Val Chapple
Cody Dhein

TSP Project

Group #1

Cody Dhein
Val Chapple
(Group of 2)

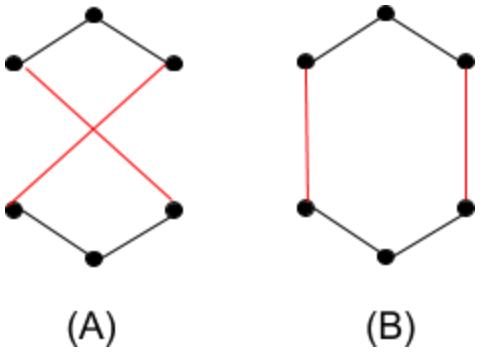
Algorithms

- 2-Optimization
- Minimum Spanning Tree implementation
- Nearest Neighbor Greedy with kd-Tree implementation

2-Optimization for Tour Improvement

Description

The idea behind 2-Optimization (2-Opt) is that we can take two existing edges in the graph, remove them, and then reconnect the two paths such that they become a single connected path again. This operation is performed only if the newly reconnected graph has an overall shorter distance than the original.



Graph (A) above is before the 2-Opt swap and graph (B) is after.

Two general methods for using 2-Opt as an improvement heuristic are 2-Opt First Improvement and 2-Opt Best Improvement.

2-Opt First Improvement (2-OptFI) performs a search for a 2-Opt edge swap that will reduce the overall distance and completes this swap as soon as it is discovered, then returning to the start of the graph and searching again.

2-Opt Best Improvement (2-OptBI) performs a similar search for edge swaps, but instead of performing the swap once an improvement is found, it instead searches the entire graph at each iteration and performs only the ‘best’ swap (the one that decreases the distance by the most), before starting again from the top.

Val Chapple
Cody Dhein

Most information on these two variations of 2-Opt was obtained from "First vs. best improvement: An empirical study" by Pierre Hansen and Nenad Mladenovic From this it is concluded that if the initial tour is chosen at random, 2-OptBI results in slower less optimal results on average compared to 2-OptFI. However the reverse is true if the initial tour is constructed through a 'greedy' constructive heuristic.

Discussion

Other improvement algorithms were looked at such as 3-Opt and k-Opt as well as other methods for improving the speed of 2-Opt. In the end 2-Opt was chosen due to time constraints and the complexity involved in some of the other methods. On average I was able to obtain results regularly within 1.10 times the optimal for samples 1 and 2. 3-Opt or k-Opt likely would have given us better results, but would have also increased the already large run time.

One method for decreasing that run time was found in "Heuristics for the Traveling Salesman Problem" by Christian Nilsson. In the article, Christian describes a method of speeding up a 2 or 3-Opt search by keeping a list of each nodes closest neighbors. The idea would be that given an edge (c1,c2), one would only look for potential swap edges where at least one node is a nearest neighbor to c1 or c2. There was an attempt to implement this, but I was unable to iron out the bugs in time for sufficient testing.

Both 2-OptBI and 2-OptFI are used in our final program.

Pseudo Code

```
twoOptSwap(route, i, j): # swaps edges (i-1,i), (j,j+1) with (i-1,j), (i,j+1)
 # route : current route
 # i, j : index of the nodes whose edges are being swapped
       newRoute = route[0:i] # copy route from 0 up to i (not including i)
       tmp = route[i:j+1] # copy route from i to j + 1 (incluiding j)
       tmp = tmp.reverse
       newRoute = newRoute + tmp
       tmp = route[j+1:end] # copy route from j+1 to the end
       newRoute = newRoute + tmp
       route = newRoute
```

Val Chapple
Cody Dhein

```
twoOptFI(path):
	changeAmt = -10

	while changeAmt < 0:
		!restart
		for i = 1; i < path.length - 2:
			for j = i+1; j < path.length - 1:

				# sum of the dist of edges (i-1,i) + (j,j+1)
				curSum = dist(path[i-1], path[i]) + dist(path[j], path[j+1])

				# sum if swapping i for j
				newSum = dist(path[i-1], path[j]) + dist(path[i], path[j+1])

				changeAmt = newSum - curSum

				if changeAmt < 0:
					twoOptSwap(path, i, j)
					GoTo !restart


twoOptBI(path):
	bestChange = -10

	while bestChange < 0:
		bestChange = 0

		for i = 1; i < path.length - 2:
			for j = i+1; j < path.length - 1:

				# sum of the dist of edges (i-1,i) + (j,j+1)
				curSum = dist(path[i-1], path[i]) + dist(path[j], path[j+1])

				# sum if swapping i for j
				newSum = dist(path[i-1], path[j]) + dist(path[i], path[j+1])

				testChange = newSum - curSum

				if testChange < bestChange:
					bestChange = testChange
					best_i = i
					best_j = j

		if bestChange != 0:
			twoOptSwap(path, best_i, best_j)
```

Val Chapple
Cody Dhein

Minimum Spanning Tree (MST) for Tour Generation

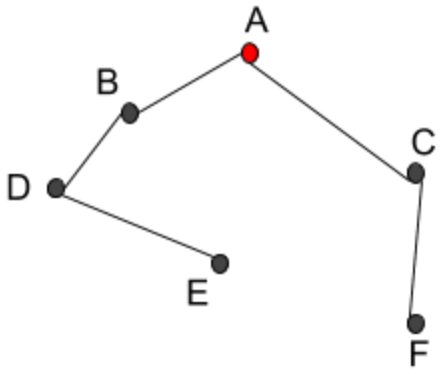
Researcher

Cody Dhein

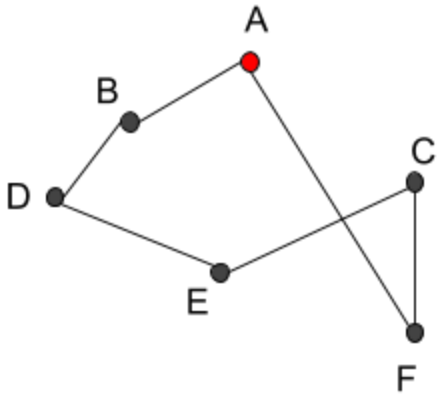
Description

A minimum spanning tree is (from Wikipedia) a subset of edges of a connected undirected graph that connects all vertices together with no cycles and the minimum possible edge weight. The fact that an MST touches each vertex and uses a minimum weight makes it a great starting point for creating an optimal TSP tour.

The idea is to use an algorithm to first build an MST and then perform a preorder traversal of the tree and add each node to the tour in the order they are visited.



Start node for MST = A.



Path generated via pre-order traversal of MST. A,B,D,E,C,F,A

Discussion

Several methods of building the MST were investigated. Prim's was used for the initial implementation before trying Kruskal's and attempting to implement Boruvka's. I found Boruvka's to be quite complex to implement and decided to forego attempting to implement it due to the realization that it probably would not yield better results.

Discussed in "State-of-the-Art Algorithms for Minimum Spanning Trees" by Jason Eisner is the indication that when it comes to sufficiently dense graphs, Prim's algorithm is optimal since it can freely disregard edges without having to sort them once the nodes are in the MST. Because of this I chose to stick with Prim's algorithm for MST creation.

My initial implementation of Prim's algorithm used Python's heapq module. The above paper and other investigation pointed to the fact that for sufficiently large and dense graphs, a fibonacci heap was actually faster. I was able to locate a module, fibonacci_heap_mod.py, that implements this data

Val Chapple
Cody Dhein

structure created by Keith Schwarz and Dan Stromberg. My hope was for this structure to speed up the route building process for the larger data sets, and though this may have been the case, I was unfortunately still unable to get tsp_example_3 to finish within 5 days.

Pseudo Code

```
buildRoute (graph, startNode):
	numNodes = graph.count

	preWalk = [] # will hold tour

	startNode.visited = True
	preWalk.append(startNode)

	# valid choices are all nodes connected to startNode
	initialChoices = graph.connectedTo(startNode)

	fibHeap = new Fibonacci Heap

	for choice in initialChoices:

		 # enqueue the startNode and choice as a tuple for the 'value' and the distance
		from start to that node as the 'priority'

		fibHeap.enqueue((startNode, choice) dist(startNode,choice))

	while preWalk.length < numNodes:

		# get minimum priority element
		selected = fibHeap.dequeue_min()

		fromCity = selected[0]
		toCity = selected[1]

		if toCity.visited == True:
			continue # discard result
		else:
			toCity.visited = True
			preWalk.append(toCity)

			newChoices = graph.connectedTo(toCity)

			for choice in newChoices:
				fibHeap.enqueue((toCity, choice), dist(toCity,choice))

	preWalk.append(preWalk[0])

	graph.tour = preWalk
```

Val Chapple
Cody Dhein

MST Results: Example Instances

	MST with 2-Opt
Example 1 (76 cities)	**Distance**: 108,323 **Time**: 1.02 sec
Example 2 (280 cities)	**Distance**: 2717 **Time**: 9.6 sec
Example 3 (15112 cities)	**Unable to complete**

MST Results: Competition Instances

	Unlimited Time	**3 Minute Time Limit**
Test 1 (50 cities)	**Distance:** 5373 **Time:** .16 sec	**Distance:** 5373 **Time:** .16 sec
Test 2 (100 cities)	**Distance:** 7411 **Time:** 1.17 sec	**Distance:** 7411 **Time:** 1.17 sec
Test 3 (250 cities)	**Distance:** 12371 **Time:** 23.73 sec	**Distance:** 12371 **Time:** 23.73 sec
Test 4 (500 cities)	**Distance:** 17385 **Time:** 109 sec	**Distance:** 17385 **Time:** 109 sec
Test 5 (1000 cities)	**Distance:** 24552 **Time:** 885 sec (=14.8 min)	**Unable to complete**
Test 6 (2000 cities)	**Unable to complete**	**Unable to complete**
Test 7 (5000 cities)	**Unable to complete**	**Unable to complete**

Val Chapple
Cody Dhein

Nearest Neighbor with kd-Tree Implementation

Researcher

Valerie Chapple

kd-Tree Description

The nearest neighbor algorithm examines any city to start and repeatedly selects the nearest city to be the next city visited. This continues until all cities have been visited. The tour distance, then, is the sum of the distances from one city to the next, with the last city connected to the start city to make a circuit.

The nearest neighbor algorithm can be implemented with kd-trees. The *k* in kd-trees is in reference to the number of dimensions used to analyze the data. For the provided TSP problem, the number of dimensions, k, is 2 (x and y). This is described as a kd-tree with 2 dimensions.

A basic kd-tree is constructed of nodes that have data with dimensions (that is, the city, city x-position, and city y-position), a left subtree, a right subtree, and the node's dividing dimension.

The dividing dimension is in reference to one of the k dimensions, such as x or y. The dividing dimension is important in determining how to divide the data between the left and right subtrees. The node itself is the median of the data in reference to the dividing dimension only. For example, if node A has a value of $x = 5$ and $y = 10$ and a dividing dimension of x, then all the data in the left subtree of A has an $x \leq 5$ and all the data in the right subtree of A has an $x > 5$.

The dividing dimension often changes at each level of the tree, usually iterating through all the dimensions on a loop. That is, a node with a depth of 1 splits on the dimension x, a node with a depth of 2 splits on the dimension y, a node with a depth of 3 splits on the dimension x, a node with a depth of 4 splits on the dimension y, and so forth.

The kd-tree can be used to find the nearest neighbor of the target city by traversing the kd-tree. To go from one node to the other, the current node's value for it's dividing dimension is compared the the target city's value for that same dividing dimension. The result of this comparison determines if the next node to examine is the left-subtree or the right-subtree. Additionally at each node, the kd-tree can be used to keep track of a number of nearest neighbors.

Since the data is cut at the median every level of the tree, it is possible that there exists a data point that is closest to the target data point but on a different subtree. A solution to this is to use what is called a hypersphere. This hypersphere in an x-y plane is simply a circle that is coplanar with the center of this circle at the target data point. The radius of this circle is the current nearest neighbor's distance from the target. If the hypersphere is contained within the current subtree, then no other subtree needs to be examined because either the nearest neighbor is the current nearest neighbor or is on the same subtree. If the hypersphere radius is enough to cross the median line, then the other subtree must be examined for the nearest neighbor. This hypersphere implementation was actually

removed in the final program, due to the implementation not correctly handling the radius comparison when the node being examined was the same as the target node. Therefore this kd-tree implementation assumes the second subtree is a viable option, but examines it as late as possible.

kd-Tree Discussion

I personally chose to research this algorithm because I gravitated towards using a tree representation, as most tree representations I have worked with have been faster than other algorithms (such as, a binary search tree and merge sort). I also was fascinated by the kd-tree because I had read many articles and done research of k-nearest-neighbors with machine learning, and once I realized that the kd-tree is a way to implement the nearest neighbor algorithm, I was very excited to learn more and try it out in this NP-Hard problem.

Additionally, I chose to implement this algorithm for our final project due to its speed, even with python. I found that the time to build a kd-tree is $O(n\ lg\ n)$ and this is much faster than building a two dimensional matrix of all the distances between each city, $O(n^2)$. Also, the search for the nearest neighbor after constructing the kd-tree is close to $O(lg\ n)$, assuming a constant number of dimensions. This rate increases dramatically though, as the dimensions, k, increases.

For unlimited time, the route determined by the kd-tree was then passed to a 2-opt algorithm to optimize the circuit. However, this python implementation with the 2-opt algorithm takes hours to run for data sets greater than 1000.

kd-Tree Pseudo Code

To implement a kd-tree for constructing a route for the travelling salesman problem, three key concepts/processes are required: the anatomy of a kd-tree node, the recursive construction of a kd-tree, and the repeated searching for nearest neighbor in the kd-tree.

Anatomy of a kd-tree node with 2 dimensions

Input:

- data = x and y positions and additional information, such as city ID
- dim = the dividing dimension of this node (x or y)
- left = pointer to another kDNode
- right = pointer to another kDNode
- mark = boolean for removing a node from a search. Default is False

Output: initialized node for kd-tree

```
kDNode( data, dim, left, right, mark):
       Return constructed kd-tree node with given initial values
```

Val Chapple
Cody Dhein

Recursive construction of a kd-Tree with 2 dimensions

Input:
- data list that contains x and y and additional information, such as city ID.
- depth is an integer of the depth of the recursion tree.

Output: node of a kd-tree.

```
kDTree( data, depth ):
	if length of input < 1
		return empty tree
	if depth is even:
		data = sort( data by x dimension )
		dim = x
	else
		data = sort( data by y dimension )
		dim = y
	midIndex = median( data )
	leftData = data[0] to data[midIndex]
	rightData = data[midIndex+1] to end
	return kDNode with
		data = data[midIndex],
		dim = dim,
		left = kDTree( leftData, depth + 1 ),
		right = kDTree( rightData, depth + 1 )),
		mark = False
```

Construct TSP route using nearest neighbor with kd-Tree with 2 dimensions

Input:
- root a valid kd-tree root node to search
- n integer of the number of nodes in the tree

Output: route as ordered list of nodes of tree for route to be taken, starting at root node

```
kDTreeRoute( root, n ):
	target = root
	route is empty array of nodes
	append root to route and mark root as visited
	while length of route < n
		Q is empty min priority queue
		bestDist = infinity (or max value)
		bestNode is null
		add root to Q with key=0
		while Q is not empty:
			node = removeMin( Q )
			if node does not exist:
				continue while loop
			dist = distance from target to node
			if dist < bestDist & node is unvisited
				bestDist = dist
				bestNode = node
			dim = dimension of node (x or y)
			if value at target[dim] <= value of node[dim]:
```

```
                        add node's left subtree to Q with key=0
                        add node's right subtree to Q with key=dist
                  else
                        add node's right subtree to Q with key=0
                        add node's left subtree to Q with key=dist
            end while loop
            if bestNode exists
                  append bestNode to route
                  mark bestNode as visited
                  target = bestNode
      end while loop
      return route
```

kd-Tree Results: Example Instances Unlimited Time

	Unlimited Time kd-Tree with the 2-Opt algorithm
Example 1 (76 cities)	**Distance**: 121471 **Time**: 0.757 sec
Example 2 (280 cities)	**Distance**: 3034 **Time**: 42.449 sec
Example 3 (15112 cities)	**Distance**: 1924453 **Time**: 21 hours

kd-Tree Results: Competition Instances

	Unlimited Time kd-Tree with the 2-Opt algorithm	**3 minute Time Limit** kd-Tree without optimization
Test 1 (50 cities)	**Distance**: 5630 **Time**: 0.408 sec	**Distance:** 7300 **Time:** 0.01737 sec
Test 2 (100 cities)	**Distance**: 8522 **Time**: 0.981 sec	**Distance:** 10195 **Time:** 0.0232 sec
Test 3 (250 cities)	**Distance**: 13646 **Time**: 25.1 sec	**Distance:** 17790 **Time:** 0.0453 sec
Test 4 (500 cities)	**Distance**: 18402 **Time**: 205 sec (= 3.4 minutes)	**Distance:** 23230 **Time:** 0.0947 sec
Test 5 (1000 cities)	**Distance**: 25777 **Time**: 1810 sec (= 30.2 min)	**Distance:** 33902 **Time:** 0.248 sec
Test 6 (2000 cities)	**Distance**: 35716 **Time**: 24600 sec (= 6.8 hours)	**Distance:** 46474 **Time:** 0.638 sec

Val Chapple
Cody Dhein

Test 7 (5000 cities)	**Did not complete**	**Distance:** 76037 **Time:** 3.21 sec

Final Program

The final program controlled which algorithm implement based on the number of cities in the problem. The MST was chosen for smaller data sets, and the kd-tree for larger data sets. For the 3-minute time limit the MST algorithm was used to calculate data sets smaller than 500, and for the unlimited time, the MST algorithm was used to calculate data sets smaller than 1000. All remaining data set sizes, up to 15000, were computed by the kd-tree and optimized if time allowed.

Best Results: Example Instances Unlimited Time

	Unlimited Time	**Algorithm**
Example 1 (76 cities)	**Distance**: 110096 **Time**: 0.5767 sec	MST with 2-opt
Example 2 (280 cities)	**Distance**: 2851 **Time**: 13.369 sec	MST with 2-opt
Example 3 (15112 cities)	**Distance**: 1924453 **Time**: 21 hours	kd-tree with 2-opt

Best Results: Competition Instances

Unlimited Time: MST for Tests 1 through 5 and kd-Tree with optimization for 6 and 7.
3 Minute Limit: MST for Tests 1 through 4 and kd-Tree without optimization for 5 through 7.

	Unlimited Time	**3 Minute Time Limit**
Test 1 (50 cities)	**Distance:** 5373 **Time:** 0.16 sec	**Distance:** 5548 **Time:** 0.1489 sec
Test 2 (100 cities)	**Distance:** 7411 **Time:** 1.17 sec	**Distance:** 7411 **Time:** 1.145 sec
Test 3 (250 cities)	**Distance:** 12371 **Time:** 23.73 sec	**Distance:** 12815 **Time:** 20.96 sec
Test 4 (500 cities)	**Distance:** 17385 **Time:** 109 sec	**Distance:** 17820 **Time:** 109.3 sec
Test 5 (1000 cities)	**Distance:** 24552 **Time:** 885 sec (=14.8 min)	**Distance:** 33902 **Time:** 0.231 sec
Test 6 (2000 cities)	**Distance**: 35716 **Time**: 24600 sec (= 6.8 hours)	**Distance:** 46474 **Time:** 0.632 sec

Val Chapple
Cody Dhein

Test 7 (5000 cities)	**Distance:** 76026 **Time:** 3.027 sec	**Distance:** 76026 **Time:** 3.027 sec