TSP Project Report

CS 325
Group 28
Kevin Choi
Erick Brownfield
Howard Chen

We researched 3 algorithms for solving the Euclidean TSP: Held-Karp TSP algorithm, the Triangle Inequality TSP algorithm, and the Christofides TSP algorithm.

Algorithm: Held-Karp algorithm

Source: *Algorithms* by S. Dasgupta, C.H. Papdimitriou, U. V. Vazirani, Mcgraw-Hill, 2006. See Chapter 6, 188-189.

Description: This is an exhaustive algorithm using dynamic programming. It breaks the original problem into $2^n n$ subproblems and then recursively solves them while recording the solutions. Each subproblem takes approximately n time to solve, resulting in $O(2^n n^2)$ time complexity. The solutions to subproblems are then used to solve the original problem. The space complexity is $O(2^n n)$, needed to record subproblem solutions. It is conjectured to approximate the solution to no worse than a factor of 4/3 more than the optimum.

While still a brute-force algorithm, it is a drastic improvement over a naive brute-force approach which is O(n!)

Pseudocode:

```
C = input with coordinates x and y
d = distance matrix of n x n

d[1, 1] = 0
for i = 2 to n
        for j = 2 to n
                d[i, j] = nearestint(root(square($C_i x - C_j x$) + square($C_i y - C_j y$))

S = all possible subset of cities containing origin
j = destination city, not including origin
A = 2d array of all subsets S by all destinations j

for 2 to n
        for each S of size n
        for each j in S
            $A[S_i, j]$ = min(A[S – {j}, k] + d[j, k]]) where k != j and is in S
return min($A[S_n, j]$ + d[j, 1])
```

Textbook Algorithm: Triangle Inequality TSP

Source: The class textbook: *Introduction to Algorithms, 3rd edition*, by Cormen et al, MIT Press, 2009

Description: This algorithm finds an approximate solution to the Euclidean TSP by first computing a minimum spanning tree T of the graph G, and then using the preorder traversal of T to generate a Hamiltonian cycle of the graph G. This Hamiltonian cycle is the approximate solution, whose cost is within a factor of 2 of the optimum.

Pythonic Pseudocode:
G – Graph
c – cost function (weights) of edges

```
def approx2_TSP(G,c):
    r = G.root
    MST = MST_Prim(G, c, r)
    H = preorder_traversal(MST)
    return H

def MST_Prim(G,c,r):
    for each u in G.vertices:
        u.key = inf
        u.parent = None
    r.key = 0
    Q = min_priority_queue(G.V)
    while Q is not empty:
        u = extract_min(Q)
        for each vertex v that are adjacent to u:
            if v is in Q and c(u, v) < v.key:
                v.parent = u
                v.key = c(u,v)

def preorder_traversal(root):
    if root is None:
        return
```

```
nodes = Stack(root)
ret = []
while nodes is not empty:
    temp = nodes.pop()
    ret.append(temp)
    if temp.right is not None:
        nodes.append(temp.right)
    if temp.left is not None:
        nodes.append(temp.left)
return ret
```

Christofides Algorithm (modified)

Source: A paper provided by the professor:

Orlis, Ch., Kartsiotis, G., Samaras, N., Margaritis, K. (2011). "Two new variants of Christofides heuristic for the static TSP and a computational study of a nearest neighbour approach for the dynamic TSP", In Proc. 1st International Symposium & 10th Balkan Conference on Operational Research (BALCOR 2011), Vol. 2, 22–25 September, Thessaloniki, Greece, pp. 376–384.

The class textbook: *Introduction to Algorithms, 3rd edition*, by Cormen et al, MIT Press, 2009.

Description: To construct an approximate solution, this algorithm computes a minimum spanning tree T of a graph G. The algorithm then takes the complete subgraph G* consisting of the odd-degree vertices in T, and computes an *approximation* of the minimum weight perfect matching M on G*. The edges in M are then combined with the edges of T to form a multigraph H (multiple edges allowed). Finally, the algorithm finds an Euler tour of H and removes from H all repeated vertices, giving us the approximate solution H*. H* is within a factor of 3/2 of the optimum solution.

Pseudocode:

Input: Complete graph G = (V, E), and a weight function w, such that w(e) gives the weight of a particular edge e.

Output: a sequence S that represents the approximate solution to the traveling salesman problem.

```
Christofides(G, w):
    T = Kruskal-MST(G, w)
    E* = Complete-Graph-Edges(Odd_Vertices(T))
    M = Min-Perfect-Match(Odd_Vertices(T), E*)
    H = Combine(M, T.E)
    V* = Euler-Tour(G.V, H)
    S = Clean-up(V*)
    return S

//Returns the edges making up an MST of G
Kruskal-MST(G, w):
    A = { }
    for each vertex v in G.V:
        Make-Set(v)
    sort the edges of G.E into nondecreasing order by weight w
    for each edge (u, v) in G.E, taken in nondecreasing order by weight:
        if Find-Set(u) != Find-Set(v):
            A = A  \union { (u,v) }
    return A

//Returns the set of edges that, when combined with the vertices in V, make up a
//      complete graph.
Complete-Graph-Edges(V):
    A = { }
    for each u in V:
        for v in V:
            if (u, v) not in A and (v, u) not in A and u != v:
                A = A \union { (u, v) }
    return A

//Takes a set of edges T and determines which vertices have odd degree
Odd_Vertices(T):
    A = { }
    for each edge (u, v) in T:
        if u not in A and adj[u].length is odd:
            A = A \union { u }
        if v not in A and adj[v].length is odd:
            A = A \union { v }
    return A

//Returns a set of edges representing an approximate perfect matching
//      of the graph represented by V and E.
Min-Perfect-Match(V, E):
    Let v be an arbitrary vertex in V
    Unvisited = V - { v }
    C = ( v )
    last = v
    while Unvisited is not empty:
        let u be the vertex in Unvisited closest to last
        Unvisited = Unvisited - { u }
        C.append(u)
        last = u

    //  At this point, C is a tour over the graph represented by V and E
    PM_1 = { }
    total_1 = 0
    i = 0
    while i < C.length - 1
        total_1 = total_1 + distance(C[i], C[i + 1])
        PM_1 = PM_1 \union { (C[i], C[i + 1])  }
        i = i + 2

    PM_2 = { }
    total_2 = 0
    i = 1
    while i < C.length - 2:
        total_2 = total_2 + distance(C[i], C[i + 1])
        PM_2 = PM_2 \union { (C[i], C[i + 1]) }
        i = i + 2
    total_2 = total_2 + distance(C[0], C[C.length - 1])

    // Return the approximated perfect match with the lower weight
    if total_1 < total_2
        return PM_1
    else
        return PM_2

//Returns a multigraph represented by combining the edge sets E1 and E2 into a multiset
Combine(E1, E2):
    Let A represent a multiset
    A = E1 \union E2
    return A

// Returns an Euler tour of the graph represented by V and E,
//      where E is a set of edges that represents a multigraph
//      and where all vertices (represented by set V) have even degree
Euler-Tour(V, E):
    Let A be an empty queue.
    for each vertex v in V:
        let cur = v
        if adjacency[cur].length != 0:
            B = ( v )     //sequence with 1 term
            while adjacency[cur].length != 0:
                remove some adjacent vertex b from adjacency[cur]
                B = B.append(b)
                E = E - { (cur, b)   }
                cur = b
            A.enqueue(B)  //add the sequence to A

    //Now we have the sequences that make up the Euler tour
    base = A.dequeue()
    while A is not empty:
        cycles = A.dequeue()
        Insert-Into(base, cycles)
            // Insert-Into is a function that inserts a cycle into the base sequence
            //        at the first occurrence of cycle[0] in base
    return base

//Takes an Euler-tour S represented by a sequence of vertices and removes all vertices that
//        were visited more than once
Clean-up(S):
    A = { }
    P = ( )
    for each vertex v in the sequence S:
        if v is not in A
            A = A \union { v }
            P.append(v)
    return P
```

The Algorithm We Chose for Our Program

The algorithm we chose to implement was the modified Christofides algorithm that was described above. ***See the section above on Christofides for the verbal description and pseudocode of the algorithm we used*.**

We chose this algorithm primarily because it seemed like a good compromise between the textbook algorithm, a 2-approximation, and the Held-Karp algorithm, which is conjectured to be a 4/3 algorithm, but not proven, and which seems fairly complex. We thought that the Christofides algorithm would be fairly simple to implement, based on its verbal description, which is all we had at first, and we thought its performance as a 3/2 approximation was pretty good.

In practice it was actually rather difficult to implement, and we had to approximate the process of finding the perfect matching. In hindsight it may have been better to implement Held-Karp.

Runtimes

The table below shows the best tours we obtained for the 3 example instances and the 7 competition instances, as well as the process time for each tour. Performance was surprisingly bad for tsp_example_3.txt and test-input-7.txt. We were unable to meet the 3 minute time limit for test-input-7.txt.

Input	**Best Tour**	**Process Time (s)**
tsp_example_1.txt	130346	0.06
tsp_example_2.txt	3106	0.78
tsp_example_3.txt	1915435	1347
test-input-1.txt	6788	0.02
test-input-2.txt	8697	0.11
test-input-3.txt	14823	0.65
test-input-4.txt	20154	2.46
test-input-5.txt	28405	10.06
test-input-6.txt	39768	43.71
test-input-7.txt	62858	355.63