APPENDIX E

Code of the algorithm in C++

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <cmath>
#include <time.h>
#include <assert.h>
#include <fstream>
#include <ctime>
#include <string>


using namespace std;

float min = 1000000;
float max = 0;
const int num_node = 1800;
const int num_chromosome = 150;
const int num_parents = 50;
const int range = 4;
const int num_facility = 20;
const double mutation_rate = 0.05;
const double replacement_prob = 0.7;
const int num_iter = 1000;
const int optimal = 51280;
const double prob_repairing = 0.05;
string problem_demand = "so1800_demand.txt";
string problem_distance = "so1800_distance.txt";
string problem_result = "so1800-2p-Max_Rep.txt";
string improving = "a";
string xover_operator = "b";

struct node {
```

```
      double x_cord;
      double y_cord;
      int demand;
      int satisfied_demand;
};

node Array_node[num_node];
double distance_node[num_node][num_node];

struct chromosome {
      int soln[num_node];
      int satisfied[num_node];
      int fitness;
      double prob;
      int index;
};

struct population {
      chromosome chros[num_chromosome];
};

struct parent {
      chromosome parents[num_parents];
};

parent mating_pool;

population initial;

void init_pop(void) { //Generating Initial Population

      for (int i = 0; i < num_chromosome; i++) {
            int counter = 0;
            while (counter <= num_facility - 1)
            {
                  int rand_num = rand() % num_node;
                  if (initial.chros[i].soln[rand_num] == 0) {
```

```
                    initial.chros[i].soln[rand_num] = 1;
                    counter += 1;
               }
          }
          initial.chros[i].index = i;
     }
}

int total_demand[num_node];

void demand_satisfied_node() {
//Calculates each nodes' satisfaction portion in case facility opened for Gre_Rep

     for (int i = 0; i < num_node; i++)
     {
          for (int j = 0; j < num_node; j++)
          {
               if (distance_node[i][j] < range) {
                    total_demand[i] += Array_node[j].demand;
               }
          }
          Array_node[i].satisfied_demand = total_demand[i] + Array_node[i].demand;
     }
}

void ordering() { //Ordering nodes according to their demand_satisfied_node() for Gre_Rep
     int i, j;
     node temp;

     for (i = 0; i < num_node; i++)
     {
          for (j = i + 1; j < num_node; j++)
          {
               if (Array_node[i].satisfied_demand < Array_node[j].satisfied_demand)
               {
                    temp = Array_node[i];
                    Array_node[i] = Array_node[j];
```

```
                    Array_node[j] = temp;
                    i = 0;
               }
          }
     }
}

void satisfied_vec_func(chromosome &x) { //Satisfied vector calculation
     for (int i = 0; i < num_node; i++)
     {
          x.satisfied[i] = 0;
     }
     for (int i = 0; i < num_node; i++) {

          for (int j = 0; j < num_node; j++) {

               if ((x.soln[i] == 1) && (distance_node[i][j] < range)) {
                    x.satisfied[j] = 1;
               }
               else {
                    if (x.satisfied[j] != 1)
                         x.satisfied[j] = 0;
               }
          }
     }
}

int fitness_func(chromosome x) { //Fitness function calculation
     x.fitness = 0;
     int sum = 0;
     for (int i = 0; i < num_node; i++) {
          sum = x.satisfied[i] * Array_node[i].demand + sum;
     }
     x.fitness = sum;
     return x.fitness;
}
```

```
void prob_func(population &gen) { //Calculating probabilities of chromosomes
	for (int i = 0; i < num_chromosome; i++) {
		if (gen.chros[i].fitness < min) {
			min = gen.chros[i].fitness;
		}
		else if (gen.chros[i].fitness > max)
		{
			max = gen.chros[i].fitness;
		}
	}
	for (int i = 0; i < num_chromosome; i++) {
		gen.chros[i].prob = (gen.chros[i].fitness - min) / (max - min);
	}
	float sum = 0;
	for (int i = 0; i < num_chromosome; i++)
	{
		sum += gen.chros[i].prob;
	}
	gen.chros[0].prob = gen.chros[0].prob / sum;
	for (int i = 1; i < num_chromosome; i++)
	{
		gen.chros[i].prob = gen.chros[i - 1].prob + (gen.chros[i].prob / sum);
		//Finding Cumulative Probabilities
	}
}

float RandomFloat(float a, float b) {
	float random = ((float)rand()) / (float)RAND_MAX;
	float diff = b - a;
	float r = random * diff;
	return a + r;
}

void parent_selection(population gen) { //Parent Selection

	for (int h = 0; h < num_parents; h++)
	{
```

```
            float rnd = RandomFloat(0, 1);
            float diff = 1;
            for (int j = 0; j < num_chromosome; j++)
            {
                  if (((gen.chros[j].prob - rnd) > 0) && ((gen.chros[j].prob - rnd) < diff)) {
                        diff = abs(gen.chros[j].prob - rnd);
                        mating_pool.parents[h] = gen.chros[j];
                        mating_pool.parents[h].index = gen.chros[j].index;
                  }
            }
      }
}

chromosome crossover(chromosome parent1, chromosome parent2) //Crossover Operator(1-point)
{
      xover_operator = "1-Point Crossover";
      int k = rand() % num_node; //Cut Point
      chromosome offspring;
      for (int i = 0; i < k; i++)
      {
            offspring.soln[i] = parent2.soln[i];
      }
      for (int i = k; i < num_node; i++)
      {
            offspring.soln[i] = parent1.soln[i];
      }
      return offspring;
}

chromosome crossover_2p(chromosome parent1, chromosome parent2) //Crossover Operator(2-point)
{
      xover_operator = "2-Point Crossover";
      int k1 = rand() % num_node; //Cut Point
      int k2 = rand() % num_node;
      int temp = 0;
      if (k1 > k2)
```

```
	{
		temp = k1;
		k1 = k2;
		k2 = temp;
	}
	chromosome offspring;
	for (int i = 0; i < k1; i++)
	{
		offspring.soln[i] = parent1.soln[i];
	}
	for (int i = k1; i < k2; i++)
	{
		offspring.soln[i] = parent2.soln[i];
	}
	for (int i = k2; i < num_node; i++)
	{
		offspring.soln[i] = parent1.soln[i];
	}
	return offspring;
}

chromosome max_imp_repairing(chromosome x) { //Repairs using marginal improvements Max_Rep
	improving = "Maximum Improving";
	int sum = 0;
	int max_demand = 0;
	int min_demand = 1000000;
	int index = 0;
	satisfied_vec_func(x);
	x.fitness = fitness_func(x);
	int current_fitness = x.fitness;
	chromosome y;
	y = x;
	int deviation[num_node];

	for (int i = 0; i < num_node; i++)
	{
		sum += x.soln[i];
```

```
}
if (sum < num_facility) {
      while (sum < num_facility)
      {
            for (int i = 0; i < num_node; i++)
            {
                  if (y.soln[i] == 0)
                  {
                        y.soln[i] = 1;
                        satisfied_vec_func(y);
                        y.fitness = fitness_func(y);
                        deviation[i] = y.fitness - current_fitness;
                  }
                  y = x;
            }
            for (int i = 0; i < num_node; i++)
            {
                  if (deviation[i] > max_demand)
                  {
                        max_demand = deviation[i];
                        index = i;
                        deviation[i] = 0;
                  }
            }

            x.soln[index] = 1;
            sum++;
            max_demand = 0;
      }
}

if (sum > num_facility) {
      while (sum > num_facility)
      {
            for (int i = 0; i < num_node; i++)
            {
                  if (y.soln[i] == 1)
```

```
                    {
                        y.soln[i] = 0;
                        satisfied_vec_func(y);
                        y.fitness = fitness_func(y);
                        deviation[i] = current_fitness - y.fitness;
                    }
                    y = x;
                }
                for (int i = 0; i < num_node; i++)
                {
                    if ((deviation[i] < min_demand) && (deviation[i] > 0))
                    {
                        min_demand = deviation[i];
                        index = i;
                        deviation[i] = 0;
                    }
                }

                x.soln[index] = 0;
                sum--;
                min_demand = 1000000;
            }
        }
        return x;
}

chromosome repairing(chromosome x) { //Randomly Repairing Rand_Rep
        improving = "Random Repairing";
        int sum = 0;
        for (int i = 0; i < num_node; i++)
        {
            sum += x.soln[i];
        }

        if (sum < num_facility) {
            while (sum <= num_facility - 1)
            {
```

```
                int rand_num = rand() % num_node;
                if (x.soln[rand_num] == 0) {
                      x.soln[rand_num] = 1;
                      sum += 1;
                }
          }
     }
     if (sum > num_facility) {
          while (sum >= num_facility + 1) {
                int rand_num = rand() % num_node;
                if (x.soln[rand_num] == 1) {
                      x.soln[rand_num] = 0;
                      sum -= 1;
                }
          }
     }
     return x;
}

chromosome alt_repairing(chromosome x) { //Repairing according to demand_satisfied_node Gre_Rep
     improving = "Greedy Repairing";
     int sum = 0;
     float rand_num = 0;
     for (int i = 0; i < num_node; i++)
     {

          sum += x.soln[i];
     }

     int maks = 0;

     while (sum < num_facility) {
          for (int i = 0; i < num_node; i++)
          {
                rand_num = RandomFloat(1, 0);
                if ((x.soln[i] == 0) && (rand_num < prob_repairing))
                {
```

```
                    x.soln[i] = 1;
                    sum++;
                }
            }
        }

        while (sum > num_facility)
        {
            for (int i = num_node; i > -1; i--)
            {
                rand_num = RandomFloat(1, 0);

                if ((x.soln[i] == 1) && (rand_num < prob_repairing))
                {
                    x.soln[i] = 0;
                    sum--;
                }
            }
        }

        return x;
}

chromosome mutation(chromosome x) { //Mutation operator (swap two elements of array)
        float rand_num = RandomFloat(1, 0);
        int rnd = rand() % num_node;
        if (rand_num < mutation_rate) {
            for (int i = 0; i < num_node; i++)
            {
                swap(x.soln[rnd], x.soln[rand() % num_node]);
            }
        }
        return x;
}

int main() {
        int bekle;
```

```
int mak_out = 0;
ofstream result("Result.txt"); //Prinout solutions in Result.txt
double timer = 0;
double timer_out = 0;
int sum_time = 0;
int sum_best = 0;

for (int k = 0; k < 10; k++) //k= number of replication
{

      int start_s = clock();
      // Start timer

      srand((int)time(NULL));

      ofstream out(problem_result);
      ifstream get_demand; //Get demand values from problem set

      get_demand.open(problem_demand);
      for (int i = 0; i < num_node; i++)
      {
            get_demand >> Array_node[i].demand;
      }

      get_demand.close();

      ifstream get_distance; //Get distance values from problem set

      get_distance.open(problem_distance);

      for (int i = 0; i < num_node; i++)
      {
            for (int j = 0; j < num_node; j++)
            {
                  get_distance >> distance_node[i][j];
            }
      }
```

```
        get_distance.close();

        //demand_satisfied_node(); //Calculate each node's portion of satisfaction

        //ordering(); //Order nodes according to portion of satisfaction

        init_pop(); //Initialize a population

        for (int iter = 0; iter < num_iter; iter++) //Generates num_iter generations
        {

              for (int i = 0; i < num_chromosome; i++) //Calculating satisfied vector
              {
                    satisfied_vec_func(initial.chros[i]);
              }

              for (int j = 0; j < num_chromosome; j++) //Calculate fitness values of population
              {
                    initial.chros[j].fitness = fitness_func(initial.chros[j]);
              }

              prob_func(initial); // Calculate Probabilities for mating

              parent_selection(initial); //Select num_parents parents from initial population

              chromosome offsprings[num_parents];

              for (int j = 0; j < num_parents; j += 2)
              //Crossing Mating Pool and producing offsprings with 2 parents
              {
                    for (int i = 0; i < 2; i++)
                    {
                          offsprings[i + j] = crossover_2p(mating_pool.parents[j],
mating_pool.parents[j + 1]);
                          swap(mating_pool.parents[j], mating_pool.parents[j + 1]);
                    }
```

```
                }



/*
                for (int j = 0; j < num_parents; j += 3)
                //Crossing 3 parents and producing offsprings
                {
                      for (int i = 0; i < 3; i++)
                      {
                            offsprings[i + j] = crossover(mating_pool.parents[j],
mating_pool.parents[j + 1]);
                            swap(mating_pool.parents[j+1], mating_pool.parents[j]);
                            swap(mating_pool.parents[j], mating_pool.parents[j + 2]);
                      }
                }
                */

                for (int i = 0; i < num_parents; i++)//Filling missing values of offspring
                {
                      satisfied_vec_func(offsprings[i]);
                      offsprings[i].fitness = fitness_func(offsprings[i]);
                }

                for (int i = 0; i < num_parents; i++) //Repairing Offspring
                {
                      offsprings[i] = max_imp_repairing(offsprings[i]);
                }

                for (int i = 0; i < num_parents; i++)//Mutating offsprings
                {
                      offsprings[i] = mutation(offsprings[i]);
                }

                for (int i = 0; i < num_parents; i++)//Filling missing values of offspring
                {
                      satisfied_vec_func(offsprings[i]);
```

```
                    offsprings[i].fitness = fitness_func(offsprings[i]);
               }

               float rnd_float = RandomFloat(1, 0);

               for (int i = 0; i < num_parents; i += 2) //Parent Replacing
               {
                    for (int j = i; j < i + 2; j++)
                    {

                         if ((offsprings[i].fitness > mating_pool.parents[j].fitness) &&
(rnd_float < replacement_prob)) {
                              offsprings[i].index = mating_pool.parents[j].index;
                              initial.chros[mating_pool.parents[j].index] = offsprings[i];
                              break;
                         }
                    }
               }

               /*
               for (int i = 0; i < num_parents; i += 3) //Parent Replacing for 3 parents mating
               {
                    for (int j = i; j < i + 3; j++)
                    {

                         if ((offsprings[i].fitness > mating_pool.parents[j].fitness) &&
(rnd_float < replacement_prob)) {
                              offsprings[i].index = mating_pool.parents[j].index;
                              initial.chros[mating_pool.parents[j].index] = offsprings[i];
                              break;
                         }
                    }
               }
               */
```

```
	int maksimum = 0;
	for (int i = 0; i < num_chromosome; i++)
	{
		if (initial.chros[i].fitness > maksimum)
		{
			maksimum = initial.chros[i].fitness;
			//Finds maximum fitness value in population
			mak_out = maksimum;
		}
	}
	if (maksimum==optimal)
	{
		int stop_f = clock();
		timer = (stop_f - start_s) / double(CLOCKS_PER_SEC);
		cout << "FOUND OPTIMAL time: " << timer << endl; //If it finds optimal stops
		cout << maksimum << endl;
		out << "Calculation Time: " << timer << endl;
	}
	out << maksimum << endl;

	prob_func(initial); // Calculate Probabilities of new population for mating

	if (iter == num_iter - 1) {
		cout << "END";
		int stop_s = clock();
		timer = (stop_s - start_s) / double(CLOCKS_PER_SEC) ;
		cout << "time: " << timer << endl;
		cout << maksimum << endl;  //Printout maximum fitness value of the population
		out << "Calculation Time: " << timer << endl;
		timer_out = timer;
	}
}
for (int i = 0; i < num_chromosome; i++)
```

```
            {
                  for (int j = 0; j < num_node; j++)
                  {
                        initial.chros[i].fitness = 0;
                        initial.chros[i].prob = 0;
                        initial.chros[i].soln[j] = 0;
                  }
            }
      }
      cin >> bekle;
      return 0;
}
```