Ethan Takla
Kenneth Thompson
Adam Sunderman
Group 38

Project 3 Group 38: TSP

Research

1. *Genetic Algorithm*: Although this algorithm doesn't have complexity that we can find, it does a pretty good job at finding a reasonable answer for the traveling salesman problem. A genetic algorithm tries to mimic biological evolution by taking organisms, producing offspring from the fittest ones, and letting mutations occur from generation to generation. In this case, each of the "organisms" would be a single permutation in the set of possible solutions to the TSP problem. Every generation, routes can have "offspring" with each other (two new routes), and additionally the paths can be randomly mutated, just as DNA can change during reproduction due to copying errors, etc. Below is the general flow of our GA program

```
for i in 0..N
    population[i] = randomRoute()

evalPopFitness()
genElites()
population.clear()

while(!stopCriteria)

    while population.size() < N
        genOffspring()

    evalPopFitness()
    genElites()
```

The most difficult part of using a genetic algorithm for the TSP is choosing an effective method of producing offspring and mutations for the routes. If one were to simply do a simple two point crossover on two routes, their offspring wouldn't have valid routes. Similarly, if one were to randomly mutate a city in the route, the route would most likely not be a valid hamiltonian cycle anymore. Simple GA's do the aforementioned processes, and simply discard any invalid routes and keep the valid ones. However, this can lead to serious inefficiency as most of the routes will be invalid. An area of research has been dedicated to creating "specialized operators", which are offspring/mutation methods that *only* produce valid routes, or "chromosomes". Some of the most prominent methods are the Partially-Mapped Crossover (PMX) and Alternating-Position Crossover (AP), both of which have their flaws. The PMX method of offspring generation doesn't necessarily keep the traits of both parents intact to-well. For this assignment, inversion sequences of the routes will be used to create chromosomes that can be used in conventional crossover and mutation methods, a method proposed by Gokturk Ucoluk of the Middle East Technical University in Ankara, Turkey. Described in greater detail

later in this paper, Ucoluk found that if you convert routes to their respective inversion sequences, then perform crossovers on these sequences, valid routes will always be generated from the new inversion sequences. Below is the pseudocode, adapted from Ucoluk's paper that is used in our genetic algorithm solution.

```
//Chromosome to route generation
vector<int> pos(chrom.size(),0)

for i in chrom.size()..0
    for m in i to chrom.size()
        if pos[m] >= chrom[i] +1
            pos[m] += 1
    pos[i] = chrom[i] + 1

for i in 0 to chrom.size()
    route[pos[i]] -1] = i;

return route
```

```
//Inversion sequence/chromosome generation
for i in 0..route.legnth()

    chrom[i] = 0
    m = 0

    while route[m] != in
        if route[m++] > i
            chrom[i] += 1

return chrom
```

Essentially, a "chromosome" is the inversion sequence of a permutation of a route. Converting the routes into this form allows us to do simple crossovers and mutations before converting the chromosomes back into permutations. A pitfall of fitness-based genetic algorithms is that they can find a local minimum solution, not the global minimum. Different mutation rates, crossover rates, population sizes, etc. can be used to help it find the best solution, but purely fitness based searches usually can't find the optimal solution.

2. 2-OPT Heuristic:

The 2-OPT Heuristic is an optimization for the traveling salesman problem first developed by G.A. Croes in 1958. The algorithm takes picks two edges in a tsp solution and reconnects them effectively reordering the route. This requires that the algorithm compare every possible combination of the swapping mechanism. If the new route is less than the previous route, the new route is saved and two new edges are picked to swap and the algorithm runs again. If the new route is greater than the old route the two picked edges attempt to reconnect in every possible way that forms a valid route. If the two edges cannot be reconnected in any manner that produces a shorter route then two new edges are picked and the algorithm starts over. 2-OPT its family of optimizations employ a local search heuristic rather than a constructive one. The 2-OPT heuristic is a special case of the K-OPT heuristic. The K-OPT heuristic operates on a variable number of edges represented by K. Although K can be any number of edges K-OPT algorithms often are seen implemented as either the 2-OPT or 3-OPT variety.

Resources:
https://www.seas.gwu.edu/~simhaweb/champalg/tsp/tsp.html
https://en.wikipedia.org/wiki/2-opt

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

Nearest Neighbor (Greedy):

Nearest Neighbor is a greedy algorithm that follows a simple procedure for solving TSP. Nearest Neighbor starts with a randomly chosen city and it adds the nearest but not yet visited city to the tour. This is repeated until all cities are visited.

The algorithm steps are as follows:

Step 1: A vertex will be picked randomly as the current vertex.
Step 2: Shortest edge will be chosen that connect current vertex to the nearest unvisited vertex V.
Step 3: Current vertex is vertex V.
Step 4: Vertex V gets marked as visited.
Step 5: If all vertices in the domain are visited, then terminate the procedure.

The output of the algorithm is the sequence of all visited vertices. Based on research, this creates a short tour, but usually not an optimal one. Since it is greedy in nature, it can sometimes miss some shorter routes that can be detected with human insight. Therefore the chance is very likely that Nearest Neighbor will not find the optimal tour even if one exists. Generally, if the last few stages are comparable in length to the first stages, then the tour is somewhat reasonable. However, if the last few stages are much greater, then it's likely there are better tours that could be found. Usually for N cities randomly distributed on a plane, the algorithm on average yields a path 25% longer than the shortest path possible. There does exist many specially arranged city distributions which makes this algorithm give the worst results. To offset this downfall of the greedy approach we and many others implement Nearest Neighbor with an optimization algorithm, such as 2-OPT above.

Resources:
http://cs.indstate.edu/~zeeshan/aman.pdf
http://www.isaet.org/images/extraimages/IJCSEE%200101308.pdf
https://www.math.ku.edu/~jmartin/courses/math105-F11/Lectures/chapter6-part4.pdf
https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm#Example_with_the_traveling_salesman_problem

Description of Algorithms

1. *Genetic Algorithm*: In the context of this problem, each city has ID which is a natural number. It follows that each route is a permutation of the cities in {1,2,3,...,N}, and has a length of N. A route of 3 0 2 1 would travel from city 3, to 0, to 2, to 1.

Before the genetic algorithm (GA) is run, a population of n individuals must first be initialized with a random but valid route, and subsequently their fitness's evaluated. In the context of the problem, the fitness of a route would simply be the length of the route itself. Finding the euclidian distance between two nodes can be computationally expensive over many iterations, so a C x C matrix of pre-calculated distances is generated beforehand, where C is the size of the chromosome, or the number of cities in a route. To find the distance between cities a and b, one can simply address the matrix at [a][b]. Once the GA has been initialized, a user specified percentage of "elites" is found, meaning the top (for example 15%) individuals with the highest fitness are stored inside a reproduction pool. From this pool, a first parent is randomly chosen, which has a user specified probability (usually around 80%) of "mating" with another randomly chosen route (sending two offspring to the next generation). If it is not chosen for mating, it's genes are simply passed on to the next generation. Every time genes are passed to the new generation by either of the two aforementioned methods, each value in the chromosome has a user defined chance (we used 0.007) of "mutating", or changing to valid random value. This allows the GA to search new routes when crossover reproduction isn't enough. In our runs, we used a population size of 1000, mutation rate of 0.07, crossover rate of 0.85, elite rate of 0.15, and performed 10-point crossovers. The GA was considered converged when we saw the same fitness value for the fittest elite 40 times.

Generating an inversion sequence, or the "chromosome" for a route, isn't very intuitive, but is computationally simple when it comes to implementation. If one takes a position p_j in the inversion sequence, p_j will be equal to the number of elements that are greater than and to the left of city j in the route. So, for example, the route 3 0 2 1 would have an inversion sequence of 1 2 1 0. p_2, the third element in the inversion sequence, has a value of one, meaning there is one number to the left and greater than two in the route. The valuable quality of this representation is that p_j must be within the range of 0 and C – j, where C is the length of the route. When one does a crossover between two chromosomes (swap their values within a specified range), this property ensures that the offspring will always be valid inversion sequences, and thus will produce new valid routes. When applying a mutation, one can simply randomly choose a value in-between 0 and C – j, again ensuring a valid inversion sequence. Once crossovers and mutations have occurred, the chromosomes/inversion sequences are translated back into route permutations. Using this method, the offspring keep some characteristics of both parents.

2. *Nearest Neighbor (Greedy):* The typical nearest neighbor implementation will construct tsp routes based on a random start point. In our implementation we took a different approach. Instead of picking a start city at random, and possibly getting a less than optimal route, our nearest neighbor algorithm will build n valid routes starting from city n. When done the algorithm keeps the route with the lowest cost and then the 2-OPT heuristic attempts to shorten the routes total distance even further.

```
NearestNeighbor(vector cities, vector tour, int startCity){
        tour.push_back(startCity);
        best = infinity;
```

```
        while(cities.size() > 0){
                for(i=0; i<cities.length(); i++){
                        if(distance(tour.back(), cities[i]) < best){
                                best = i;
                        }
                }
                tour.push_back(cities[i]);
                cities.delete(i);
        }
}
```

3. *2-OPT Heuristic:* The 2-OPT heuristic we implemented takes a valid tour from Nearest Neighbor and attempts to build a better tour by reconnecting random edges. The algorithm picks two random city points, call them 'low' and 'high' since their positions in the data structure matter. It will reconstruct 5 new routes in the following orders and fashion.

New route 1 = startRoute[high] to startRoute.size() +
startRoute[low] to startRoute[high] +
startRoute[0] to startRoute[high]

New route 2 = startRoute[high] to startRoute.size() +
startRoute[0] to startRoute[low] +
startRoute[low] to startRoute[high]

New route 3 = startRoute[low] to startRoute[high] +
startRoute[0] to startRoute[low] +
startRoute[high] to startRoute.size()

New route 4 = startRoute[low] to startRoute[high] +
startRoute[high] to startRoute.size() +
startRoute[0] to startRoute[low]

New route 5 = startRoute[0] to startRoute[low] +
startRoute[high] to startRoute.size() +
startRoute[low] to startRoute[high]

If new route 1 is not an improvement then new route 2 is tried and so on until all five routes have been checked. The new route with the lowest tour length will be used as the new route and 2-OPT will re-run looking for more swaps on new indices.

```
twoOpt(vector tour, vector temp){
        while(improving){
```

```
                int pos1 = random_int();
                int pos2 = random_int();
                temp = new route 1;
                if(temp.tour_distance() > tour.tour_distance()){
                        temp = new route 2;
                        if(temp.tour_distance() > tour.tour_distance()){
                                temp = new route 3;
                                if(temp.tour_distance() > tour.tour_distance()){
                                        temp = new route 4;
                                        if(temp.tour_distance() > tour.tour_distance()){
                                                temp = new route 5;
                                                if(temp.tour_distance() > tour.tour_distance()){
                                                        continue;
                                                }
                                                else{
                                                        tour = temp;
                                                }
                                        }
                                        else{
                                                tour = temp;
                                        }
                                }
                                else{
                                        tour = temp;
                                }
                        }
                        else{
                                tour = temp;
                        }
                }
                else{
                        tour = temp;
                }
        }
}
```

Discussion

1. *Genetic Algorithm:* Although we implemented all three of our algorithms, the genetic one due to the fact it was a very different, interesting approach that showed promised and provided a great research opportunity. Compared to the k-opt methods, greedy, methods, etc., this is a completely

different framework and thought process.

2. *Nearest Neighbor (Greedy) Algorithm:* The choice to use the nearest neighbor was based on the simple fact that it can produce a relatively short tour in a reasonable amount of time. This allows more time to be able to optimize the path chosen by the greedy algorithm with 2-OPT.

3. *2-OPT Algorithm:* The 2-OPT algorithm was chosen because it has the potential to not only improve upon the greedy algorithm above but it could have modified any routes produced by other algorithms as well. The original plan was to implement more methods but due to time and problem complexity we were not able to implement any more methods of solving TSP. That said the 2-OPT algorithm was critical in 'helping' nearest neighbor find the best route possible, in some cases improving the shortest route by up to 20 percent.

Best tours for each algorithm

1. *Genetic Algorithm (Performance varied GREATLY on parameter tuning):*
    a. *Example 1: Route length: 139187, Duration 48.87 seconds*
    b. *Example 2: Route length: 8061, Duration 4934.910623 seconds*
    c. *Example 3: Too large of a problem for the GA. Could be possible with some optimization, but it was not able to solve this problem at all.*

2. *Greedy Algorithm w/ 2-OPT:*
    a. *Tsp_example_1.txt: Route length: 120378, Duration 180 seconds*
    b. *Tsp_example_2.txt: Route length: 2921, Duration 180 seconds*
    c. *Tsp_example_3.txt: Route length: 1964688, Duration 180 seconds*
    d. *Test-input-1.txt: Route length: 5475, Duration 180 seconds*
    e. *Test-input-2.txt: Route length: 7874, Duration 180 seconds*
    f. *Test-input-3.txt: Route length: 14352, Duration 180 seconds*
    g. *Test-input-4.txt: Route length: 19280, Duration 180 seconds*
    h. *Test-input-5.txt: Route length: 26517, Duration 180 seconds*
    i. *Test-input-6.txt: Route length: 39371, Duration 180 seconds*
    j. *Test-input-7.txt: Route length: 61460, Duration 180 seconds*

3. *Nearest neighbor*
    a. *Test-input-1.txt: Route length: 5926, Duration: .012 seconds*
    b. *Test-input-2.txt. Route length: 9503, Duration: .052 seconds*
    c. *Test-input-3.txt. Route length: 15829, Duration: .474 seconds*
    d. *Test-input-4.txt. Route length: 20215, Duration: 2.625 seconds*
    e. *Test-input-5.txt. Route length: 28685, Duration: 18.16 seconds*
    f. *Test-input-6.txt. Route Length: 40913, Duration: 131.68 seconds*