j] = min(getDistance(G, i, j), getDistance(G, i, j))

    return S[1ij]
```

1.2 K-OPT Heuristic

1.2.1 Description

K-OPT is a local search algorithm that removes K edges from a graph and then exchange/swap edges comparing each possible combination and seeing if that particular combination produces a better result. Variants of K-OPT, 2-OPT and 3-OPT, produce relatively fast and accurate approximations, while not taking a relatively long amount of time to solve. As the value K gets larger than 3, it can become very time consuming and costly to get results. Euclidean cases of TSP, which is problem presented, can be:

K-OPT algorithms can be improved by including other approximation algorithms to form an initial tour of the problem. Quick heuristic algorithms that may not give very close approximation when run independently can help K-OPT performance by quickly producing a tour that can improve the quality of edge swaps between nodes (cities/locations). Such combinations with K-OPT could be Nearest Neighbor, Prim's algorithm, minimum spanning tree and many more.

There are many algorithms such as Lin-Kernighan (LK), Lin-Kernighan-Helsguan (LkH), LKH-1 and LKH-2 which use a varying-OPT approach. This means that throughout the algorithm different numbers of edge swaps are made depending on the current situation in the problem being solved. This is handled by a complex algorithm with many decisions. The pseudocode below will more specifically cover a 2-OPT approach used in combination with some sort of quick heuristic algorithm, such as nearest neighbor, that initializes the tour.

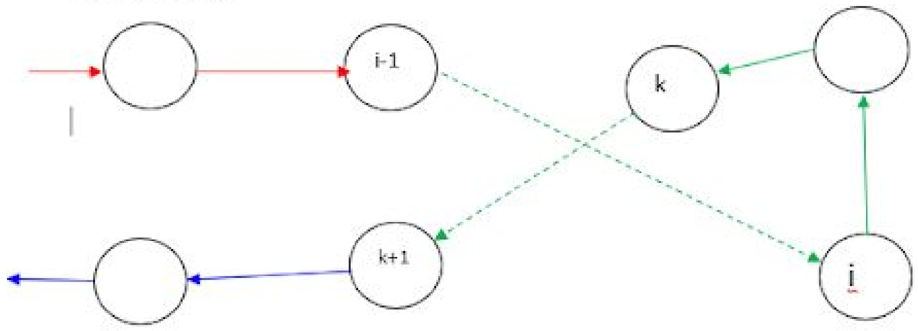
Possible Benefits of doing 2-OPT combination with another quick algorithm:
Should produce close approximations in a moderate amount of time. During testing if we find that it takes too long, then we have two separate algorithms for solving the Traveling Salesperson Problem (TSP) with only slight modifications to our code. By themselves the 2-OPT and whatever other paired algorithm we choose should not have any issues producing results in the 3-minute time limit. The results most likely will be worse but will at least yield results before the time limit is reached.

Overview of Steps [Reference 1]:

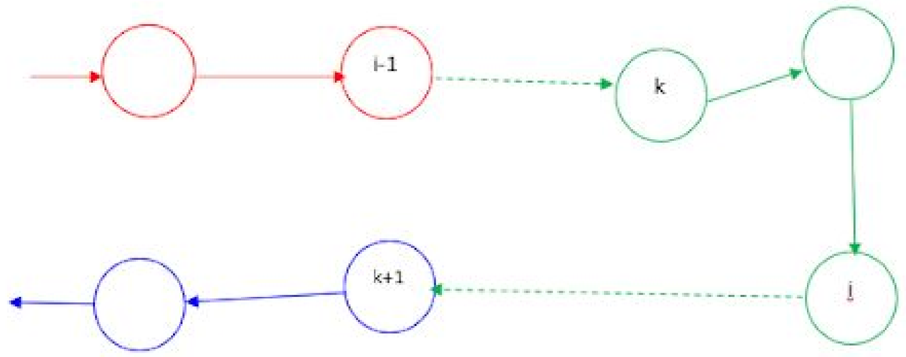
1. Parse Data
2. Create a full tour using a quick heuristic algorithm such as nearest neighbor, Prim's algorithm, etc..
3. Run k-opt algorithm
    a. Removes k edges from current tour
    b. Complete the tour by swapping/exchanging edges
    c. Select the best combination
    d. If this combination is less than current than change the current tour to reflect the edge exchange.

General Illustration of the Swap:

Before the Swap:



After the Swap:



1.2.2 Pseudocode

```
// Reference: https://paginas.fe.up.pt/~mac/ensino/docs/OR/CombinatorialOptimizationHeuristicsLocalSearch.pdf
// Parse data files
// Run a quick heuristic algorithm to set-up a beginning tour
                    // Maybe nearest neighbor or some other quick algorithm

2opt(tour, n, tourNew){
  bool improvement = TRUE
  int size = n  // number of cities
  tourNew = tour  // newTour equals existing tour to start
  while (improvement == TRUE)
  {
          bestDistance = distance(tour)  // calculates the distance of the tour
          for(int i=0; i<size-1 ; i++)
          {
     for(int k=i+1; k<size; k++)
     {
           tourNew = 2optSwap(tour, &tourNew, i, k)
           newDistance = distanceTour(tourNew)
```

```
      }
           }

           if(newDistance < bestDistance)
           {
      tour = newRoute
      improvement = TRUE
           }

           else {improvement = FALSE}
  }
}

// 2-opt swap implementation
// Reference: https://en.wikipedia.org/wiki/2-opt
TOUR swap2OPT(tour, i, k)
{
   // take tour[0] to tour[i-1] and add them in order to tourNew
   for s=0 to s<=i-1
          tourNew.addCity(s)

   // take tour[i] to tour[k] and add them in reverse order to tourNew
   for s=i to s<=k
          tourNew.addCity(s)

   // take tour[k+1] to end and add them in order to tourNew
   for s=k+1 to s<tour.size
          tourNew.addCity(s)

   return tourNew
}

int distanceTour(tour)
{
  // Loop through the tour adding up the distance

  return tourLength;
}
```

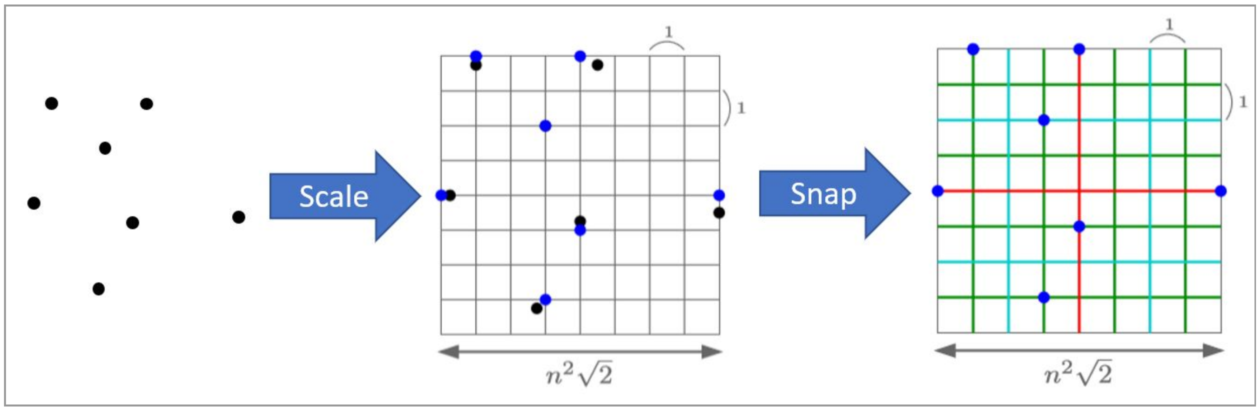
1.3 PTAS (Arora)

1.3.1 Description

Relatively recently a Polynomial Time Approximation Scheme (PTAS) for the special-case Euclidean TSP was developed by Princeton professor and theoretical computer scientist, Sanjeev Arora. The proposed algorithm advertises an arbitrarily close approximation to the Euclidean TSP with a running time that is $O(n\ (log\ n)^{O(c)})$ with an approximation that is within $\frac{1}{c}$ of the true optimal solution [1]. A PTAS algorithm is any that can approximate a true solution to within a factor of 1/c for some fixed c greater than 1 in polynomial time.For example, if c = 2 then this Euclidean TSP algorithm would guarantee a solution that is within 50% of the true solution (1 + ½)*OPT in a running time of $O(n\ (log\ n)^{2})$. To meet the requirements of the assignment we would have to compute a solution that is no worse than 25% more than the true optimal path (1 + ¼) so this algorithm would run in $O(n\ (log\ n)^{4})$, which is significantly better than exact solution algorithms that can't do better than $O(n^{2}\ 2^{n})$, such as the Held-Karp algorithm.

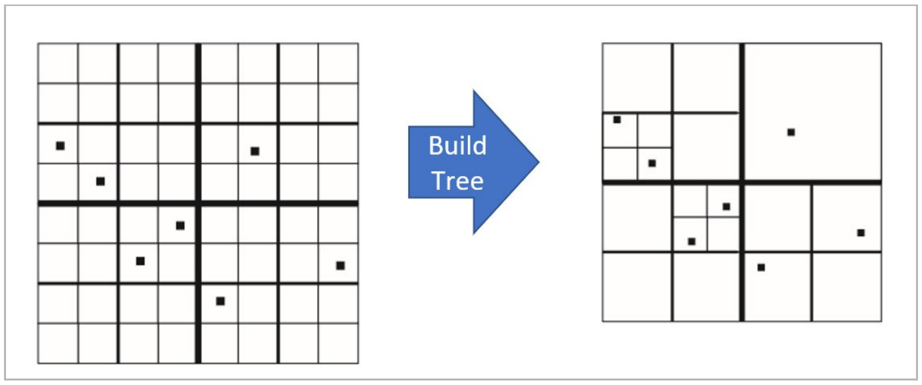
The technique involves building an approximate representation of the locations in a quadtree structure along with an approximate representation of possible paths, approximate in that they are restricted to a discrete set of 'portals' along quadtree grid lines through which they can pass. The first step in the algorithm is to transform the location data into a suitable input for the portal path optimization. This involves re-scaling the data to to ensure a minim inter-node distance and building a quadtree representation of the data by partitioning the plane containing the scaled locations. The algorithm performs best when an element of randomness is introduced, but for a cost of $O(n^4)$ a non-random version of the partitioning can be used. Figure 1.3.1.1 illustrates the concept of scaling and partitioning, representing the quadtree nodes as grid line intersections.

Figure 1.3.1.1 - Transforming Nodes



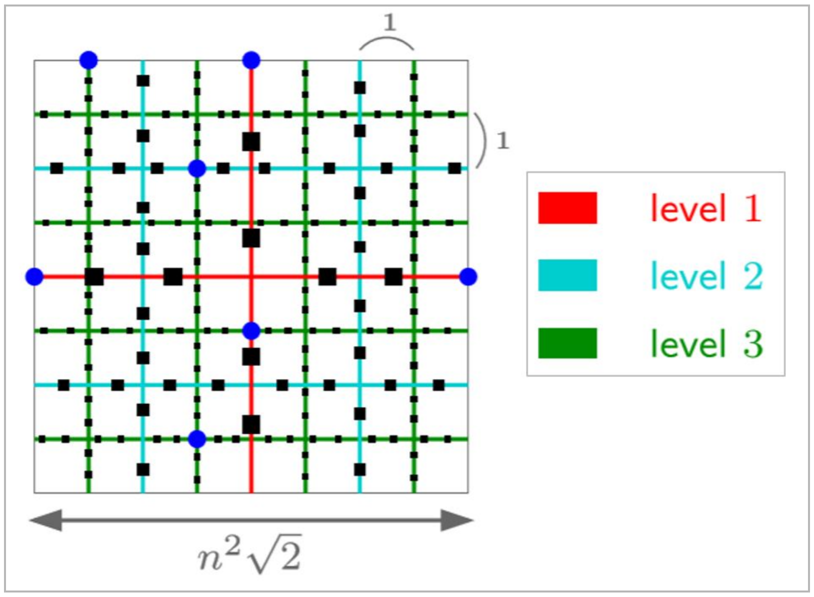
In an alternative representation, the true quadtree structure is shown in which each location is contained in a leaf node of the quadtree rather than lying on a grid line as shown above (see Fig 1.3.1.2). The partitioning continues until each single location is contained in its own quadtree leaf node.

Figure 1.3.1.2 - Quadtree Representation



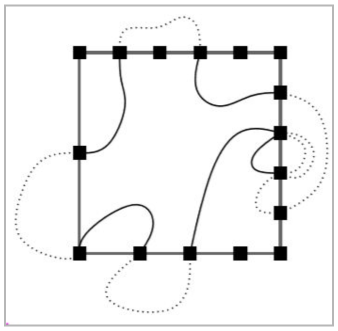
Once scaled and organized in a quadtree, the portals for approximate path construction must be created along the boundaries of the quadtree nodes.The number of portals along a boundary is dependent on the level of the quadtree at which the boundary lies since it relates to the relative density of neighboring nodes. Also, the density of the portals is dependent on the desired accuracy of the approximation. The portal spacing is inversely proportional to c, the constant described earlier that determines the approximation accuracy and the running time, with a coefficient equal to log n, where n is the number of nodes. The depth of the tree is log n, hence the dependence of the portal density on it. FIgure 1.3.1.3 below illustrates an example of portal creation.

Figure 1.3.1.3 - Portal Representation



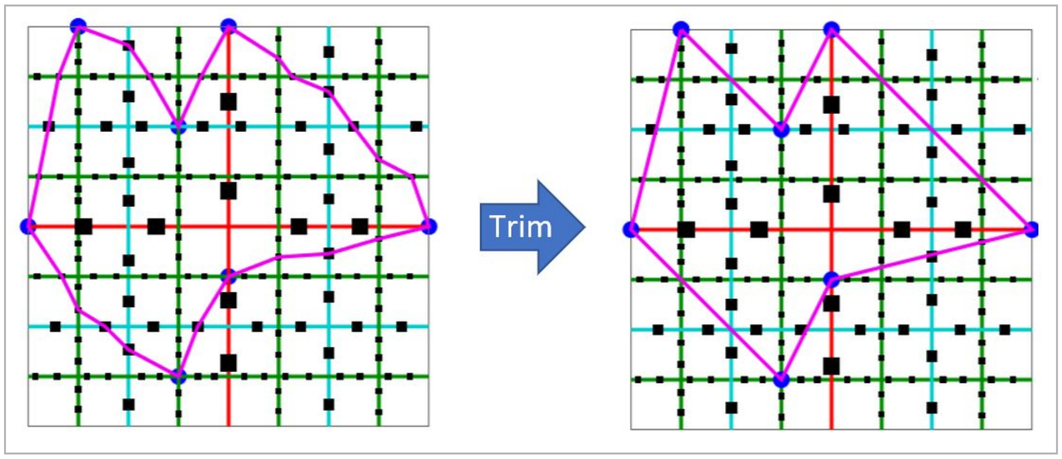
Building the approximately optimal path from the quadtree and strategically placed portals relies upon a bottom-up dynamic programming algorithm. Starting at the leaf nodes (smallest grid squares), the shortest '2-light' portal respecting tour is determined and tours at higher levels are created by combining optimal tours at lower levels. 'Portal-respecting' means that the edges of the tour must only cross grid lines at portals. '2-light' means that each portal can only be visited twice. The edges are also not allowed to self-intersect. Figure 1.3.1.4 illustrates the concept of a portal-respecting 2-light tour of a portion of the quadtree.

Figure 1.3.1.4 - Portal-Respecting Tour



When the dynamic programming algorithm reaches the top level of the quadtree (the entire bounding box of the scaled locations), the optimum combination of 2nd-level tours are selected to create an optimal 2-light portal-respecting tour that reaches all of of the locations. At this point, the paths are 'bent', as shown in Figure 1.3.1.5, so the final step is to trim the paths at all portals that are not locations. From these trimmed edges the total tour length can be computed, but it must be rescaled to undo the transformation that was applied to the original locations.

Figure 1.3.1.5 - Trimming the Portal-Respecting Path



There are many special situations that must be handled in an implementation of this method and complexities that are being glossed over due to both a desire to be brief and a lack of full understanding, but the issues include:

- Multiple locations from the original problem may map to single portals
- Shortcuts to finding the minimum portal-respecting path must be used
- The quadtree creation must have some randomization applied to improve the efficiency

Ultimately, this solution was deemed too complex and difficult to implement by our team. We were unable to find any helpful pseudocode and even the author of the research paper in which the method was presented acknowledged that their demonstration algorithm, for which no code was published, was quite slow and that it was probably due in part to an inefficient implementation. The bottom line is that this algorithm has a compelling theoretical basis, but is not nearly as practical to implement as other approximation schemes for the special Euclidean TSP. We also had little confidence that we would be able to implement a solution at all in the time available.

1.3.2 Pseudocode

```
// Compute a transformation of the problem instance
// Cities are given in array V
ComputeScale(V)
  n <- V.size
  L_Scaled = n * n * sqrt(2)

  // Compute the current bounding box size for scaling
  minX, minY, maxX, maxY <- +Inf, +Inf, -Inf, -Inf
  For i = 1 to n
    If V[i].x < minX then minX = V[i].x
    If V[i].x > maxX then maxX = V[i].x
    If V[i].y < minY then minY = V[i].y
    If V[i].Y > maxY then maxY = V[i].y
  L_Actual = max(maxX-minX, maxY-minY)
  Return L_Scaled/L_Actual

// Scale and snap locations to quadtree gridlines O(n log n)
// V is an array of cities.
ComputeGrid(V)
  G <- V
  // Snap points
  For i = 1 to V.size
      // Find nearest grid location
      G[i] = NearestGrid(V[i])
  Return G

// Generate portals at grid boundaries O(n (log n)^2)
// c is the approximation constant
```

```
CreatePortals(G, c)
  P <- Empty portal array
  //...Not sure on this and no example code could be found
  For i = 1 to 4 // Iterate quad tree levels
    For j = 1 to 2^i
       P.append(CreatePortal(G, i, j))
  Return P

// Use DP to build an optimal 'portal-respecting' path O(n (log n)^c)
// Grid (G), Portals (P)
ComputeBestPortalTour(G, P)
  // I really don't understand how this would be done
  Return T

// Compute real solution path length from unscaled locations O(n)
ComputeLength(T)
  // Trim out portals
  v = T.start
  d = 0
  While v.next != T.start
    // Trim edges until the next neighbor is not a portal
    While v.next is a portal
      v.next = v.next.next
    d += rnd(sqrt((v.next.x - v.x)^2, (v.next.y - v.y)^2))
    v = v.next

  Return d
```

2. Implementation

2.1 Design

After comparing our three algorithms of interest, we decided to implement the 2-OPT algorithm, using a nearest neighbor method of constructing an initial tour among the cities. The PTAS algorithm was far too complex and had too little online material available for our team to have confidence that a working solution could be developed in time to meet the project deadline. Also, it uses a variety of algorithms (quadtree operations, dynamic programming, portal selection) that we didn't fully understand how to implement. The Held-Karp algorithm looked promising in that it would give us an exact solution with much better efficiency than a simple brute force algorithm, but the running time would have been O($n^2 2^n$), which is still exponential and would likely have been too slow to solve the larger test and competition problems of thousands of cities. The 2-OPT was relatively straightforward to implement, produced results within the tolerances of our project specification and ran much faster Held-Karp.

After settling on our algorithm, we brainstormed the class objects, properties and methods that would be important in the implementation. We then converted the list into a basic UML diagram (see Fig. 2.1.1) and identified ways to group the objects to facilitate parallel code development. We managed the code with a git repository and worked first individually to develop the basic framework before working more collaboratively as we focused on the 2-OPT-specific portion of the code. See section 1.2.2 for the 2-OPT algorithm pseudocode. The high-level program pseudocode and nearest neighbor pseudocode is shown below.

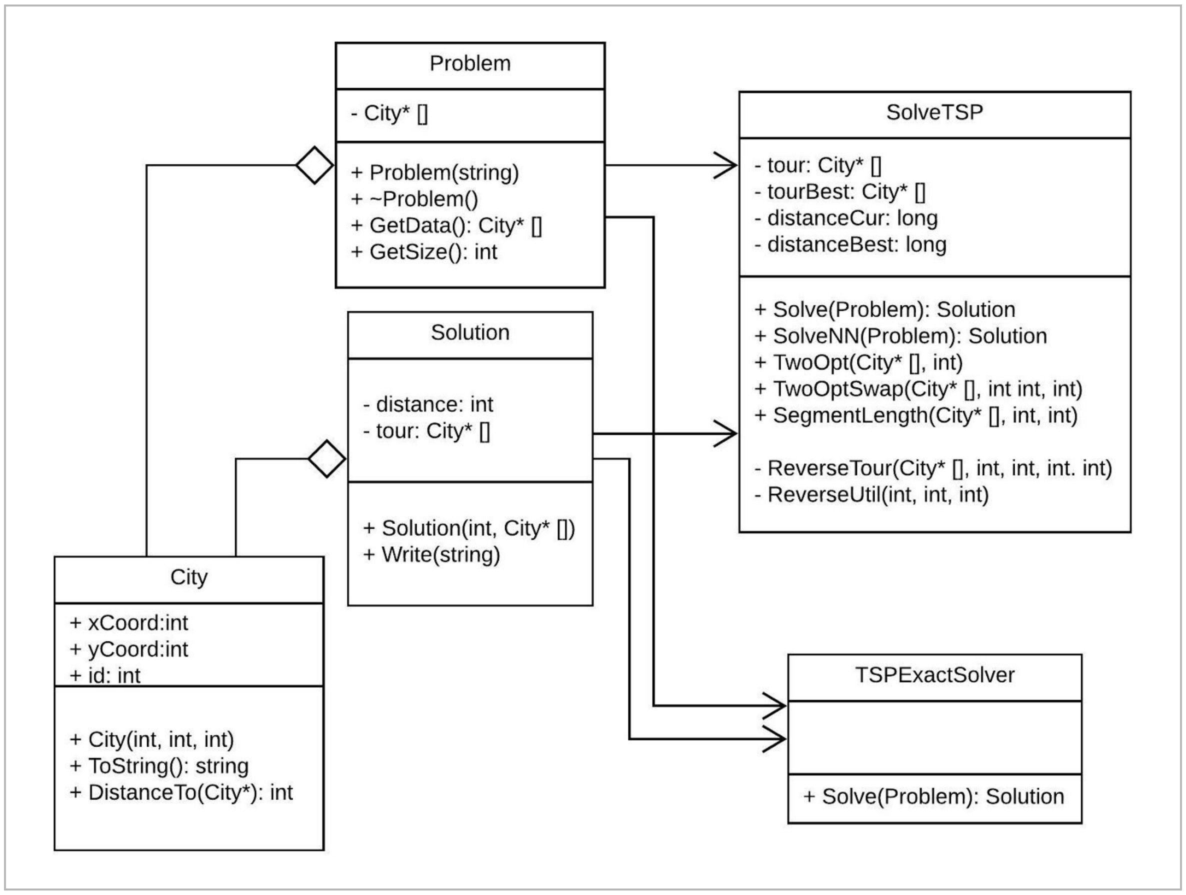
Pseudocode

```
Main()
  ReadProblem(P)
  If P.size < EXACT_SIZE
    S <- SolveExact(P)
  Else
    S <- SolveAppx(P)
  WriteSoln(S)

SolveExact(P)
  If P.size < 2
    Return Solution(0, P.cities)

  Int minDistance = -1
  Tour shortestPath

  For every permutation of P.cities
    Int currentPermutationDistance = 0;
    For every vertex in permutaion
      currentPermutationDistance += currentCity.distanceTo(prevCity);
    If minDistance == -1 or currentPermutationDistance < minDistance
      minDistance = currentPermutationDistance
      shortestPath = currentPermutation

    Return Solution(minDistance, shortestPath)

SolveAppx(P)
  T_Best <- GetNNTour(P)
  Do
    T_Curr <- TwoOptSwap(T_Best)
  While (T_Curr.dist < T_Best.dist)
  Return Solution(T_best.dist, T_best)

GetNNTour(P)
  currentCity = P.cities.at(randomIndex);
```

```
Tour nn
nn.push(currentCity)
currentCity.visited = true
totalDistance = 0
allVisited = false
While (!allVisited)
  allVisited = true
  nextNeighborDistance = -1
  For every city in P.cities
    If city.visited = true
      Continue

    potentialNeighborDistance = city.distanceTo(currentCity)
    If nextNeighborDistance == -1 or
      potentialNeighborDistance < nextNeighborDistance
        nextNeighbor = city
        nextNeighborDistance = potentialNeighborDistance
        allVisited = false

  If nextNeighborDistance != -1
    nextNeighbor.visited = true
    nn.push(nextNeighbor)
    currentCity = nextNeighbor
    totalDistance += nextNeighborDistance

Return Solution(totalDistance, nn)
```

We decided to implement the algorithm in C++ rather than Python to ensure that it executed as quickly as possible.

Figure 2.1.1 - UML Diagram



2.2 Initial Implementation

We first implemented the City, Problem and Solution classes. Subsequently, two team members worked on two different methods of building the nearest neighbor path while the third team member began implementing the 2-OPT algorithm. One team member implemented the nearest neighbor solution using a brute-force nearest neighbor algorithm with a running time of O(n^2). Another team member attempted to implement a 2-D tree to enable the construction of a nearest neighbor path in O($n\ lg\ n$), but the brute-force algorithm proved to be good enough for our time limits and was much simpler so we abandoned the more efficient method to focus on other details of the implementation. One team member also implemented a brute force exact solver to use on very small problems and a main function that branches to use our exact or approximate solver, depending on the number if cities.

In our first ‘working’ implementation of an approximate TSP solver we used a nearest neighbor path constructed starting from the first city read in from the file as the starting point. The 2-OPT algorithm then iterated through the list of cities, comparing edge swaps with every other city in the tour in $O(n^{2)})$ and accepted edge swaps if the entire length of the tour reduced. When the edges are swapped, the intermediate city ordering had to be reversed and this was originally done tentatively to compute the new tour length, then reversed again if the tour length increased. At this stage, we also were ignoring wrap-around from one end of our array representation of the tour to the other. We were able to find solutions for smaller numbers of cities, but the implementation was too slow to run test case 3 with over 15,000 cities in under 3 minutes.

2.3 Refinements

Programming collaboratively, we made several refinements to our algorithm that increased the speed, gave us shorter optimal paths and eliminated some bugs that were introduced by some of our efficiency improvements. Table 2.3.1 below outlines the changes and motivations.

Table 2.3.1 - Implementation Refinements

Change	**Motivation**
Modified the distance calculation in the City class	We were originally incorrectly truncating the inter-city distance rather than rounding as the specification dictated.
Check only the change in swapped edge lengths rather than the entire tour length change.	Only the local edge changes affect the total tour length so this increased execution speed without compromising the optimum.
Reverse the tour between swapped edge cities only if it improves the result.	Initially, every swap check caused reversal and re-reversal happened every time the swap was rejected, which was most of the time, The reversal now happens rarely and only if it is beneficial. This was required to meet the time requirements on test case 3.
Use a random city index to start the nearest neighbor tour construction, do it several times and retain the best solution as the starting point for 2-OPT.	Improves the accuracy of the solution, but could only be done a few times for large problems.
Run 2-OPT multiple times until the solution is no longer improved	This provides a more accurate result than running 2-OPT only once on most cases.
Set a limit of checking edges up to the next 2000 cities.	This sped up the run time for the larger problems, but did slightly affect the best distance found.

3. Results

*Note: no need to run unlimited on the Competition Tour results, since running the full 180 seconds would produce almost the same results. When 500 samples or less are used our algorithm converges on a smallest value every time.

Competition Tour Results - 3 Minute Time Limit		
Input File	*Distance*	*Time [sec]*
test-input-1.txt	5401	0.03
test-input-2.txt	7467	0.11
test-input-3.txt	12613	0.90
test-input-4.txt	17545	2.31
test-input-5.txt	24434	12.17
test-input-6.txt	34644	72.38
test-input-7.txt	54639	70.58

Example Tour Results - Unlimited Time		
Input File	*Distance*	*Time [sec]*
tsp_example_1.txt	110987	0.06
tsp_example_2.txt	2681	0.90
tsp_example_3.txt	1753583	237.20

4. References

[1] Arora, Sanjeev. “Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problem.” *Association for Computing Machinery, Inc.*, 1515 Broadway, New York, NY 10036, USA.

[2] https://graphics.stanford.edu/courses/cs468-06-winter/Slides/steve_tsp_ptas_winter.pdf

[3] https://www2.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html

[4]https://paginas.fe.up.pt/~mac/ensino/docs/OR/CombinatorialOptimizationHeuristicsLocalSearch.pdf