CS 5500 Homework 8

Sally Devitry (A01980316)

Approach/Implementation

In this assignment, we were tasked with solving the traveling salesperson problem. The traveling salesperson problem is to visit a number of cities in the minimum distance. The given 100 pairs of integers represent geographic locations of cities on a Cartesian grid. The cost to travel between any two cities is equivalent to the Euclidian distance between them.

My assignment uses a genetic algorithm to solve the travelling salesperson problem. Genes are represented as strings of numbers, the order in which the cities are travelled. Mutations are created by swapping two numbers in the order string.

My code computes 1000 generations with a population of 10 each generation. The fitness of each 'gene' is calculated by determining the euclidean distance of that path from start to finish.

Process 0 initializes the original population, then all other processes perform iterations of crossing over and mutating. The bestYet path is kept track of to be printed at the end.

Run Commands

The program can be run with 2 or more process.

```
mpirun -np 2 a.out
mpirun -np 4 --oversubscribe a.out
```

Some example output (2 processes):

```
Generation 1
FITNESS VALUES
46137486
46327153
47133974
47795681
48556196
46830444
49121099
49335735
51409899
51931589
```

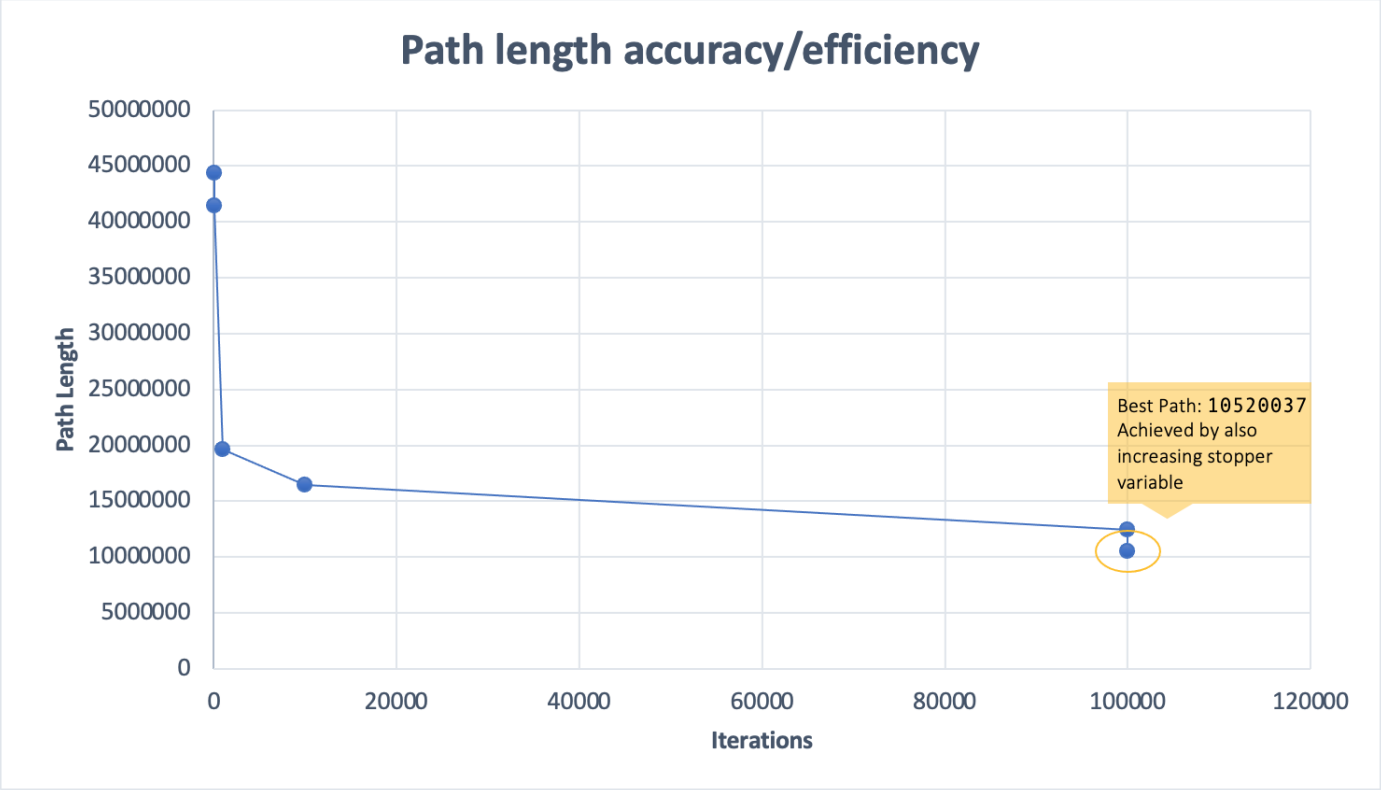
```
BEST PATH LENGTH = 45346114

...

Generation 999
FITNESS VALUES
10190345
11733750
12100837
12407480
12439644
12844479
12965804
13189652
13362106
14208230
BEST PATH LENGTH = 10190345

Generation 1000
FITNESS VALUES
10190345
11733750
12100837
12407480
12439644
12844479
12965804
13189652
13362106
14187650
BEST PATH LENGTH = 10190345
```

Effectiveness Visualization



Concluding Remarks

Overall, this assignment was one of my favorites so far. I thought that a genetic algorithm would be really difficult to understand, but it came easier than expected. It is a very cool way to solve np-hard problems in a short amount of time.

Appendix (Code)

```
#include <iostream>
#include <fstream>
#include <mpi.h>
#include <unistd.h>
#include <stdlib.h>
#include <vector>
#include <math.h>
// #include <bits/stdc++.h>
#include <limits.h>

using namespace std;
```

```
#define MCW MPI_COMM_WORLD

#define numCities 100
#define popSize 10

int cities[numCities][2] = {{179140, 750703}, {78270, 737081}, {577860, 689926}, {628150, 597095}, {954

// string defines the path traversed by the salesperson
// the fitness value of the path is stored in an integer
struct order
{
  string gene;
  int fitness;
};

//return a random number for swapping
int rand_num(int start, int end)
{
  int r = end - start;
  int rnum = start + rand() % r;
  return rnum;
}

// Mutated gene is a string with a random swap of two cities to create variation
string mutatedGene(string gene)
{
  while (true)
  {
    int r = rand_num(1, numCities);
    int r1 = rand_num(1, numCities);
    if (r1 != r)
    {
      char temp = gene[r];
      gene[r] = gene[r1];
      gene[r1] = temp;
      break;
    }
  }
  return gene;
}

//check if the character has already occurred in the string
bool isRepeat(string s, char ch)
{
  for (int i = 0; i < s.size(); i++)
  {
    if (s[i] == ch)
      return true;
  }
  return false;
}
```

```
//return a valid gene string required to create the population
string create_gene()
{
  string gene = "0";
  while (true)
  {
    if (gene.size() == numCities)
    {
      gene += gene[0];
      break;
    }
    int temp = rand_num(1, numCities);
    if (!isRepeat(gene, (char)(temp + 48)))
      gene += (char)(temp + 48);
  }
  return gene;
}

// Function to return the fitness value of a gene. The fitness value is the euclidean distance
int calc_fitness(string gene)
{
  int eucDist = 0;
  for (int i = 0; i < gene.size() - 1; i++)
  {
    eucDist += sqrt((pow(int(cities[gene[i]-48][0])-int(cities[gene[i+1]-48][0]), 2)) + (pow(int(cities
  }
  return eucDist;
}

// Comparator for gene struct.
bool lessthan(struct order t1,
              struct order t2)
{
  return t1.fitness < t2.fitness;
}

int cooldown(int temp)
{
  return (90 * temp) / 100;
}

void runTSP(int rank)
{
  struct order bestYet;
  bestYet.gene = "0";
  bestYet.fitness = 10000000000;
  int stopper = 0;
  int gen = 1;
  // stop at this number of gene iterations
  int maxGen = 1000;
```

```
vector<struct order> population;
struct order temp;

int temperature = 1000000000;

if (rank == 0) {
  // Populating the gene pool.
  for (int i = 0; i < popSize; i++)
  {
    temp.gene = create_gene();
    temp.fitness = calc_fitness(temp.gene);
    if (temp.fitness < bestYet.fitness) {
      bestYet.fitness = temp.fitness;
    }
    population.push_back(temp);
  }

  cout << "\nInitial population: " << endl
      << "gene     FITNESS VALUES\n";
  for (int i = 0; i < popSize; i++)
    cout << population[i].fitness << endl;
  cout << "\n";
}

// MPI_Barrier(MCW);

// perform crossing and gene mutation.
while (gen <= maxGen)
{

  sort(population.begin(), population.end(), lessthan);
  vector<struct order> new_population;

  for (int i = 0; i < popSize; i++)
  {
    struct order p1 = population[i];

    while (true)
    {
      if (stopper > 1000000) {
        break;
      }
      string new_g = mutatedGene(p1.gene);
      struct order new_gene;
      new_gene.gene = new_g;
      new_gene.fitness = calc_fitness(new_gene.gene);

      if (new_gene.fitness <= population[i].fitness)
      {
        if (new_gene.fitness < bestYet.fitness) {
          bestYet.fitness = new_gene.fitness;
        }
```

```
          new_population.push_back(new_gene);
          break;
        }
        else
        {
          // Accept the bad children at a probablity
          float prob = pow(2.7, -1 * ((float)(new_gene.fitness - population[i].fitness) / temperature))
          if (prob > 0.3)
          {
            if (new_gene.fitness < bestYet.fitness) {
              bestYet.fitness = new_gene.fitness;
            }
            new_population.push_back(new_gene);
            break;
          }
        }
        stopper++;
      }
    }

    temperature = cooldown(temperature);
    population = new_population;
    cout << "\nGeneration " << gen << " \n";
    cout << "FITNESS VALUES\n";

    for (int i = 0; i < popSize; i++) {
      cout << population[i].fitness << endl;
    }
    gen++;
    cout << "BEST PATH LENGTH = " << bestYet.fitness << endl;
  }
}

int main(int argc, char **argv)
{
  int rank, size;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MCW, &rank);
  MPI_Comm_size(MCW, &size);
  MPI_Request request;

  if (rank == 0) {
    runTSP(rank);
  }
  MPI_Finalize();
  return 0;
}
```