Project 5

Danny Foley
Computer Science and Engineering
Speed School of Engineering
University of Louisville, USA
danny.foley@louisville.edu

Introduction

In Project 5, the assignment was to compute the shortest possible route from the Traveling Salesperson Problem. The traveling salesperson problem is a common problem where it is representing an actual real-life issue where a salesperson would need to figure out the best way to travel to each city in the area so that they can reduce time and costs.

Approach

For this assignment a genetic algorithm approach and wisdom of crowds approach were to be implemented. The information to use was given in a text file (Figure 1). With this approach, there were to be two different settings for two different parameters used. In this project, the two settings changed were mutation method and population size. The project contains a crossover method and two mutation methods.

Crossover Method

The crossover method used is Davis' Order Crossover or OX1. In this method, a section of the population is taken out from parent 1 and replicated within a child, then the remaining information from parent 2 that is not the same as what is in the child will be replicated. This is done again for child 2, where the parents' roles are reversed. The crossovers would occur between a population in the top 10 percent and a population in the top 50 percent based on their respective total distance. See Figure 3 for a generalized visual and Figure 4 for the code implementation.

Mutation Methods

The mutation method is the swap mutation where two different nodes in a single populous are swapped around. See Figure 4 for a generalized visual and Figure 5 for the code implementation. There was a 33% mutations rate.

Genetic Algorithm

The main algorithm was used with an iteration cap of 100, meaning that after 100 generations of no better route found, the generations would no longer be created. After creating a random population of size fifty or hundred, the population would be sorted from best to worst and go under a crossover and mutation. This algorithm assumes that the best route is found once the iteration cap is met. The code for this algorithm is found in Figure 6.

Wisdom of Crowds Algorithm

The algorithm created uses the results from the genetic algorithm and processes through the top 20 percent of all routes returned from the genetic algorithm. The frequency of each edge is found and inserted into a 2D vector. Once the most frequent edge is found, then the chain begins and the most frequent edge for the city at the end of the chain is used until the entire chain is filled with the total number of cities available. Once the chain is full the chain is wrapped back to the starting city, creating the route. This code implementation can be seen in Figure 7.

Results

Random11.tsp				
	GA Time(ms)	GA + WoC Time(ms)	GA Distance	GA + WoC Distance
Run 1	3614	3624	351	407
Run 2	3725	3741	351	398
Run 3	3954	3969	351	404
Average	3764	3778	351	403

Random22.tsp				
	GA Time(ms)	GA + WoC Time(ms)	GA Distance	GA + WoC Distance
Run 1	8154	8213	424	456
Run 2	6791	6838	434	495
Run 3	8168	8228	445	613
Average	7704	7760	434	521

Random44.tsp				
	GA Time(ms)	GA + WoC Time(ms)	GA Distance	GA + WoC Distance
Run 1	32939	33123	631	920
Run 2	78380	78688	592	752
Run 3	39785	40048	595	827
Average	50368	50620	606	833

Random77.tsp				
	GA Time(ms)	GA + WoC Time(ms)	GA Distance	GA + WoC Distance
Run 1	49500	49933	1150	1598
Run 2	39785	40048	1164	1446
Run 3	67077	67485	1140	1410
Average	52121	52489	1151	1485

Random97.tsp				
	GA Time(ms)	GA + WoC Time(ms)	GA Distance	GA + WoC Distance
Run 1	79262	79906	1495	2134
Run 2	79540	80062	1587	2165
Run 3	65375	65918	1653	2092
Average	74726	75295	1578	2130

Random222.tsp				
	GA Time(ms)	GA + WoC Time(ms)	GA Distance	GA + WoC Distance
Run 1	354979	357284	3849	5249
Run 2	240564	242626	3963	6011
Run 3	342861	345156	3886	5641
Average	312801	315022	3899	5634

Discussion

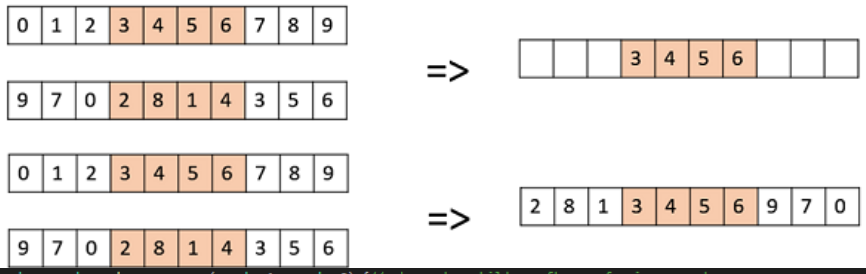
For all of the different outputs given by the program refer to Figures 8, 9, 10, 11, and 12. Figure 8 shows the final graphical solution found by GA, Figure 9 shows the final graphical solution found by GA + WoC, Figure 10 shows the Cost vs Generation graph for the solution when using GA, Figure 11 shows the final terminal output that includes the distance and the printed route with city numbers for both GA and GA+WoC.

The results were not expected. The assumption would have been that using this method of GA + WoC would bring about a better result that just using GA, however this is most likely due to the algorithm devised to create the resulting wisdom of crowds. If another, better, algorithm was used, then it is likely that the GA + WoC method would result in the best solution. Each time the number of cities increased, the difference between the average GA and GA+WoC results would also increase. The runtime difference between each method was about the same across all different numbers of cities, except for the case of the 222 cities problem.

Figures

```
NAME: concorde100
TYPE: TSP
COMMENT: Generated by CCutil_writetsplib
COMMENT: Write called for by Concorde GUI
DIMENSION: 100
EDGE_WEIGHT_TYPE: EUC_2D
NODE_COORD_SECTION
1 87.951292 2.658162
2 33.466597 66.682943
3 91.778314 53.807184
4 20.526749 47.633290
5 9.006012 81.185339
6 20.032350 2.761925
7 77.181310 31.922361
8 41.059603 32.578509
9 18.692587 97.015290
10 51.658681 33.808405
```

Figure 1




```
vector<graph> order_crossover(graph p1, graph p2) {//returns two children after performing an order crossover.
    vector<graph> c;
    graph c1;
    graph c2;
    int size = p1.all_cities.size();
    //------------------------------------------- create two crossover points
    int rdp1 = (rand() % size);
    int rdp2 = rdp1 + (rand() % (size-rdp1));
    //------------------------------------------- use the crossover algorithm OX1
    for (int i = rdp1; i <= rdp2; i++) {
        c1.cities_out.push_back(p1.cities_in[i]);
        c2.cities_out.push_back(p2.cities_in[i]);
    }
    for (int i = rdp2; i < size; i++) {
        if (c1.check(p2.cities_in[i].num)) {
            c1.cities_out.push_back(p2.cities_in[i]);
        }
        if (c2.check(p1.cities_in[i].num)) {
            c2.cities_out.push_back(p1.cities_in[i]);
        }
    }
    for (int i = 0; i < rdp2; i++) {
        if (c1.check(p2.cities_in[i].num)) {
            c1.cities_out.insert(c1.cities_out.begin(), p2.cities_in[i]);
        }
        if (c2.check(p1.cities_in[i].num)) {
            c2.cities_out.insert(c2.cities_out.begin(), p1.cities_in[i]);
        }
    }
    //------------------------------------------- make sure the children have everything else they need and ship them off
    c1.afterCrossover();
    c2.afterCrossover();
    c.push_back(c1);
```

Figure 3

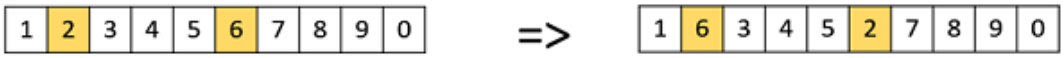


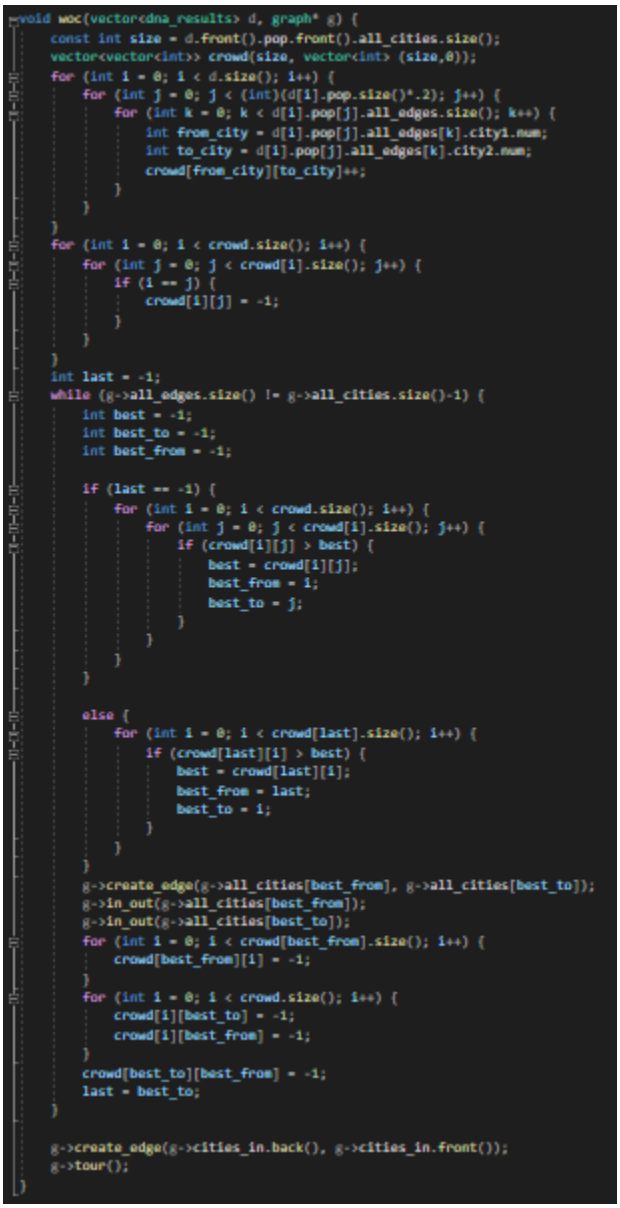
Figure 4

```
vector<graph> swap_mutation() {//goes through population and swaps two cities
    vector<graph> m;
    for (int i = 0; i < pop.size(); i++) {
        m.push_back(pop[i]);
    }
    for (int i = 0; i < m.size(); i++) {
        int rate = rand() % 2; //there is a 50% mutation rate
        if (rate == 1) {
            //cout << "MUTATING" << endl;
            int m1 = rand() % (int)(m[i].all_cities.size()-1);
            int m2 = rand() % (int)(m[i].all_cities.size());
            node mut;
            if (m1 == m2) {
                m2 += 1;
            }
            mut = m[i].cities_in[m2];
            m[i].cities_in[m2] = m[i].cities_in[m1];
            m[i].cities_in[m1] = mut;
            m[i].afterMutation();
        }
    }
    return m;
}
```

Figure 5

```
vector<dna_results> dna(FileInfo I, string mutate, string crossover, int iterations, int pop_size) {
    int gen = 0;
    int index = 0;
    long double best_dist = LLONG_MAX;
    vector<dna_results> dna;
    dna_results d;
    //------------------------------------ creates a population (of pop_size) of routes.
    for (int i = 0; i < pop_size; i++) {
        graph g;
        g.insertion(I);
        d.pop.push_back(g);
    }
    d.rank_tours();
    //------------------------------------ create new generations with a crossover and mutation until no progress is made for x generations
    while (index < iterations) {
        gen++;
        //cout << "Gen: " << gen << endl;
        index++;
        dna_results new_generation_c = crossover_method(d, crossover);
        dna_results new_generation = mutation_method(new_generation_c, mutate);
        d.pop.clear();
        d.pop.insert(d.pop.begin(), new_generation_c.pop.begin(), new_generation_c.pop.end());
        d.generation = gen;
        if (d.pop[0].dist < best_dist) {
            index = 0;
            best_dist = d.pop[0].dist;
        }
        dna.push_back(d);
    }
    return dna;
}
```

Figure 6



```
void woc(vector<dna_results> d, graph* g) {
    const int size = d.front().pop.front().all_cities.size();
    vector<vector<int>> crowd(size, vector<int> (size,0));
    for (int i = 0; i < d.size(); i++) {
        for (int j = 0; j < (int)(d[i].pop.size()*.2); j++) {
            for (int k = 0; k < d[i].pop[j].all_edges.size(); k++) {
                int from_city = d[i].pop[j].all_edges[k].city1.num;
                int to_city = d[i].pop[j].all_edges[k].city2.num;
                crowd[from_city][to_city]++;
            }
        }
    }
    for (int i = 0; i < crowd.size(); i++) {
        for (int j = 0; j < crowd[i].size(); j++) {
            if (i == j) {
                crowd[i][j] = -1;
            }
        }
    }
    int last = -1;
    while (g->all_edges.size() != g->all_cities.size()-1) {
        int best = -1;
        int best_to = -1;
        int best_from = -1;

        if (last == -1) {
            for (int i = 0; i < crowd.size(); i++) {
                for (int j = 0; j < crowd[i].size(); j++) {
                    if (crowd[i][j] > best) {
                        best = crowd[i][j];
                        best_from = i;
                        best_to = j;
                    }
                }
            }
        }

        else {
            for (int i = 0; i < crowd[last].size(); i++) {
                if (crowd[last][i] > best) {
                    best = crowd[last][i];
                    best_from = last;
                    best_to = i;
                }
            }
        }
        g->create_edge(g->all_cities[best_from], g->all_cities[best_to]);
        g->in_out(g->all_cities[best_from]);
        g->in_out(g->all_cities[best_to]);
        for (int i = 0; i < crowd[best_from].size(); i++) {
            crowd[best_from][i] = -1;
        }
        for (int i = 0; i < crowd.size(); i++) {
            crowd[i][best_to] = -1;
            crowd[i][best_from] = -1;
        }
        crowd[best_to][best_from] = -1;
        last = best_to;
    }

    g->create_edge(g->cities_in.back(), g->cities_in.front());
    g->tour();
}
```

Figure 7

Generation: 24
Distance: 351.045880

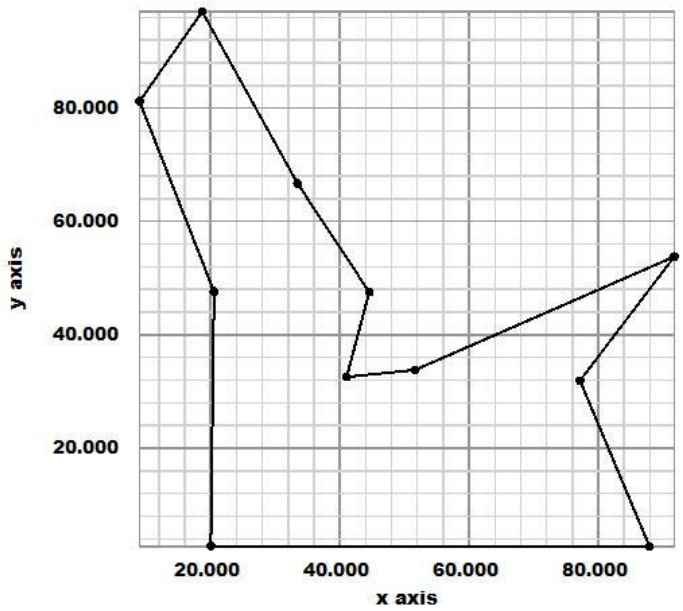


Figure 8

Wisdom of Crowds
Distance: 364.656306

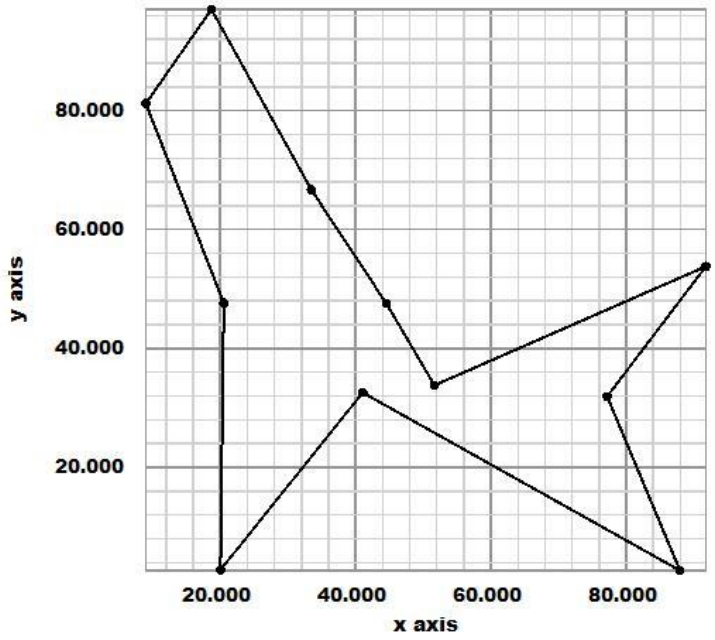


Figure 9

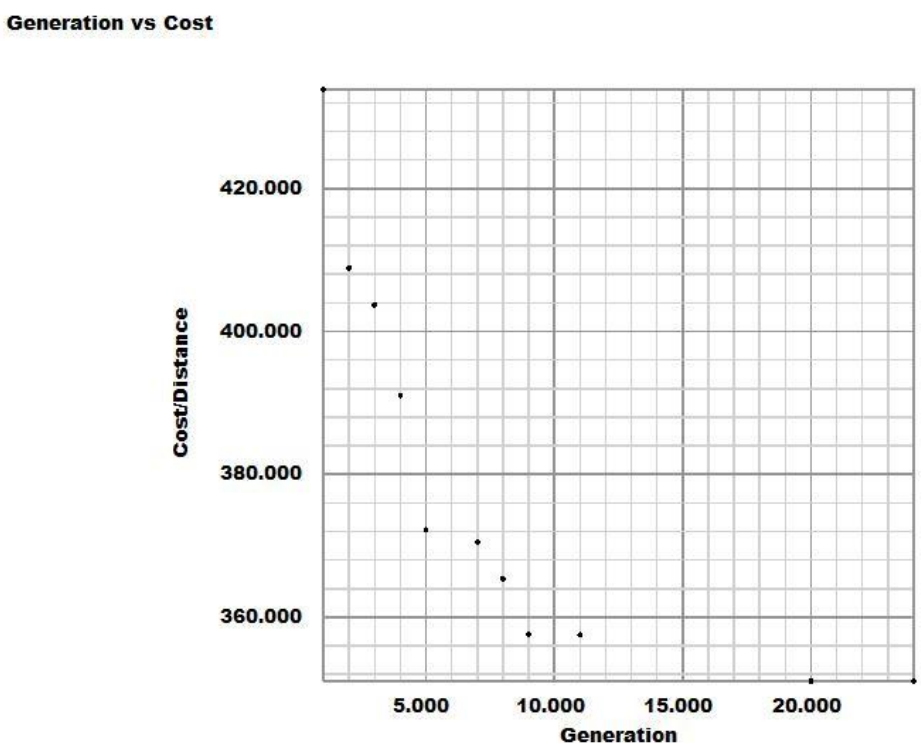


Figure 10

```
GA ONLY Elapsed Time in ms: 7426.00
Best GA Generation: 122
Best GA Distance: 351.046
Best GA Route:
11->8->10->3->7->1->6->4->5->9->2->11


GA + WOC Time in ms: 7442.00
WOC Distance: 366.262
WOC Route:
9->5->2->11->3->1->7->10->8->6->4->9
```

Figure 11

References

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm