CSS534 – Project 1 – Solve the Traveling Salesman Problem using Genetic Algorithms and OpenMP

Nathaniel J. Grabaskas
Computing and Software Systems, University of Washington

Table of Contents

Section 1: Introduction

I had two initial goals: solving the Traveling Salesman Problem (TSP) using Genetic Algorithms (GA) and optimizing it for a Parallel Environment (PE) using OpenMP. For all examples I used a population of 36 cities and a generation size of 50,000 chromosomes. For each generation I take the fittest 25,000 chromosomes and use those to create an additional 25,000 offspring. This is done for 150 generations.

First, I started with implementing a Greedy Crossover (GX) and can be seen in section 2. This method proved to be acceptable with a speed-up around ~7 times and found a fastest route of < 449. Next, I moved onto an implementation of the Greedy Crossover using Nearest Neighbor (GXNN) and the results can be seen in section 3. In section 4, I discuss the methods and strategy to increase performance using parallelization through OpenMP. After that in section 5 I discuss some of the challenges that I had to overcome. In the conclusion I discuss the overall results achieved.

Section 2: Greedy Crossover (GX)

The GX takes the first city of parent[i] and finds the shortest connected city from both parent[i] and [i+1] and uses that city as the next city in the chromosome. If the city is the last city in the trip it then takes the first city as the connected city. See figure 1.

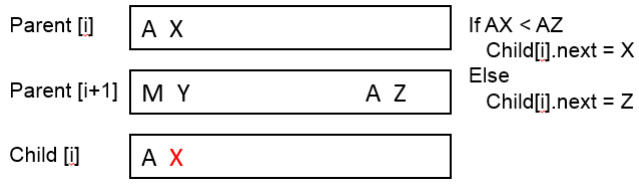

Figure 1: Shows the GX where distance from A to X is less than from A to Z. If both X and Z were already a part of child[i] a new city would be chosen at random.

This algorithm proved to run at ~7 times speed-up or an execution time of < 4M. Depending on the mutation rate this approach finds slightly different optimal routes. A mutation rate of 40 was found to be the best rate and returned a trip with 447.638 distance. Output and code can be seen in Appendix A.

Section 3: Greedy Crossover Nearest Neighbor (GXNN)

GXNN is similar to GX, but it attempts to find an optimal trip by instead of selecting a city at random when both connections are already in the chromosome it selects the nearest neighbor to the current city and uses that as the next city in the chromosome.

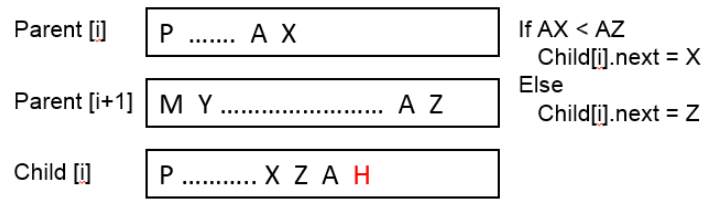

Figure 2: Since X and Z are already a part of child[i] the algorithm selects H as the nearest neighbor that is available.

GXNN was able to find a fastest trip of 447.388 with a mutation rate of 40. Unfortunately, the added time to find the nearest neighbor brought the speed-up down to ~2 times or an execution time of ~12M. However, it did prove to be more efficient in the mutation needing only 29 generations to find the fastest trip. Output and code can be seen in Appendix B.

Section 4: Parallelization Techniques and Methods

Several methods were used to speed-up the overall performance in GX and GXNN implementations. To increase performance before parallelization a calculate distance function was written. This function populated a CITIES+1 * CITIES+1 array that gives the distance of every city to every other city including each city's distance from {0,0}. This distance array resulted in a 50% speed-up from the method of calculating each distance in the evaluate() function. Statements in select() and populate() were changed from a string copy function to a simple assignment operator and led to 5% increase in speed.

Only certain sections of the overall program were parallelized. The section of the evaluate function that calculates each chromosome's fitness was parallelized. Each thread is given a private variable (temp_sum) to calculate the trip distance and a private copy of the distance array to ensure there are no access conflicts. Distances array is small enough that it can fit in the cache without any problems, array size = 5,476 bytes.

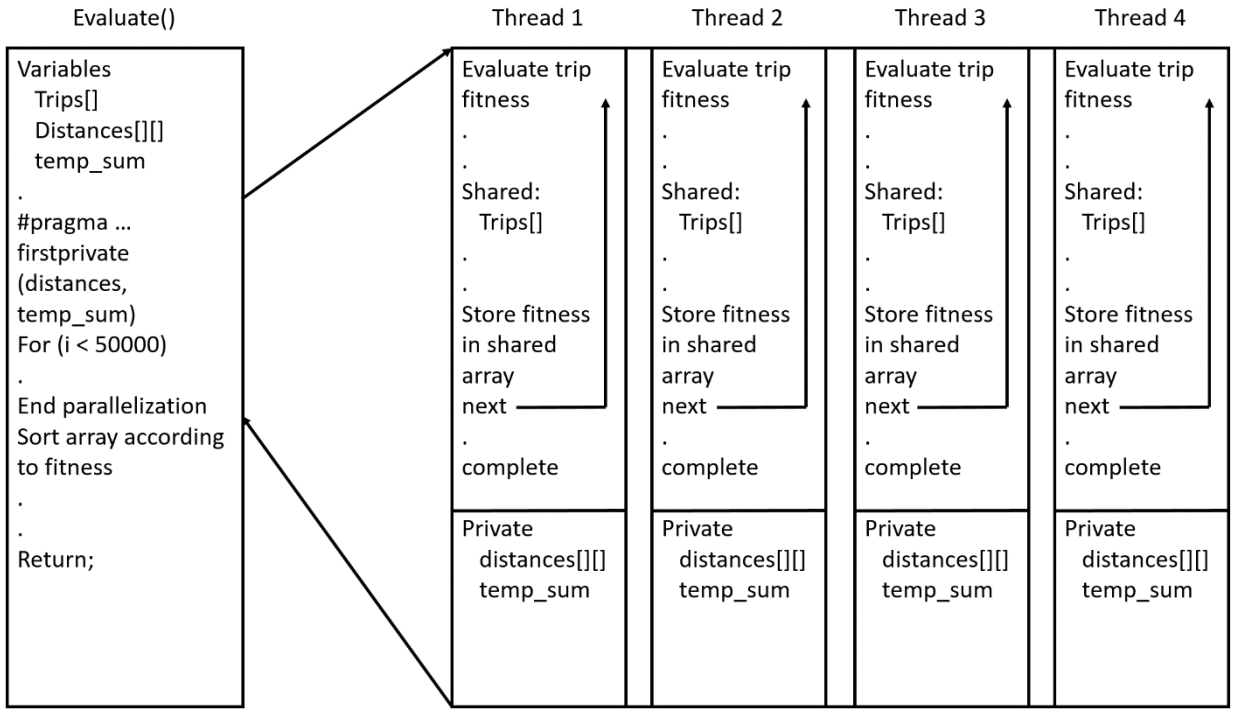

Figure 3: Shows the parallelization of the evaluate() function.

The crossover function "for" loop operates on both parent[i] and [i + 1] during the same iteration and does so in sequence TOP_X / 2 times. This allowed for easy parallelization using the standard OpenMP command "#pragma omp parallel for". Each thread was given a private copy of distances to ensure there are no access conflicts.

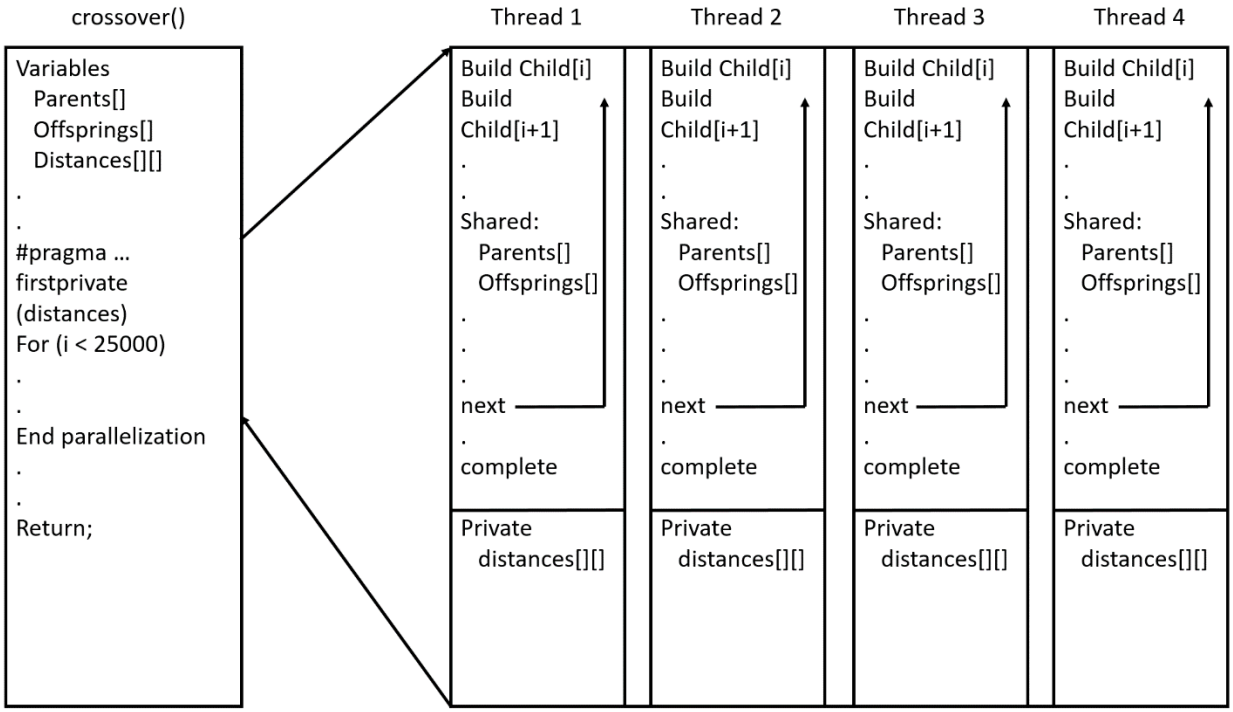

Figure 4: Shows the parallelization of the crossover() function.

Section 5: Challenges

GX and GXNN both proved to be difficult to implement. One of the major challenges was inconsistent results from the function std::strchr. This was initially used to check if a city was already present in an offspring but was found to be inconsistent in its results. To correct this problem I simply wrote a chk_contains() function that returns true if the trip contains the city or false if it does not.

The mutate function needs random numbers in order to be able to function correctly. And the standard C++ rand() function is not parallel safe. The alternative function rand_r() is parallel safe but requires that a seed be passed to it and the seeds needs changed each iteration. Using the rand_r() did not produce an increase in speed thus was not used. In order to produce consistent results srand() cannot be set with time(NULL).

Finding the ideal mutation rate seemed to be a bit of a stab in the dark. A simple “for” loop was added to run the algorithm from a mutation rate of 5 – 100. This showed that a mutation rate of 40 yields the fastest trip.

Section 6: Conclusion

Run on a single thread the algorithm takes time ~11272740 to execute. Running on 4 threads the program takes time ~3799415 to execute. This is a parallelized speed-up of 11272740/3799415 = 2.967 times when compared to itself. When compared to the example given by Dr. Fukuda this a speed-up of 26234614/3799415 = 6.904. The fastest route found is distance = 447.638 and route = V1YZHUE20J6OI84TNXGK9FAL7R3DBQPMSWC5. Source code is in Appendix C.

The optimal number of threads was found to be 8 on the UWB Linux machines. The algorithm took ~3403679 time to execute for a relative speed-up of 11272740/3403679 = 3.312 times faster.

Appendix A – GX

Greedy Crossover with 4 Threads

```
# threads = 4
generation: 0
generation: 0 shortest distance = 1265.72        itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 1040.42        itinerary = V1WMPC20E5SBAF97DLGN4RQJT3UHYZ8I6KXO
generation: 2 shortest distance = 1001.2         itinerary = IX4A9KG7LEYH2W5CMSVF8TNOR3DQBUZ061JP
generation: 3 shortest distance = 865.717        itinerary = V1ZHY02EW5SCMOIN4T9KFG68XR3DPJLAUQB7
generation: 4 shortest distance = 791.005        itinerary = V1UJ7LAX8N4F9OKTG602EQ3PBDRMSWC5HZYI
generation: 5 shortest distance = 667.54         itinerary = V1IO6UHYEZJ7LFK98TN4XAGR3DB05CWPQMS2
generation: 6 shortest distance = 652.722        itinerary = V1IO6UHYEZJ7LFK98TN4XAGR3DBPMSWC5Q02
generation: 7 shortest distance = 596.984        itinerary = V1YZU0EH5CWS2MPBQDJ6OI4T8XFK9N3R7LGA
generation: 9 shortest distance = 582.219        itinerary = V1I6O8NT4GXKJ20EHZYU5CWSMPQBDAL73RF9
generation: 10 shortest distance = 575.384       itinerary = V1YZH5MPWSC2UE6OI84NTXKGF9AJ0L7R3DBQ
generation: 11 shortest distance = 569.665       itinerary = V1YZH5CWS20J6OI8TNX49FKGAL7DR3BQEUMP
generation: 12 shortest distance = 556.211       itinerary = V1YZHUOI684NTXKF9J0E2SW5CPMBDQ3RGAL7
generation: 13 shortest distance = 548.369       itinerary = V1YZH5CWSMPQBDR37LA9FKXTN4GI6O8U20EJ
generation: 14 shortest distance = 528.14        itinerary = V1YZH5CWSMP20EUJDBQR37LAXFK9486OINTG
generation: 15 shortest distance = 480.607       itinerary = V1YZH5CWS20EJU6OI84NTXKG9FAL7R3DBQPM
generation: 20
generation: 22 shortest distance = 473.331       itinerary = V1YZHUE02SWC5MPQBDR37LAF9KGXNT84IO6J
generation: 25 shortest distance = 471.246       itinerary = V1YZHUE02WSC5MPQBDR37LAFK9GXNT48IO6J
generation: 26 shortest distance = 470.101       itinerary = V1YZHUE20J6OI84NTX9FKGAL7R3DQBMPSWC5
generation: 31 shortest distance = 468.08        itinerary = V1YZHUJ6OI84NTGXKF9AL7R3DBQPMSWC520E
generation: 32 shortest distance = 458.487       itinerary = V1YZHUE20J6OI84NTXKG9FAL7R3DBQPMSWC5
generation: 40
generation: 48 shortest distance = 457.132       itinerary = V1YZHUE025CWSMPQBDR37LA9FKGXNT48IO6J
generation: 55 shortest distance = 455.439       itinerary = V1YZHUE025CWSMPQBD3R7LAFK9GXNT48OI6J
generation: 60
generation: 61 shortest distance = 453.554       itinerary = V1YZHUE025CWSMPQBDR37LAF9KGXNT48OI6J
generation: 63 shortest distance = 449.552       itinerary = V1YZHUE20J6OI84NTXGK9FAL7R3DBQPMSWC5
generation: 80
generation: 82 shortest distance = 447.638       itinerary = V1YZHUE20J6OI84TNXGK9FAL7R3DBQPMSWC5
generation: 100
generation: 120
generation: 140
elapsed time = 3799415
```

Greedy Crossover with 8 Threads

```
# threads = 8
generation: 0
generation: 0 shortest distance = 1265.72        itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 1040.42        itinerary = V1WMPC20E5SBAF97DLGN4RQJT3UHYZ8I6KXO
generation: 2 shortest distance = 1001.2         itinerary = IX4A9KG7LEYH2W5CMSVF8TNOR3DQBUZ061JP
generation: 3 shortest distance = 865.717        itinerary = V1ZHY02EW5SCMOIN4T9KFG68XR3DPJLAUQB7
generation: 4 shortest distance = 791.005        itinerary = V1UJ7LAX8N4F9OKTG602EQ3PBDRMSWC5HZYI
generation: 5 shortest distance = 667.54         itinerary = V1IO6UHYEZJ7LFK98TN4XAGR3DB05CWPQMS2
generation: 6 shortest distance = 652.722        itinerary = V1IO6UHYEZJ7LFK98TN4XAGR3DBPMSWC5Q02
generation: 7 shortest distance = 596.984        itinerary = V1YZU0EH5CWS2MPBQDJ6OI4T8XFK9N3R7LGA
generation: 9 shortest distance = 582.219        itinerary = V1I6O8NT4GXKJ20EHZYU5CWSMPQBDAL73RF9
generation: 10 shortest distance = 575.384       itinerary = V1YZH5MPWSC2UE6OI84NTXKGF9AJ0L7R3DBQ
generation: 11 shortest distance = 569.665       itinerary = V1YZH5CWS20J6OI8TNX49FKGAL7DR3BQEUMP
generation: 12 shortest distance = 556.211       itinerary = V1YZHUOI684NTXKF9J0E2SW5CPMBDQ3RGAL7
generation: 13 shortest distance = 548.369       itinerary = V1YZH5CWSMPQBDR37LA9FKXTN4GI6O8U20EJ
generation: 14 shortest distance = 528.14        itinerary = V1YZH5CWSMP20EUJDBQR37LAXFK9486OINTG
generation: 15 shortest distance = 480.607       itinerary = V1YZH5CWS20EJU6OI84NTXKG9FAL7R3DBQPM
generation: 20
generation: 22 shortest distance = 473.331       itinerary = V1YZHUE02SWC5MPQBDR37LAF9KGXNT84IO6J
generation: 25 shortest distance = 471.246       itinerary = V1YZHUE02WSC5MPQBDR37LAFK9GXNT48IO6J
generation: 26 shortest distance = 470.101       itinerary = V1YZHUE20J6OI84NTX9FKGAL7R3DQBMPSWC5
generation: 31 shortest distance = 468.08        itinerary = V1YZHUJ6OI84NTGXKF9AL7R3DBQPMSWC520E
generation: 32 shortest distance = 458.487       itinerary = V1YZHUE20J6OI84NTXKG9FAL7R3DBQPMSWC5
generation: 40
generation: 48 shortest distance = 457.132       itinerary = V1YZHUE025CWSMPQBDR37LA9FKGXNT48IO6J
generation: 55 shortest distance = 455.439       itinerary = V1YZHUE025CWSMPQBD3R7LAFK9GXNT48OI6J
generation: 60
generation: 61 shortest distance = 453.554       itinerary = V1YZHUE025CWSMPQBDR37LAF9KGXNT48OI6J
generation: 63 shortest distance = 449.552       itinerary = V1YZHUE20J6OI84NTXGK9FAL7R3DBQPMSWC5
generation: 80
generation: 82 shortest distance = 447.638       itinerary = V1YZHUE20J6OI84TNXGK9FAL7R3DBQPMSWC5
generation: 100
generation: 120
generation: 140
elapsed time = 3403679
```

Source Code for Crossover()

```
/*generate 25,000 offspring from the parents
*
* @param parents[TOP_X]: top 25000 chromosomes
* @param offsprings[TOP_X]: will hold new bottom 25000 chromosomes
* @param distances[CITIES+1][CITIES+1]: used to calculated nearest connection
*/
extern void crossover(Trip parents[TOP_X], Trip offsprings[TOP_X], float distances[CITIES
+ 1][CITIES + 1])
{
	//parallelize
	#pragma omp parallel for firstprivate(distances)
	for (int i = 0; i < TOP_X; i += 2)
	{
		offsprings[i].itinerary[0] = parents[i].itinerary[0];

		for (int j = 0; j < CITIES - 1; j++)
		{
			//find starting city
			char current_city = offsprings[i].itinerary[j];
			int index_city = (current_city >= 'A') ? current_city - 'A' :
				current_city - '0' + 26;

			//find first connection
			int temp = 0;
			char connection1;
			while (parents[i].itinerary[temp++] != current_city);
			if (temp <= CITIES - 1)
				connection1 = parents[i].itinerary[temp];
			else
				connection1 = parents[i].itinerary[0];
			int index_connection1 = (connection1 >= 'A') ? connection1 - 'A' :
				connection1 - '0' + 26;

			//find second connection
			temp = 0;
			char connection2;
			while (parents[i + 1].itinerary[temp++] != current_city);
			if (temp <= CITIES - 1)
				connection2 = parents[i + 1].itinerary[temp];
			else
				connection2 = parents[i + 1].itinerary[0];
			int index_connection2 = (connection2 >= 'A') ? connection2 - 'A' :
				connection2 - '0' + 26;

			//find shortest connection
			char shortest1, shortest2;
			if (distances[index_city][index_connection1] <
				distances[index_city][index_connection2])
			{
				shortest1 = connection1;
				shortest2 = connection2;
			}
			else
			{
				shortest1 = connection2;
				shortest2 = connection1;
```

```
                        }

                        //ensure the city is not already present in the trip
                        if (!chk_contains(&offsprings[i], shortest1, j + 1))
                                offsprings[i].itinerary[j + 1] = shortest1;
                        else if (!chk_contains(&offsprings[i], shortest2, j + 1))
                                offsprings[i].itinerary[j + 1] = shortest2;
                        else
                                select_city(&offsprings[i], &parents[i+1], j + 1);
                }

                //creat child 2 as complement of child 1
                //create complement table
                char complements[CITIES];
                for (int j = 0, k = 35; j < CITIES; j++, k--) //work from outside in
                {
                        char complement_start = parents[i].itinerary[j];
                        char complement_end = parents[i].itinerary[k];
                        int index = (complement_start >= 'A') ? complement_start - 'A' :
                                complement_start - '0' + 26;
                        complements[index] = complement_end;
                }

                //build child 2
                for (int j = 0; j < CITIES; j++)
                {
                        char current_city = offsprings[i].itinerary[j];
                        int index = (current_city >= 'A') ? current_city - 'A' :
                                current_city - '0' + 26;
                        offsprings[i+1].itinerary[j] = complements[index];
                }
        }

}
```

Appendix B – GXNN

Greedy Crossover with nearest neighbor and 8 Threads

```
generation: 0
generation: 0 shortest distance = 1265.72        itinerary = V1SPMBQAN26G4J37DX8OTF95ZUH0EYRLCWKI
generation: 1 shortest distance = 932.618        itinerary = 6O9TKNFA7JI1EB2UZCYSW5H0R3DMPQLGX48V
generation: 2 shortest distance = 713.297        itinerary = V1EUHZYI69FO8KX4GNTADQ2WSC5MPB0JR37L
generation: 3 shortest distance = 630.337        itinerary = O6IV1YZHUEJ02WSC5PQBL7F98TN4XKGAR3DM
generation: 4 shortest distance = 571.265        itinerary = O6IV1YZHUEJ02WSC5PMBQDR37FK94XNT8GAL
generation: 5 shortest distance = 546.193        itinerary = V1YZH20UEJO6I8TN49KFXG7LA3RBDQPMSWC5
generation: 6 shortest distance = 522.819        itinerary = V1YHZE02WC5SMPBDQ3R7LAKF948TNXGJU6OI
generation: 7 shortest distance = 494.598        itinerary = V1YZH5WSC20EUJ6OI84N9FKGXTAL7R3DQBPM
generation: 10 shortest distance = 469.016       itinerary = V1YZHUE20J6OI84NTXGKF9AL7R3DBQPMSC5W
generation: 11 shortest distance = 453.709       itinerary = V1YZHUE20J6OI84NTXGKF9AL7R3DBQPMSWC5
generation: 20
generation: 21 shortest distance = 449.302       itinerary = V1YZH5CWSMP20EUJ6OI84NTXGK9FAL7R3DBQ
generation: 29 shortest distance = 447.388       itinerary = V1YZH5CWSMP20EUJ6OI84TNXGK9FAL7R3DBQ
generation: 40
generation: 60
generation: 80
generation: 100
generation: 120
generation: 140
elapsed time = 12307417
```

Source Code for Nearest Neighbor section

```
void random_city(Trip* offsprings, Trip* parents, int current_index, float
distances[CITIES+1][CITIES+1])
{
	for (int j = 0; j < CITIES; j++)
	{
		char c_city = offsprings[0].itinerary[current_index - 1];
		int c_index = (c_city >= 'A') ? c_city - 'A' : c_city - '0' + 26;
		float smallest = 10000;
		int smallest_i;

		//find nearest != 0
		for (int k = 0, m = 0; k < CITIES; k++)
		{
			c_city = parents[0].itinerary[k];
			m = (c_city >= 'A') ? c_city - 'A' : c_city - '0' + 26;
			if (!chk_contains(&offsprings[0], c_city, current_index))
			{
				if (distances[c_index][m] < smallest &&
					distances[c_index][m] != 0)
				{
					smallest = distances[c_index][m];
					smallest_i = k;
				}
			}
		}
		if (smallest_i < 0 || smallest_i > 35)
		{
			for (int j = 0; j < CITIES; j++)
			{
				if (!chk_contains(&offsprings[0], parents[0].itinerary[j],
					current_index))
				{
```

```
                                    offsprings[0].itinerary[current_index] =
                                           parents[0].itinerary[j];
                                    j = CITIES;
                             }
                      }
               }
               else
                      offsprings[0].itinerary[current_index] =
                             parents[0].itinerary[smallest_i];

       }

}
```

Appendix C – Source Files