CS-241 Final Exam: Genetic Algorithm for the Traveling Salesperson Problem

The problem:

Given a connected, weighted graph where:

a. There is not necessarily an edge between every two nodes.
b. Every edge is bidirectional; however, an edge may have a different cost in the reverse direction.
c. There is a special start node that must be the first node visited in any tour.
d. There is a special end node that must be the last node visited in any tour.
e. The Triangle Inequality does not necessarily hold: for example,

   cost(A,B) + cost(B,C) may be less than cost(A,C)

Use a Genetic Algorithm to find a low cost (not necessarily the least cost) tour that:

a. Visits every node at least once.
b. May visit some nodes more than once.
c. Starts at the start node and ends and the end node.
d. May not include all edges.

Data File Format

Each input graph file will be a comma delimited, ASCII file of the form:

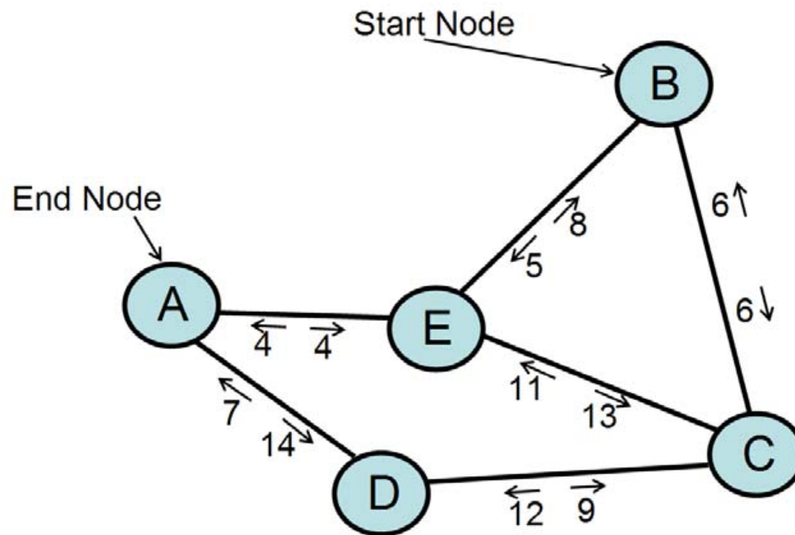
*city*1*, city*2*, cost*12*, cost*21

a. *city1* and *city2* are character arrays.
b. *cost*12 is an integer that represents the cost of flying from *city*1 to *city*2.
c. *cost*21 is an integer that represents the cost of flying from *city*2 to *city*1.
d. All costs will be >$0 and <=$10,000.
e. The first line of the file will be: *startCity*, *endCity* (with no cost given). The start city may or may not be the same as the end city.

For simplicity, you may assume:

1. No input graph will contain more than 100 nodes (cities)
2. No input line (*city1*, *city2*, *cost12*, *cost21*) will contain more than 1000 characters.
3. No cost for a single flight with be larger than $10000
4. All costs are integer dollar amounts.
5. No genome need represent a tour longer than 1000 nodes. If you are constructing a genome that reaches 1000 nodes and is not a complete tour, then start over. This applies to constructing the initial population, crossover, and mutation.

Example Graph:



Both Data file 1 and 2 are representations of this graph in the specified file format:

Data file 1	Data file 2
B,A B,C,6,6 B,E,5,8 A,E,4,4 E,C,13,11 A,D,14,7 C,D,12,9	B,A D,C,9,12 C,E,11,3 B,C,6,6 E,B,8,5 A,E,4,4 A,D,14,7

Represented with the given data structures:

```
struct node
{ char *name;
  struct halfEdge *next;
  int edgeCount;
};
```

```
struct halfEdge
{ struct node *city;
  int cost;
  struct halfEdge *next;
};
```

```
struct node *cityList[MAX_CITIES];
int cityCount = 0;
struct node *startCity, *endCity;
```

There are 5 nodes. They are created in the first seen in the input file. Thus, from data file 1 the order of the nodes in **cityList** would be: B, A, C, E, D.

There are 12 different half edges:

cityList[0]->name: B, **halfEdge:**{ →C(6) →E(5) → null }
cityList[1]->name: A, **halfEdge:**{ →E(4) →D(14) → null }
cityList[2]->name: C, **halfEdge:**{ →B(6) →E(11) →D(12) → null }
cityList[3]->name: E, **halfEdge:**{ →B(8) →A(4) →C(13) → null }
cityList[4]->name: D, **halfEdge:**{ →A(7) →C(9) → null }

The Genome:

In a genetic algorithm, a genome represents a valid solution. Not necessarily the optimal solution or even necessarily a good solution. In this problem, a genome is any valid tour of the cities. Thus, it is any tour that starts at the starting city, ends at the ending city and includes every city at least once.

The ***fitness*** of a solution is the sum of the cost of each edge (flight) in the tour.

A "*good*" solution is one that has a significantly lower total cost than a random solution.

It is unlikely that a good solution will include visiting every city more than an average of ten times. Our problem has a maximum of 100 cities. Thus, we can make the problem easier to code by limiting the maximum tour length to 1000 cities. That is, limiting the genome to a maximum of 1000 genes. This makes the problem easer to code for four major reasons:

1) When building a genome, rather than using malloc to create a new gene (a reference to a city node) and adding the gene in a linked list, the genome can be an array of city nodes.
2) When doing crossover, mutation, and other operations, storing the genome as an array of city nodes grants random access to each gene. For example, **`genome->gene[6]`** is the sixth gene in the tour. To access the sixth gene in a linked list representation, we need to start at the first, and count.
3) An array representation makes it easier to adopt the maze algorithm to this problem.
4) The graph is represented as a linked list. The graph, however, is static in this application: Neither nodes nor edges are ever removed, or changed in any way. This means we do not need to worry about malloc for each node and halfEdge causing a memory leak. The genomes, however, are constantly being created and destroyed. By using an array representation we can malloc memory for every genome at the start of the program. The length of a genome changes not by calling malloc to get a new node for a linked list, but by adding one to **`genome->length`**. A genome is erased by setting **`genome->length`** to 0.

Representation of genome structure:

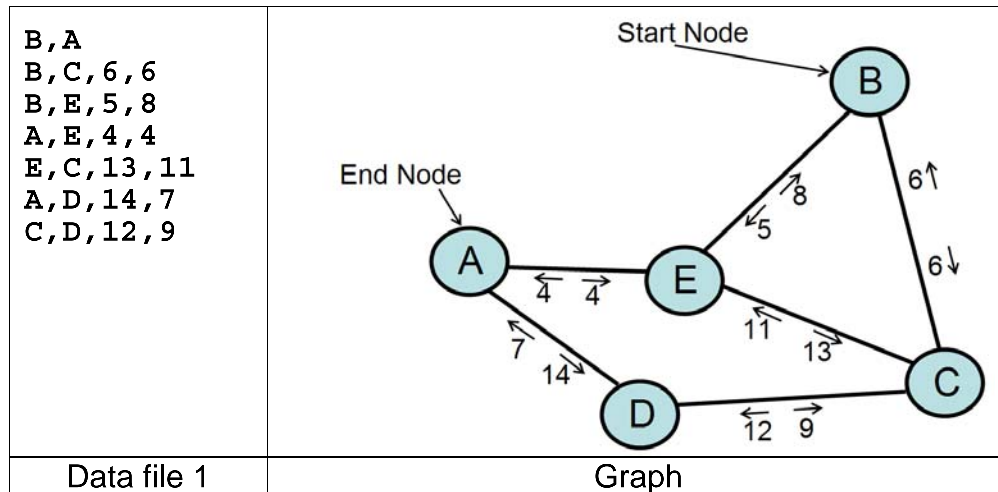
```
#define MAX_TOUR_LENGTH 1000

struct genome
{ int cost;
  int length;
  struct node *gene[MAX_TOUR_LENGTH];
};
```

malloc memory only once, at the program start for each population location. Change the population by changing the values within the structure.

```
void mallocGenoms()
{ int i;
  for (i=0; i<=populationSize; i++)
  { population[i] = malloc(sizeof(struct genome));
  }
}
```

Example Genome:



One valid tour of this graph is BCEADA. This tour is valid because it starts at the start node (B), ends at the end node (A), includes all nodes (A, B, C, D, and E), and all edges are present in the graph: BC, CE, EA, AD, and DA. The cost of this tour is 6+11+4+14+7 = 42.

The following code shows how the structure is used by hard coding the BCEADA genome. Of course, this is not random, and the length and cost should not be constants but calculated as the tour is extended.

```
void setGenome(struct genome* g)
{ g->gene[0] = cityList[0]; //B
  g->gene[1] = cityList[2]; //C
  g->gene[2] = cityList[3]; //E
  g->gene[3] = cityList[1]; //A
  g->gene[4] = cityList[4]; //D
  g->gene[5] = cityList[1]; //A

  g->length = 6;
  g->cost = 42;
}
```

Then, the genome could be printed with:

```
void printGenome(struct genome* g)
{ int i;
  int length = g->length;
  printf("%6d: { ", g->cost);
  for (i=0; i<length; i++)
  { printf("[%s] ",g->gene[i]->name);
  }
  printf("}\n");
}
```

With output: `42: { [B] [C] [E] [A] [D] [A] }`

Grading:

The first code I posted had a bug which was not found until Sunday 5/10. Some of you no doubt struggled with that bug. Also, due to so many other obligations, I just have not gotten to post the test code as quickly as promised. Therefore, I have relaxed the grading. At this point, I just cannot make you interface to my test programs - even if I had them done now, which I do not.

- There are 4 parts to this project:
  1. Produce a “random”, valid starting population.
  2. Calculate fitness and sort the population by fitness.
  3. Given a pair of parents, use crossover to produce a pair of children.
  4. Given a genome to mutate and a second genome from which to “repair” invalid edges, produce the mutated genome.
- Given various test graphs and a population number, produce a “random”, valid starting population. Demonstrate this using the **`printGenome(struct genome* g)`** function provided in geneticTest.c. using TravelingSalesPerson_Test0.txt (with a population of 5) and TravelingSalesPerson_Test1.txt (with a population of 200): **80/100 points.** It just like with the maze program, “random” does not guarantee no duplicates. It is ok if you get some duplicate genomes.
- Get any one of the other parts correct: Demonstrate this by your own output in your own format using TravelingSalesPerson_Test1.txt: **95/100 points.**
- Get any two of the other parts correct: Demonstrate this by your own output in your own format using TravelingSalesPerson_Test1.tx: **110/100 points.**
- Get any three of the other parts correct: Demonstrate this by your own output in your own format using TravelingSalesPerson_Test1.tx: **125/100 points.**
- Put it all together and get a running GA for **150/100 points.**
- The final will be graded not as a project, but as the second exam: averaged with the first and together counting as 20% of the course grade.

Partial credit will be given for parts that you get partly correct.