Project 4

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

Introduction

In Project 4, 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 was 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 first mutation method is the swap mutation where two different nodes in a single populous are swapped around. See Figure 5 for a generalized visual and Figure 6 for the code implementation.

The second mutation method is the inverse mutation where a section of five nodes in a single populous is reversed in its order. See Figure 7 for a generalized visual and Figure 8 for the code implementation.

For both methods, there was a mutation rate of 50 percent.

Population Size

The population size was altered between trials to gauge the change it would have on the results, the two population sizes used were fifty (50) and one hundred (100).

Genetic Algorithm

The main algorithm was used with an iteration cap of 150, meaning that after 150 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 9.

UI

The user interface for this project is using an outside application “plot.exe”. After the genetic algorithm is complete, the populations of each generation that created a new all

time best route were saved and made into a graphical representation. These graphs were combined into a .gif file so that the user can see the progress made by the algorithm. A graph showing the cost vs generation is also created. The best route is also displayed as well as the output showing the best generation, distance, and total number of generations. See Figure 10, Figure 11, and Figure 12.

Results

The results are as follows:

	Average Time	Average Distance
POP 50 SWAP	63.75	1624.6275
POP 100 SWAP	65	1631.3825
POP 50 INVERSE	87	1515.81
POP 100 INVERSE	82.5	1740.1475

https://drive.google.com/drive/folders/1-j47yxuH9KRXL33WpJ_WdiltWikPy9x8?usp=sharing

This link above contains output data that the user would see.

Discussion

The results were not expected. The assumption would have been that having a higher population would have cause the results to be better. Based on the results found, it does not appear that anything really changes the results for the better. If more sample runs could be performed, there could be a trend found.

The biggest issue once again had nothing to do with the implementation of crossovers or mutations, the issue was just getting a graphical representation and finding a way to show the improvement in a visual way. Other than the UI issues, the only real issue was trying to come up

with another crossover method, which was not done. If this project could be repeated, another crossover method would be used as well as more mutation methods and iterations.

After having used GA, the ability to create a better population was inspiring. It felt like the algorithm could find the best route in some cases, like when using a smaller number of cities. Using a set of 30, rather than 100 cities produced a fairly accurate best route similar to that of the greedy approach. Using GA may be a necessity going forward in these projects if the final end goal is to get as close to the best route as possible.

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

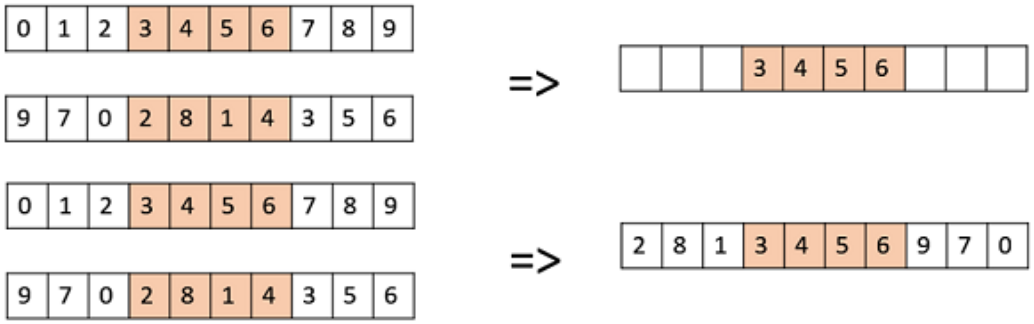

Figure 3

```
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);
    c.push_back(c2);
    return c;
}
```

Figure 4

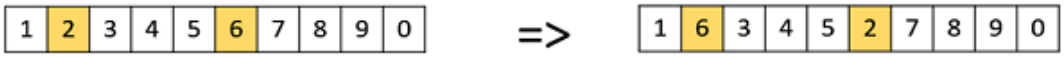

Figure 5

```
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 6

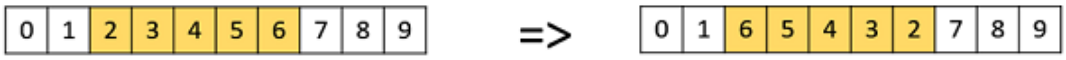

Figure 7

```
vector<graph> inverse_mutation() {//takes a section of link 5 in the tour and reveres
    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; //50% mutation rate
        if (rate == 1) {
            //cout << "MUTATING" << endl;
            int m1 = rand() % (int)(m[i].all_cities.size()-5);
            int m2 = m1 + 5;

            for (int i = 0; i < m.size(); i++) {
                vector<node> muts;
                for (int j = m2; j > m1; j--) {
                    muts.push_back(m[i].cities_in[j]);
                }
                int index = 0;
                for (int j = m1; j < m2; j++) {
                    m[i].cities_in[j] = muts[index];
                    index++;
                }
            }
        }
    }
    return m;
}
```

Figure 8

```
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 9

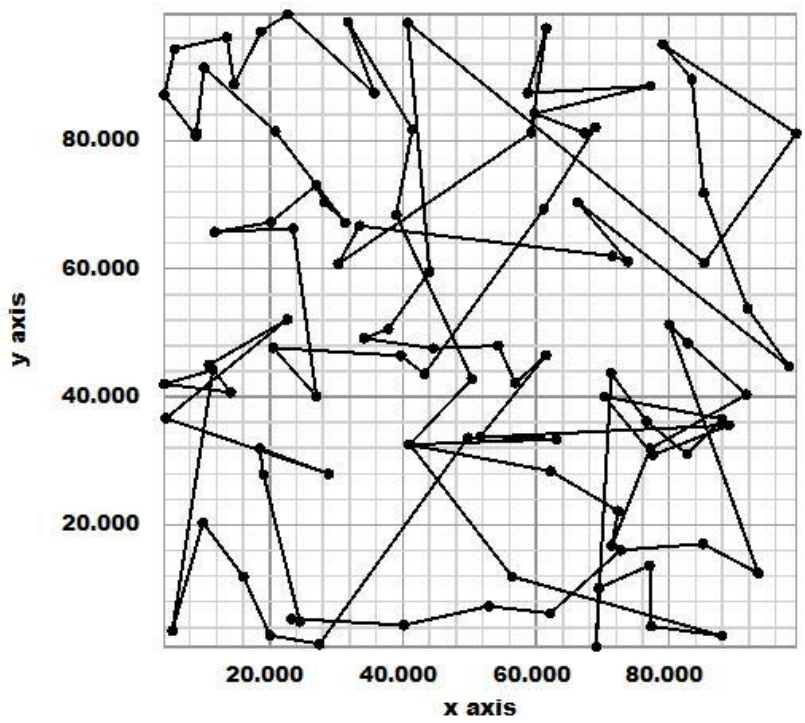

Figure 10

```
FULL FILE PATH: C:\Users\mathe\source\repos\TSP\TSP\Random100.tsp
Elapsed Time in seconds: 76.00
Total Generations: 463
Best Generation: 363
Best Distance: 1545.91
Best Route:
58->80->82->57->9->27->88->67->65->48->16->8->68->1->92->73->43->95->17->75->36->38->74->89->37->90->20->6->25->13->40->
64->31->33->59->35->84->51->29->56->98->60->81->79->15->30->24->71->41->39->66->7->72->85->83->54->94->10->93->70->99->1
1->18->12->49->22->53->91->32->63->19->3->21->34->23->86->2->87->96->45->52->28->100->55->61->46->69->47->4->97->77->62-
>26->50->14->76->78->44->5->42->58
Run the Python Code to Create a Gif, Once Created Then:
Press any key to continue . . .
```

Figure 11

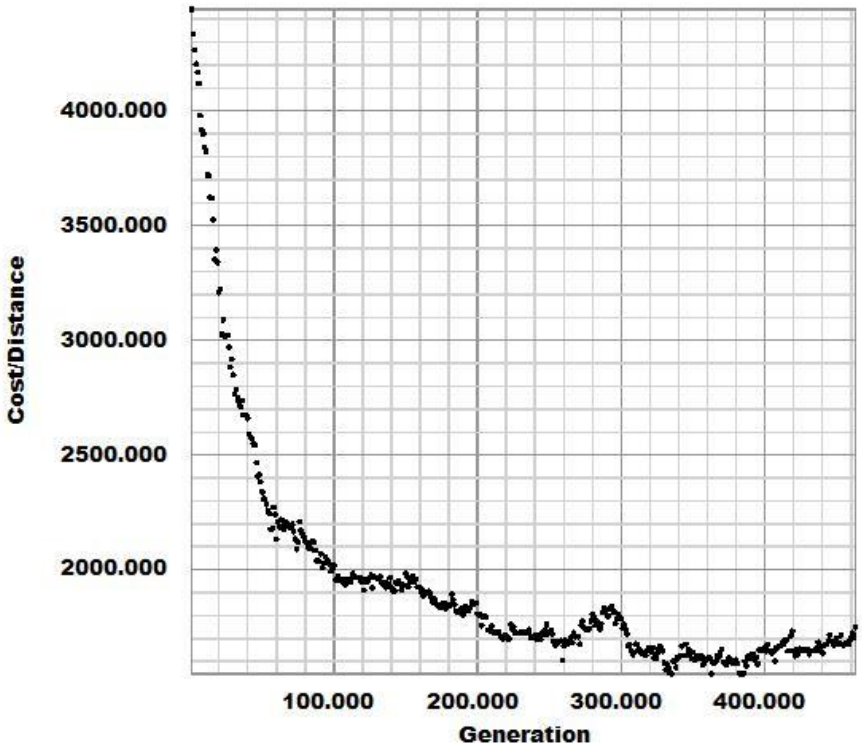
Generation vs Cost
Cost/Distance
4000.000
3500.000
3000.000
2500.000
2000.000
100.000
200.000
300.000
400.000
Generation

Figure 12

References

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

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