University of South-Eastern Norway, Kongsberg

OOP-4200
Assignment 2.

Travelling Salesman Problem – Genetic Algorithm

Victor Johan Hansen
October 2019

1 Introduction

In this assignment we will implement a solver for the *Travelling Salesman Problem (TSP)*[1] that uses genetic algorithms.

1.1 Nomenclature

Genetic algorithm: a process of natural selection where the goal is to advance towards a fitter solution.

Genome or *genetic material* is a way of representing solutions to a problem as a sequence of values. Each genome has its own set of values that specifies a potential solution. The members of our population are made up of genomes.
E.g., with a problem of *n* points, a genome might be:

$$[(n-2) \ \ldots\ n \ \ldots\ 1 \ \ldots\ (n-1)]$$

For our TSP-solver this will be the order that the points are visited.

Fitness function: is the length of the circuit formed by visiting *n* points in a specified order. Members of the population with *fitter* genomes will have a shorter path; *less fit* individuals will specify a longer path.

Crossover (recombination): the best solutions are used to generate offspring (i.e. children) from the existing population (i.e. parents).

Mutation: the genomes are mutated in some random way that allows us to generate some interesting solutions.

[1] https://en.wikipedia.org/wiki/Travelling_salesman_problem

2 Design and implementation

Written in C++ (C++11). Compiled with Clang.

2.1 Point.h

This class will represent the *xyz*-coordinates for each point from the input .txt-files.

2.1.1 calcDist

`double calcDist(const Point& p) const` is declared here. This function computes the distance between two points P1 and P2 in $\mathbf{R}^3$:

$$\sum_{i=0}^{order.size()-1} d(P_i, P_{i+1}) = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2 + (z_{i+1} - z_i)^2}$$

2.2 tsp-ga.h

This header contains the class for the TSPGenome. The class has two data members: `circuitLength`, which is the total distance covered when the given points are visited in a specified order. And `visitOrder -` a vector which contains the points we will visit. The following operations for the TSPGenome are declared in the class: `computeCircuitLength`, `mutate` and methods for returning `visitOrder` and `circuitLength`. The header file also includes two functions `TSPGenome crosslink` and `TSPGenome findAShortPath` which are not members of the class but are functions with the signature of the class.

2.3 tsp-ga.cpp

This file contains the implementation of the TSPGenome operations, as well as some other functions.

2.3.1 ComputeCircuitLength

`computeCircuitLength(const std::vector<Point>& points):`
Computes the circuit length from passing through the points in the order specified in the object. This will be our fitness function.

```
for (int i = 0; i < visitOrder.size(); i++) {
        circuitLength +=
        points[visitOrder[i]].calcDist(points[visitOrder[(i+1)%(visitOrder.size())]]);
  }
}
```

`(i+1) % (visitOrder.size())` allow us to compute a round trip for the circuit length. If `visitOrder.size()` = *n* and we've reached the end of our trip where `i=n-1`, we get `((n-1)+1) % (n)` = 0, which is the start of our trip.

2.3.2 Crosslink

`TSPGenome crosslink(const TSPGenome &g1, const TSPGenome &g2)` – is a mechanism used to produce an offspring from two parents. We copy g1's order-values if the values are unique at indexes `[0, index-1]` to a new vector (`newOrder`) which holds the offspring's order. `std::set<int> myset` ensures that we only store unique elements in a specific order.

```
std::vector<int> newOrder;
std::set<int> myset;
int index = rand() % g1.getOrder().size();
for (int i = 0; i < index; i++) {
      myset.insert(g1.getOrder()[i]);
      newOrder.push_back(g1.getOrder()[i]);
  }
...
```

Next we copy each value from g2's order to `newOrder`, but only if g2's vector doesn't contain elements that already appear in `newOrder`. This is done to ensure that each value appears exactly once in `newOrder`. Finally, we can use the `newOrder` to create a new TSPGenome object.

```
for (int i = 0; i < g2.getOrder().size(); i++) {
      if(myset.count(g2.getOrder()[i]) == 0) {
            newOrder.push_back(g2.getOrder()[i]);
            myset.insert(g2.getOrder()[i]);
      }
}
return(TSPGenome(newOrder));
...
```

2.3.3 findAShortPath

Now we want to find a short path. First we generate an initial population of random genomes, where the genome size is equivalent to the number of points (e.g. 12). The population size will be specified by the user (e.g. 1000).

```
std::vector<TSPGenome> population;
for (int i = 0; i < populationSize; i++) {
      population.push_back(TSPGenome(points.size()));
} ...
```

Next we compute the circuit length for each member of the population for several generations, the number of generations will be specified by the user (e.g. 100).

```
for (int gen = 0; gen < numGenerations; gen++) {
      for (int i = 0; i < populationSize; i++) {
            population[i].computeCircuitLength(points);
      } ...
```

We pass the function `isShorterPath` to `sort()`, this allow us to sort the genomes by fitness. `isShorterPath` will return true if g1 has a shorter circuit length than g2. We want to keep the fittest members of the population by replacing the remaining members with new genomes produced from the fittest ones. The argument `keepPopulation` specifies how many members to keep from the current population.

```
std::sort(population.begin(), population.end(), isShorterPath);
for (int i = keepPopulation; i < populationSize; i++) {
      std::size_t g1 = rand() % (keepPopulation);
      std::size_t g2 = rand() % (keepPopulation);
      population[i] = crosslink(population[g1], population[g2]);
    } ...
```

Next we apply mutation to the population; the amount of mutation is specified by `numMutations`. We call `mutate()` for a random genome in the population. Then we return the best solution found so far. `mutate()` swaps two randomly selected values in the vector `visitOrder`, which is a TSPGenome data member.

```
for (int i = 0; i < numMutations; i++) {
      population[1+rand() % (populationSize-1)].mutate();
    }
  ...
  return population[0]; // best solution so far
}
```

2.4 tsp-main.cpp

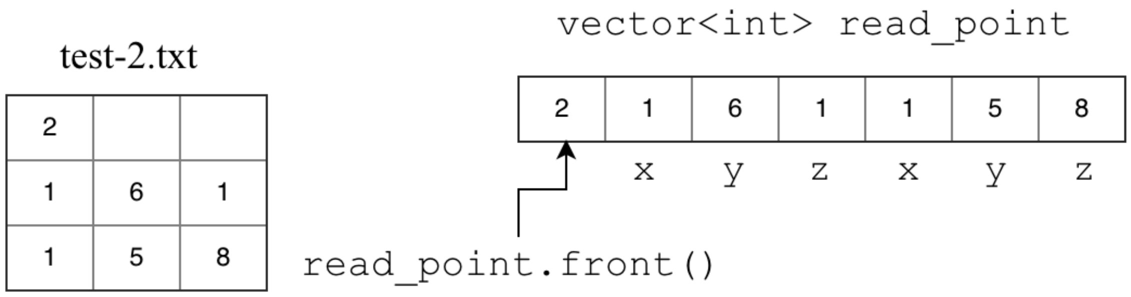
This is the main program, we read points from a .txt-file and load the points into our genetic algorithm.

2.4.1 read_txt_and_fill

I use the input file stream class `ifstream` to read points from a .txt-file into a vector. The user specifies how many points to read.

```
static void read_txt_and_fill(std::vector<Point>& txt_points) {
    std::ifstream myfile;
    std::vector<int> read_point;
    std::cout<<"\nHow many points?\n(2, 5, 8, 10, 12, 30, 60, 100, 300, 500): ";
    int size;
    std::cin>>size;
    std::string filestr ="test-txt/test-"+std::to_string(size)+".txt";
    myfile.open(filestr);
    if (myfile.is_open()) {
      std::cout<<"\n** reading from .txt-file **\n";
        int points;
        while (myfile>>points) {
            read_point.push_back(points);
        }
        myfile.close();
    }
    else std::cout<<"Unable to open file.";
...
```

Then we parse the data from the vector. The first value tell us the number of points in the file, the following values are *xyz*-coordinates of the Point class.



We increment the index *n* by 3 to properly place the values into its corresponding coordinates.
E.g. for 5 points the *x*-coordinates will be at located at indexes:
`read_point[1, 4, 7, 10, 13]`.

```
int n=1;
for (int i=0; i<read_point.front(); i++) {
      Point new_Points(read_point.at(n),read_point.at(n+1),read_point.at(n+2));
      txt_points.push_back(new_Points);
      n+=3;
    }
}
```

2.4.2 main()

The program takes the following command-line arguments:

argv[0]	argv[1]	argv[2]	argv[3]	argv[4]
./tsp	population	generations	keep	mutate

```
argc = 5
```

argc *is the argument count, and* argv *is an argument vector.*

- **population** specifies the population size.
- **generations** specifies how many generations to run the algorithm for.
- **keep** is a float value in the range 0-1 that specifies the percentage of the population that should be preserved from generation to generation.
- **mutate** is a positive float value that specifies how many mutations to apply to each member of the population.

```
unsigned int popSize = (int) atoi(argv[1]);
unsigned int numGen = (int) atoi(argv[2]);
int keepPop = (int) ( atof(argv[3]) * ((float)popSize));
int numMut = (int) (popSize * atof(argv[4]));
```

We use srand(time(nullptr)) to generate a new random sequence every time the program runs. Next we find a short path based on the command-line arguments. Finally, we calculate the circuit length for this path, and print both the path and circuit length.

```
srand(time(nullptr));
TSPGenome path(points.size());
path = findAShortPath(points, popSize, numGen, keepPop, numMut);
std::cout<<"\nShortest distance: "<<path.getCircuitLength()<<std::endl;
printPath(path.getOrder());
```

3 Testing the TSP-GA solver

```
$ ./tsp 1000 100 0.4 1.2
```

Parameters	
Population	1000
Generations	100
Keep	0.4
Mutation	1.2

The values selected for the parameters are arbitrary.

Points	Shortest distance	Shortest path
2	14.1421	0 1
5	27.1016	2 3 1 0 4
8	34.1513	5 2 3 4 7 0 6 1
10	42.5904	5 3 0 7 8 2 4 9 6 1
12	47.6887	5 0 9 10 11 1 4 6 2 8 3 7
30	1055.79	*
60	2264.69	*
100	4035.76	*
300	14502.1	*
500	25898.5	*

**too long path.*

Note the *shortest distance* shown in the table isn't necessarily the **shortest** distance. It's safe to assume that if we add more generations we will most likely achieve a shorter distance. I have also discovered that we obtain shorter distances if we set the "keep"-parameter at low values, e.g. 0.1.

PS: I have at times encountered different distances for the same parameters, this occurs when I run tests with 30 points and more.