ECS 60 **Previous Challenge Programs**

Challenge #1

For this assignment, you will be writing a program determine the paths to connect all of the cities on a map.

Here are the specifications:

1. CreateFile.cpp is available and worth perusing.
   - 1.1. The name of a data file reflects parameters used to create it. For example, map-20-1000-5.txt has 20 cities, a map width and height of 1000, and was created using a seed of 5 for the random number generator.
   - 1.2. The first line of the file indicates the number of cities, and the width. All maps are square.
   - 1.3. The next lines provide the coordinates of the cities. The origin of the coordinate system is the lower left corner.
   - 1.4. The rest of the lines are the elevations of each plot in the map, starting from the upper left and working across, and then down. Thus, the map in the file matches the map stored with respect to its origin.
2. General Operation
   - 2.1. main() calls readFile() to read the file into two two-dimensional maps, as well as an array of the cities.
   - 2.2. The CPUTimer is then started.
   - 2.3. One of the plot maps is passed to the Router constructor. When the constructor returns, that map is destroyed, so you must make some copy in your constructor.
   - 2.4. `void Router::findRoutes(const CityPos *cityPos, int cityCount, Edge *paths, int &pathCount)` is the real workhorse of the program. This function must place Edges in paths sufficient to connect all of the cities, and set pathCount to the number of Edges used in the paths array.
     - 2.4.1. Paths may split at intermediate points between cities!
   - 2.5. When findRoutes() returns, the CPUTime is printed.
   - 2.6. checkRoute() ensures that all submitted Edges are for adjacent plots. It also uses dfs to determine the total cost, and to ensure that all cities are connected to each other. If it detects an error, it posts a message indicating the problem, and exits.
3. Grading
   - 3.1. Performance will be tested with three map files, each with twenty cities, and a width of 1000.
   - 3.2. (20 points) Correctly connect all of the cities. This is indicated by having no messages from checkRoute(). If a program does not correctly connect the cities, then it will receive zero for the entire assignment.
   - 3.3. (15 points) CPU time: min (18, 15 * Sean's CPU Time / Your CPU Time);
     - 3.3.1. CPU time may not exceed 60.
     - 3.3.2. Programs must be compiled without any optimization options. You may not use any precompiled code, including the STL and assembly.
   - 3.4. (15 points) Total cost of the paths in the paths array :: min(17, 15 * Sean's total / Your total)
     - 3.4.1. Cost to connect adjacent plots = (change in elevation)^2 + 10.
     - 3.4.2. Each non-edge plot has eight adjacent plots.
     - 3.4.3. My cost may be optimal.

```
typedef struct{
  int x;
  int y;
} CityPos;

typedef struct {  // start and end should be adjacent!
  short startX;
  short startY;
  short endX;
  short endY;
} Edge;

int main(int argc, char* argv[])
{
  short **map, **map2;
  int width, cityCount, pathCount;
  CityPos *cityPos;
  Router *router;
  Edge *paths;
  CPUTimer ct;
  readFile(argv[1], &map, &map2, &width, &cityPos, &cityCount);
  paths = new Edge[4 * width * width];  // maximum number of edges
possible
  ct.reset();
  router = new Router(map, width);

  for(int i = 0; i < width; i++)
    delete [] map[i];
  delete [] map;

  router->findRoutes((const CityPos*) cityPos, cityCount, paths,
pathCount);
  cout << "CPU time: " << ct.cur_CPUTime() ;
  checkRoute(map2, cityPos, cityCount, paths, pathCount, width);
  return 0;
} // main()

[davis@lect2 p5]$ cat map-3-5-1.txt
3 5
1 3
0 2
0 3
  3 55 29  1  2
  5 62 46  3  6
  3 59 83  4  8
  5 72 51  3  6
  4 71 34  2  6
[davis@lect2 p5]$

[davis@lect2 p5]$ router.out map-3-5-1.txt
CPU time: 0 Total cost: 3505
[davis@lect2 p5]$
```

Challenge 2

You are to write a program that determines the minimum blood flow pattern for a brain. You program will be given a list of brain cells and how blood vessels connect them, starting from the carotid artery, and ending with the jugular vein. Your program will determine how much blood should flow between connected brain cells during a series of pulses until all the brain cells have received at least one full blood cell.

Here are the specifications:

4. CreateFile.cpp is available and worth perusing.
   - 4.1. The name of a data file reflects parameters used to create it. For example, brain-25-100-5.txt has 25 brain cells, 100 blood vessels, and was created using a seed of 5 for the random number generator.
   - 4.2. The first line of the file contains the number of brain cells followed by the number of blood vessels, and the length of the shortest path between the first and last cells.
   - 4.3. Each succeeding lines contains information about one blood vessel: <ID> <upstream brain cell ID> <downstream brain cell ID> <carrying capacity>
5. Brain Cells
   - 5.1. Each brain cell has a unique ID. They are assigned their IDs randomly, though the IDs are contiguous. However, the first cell encountered by the carotid artery is ID 0, and the last cell before the jugular vein will have the largest ID.
   - 5.2. Your program should indicate when every brain cell has received a full blood cell.
   - 5.3. Each brain cell now empties exactly one full blood cell during the life of the program. Thus, if cell B is downstream from cell A, and cell A has never received a full blood cell before, then cell A must receive at least two full blood cells during a pulse so that at least one full blood cell will make it to cell B. Note that blood cells are not eaten by the brain cells, and must continue along the blood vessels once emptied. **NOTE: This is a change from the original spec that had each brain cell eat up to 10 full blood cells.**
6. Blood Vessel
   - 6.1. Each blood vessel has a unique ID. They are assigned their IDs randomly, though the IDs are contiguous, and start at zero.
   - 6.2. Each blood vessel connects two brain cells. The blood flows from the first cell specified to the second cell.
   - 6.3. Up to five blood vessels can connect to a single brain cell, except the last may have any number of vessels leading into it.
   - 6.4. Each blood vessel has a blood cell carrying capacity.
   - 6.5. While in a real brain it takes time to transport blood cells, we will assume that they are transported instantly through the whole brain. (You really don't want to simulate time do you?).
     - 6.5.1. Oddly, the system may be thought of as empty at the end of each pulse.
   - 6.6. There are no cycles in the blood vessels for this could cause recycling of empty blood cells.
   - 6.7. The total number of blood cells arriving at a brain cell must equal the total number leaving. When a brain cell empties a blood cell, the number of full cells leaving the brain cell would be one less than entering, and the number of empty cells would be one more than entering. The first and last cells do not obey this conservation of blood rule. The first cell can provide an infinite number of full blood cells, and the last cell can absorb an infinite number of blood cells.
   - 6.8. Each pulse, your class is provided two arrays of ints with one position for each blood vessel. One array is for the flow of full blood cells, and the other is for the flow of empty blood cells in the blood vessels. For each pulse, the class fills each position with the number of blood cells of each type that the corresponding vessel must carry.
7. Grading
   - 7.1. Performance will be tested with three data files, each with 5000 brain cells, and 10000 vessels . Each of these values is the maximum possible for the program.
   - 7.2. (10 points) Correctly feed all cells. This is indicated by having no messages from checkFlows(). If a program does not correctly process the instructions, then it will receive zero for the entire assignment.
   - 7.3. (20 points) CPU time: min (23, 20 * Sean's CPU Time / Your CPU Time);

7.3.1. The program terminates when all brain cells have been fed at least one full blood cell, or the number of pulses reaches **10000**. This is a change from 1000 in the original specs.

7.4. (20 points) Simulated pulses to feed all brain cells

7.4.1. min(23, 20 * (Sean's simulated pulses) / (Your simulated pulses)

```
int main(int argc, char *argv[])
{
  int vesselCount, cellCount, depth, *emptyFlows, *fullFlows, pulses = 0,
totalFed = 0, theirTotalFed;
  Vessel *vessels, *vessels2;
  Cell *cells;
  CPUTimer ct;

  if(argc != 2)
  {
    cout << "Usage: blood.out filename\n";
    return 1;
  } // if wrong number of arguments

  readFile(argv[1], &vessels, &vessels2, &vesselCount, &cellCount,
&depth);
  cells = new Cell[cellCount];
  emptyFlows = new int[vesselCount];
  fullFlows = new int[vesselCount];
  ct.reset();
  Blood *blood = new Blood(vessels2, vesselCount, cellCount, depth);
  delete [] vessels2;

  do{
    theirTotalFed = blood->calcFlows(fullFlows, emptyFlows);
    checkFlows(vessels, fullFlows, emptyFlows, cells, vesselCount,
cellCount,
      pulses, &totalFed);
    if(theirTotalFed != totalFed)
      cout << "At pulse #" << pulses << " your number fed, " <<
theirTotalFed
        << ", does not match ours, " << totalFed << endl;
  } while(++pulses < 10000  && totalFed < cellCount);

  cout << "Time: " << ct.cur_CPUTime() << " Ticks: " << ct.cur_CPUTicks()
    << " Pulses: " << pulses << endl;

  for(int i = 0; i < cellCount; i++)
    if(!cells[i].fed)
      cout << "Cell #" << i << " has not been fed.\n";

  return 0;
}  // main()
```

Challenge #3

You are to write an efficient C++ class, Evac, that will determine the routes taken to evacuate the people of an area through one destination city. Your class constructor will be passed an array of City. The City class, and the Road class objects it contains, are listed on the back of this handout. After main() destroys the City array, it calls Evac::evacuate, passing it the ID of the destination city. The method fills an array or EvacRoutes with the movement of the people out of the evacuated cities into other cities. I have provided the driver program EvacRunner.cpp, and the necessary class stubs evac.h. You may add any classes and/or code you wish to evac.cpp, and evac.h. You may also use, and alter any Weiss files. CreateRoads.out was compiled from CreateRoads.cpp, and creates the evac files used for testing.

Further specifications:

1. Command Line parameter is the name of the City file.
2. Evac Files
   - 2.1. Since EvacRunner.cpp handles all file input, you need not concerned with how to read them.
   - 2.2. File names are appended with three numbers: number of cities, number of bidirectional roads, and the seed used for the random number generator. Note the number of roads is inaccurate where number of roads > (number of cities)^2 / 2, because there cannot be that many unique roads.
   - 2.3. The first line of the file contains the number of cities, and the number of roads.
   - 2.4. The second line of the file contains the ID of the city that is the destination.
   - 2.5. Each of the remaining lines of the file contains information about one city including its ID, coordinates, population, number of roads to adjacent cities, and a list of its roads.
     - 2.5.1. There is at most one bidirectional road between any pair of cities. Each direction of a road has the same peoplePerHour, but a different ID. So there are two times as many IDs as bidirectional roads.
     - 2.5.2. Road information has the following format: <destinationCityID> <peoplePerHour><roadID>
3. Evacuation
   - 3.1. Your class must keep a simulated clock that is initialized to 1, and counts hours. Each Road has a variable indicating how many people it can transport in an hour. You will note that the EvacRoute class has a variable to hold the time of transport.
   - 3.2. Non-destination cities can only hold as many evacuees as their original population. For example, a City with a population of 4000 that has 300 people entering it must have at least 300 people leaving it during the same hour. The number of evacuees in a city is checked at the end of each simulated hour.
   - 3.3. Only the destination city may have more people enter than leave the city.
4. Measurements
   - 4.1. CPU time starts just before constructing your Evac object in main(), and ends after your evacuate() function returns. Thus, your destructors will not be called during CPU time.
     - 4.1.1. You may not have any global variables since they would be constructed before CPU time starts.
   - 4.2. Evac time is the number of simulated hours it took your class to completely empty the area.
5. Grading
   - 5.1. The program will be tested using three 1000 city, 10,000 road files. The measurements will be the total of the three runs. This would simulate about 150,000 evacuees.
   - 5.2. If there are ANY error messages from EvacRunner, then the program will receive zero.
   - 5.3. CPU Time score = min(18, 15 * Sean's CPU / Your CPU)
   - 5.4. Simulated time score = min(18, 15 * Sean's Simulated time / Your simulated time)

```
class Road
{
public:
  int destinationCityID;
  int peoplePerHour;
  int ID;
}; // class Road

class EvacRoute
{
public:
  int roadID;
  int time;
  int numPeople;
  bool operator< (const
EvacRoute &rhs) const;
}; // EvacRoute;
class City
{
public:
  int ID;
  int x;
  int y;
  int population;
  int evacuees;
  Road *roads;
  int roadCount;

  City();
  ~City();
}; // class City
```

```
int main(int argc, char* argv[])
{
  int numCities, numRoads, routeCount, evacID;
  ifstream inf(argv[1]);
  inf >> numCities >> numRoads;
  City *cities = new City[numCities];
  readCities(inf, cities, evacID, numCities);
  EvacRoute *evacRoutes = new EvacRoute[numCities * 5000];
  CPUTimer ct;
  ct.reset();
  Evac *evac = new Evac(cities, numCities, numRoads);
  delete [] cities;
  evac->evacuate(evacID, evacRoutes, routeCount);
  cout << "CPU Time: " << ct.cur_CPUTime() << endl;
  ifstream inf2(argv[1]);
  inf2 >> numCities >> numRoads;
  cities = new City[numCities];
  Road2 *roads = new Road2[2 * numRoads];
  readCities2(inf2, cities, evacID, numCities,roads);
  checker(cities, evacID, numCities, evacRoutes, routeCount, roads);
  return 0;
}

[davis@lect15 private]$ evac.out cities-1000-10000-3.txt
CPU Time: 5.04
Evacuation hours: 24796
[davis@lect15 private]$ evac.out cities-1000-10000-4.txt
CPU Time: 2.14
Evacuation hours: 10388
[davis@lect15 private]$ evac.out cities-1000-10000-5.txt
CPU Time: 0.6
Evacuation hours: 2769
[davis@lect15 private]$ evac.out cities-1000-10000-6.txt
CPU Time: 1.52
Evacuation hours: 6890
[davis@lect15 private]$
```