Christopher Rico, Gerry Ashlock, Greg Tsang
Group Project
CS 325
6/8/18

Traveling Salesman Problem (TSP)

The traveling salesman problem is a well known and well researched problem in computer science. It is classified as an NP-Hard problem and is the subject of many research papers and algorithms. If given a list of cities and the distances between them, the traveling salesman problem aims to find the shortest possible path that visits each city exactly once and returns back to the starting city.

Each of the three group members researched a different algorithm for solving, or approximating, this problem. Each of the group members wrote up a short but concise description of how their chosen algorithm works and provided pseudocode. Once the research was complete, we came together and decided on which of the three algorithms we wanted to implement to find the best – or close to the best – solution to the traveling salesman problem. Once the chosen algorithm was implemented, we ran it through some example TSP cases with known optimal solutions, and attempted for a ratio of 1.25 (experimental solution : optimal solution). Once the algorithm was fully implemented and producing solutions that were optimal enough, we then ran some TSP problems that did not have known optimal solutions for a class wide competition to see who could create the best algorithm for solving the traveling salesman problem.

Algorithm #1: Ant Colony Optimization

The ant colony optimization was originally created in 1992 as part of Marco Dorigo's Ph.D. thesis. Basically the algorithm mimics how ants forage for food in the real world. Ants foraging for food basically simplifies down to a shortest path problem (minimize the energy expended to get the food). As ants move, they produce pheromones which other ants can follow. As more ants use a path, the pheromone level increases on that path. Ants are more likely to follow trails that have higher pheromone levels than lower levels of pheromones. This results in autocatalytic behavior. As more ants follow the trail, the trails being increasingly more attractive to fellow ants. This "Positive Feedback Loop" is the baseline for how the algorithm actually works.

There are a few things that we need to adapt for this to work for our virtual ants. First, actual ants can become stuck in almost infinite loops when the path is too complicated. To overcome this, our virtual ants will be given some memory so they can track their trip and eliminate loops (if needed) once they have arrived at the end destination. The virtual ants will only lay pheromone on its return trip so that the solution is constructed on its way back and infinite loops are not created, since the loops were removed after getting to the destination. We will also allow the ants to lay proportional amounts of pheromone so that faster, better paths are prioritized over slower ones. We will also be "evaporating" a portion of the pheromone so that any new paths are given a chance to be searched. If a new better path is added after the ants have begun, that path is almost always ignored by real ants since the pheromone level of the current best path is too strong to ignore. The basic steps for this algorithm is laid out below.

1. Initialize all of the arcs with a uniform pheromone level
2. Place ants at the starting location
3. Ants progress forward by probabilistically selecting the next location based on the relative pheromone levels of the surround nodes along with the distance to those nodes
4. Eliminate loops in the path the ant followed
5. Retrace Steps
6. Updated the trail by “evaporating” a portion of the pheromone (to allow for new better paths to be found)
7. Lay proportional pheromone to the retraced paths
8. Loop or exit

Pseudocode for Ant Colony Optimization

```
Get Coordinates of the cities
For i = 1 -> MAX_ITERATIONS
    for j = 1 -> NUM_ANTS
        for k = 1 -> NUM_CITIES
            for m = 1 -> AVAILABLE_CITIES
                calculate probability of choosing city
            get random number and choose city based on probabilities
            move ant to chosen city
        Loop
        Calculate Length of path
    Loop
    Update pheromone based on evaporation and each ants path and path length
Loop
```

Sources:
https://www.youtube.com/watch?v=xpyKmjJuqhk
https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms
https://en.wikipedia.org/wiki/Travelling_salesman_problem

Algorithm #2: **Nearest Fragment Operator**

The Nearest Fragment (NF) Operator is a variation on the Nearest Neighbor (NN) constructive heuristic. The NF method is a greedy algorithm that examines edges by increasing weight, and connects the nearest unvisited city to a group (fragment) of visited cities, successively connecting cities until all have been visited while avoiding to:

- Close a tour while unvisited cities exist
- Add an edge that includes a city which is already linked to two edges

Because the goal of the TSP is to find a tour whose cost is minimal, the NF heuristic aims to examine and select edges that have the minimum weight possible – a greedy approach – while still meeting the constraints of the traveling salesman problem.

We chose this algorithm because while a NN greedy approach is straightforward to understand, the NF variation on the NN greedy approach has been proven to yield solutions that are much closer to optimal than a simple NN greedy approach. For example, in one comparison of NN/NF TSP implementations performed by *Krari et al*, the NF implementation yielded tour results that were, on average, 14.44% greater than the optimal solution (a ratio of 1.14). The NN implementation yielded tour results that were, on average, 24.15% greater than the optimal solution (a ratio of 1.24). Clearly, the NF approach has the potential to increase the optimality of a greedy heuristic while still maintaining the simplicity of one. This is ideal for our purposes, as we do not have access to massive optimization libraries or computing power, unlike other organizations who have set out to generate solutions for the TSP.

The Nearest Fragment heuristic algorithm is fairly straightforward. It begins with a list of all possible edges E, sorted in increasing order by their weight, and an empty set T to hold the edges that will comprise the tour. The algorithm will then:

1. Examine each edge in the list, starting with the smallest.
    a. If the number of edges in T is less than the number of cities AND the addition of the edge to T will create a circuit (i.e., if a subtour will be created):
        i. Do not add the edge to T.
        ii. Continue to the next edge.
    b. If the edge contains a city which is already connected to two other cities (i.e., if a subtour will be created):
        i. Do not add the edge to T.
        ii. Continue to the next edge.
    c. If the number of edges in T is equal to the number of cities AND the addition of the edge to the tour will create a circuit:
        i. Add the edge to T and stop examining edges.
        ii. Return T.
    d. Otherwise, add the edge to T and continue to the next edge.

Pseudocode for Nearest Fragment Heuristic

```
REQUIRE: Sorted set of all edges of the problem E.
    for each e in E do
        if (e is closing T and size(T) < n ) or (e contains a city already connected to two
        others) then
                    go to the next edge
        end if
        if e is closing T and size(T) = n
        then
            add e to T
        return T
        end if
        add e to T
    end for
    return T
```

Sources:
http://www.atgc-montpellier.fr/permutmatrix/manual/SeriationPPI.htm
https://users.cs.cf.ac.uk/C.L.Mumford/howard/Multi-Fragment.html
El Krari M., Ahiod B., El Benani B. (2017) An Empirical Study of the Multi-fragment Tour Construction Algorithm for the Travelling Salesman Problem. In: Abraham A., Haqiq A., Alimi A., Mezzour G., Rokbani N., Muda A. (eds) Proceedings of the 16th International Conference on Hybrid Intelligent Systems (HIS 2016). HIS 2016. Advances in Intelligent Systems and Computing, vol 552. Springer, Cham

Algorithm #3: **Christofides**

The Christofides algorithm is named after Nicos Christofides, who published it in 1976. The algorithm guarantees an approximation ratio of 1.5, but requires that all edges obey the triangle inequality, thus forming a metric space. Christofides can be further improved by performing pairwise exchange, or 2-opt, after an initial solution has been generated.
Christofides algorithm can be broken down into the following steps:

1) Find a minimum spanning tree T of G (can use Prim's algorithm here)

```
MST-PRIM (G, w, r)
    for each u ∈ G.V
        u.key = ∞
        u.parent = NULL
    r.key = 0
    Q = G.V
    while Q != ∅
        u = EXTRACT-MIN(Q)
        for each v ∈ G.Adj[u]
            if v ∈ Q and w(u, v) < v.key
                v.parent = u
                v.key = w(u, v)
```

2) Let O be the set of vertices with odd degree in T (O has an even number of vertices)

3) Find minimum cost perfect matching M for vertices in O

```
PERFECT-MATCH (O, G)
    while O != ∅
        v = O.POP()
        length = ∞
        for u ∈ O
            if w(u,v) < length
                length = w(u,v)
                closest = u
        G.ADD-EDGE(closest, v)
        O.REMOVE(closest)
```

4) Combine M and T to create multigraph H

5) Find Eulerian circuit in H

6) Convert Eulerian circuit to Hamiltonian circuit by skipping visited nodes

The 2-opt algorithm then removes pairs of edges and checks if there is a way to reconnect the resultant paths that creates a shorter circuit.

Sources:
https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://en.wikipedia.org/wiki/Christofides_algorithm
https://www.geeksforgeeks.org/travelling-salesman-problem-set-2-approximate-using-mst/
Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Algorithm Chosen: **Nearest Fragment Operator**

We decided to implement the nearest fragment operator algorithm to solve/approximate the traveling salesman problem. We decided to implement this algorithm because it was a pretty simple algorithm to implement, in comparison to the others, along with being a well known algorithm with many resources to help us along the way. The other two algorithms were deemed too difficult or too unconventional to be implemented practically. However, while this algorithm was fairly straightforward to implement, and yielded decent results on its own, it did not provide solutions with the required optimal ratio. We decided to add a pairwise optimization (2-opt) subroutine to the algorithm to bring the optimality of the solutions closer to 1.25. Below are the results from the three example problems with known optimal solutions along with the problems with unknown optimal solutions.

Example Problems:

File Name	Known Optimal Solution	Actual Solution Found	Ratio
tsp_example_1.txt	108159	131610	1.22
tsp_example_2.txt	2579	2919	1.13
tsp_example_3.txt	1573084	1705484	1.08

Competition Problems:

File Name	Actual Solution Found	Time to Find Solution (s)
test-input-1.txt	6169	0.0136
test-input-2.txt	8463	0.0425
test-input-3.txt	13702	0.2741
test-input-4.txt	18136	1.1177
test-input-5.txt	25282	5.0379
test-input-6.txt	34917	19.0999
test-input-7.txt	55345	153.3810