Group 11

Charles Bennett

Conrad Lewin

Vlad Predovic

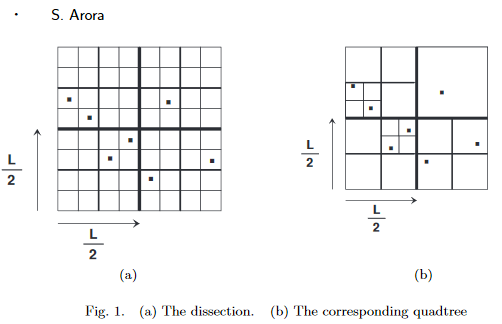
# CS 325 Project 4: The Travelling Salesman Problem

**Description of 3 methods/algorithms for solving TSP:**

**Algorithm 1: Euclidean TSP**

The Euclidean TSP approach uses the Arora-Mitchell approximation algorithm to solve the travelling salesman problem. The algorithm was discovered by Arora and Mitchell in 1998, and it can find the approximate value of a tour at an accuracy of 1+1/c, and has a running time of

n(log n)O((c)^(d-1)). The algorithm works using partitioning and dynamic programming. The plane is recursively partitioned with each tour crossing each line of the partition at most r = O(c) times. For each line in the partition, the algorithm predicts where the tour crosses the line and in what order the lines are crossed, then the algorithm recurses independently on each side of the line.



The plane is split using a randomized variant of quadtree. Each square is split into 4 smaller squares, and recursion stops when there is <= 1 node left in each square. After the squares are created, they are shifted, yet the points stay where they’re supposed to be. The same rules that apply to the quadtree also apply to the shifted quadtree, meaning there is a (1+1/c) tour that crosses each square O(c) times. The crossings happen at prespecified points, called portals. The salesman path visits all input nodes and some subset of portals (which can be visited more than once). Dynamic programming is then used to find paths that cross each edge of the quadtree a certain number of times, always crossing at portals. Dynamic programming generates a lookup table of all the costs of the optimal solutions to all instances of the problem, then chooses the best path.

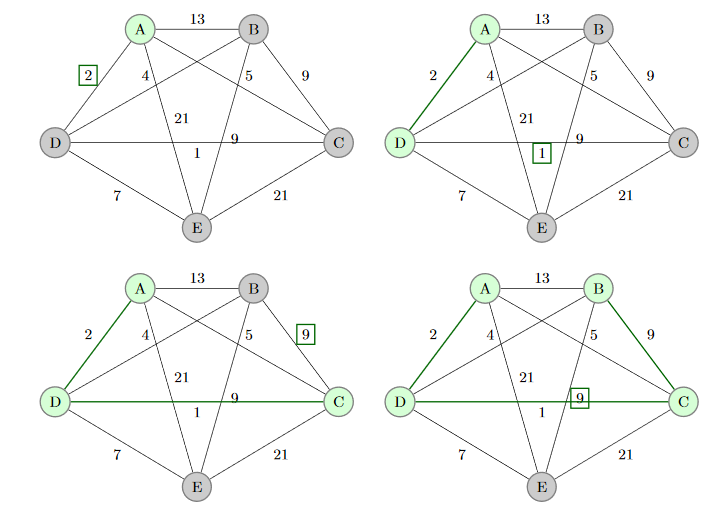
Sources: <http://graphics.stanford.edu/courses/cs468-06-winter/Papers/arora-tsp.pdf>

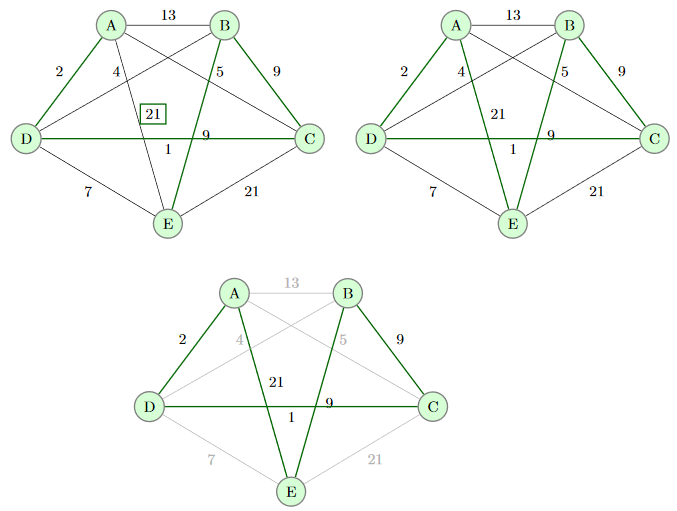
<https://open.kattis.com/problems/euclideantsp>

**Algorithm 2: Greedy**

The greedy nearest-neighbor algorithm is one of the easiest TSP algorithms to implement, and was one of the first to be used to solve TSP. The algorithm usually finds a reasonably short tour with a short running time, but it does not always find the optimal tour.

The algorithm starts at a vertex and sets it as the current vertex, then looks at all edges connected to the current vertex. It chooses the shortest edge connecting the current vertex a new vertex, then sets the new vertex as the current vertex. The previous vertex is added to a list of visited vertices, then the process repeats on the new current vertex. Once all vertices are visited, the path returns to the starting vertex and is complete, and the sequence of visited vertices is the result of the algorithm.



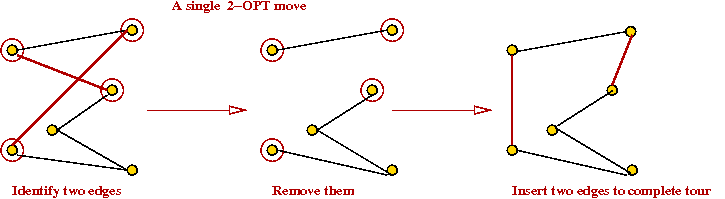


Sources: <https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm>

<http://cs.indstate.edu/~zeeshan/aman.pdf>

**Algorithm 3: 2-opt**

2-opt is a simple local search algorithm devised by Croes in 1958 for solving TSP. The algorithm consists of 3 steps. The first step is to use a greedy algorithm to find an initial solution that can be improved upon. The second step considers swapping two vertices in the initial solution, then finds the length of the new path. If the new path is shorter, the algorithm moves to step 3, and the new path is stored as the best solution. The process repeats until all swaps have been considered. Because each node can be exchanged with n-1 other nodes out of n total nodes, there are a total of n(n-1)/2 exchanges.



Sources:

<http://bardzo.be/4sem/NAI/rozne/Comparison%20of%20TSP%20Algorithms/Comparison%20of%20TSP%20Algorithms.PDF>

<https://en.wikipedia.org/wiki/2-opt>

<https://www.seas.gwu.edu/~simhaweb/champalg/tsp/figures/kopt2.png>

**Verbal description of our algorithm:**

We chose to implement a 2-opt algorithm to run the example instances. Our twoOpt algorithm takes two parameters, the map (a graph of vertices), and the source (the first item in the list of vertices). The algorithm first goes through each vertex in the list and makes an edge list of all the distances between each vertex. The edges are added together to find the minimum total edge weight. The starting solution is then created by listing out the vertices in order.

The twoOptSwap function is called in a while loop as long as 2-opt continues to improve on the starting solution. The twoOptSwap function’s parameters include the list of vertices, the starting solution, and the number of vertices. In subsequent calls, the starting solution is replaced with the current best solution.

During each iteration of the twoOptSwap, 4 nodes are compared. The coordinates of each node are gathered, and the distances between each node are calculated. If none of the vertices are duplicate, the improvement of the swap is calculated. If the improvement is larger than the previous largest improvement, then the new largest improvement is set, and the best move is recorded.

After all nodes have been examined, if there is an improvement on the initial solution, the new solution is returned and the status flag is set to true, indicating that the algorithm should continue through the while loop and keep trying to find a better solution. The algorithm stops when there are no improvements left to be made.

**Why we chose our algorithm:**

The algorithm was selected partially for its simplicity and also for the efficiency it had when dealing with large solutions. Given the time frame required for the solution to run in, we needed an algorithm that could rapidly improve a solution. Considering all edges of the graph gives the algorithm a running time of O(n2), where n is the number of vertices. The algorithm produces a much more accurate result compared to a simple nearest neighbor greedy algorithm due to the fact that it loops continuously, trying to find a better path.

**Pseudo code:**

def twoOptSwap(verticesList, currentSolution, numVertices)

biggest\_change = 0

best\_move = None

for node1 in range(numVertices)

for node3 in range(numVertices)

node2 = (node1 + 1) % numVertices

node4 = (node3 + 1) % numVertices

Anode1 = currSol[node1].getCords()

Anode2 = currSol[node2].getCords()

Anode3 = currSol[node3].getCords()

Anode4 = currSol[node4].getCords()

len13 = Distance between nodes 1 and 3

len24 = Distance between nodes 2 and 4

len12 = Distance between nodes 1 and 2

len34 = Distance between nodes 3 and 4

if (node1 != node3 and node2 != node3 and node2 != node4)

improv = (len12 + len34) - (len13 + len24)

if (improv > biggest\_change)

biggest\_change = improv

best\_move = [node1,node2,node3,node4]

if (biggest\_change > 0)

newSol = range(0, numVertices)

newSol[0] = currSol[best\_move[0]]

n = 1

while best\_move[2] != best\_move[1]

newSol[n] = currSol[best\_move[2]]

best\_move[2] = (best\_move[2] - 1) % numVertices

n += 1

newSol[n] = currSol[best\_move[1]]

n += 1

while best\_move[3] != best\_move[0]:

newSol[n] = currSol[best\_move[3]]

best\_move[3] = (best\_move[3] + 1) % numVertices

n += 1

return (True, newSol)

else:

return (False, currSol)

def twoOpt(theMap, source)

minWeight = 0

for i in xrange(len(theMap.vertList)-1):

theMap.edgeList.append(theMap.createEdge(theMap.vertList[i], theMap.vertList(i+1]))

for edg in theMap.edgeList:

minWeight += edg.getEdgeWeight()

currOrder = [x for x in theMap.vertList]

status = True

while (status=True)

(status, currOrder) = twoOptSwap(theMap.vertList, currOrder, len(theMap.vertList))

return makeTour(theMap, currOrder, “Greedy 2-Opt”)

**Best tours for the 3 example instances (and times to obtain the tours):**

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Tour Length** | **Run Time (seconds)** |
| 1 | 121232 | 0.87 |
| 2 | 2747 | 6.97 |
| 3 | infinity | infinity |

**Best tours for competition test instances:**

|  |  |  |
| --- | --- | --- |
| **Test Case** | **Tour Length** | **Run Time (seconds)** |
| 1 | 5635 | 0.7 |
| 2 | 7616 | 5.61 |
| 3 | 13436 | 106.88 |
| 4 | 18541 | 883.67 |
| 5 | 25396 | 8216.56 |
| 6 | infinity | infinity |
| 7 | infinity | infinity |