CPS 222: Computer Science III
Project 3: Shortest Path

Important Dates
Checkpoint 1: 11:59pm, Wednesday, 11/14
Checkpoint 2: 11:59pm, Monday, 11/26
Checkpoint 3: 11:59pm, Monday, 12/3
Final deadline: 11:59pm, Monday, 12/10

At each checkpoint you will receive some high-level written comments intended to help you to improve your solutions (do not expect comments to identify every error/bug!). Each numbered part will also receive a score from the following rubric:
0 points: *No substantive attempt submitted or doesn't compile.*
1 point: *A good start but major functionality is missing or incorrect.*
2 points: *Mostly functional, but has some important errors or does not comply with the specifications.*
3 points: *Nearly complete; only minor issues remain.*
Your accumulated checkpoint scores will comprise 5% of your project grade.

Introduction

The shortest path problem is one of the most well-studied problems in computer science. In this project you will be implementing various algorithms and data structures related to this foundational problem. Immediate applications involve navigation (where vertices in the graph represent locations and edge weights represent some measurement of the cost of travel). That said, when you can abstract away the literal interpretation of the graph as locations in space, other less obvious applications of the shortest path problem arise, as you will see!

Part 1: Dijkstra's Algorithm

1) Adjacency Matrix

In *readGraph.hpp/cpp* write a function,

```
int readGraph(ifstream& fin, double**& matrix, string*& vLabels,
                                                    string**& eLabels).
```

It should read in a adjacency matrix from the given file stream. You've been provided a simple example in *graph.txt*. In this file format

- The first line contains the number of vertices in the graph, followed by the number of edges.
- Next there is a line for each vertex which contains the label for the vertex. You may assume that the labels do not contain whitespace.
- Finally there is a line for each edge. Each line consists of four items: the index of the source vertex, the index of the destination vertex, the weight for the edge, and a label for the edge. You may assume that the labels do not contain whitespace.

Your function should assume `matrix` and `vLabels`, and `eLabels` are references to uninitialized pointers. The function should point them to dynamically allocated arrays, fill them using the data in the file, and return the number of vertices in the graph. The label for non-existent edges can be the empty string. The weight of a non-existent edge should be the value *infinity*. To do this, `#include <limits>`, which gives you access to the static class template `numeric_limits`, which in turn has methods that return various values for all the numeric types. You would need to call `numeric_limits<double>::infinity()`, which returns the special double value that represents infinity. The weight from a vertex to itself should be 0.

You should start implementing *readGraph_TEST.cpp* with test cases for this function that have it read in a file and verify that it properly fills in the arrays. You can use the provided file *graph.txt*, which has a simple example graph, if you like, and/or create your own examples.

2) Dijkstra's Algorithm

In *shortestPath.hpp/cpp* write a function

```
void dijkstra(const double* const * matrix, int numVertices, int source,
                                                   double*& dist, int*& prev)..
```

The function should perform Dijkstra's algorithm on the given adjacency matrix in $\Theta(v^2)$ time. After the algorithm completes, `dist[i]` should be the shortest distance from the source to vertex *i*, and `prev[i]` should be the index of the vertex just before vertex *i* along its shortest path (it doesn't matter what `prev[source]` is). Recall that this algorithm does not need to use a binary heap – it finds the minimum distance vertex and updates distances by iterating through all vertices.

Assume that `dist` and `prev` are uninitialized pointers (so you must allocate these arrays in this function). Assume that `adjMatrix` was generated by the `readGraph` function. Create *shortestPath_TEST.cpp* and test your function out.

3) Find Some Shortest Paths

In *shortestPath.hpp/cpp* write a function

```
int getPath(int source, int dest, const int* prev, int*& path).
```

The function should use the `prev` array (generated by `dijkstra`) to find the path from `source` to `dest`. Assume `path` is a reference to an uninitialized pointer. The function should allocate an array of the appropriate size and fill it with the vertex indices along the path, in order. The function should return the size of the `path` array (i.e. the number of vertices visited by the path). Test your function in *shortestPath_TEST.cpp*.

In *matrixDijkstra.cpp* write a program that takes four command-line arguments in this order: the name of a file containing a graph, the name of the output file, and two strings, the names of the source vertex and destination vertex, respectively. It should create an adjacency matrix and pass it to your `dijkstra` function, then use `getPath` to calculate the path between the given vertices. The output file should be in the same format as the graph file. It should list *all* the vertices in the same order, but it should only include the edges on the path. You should print the edges in the order they appear along the path. So, if you run

```
$matrixDijkstra graph.txt path.txt CandyCastle PeanutBrittleHouse
```

the file *path.txt* should look like this:

```
4 2
GingerbreadPlumTrees
CandyCastle
LicoriceCastle
PeanutBrittleHouse
1 2 5.5 GumdropMountains
2 3 2.9 LollipopWoods
```

In addition to writing the output file the program should calculate the total weight of the shortest path and print it to the screen. It should also `#include <chrono>` and measure the number of microseconds it takes for Dijkstra's algorithm to run and print it to the screen. Nicely label your output.

Note: You should also create your own example graphs to test your code!

4) Adjacency List

In *readGraph.hpp/cpp* write a function,

```
int readGraph(ifstream& fin, int**& adj, double**& weights, int*& lengths,
                                          string*& vLabels, string**& eLabels).
```

This function will read in a graph file (in the same format as above), but instead produce an adjacency list. The adjacency list is represented using several arrays:

- `adj` – An array of arrays of `int`s. The array `adj[i]` should contain the indices of the vertices adjacent to vertex *i*, and **should not be any longer than necessary** to store the indices.
- `weights` – The array `weights[i]` should contain the weights on the edges going from vertex *i* to its adjacent vertices *in the same order as they appear in* `adj`. That is, `weights[i][j]` is the weight on the edge from vertex *i* to vertex `adj[i][j]`.
- `lengths` – An array of the lengths of the arrays stored in `adj` and `weights`.
- `vLabels` – An array of vertex labels.
- `eLabels` – Should be similar to `weights`, except containing the edge labels instead of the edge weights.

The time and space complexity of this function should be Θ(*e* + *v*). An STL `vector` might be helpful. Note: you may assume that the edges in the file are sorted by source index, then by destination index.

So for *graph.txt*, the arrays should be arranged like this:

```
adj[0]: [3]
adj[1]: [2]
adj[2]: [0, 1, 3]
adj[3]: [1]

weights[0]: [3.2]
weights[1]: [5.5]
weights[2]: [2.7, 5.5, 2.9]
weights[3]: [1.3]

lengths: [1, 1, 3, 1]

vLabels: [“GingerbreadPlumTrees”, “CandyCastle”, “LicoriceCastle”,
                                                   “PeanutBrittleHouse”]

eLabels[0]: [“MolassesSwamp”]
eLabels[1]: [“GumdropMountains”]
eLabels[2]: [“IceCreamSea”, “GumdropMountains”, “LollipopWoods”]
eLabels[3]: [“PeppermintForest”]
```

Stop and test this out in *readGraph_TEST.cpp*!

5) Binary Heap

In *BinaryHeap.hpp/cpp*, create a class `BinaryHeap`. This will *not* be a class template! As such, it should have separate header and source files, and should be compiled to its own object file before being linked into any programs that use it, just like old times....

Your heap will *not* be general-purpose (able to store any items and any priorities). Instead, it will assume that the priorities it is given are `double`s and the items associated with those priorities are the integers 0...*n*-1 where *n* is the initial size of the heap. This will allow for efficient implementations of operations important to this specific problem.

Note that you have some significant design leeway with regard to how you implement your heap (one restriction: **you may not use any STL classes or functions to implement your heap**). That means you should budget significant time to think through how you will give it the required capabilities listed below. Please do not jump in right away. Look over all of the requirements and consider the data structure as a whole. Brainstorm **on paper** how you will structure your class and adapt the relevant algorithms to your needs. If you are having trouble getting started, I am happy to brainstorm with you.

Your `BinaryHeap` class should have the following public methods (implement private/protected methods as you see fit):

`BinaryHeap(const double* priorities, int numItems)` – The constructor should take an array of priorities and the number of items in the array (item *i* has priority `priorities[i]`). Your heap may not alter this array. The constructor should set up the heap and must run in $\Theta(n)$ time.

`~BinaryHeap()` – The destructor should clean up any dynamically allocated members of the heap.

`int getMin() const` – Returns the item with the minimum priority, but does not alter the heap. Must run in O(1) time.

`void popMin()` – Removes the minimum element from the heap. Must run in O(log *n*) time.

`bool contains(int item) const` – Returns true if `item` is in the heap. Must be O(1).

`double getPriority(int item) const` – Returns the priority of the given item. Note that this priority might have changed since the heap was created (see below). Must execute in O(1) time.

`void decreasePriority(int item, double newPriority)` – If the item is not in the heap, or the new priority is greater than or equal to the priority associated with item, do nothing. Otherwise, change the priority of the item and fix the heap accordingly. Must execute in O(log *n*) time.

`int getSize() const` – Returns the number of items currently in the heap. Must be O(1).

`int getItem(int pos) const` – Mainly for testing/debugging. Returns the item at the given position in the heap (so, `h.getItem(0)` should be equivalent to `h.getMin()`). Must be O(1).

`int getPos(int item) const` – Mainly for testing/debugging. Returns the position of the given item in the heap (so, for instance `h.getPos(h.getMin())` should return 0). Must be O(1).

Once you have a plan, start implementing your heap. As always, create *BinaryHeap_TEST.cpp* and **thoroughly test as you go**. It will be **way** harder to debug your next Dijkstra's algorithm implementation if your heap is shaky. If you are having trouble getting a heap working with the required complexities, implement something that is simpler but less efficient so you can make progress (and then come talk to me!). Also note that later parts in the assignment do not depend on the heap, so if you are having trouble you may also consider moving on to those and coming back later.

6) Dijkstra's Algorithm Again

In *shortestPath.hpp/cpp* write a function

```
void dijkstra(const int* const * adj, const double* const * weights,
    const int* lengths, int numVertices, int source, double*& dist, int*& prev).
```

It should implement Dijkstra's algorithm on graphs represented using an adjacency list as described above. It should find the shortest paths from `source` to all other vertices in O(*e* log *v*) time.

In *listDijkstra.cpp* write a program just like *matrixDijkstra*, except using the adjacency list version of Dijkstra's algorithm. *Note: it is possible for the two versions to find different paths with the same distance, and that's okay.*

X) Giving Directions (not graded)

Once you feel good about your implementation, you can try them out on some much bigger graphs, if you want. Download *bigGraphs.zip*. Inside you will find the following files:

- *RL-all-simple.tmg* – Road network data for Rhode Island (about 500 vertices)
- *PA-all-simple.tmg* – Road network data for Pennsylvania (about 8000 vertices)
- *tmgtotxt.py* – A script that takes a tmg file and produces a graph file in the format above.
- *txttotmg.py* – A script that takes a graph file (e.g. your shortest path) and also the original tmg file, and produces a tmg file for the graph.

The graph files are from the METAL project by James D. Teresco, and are provided for educational purposes only. You can find more road networks here: http://tm.teresco.org/graphs/ (use the "simple" version of the graphs). To try out your algorithm, run the *tmgtograph.py* script to generate graph files and use the programs you wrote. Note that that the vertex labels are a little bit hard to read (they are automatically generated based on intersections). You can just grab a couple labels at random if you like. Alternatively, if you go to http://courses.teresco.org/metal/hdx/ and load the tmg file you can visualize the graph. If you click on a waypoint, it will tell you what its label is.

Your program might take a little while but note that these are *extremely* sparse graphs. There should be a dramatic runtime difference between the two versions of the algorithm! After you've generated a shortest path, you can use the *graphtotmg.py* script to produce a tmg file that represents the path. You can load this tmg file at the same website and visualize your path on the map.

---You have reached Checkpoint 1---

Files to turn in by 11:59, Wednesday 11/14:

readGraph.hpp/cpp, shortestPath.hpp/cpp, BinaryHeap.hpp/cpp, matrixDijkstra.cpp, listDijkstra.cpp, readGraph_TEST.cpp, shortestPath_TEST.cpp, BinaryHeap_TEST.cpp, Makefile

and, as always, don't forget to fill out and sign your cover sheet!

Part 2: Dynamic Programming

7) Edge List

You've studied adjacency matrices and adjacency lists. There is a third, somewhat less common graph representation called an "edge list" that is well-suited for the Bellman-Ford algorithm.

In *readGraph.hpp/cpp*, implement the function

```
int readGraph(ifstream& fin, int**& edgeList, double*& weights,
                          int& numEdges, string*& vLabels, string*& eLabels).
```

It should assume that `edgeList`, `weights`, `vLabels`, and `eLabels` are all uninitialized pointers and allocated arrays accordingly. The `edgeList` should be an array of edges, where each edge is represented by a length 2 array (the first element is the source vertex, the second is the destination). The `weights` array stores the weight of each edge, in the same order as they appear in `edgeList`. The variable `numEdges` should be set to the number of edges. The `vLabels` array is the list of vertex labels as before. The `eLabels` array stores the edge labels, in the same order as they appear in `edgeList`. Again you may assume that the edges are sorted by source index and then by destination index. The function should return the number of vertices.

So for *graph.txt*, the arrays should be arranged like this:

```
edgeList[0]: [0, 3]
edgeList[1]: [1, 2]
edgeList[2]: [2, 0]
edgeList[3]: [2, 1]
edgeList[4]: [2, 3]
edgeList[5]: [3, 1]

weights: [3.2, 5.5, 2.7, 5.5, 2.9, 1.3]

numEdges: 6

vLabels: ["GingerbreadPlumTrees", "CandyCastle", "LicoriceCastle",
                                                   "PeanutBrittleHouse"]

eLabels: ["MolassesSwamp", "GumdropMountains", "IceCreamSea",
                      "GumdropMountains", "LollipopWoods", "PeppermintForest"]
```

Test it in *shortestPath_TEST.cpp*!

8) Bellman-Ford

In *shortestPath.hpp*/cpp implement the function:

```
int bellmanFord(const int* const * edges, const double* weights,
          int numVertices, int numEdges, int source, double*& dist, int*& prev)
```

This function should implement the Bellman-Ford algorithm to find the shortest path from the vertex `source` to all other vertices in the weighted graph represented by the adjacency matrix called `graph`. It takes an edge list, as generated by your function above. The weights are not assumed to be positive.

Unlike your `dijkstra` function, this function has a return value. Recall that the Bellman-Ford algorithm can detect whether the graph contains a negative cycle by performing an extra *v*th iteration. Your implementation should perform this iteration and update `dist` and `prev`, as it does so. If no updates are necessary, the graph contains no negative cycle, and the function should return -1.

Otherwise there must be some vertex that was updated on the *v*th pass through the graph. In this case, your function should return the index of a vertex that was updated (if multiple vertices were updated, it doesn't matter which one you return). Test your function in *shortestPath_TEST.cpp.*

9) Find Some More Shortest Paths

Write the program *bellmanFord.cpp*, which is just like *matrixDijkstra* and *listDijkstra*. For now, if `bellmanFord` returns a vertex (i.e. not -1), you can simply print an error message and quit. Note that if all weights are positive, the Bellman-Ford algorithm and Dijkstra's algorithm should produce the same answer (again, with the possible exception of tie breaking).

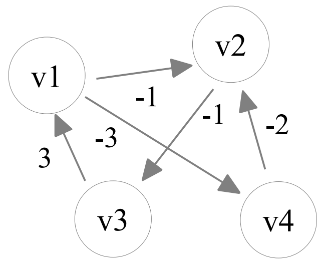
If a negative cycle is detected, your function will return the index of the vertex that was updated in the *v*th pass. **This vertex may not necessarily be part of the cycle**, but you can use it to *find* the cycle by following previous links back until you see some vertex twice. The vertices in between the two sightings comprise the cycle.

Now, in *shortestPath.hpp/cpp*, implement the function
`int getCycle(int vertex, const int* prev, int numVertices, int*& cycle).`
The function should take the vertex index returned by `bellmanFord` and use the `prev` array to find the cycle. Assume `cycle` is a reference to an uninitialized pointer. The function should allocate an array of the appropriate size and fill it with the vertex indices along the cycle, in order. Because it is a cycle, it should start and end with the same vertex, and it doesn't actually matter which vertex in the cycle that is. The function should return the size of the `cycle` array. Test this in *shortestPath_TEST.cpp.*

Hint: You can use an array or vector of `bool`*s to efficiently map each vertex to a boolean value recording whether or not you've seen it in the path so far.*

Now alter *bellmanFord.cpp* so that if `bellmanFord` detects a negative cycle, it generates a file that represents that cycle. If there are multiple cycles, it doesn't matter which one you print out. In this case, the program should print to the screen some warning that a negative cycle was detected. It should print the total weight of the cycle where it would normally print the total weight of the path, and the runtime as usual. For instance, in the graph shown here (also included in *graph2.txt*), the program might print something like:

```
Negative cycle detected!
Total weight: -3
Runtime: 16 microseconds
```



The output file might look like:

```
4 4
v1
v2
v3
v4
0 3 -3 v1->v4
3 1 -2 v4->v2
1 2 -1 v2->v3
2 0 3 v3->v1
```

Note that it is possible that your output would differ if you found a different cycle or printed the cycle with the different start/end. That's okay! Just verify that your cycle is correct.

10) Shortest Path and Currency Exchange

Consider the exchange rates between a bunch of currencies[1]:

	Pounds St.	Euros	Yen	Swiss Francs	US Dollars	Gold
Pounds St.	1	1.4599	189.05	2.1904	1.5714	0.004816
Euros	0.684978	1	129.52	1.4978	1.0752	0.003295
Yen	0.00528961	0.00772082	1	0.011574	0.008309	0.0000255
Swiss Francs	0.456458	0.667646	86.4006	1	0.7182	0.002201
US Dollars	0.0636375	0.93006	120.351	1.39237	1	0.003065
Gold	207.641	303.49	39215.7	454.339	326.264	1

Assume that each exchange costs 0.1% of the amount changed. So if you want to change £1000 (Pounds sterling) to US dollars, you would get £1000 * 1.5714 * 0.999 = $1569.8286.

The thing is, you could actually do a little better in this case by first exchanging to Swiss francs:

$$\pounds 1000 * \underbrace{(2.1904 * 0.999)}_{\text{Convert £ to SFr.}} * \underbrace{(0.7182 * 0.999)}_{\text{Convert SFr. to \$}} = \$1570$$

Okay, so it's only 18-19 cents better. But if you were a big-time currencies trader $0.18 here and there could add up! Clearly, finding the best series of exchanges is related to the shortest path problem. You might think of making a graph with exchange rates as weights (times the transaction cost) but there is a problem: the best sequence of exchanges is the one with the *largest product* of exchange rates. Bellman-Ford, on the other hand finds the path with the *minimum sum* of weights.

The trick to get around this is to instead weight the edges with -log(exchange rate). Because log(*ab*) = log(*a*) + log(*b*), this effectively turns the product into a sum. The negative makes it so what would have been the maximum is now the most negative. The upshot: finding the path with the minimum sum of -log(exchange rate)s is the same as finding the path with the maximum product of exchange rates.

In *currency.cpp* write a program that uses the Bellman-Ford algorithm to find the best exchange rate from a given currency to all other currencies. It should be a lot like *bellmanFord.cpp* except it takes a fifth argument: a floating point number representing the transaction cost.

The file *exchangeRates.txt* represents the example above in the usual format. You can use `readGraph` as you did before, but after it is done, you'll need to edit the weights. The weights should be -log(exchange rate * (1 - transaction cost)). You can use the `log` function in `<cmath>`, which computes the natural log.

The program should generate output files in the same way, but the weights in those files should be *exchange rates*, accounting for the fee (not the transformed numbers the algorithm actually works with). Similarly, instead of printing the total sum of weights to the screen, it should print the *effective exchange rate*. So, when it comes time to generate output, you'll have to turn the distances back into exchange rates. The `exp` function (also in `<cmath>`) will come in handy for this. Calling `exp(x)` returns the value e^x, so `exp` is the inverse of `log`.

1 Example taken from a presentation by Roger Sedgewick and Kevin Wayne

As long as no negative cycle is detected, the output should give the best exchange sequence from the source currency to the destination. For instance, if the source is GBP, the destination is USD, and the fee is 0.001, then it should print something like:

```
Effective Exchange Rate: 1.57
Runtime: 20 microseconds
```

and the output file should look something like

```
6 2
GBP
EUR
JPY
CHF
USD
GLD
0 3 2.1882096 GBP->CHF
3 4 0.7174818 CHF->USD
```

In this context, a negative cycle is called an *arbitrage opportunity*. Arbitrage is essentially the act of starting with some amount of *x*, performing a series of exchanges, and ending up with more *x* than you started with. Pure profit! Of course, if arbitrage were possible all the time, everything would go haywire: it would be possible to generate infinite money by just exchanging currencies. In principle, if the exchange rates are sane, then any loop should have an effective exchange rate of 1. However, when you set a small enough transaction cost, arbitrage becomes possible! If your program detects a negative cycle, it should behave much like *bellmanFord.cpp*, print a warning and generate a file representing the cycle. For instance, if the source is GBP, the destination is USD, and the fee is 0, then it might print something like:

```
Arbitrage Opportunity Detected!
Effective Exchange Rate: 1.00129
Runtime: 20 microseconds
```

The output file should look something like

```
6 3
GBP
EUR
JPY
CHF
USD
GLD
2 5 0.0000255 JPY->GLD
5 4 326.264 GLD->USD
4 2 120.351 USD->JPY
```

The fact is, with limited precision floating point numbers, arbitrage is inevitable due to rounding errors. Transaction fees for changing currencies are not just there to make money for the exchanger; they also stabilize the market by making transactions too expensive to exploit these tiny arbitrage opportunities.

---You have reached Checkpoint 2---

Files to turn in by 11:59pm, Monday, 11/26:

readGraph.hpp/cpp, shortestPath.hpp/cpp, BinaryHeap.hpp/cpp, bellmanFord.cpp, currency.cpp, readGraph_TEST.cpp, shortestPath_TEST.cpp, BinaryHeap_TEST.cpp, Makefile

and, as always, don't forget to fill out and sign your cover sheet!

Part 3: Parallel Algorithms

11-14) Parallel Dijkstra's Algorithm

Consider the following pseudocode for Dijkstra's algorithm for finding the shortest path from a source vertex to all other vertices in a weighted graph using an adjacency matrix (assume the weight of a missing edge is infinity):

```
DIJKSTRA(G, s)
1  INITIALIZE(G)
2  s.dist = 0
3  for i = 1 up to v - 1
4      x = GETMINVERTEX(G)
5      x.inTree = TRUE
6      UPDATEDISTANCES(G, x)
```

Three helper functions are put to use. The INITIALIZE function should initialize important quantities:

```
INITIALIZE(G)
1  for all vertices x in G
2      x.dist = ∞
3      x.prev = NIL
4      x.inTree = FALSE
```

The GETMINVERTEX function should return the vertex not in the tree with the minimum distance:

```
GETMINVERTEX(G)
1  minDist = ∞
2  minVertex = NIL
3  for all vertices x in G
4      if x.dist < minDist and not x.inTree
5          minVertex = x
6          minDist = x.dist
7  return minVertex
```

The UPDATEDISTANCES function should update the distances of the vertices adjacent to the given vertex:

```
UPDATEDISTANCES(G, x)
1  for all vertices y in G
2      if not y.inTree and y.dist > x.dist + G.weights[x, y]
3          y.dist = x.dist + G.weights[x, y]
4          y.prev = x
```

The main loop (DIJKSTRA line 2) is not a good candidate for parallelization. Each iteration relies upon the results of the previous iteration. However, the three helper functions can all be done in parallel. In *shortestPath.hpp/cpp* implement the function

```
void dijkstra(const double* const * graph, int numVertices, int source,
                                   double*& dist, int*& prev, int numThreads)
```

It should perform Dijkstra's algorithm using three parallel helper functions (use the recursive structure we have seen in class examples to minimize the overhead of creating threads and to balance load).

In *parallelDijkstra.cpp* write a program like the other Dijkstra programs that outputs the results of the parallel Dijkstra's algorithm. Add another command line argument that gives the number of threads to use. One major difference: you should measure and output both the CPU time and the wall clock time of the execution of `dijkstra` (see class examples). For instance, when calculating the shortest path from the Candy Castle to the Peanut Brittle House using *graph.txt* and 2 threads, your output might look something like:

```
Total distance: 8.4
CPU time: 595 microseconds
Wall clock time: 527 microseconds
```

When testing your code for correctness in *shortestPath_TEST.cpp*, **you should test each of your helper functions individually** to make sure they behave as intended.

Note: in my experience parallel Dijkstra's does not yield much, if any speed-up in practice on some computers. This has to do with the overhead of C++ threads. On lab machines you should expect to see a modest speedup for big graphs. Don't be too concerned if you test it on your own computer and don't see much speed up with multiple threads (or even slow down!)

15) Analyze Parallel Dijkstra's

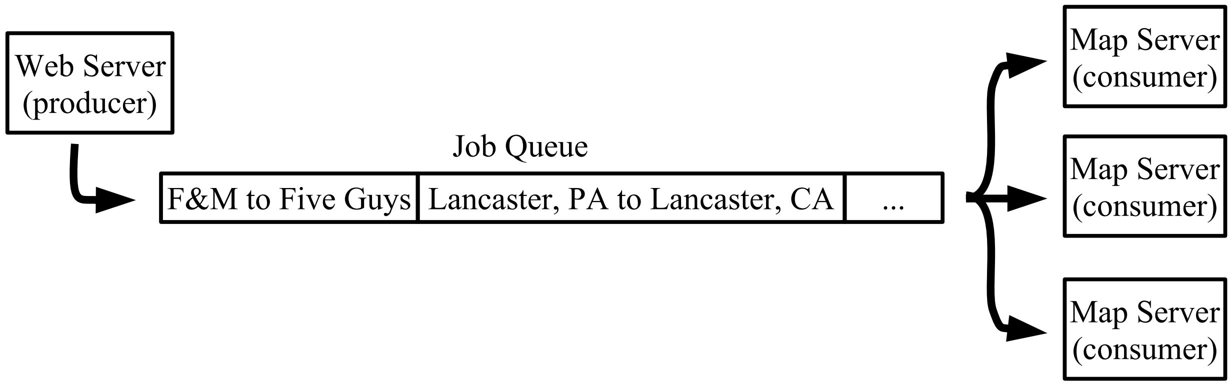
In *analysis.pdf*, analyze the parallelism of parallel Dijkstra's algorithm. Specifically....

- Derive the *work* and *span* of each of the helper functions in terms of *v*, the number of vertices.
- Using those quantities, derive the *work*, *span*, and *parallelism* of the overall algorithm.
  - *Hint: you already know the complexity of serial adjacency matrix Dijkstra's so the work had better come out to at least that complexity!*
- Recall that with an adjacency list representation and a binary heap, Dijkstra's algorithm can be performed serially in O(*e* log *v*) time, which is efficient in sparse graphs. Answer the following questions comparing parallel matrix Dijkstra's to serial list Dijkstra's. (Note: these questions are about the results of the complexity analysis, *not* the observed runtime of your particular program on your particular computer).
  1. Consider the parallel algorithm on an idealized parallel computer (i.e. with infinitely many independent processors). Given your analysis, which algorithm would you prefer to use in dense graphs? Which would you prefer in sparse graphs? Briefly explain.
  2. Consider the parallel algorithm on a computer with only 2 processors. Given your analysis, which algorithm would you prefer to use in dense graphs? Which would you prefer in sparse graphs? Briefly explain.
  3. Why would it be less straightforward to parallelize the adjacency list version of Dijkstra's? That is, what does it do differently that would be difficult to parallelize?

Note: Do not neglect the presentation of your analysis. **Please typeset your answers nicely.** You can use the equation editor in Word or OpenOffice or you can use LaTeX if you know it. Your reasoning is just as important (if not *more* important) than your answer, so make sure you take the time to think it through carefully and express yourself well. Show your derivations and clearly explain your reasoning.

16) Producers and Consumers

Say you use your shortest path algorithms to develop a Google Maps rival. Users visit a website and submit requests for routes (in the form of source and destination information). You are just getting your business started, so for now you only have a few servers calculating the routes. Requests from users (via the web server) are placed onto a queue. When a "map server" is available it takes a job from the queue and calculates the route. This is a simple example of the producer-consumer model that arises in many contexts. The general idea is just that there is some process that, over time, produces things that need to be processed (the web server in this case) and there are several (concurrent) processes devoted to processing those things (the map servers).



You've been provided with *mapqueue.cpp* which implements a simple simulation of this system. It has a queue of jobs. A producer thread puts a job on the queue every second (jobs have random durations between 1 and 5 seconds). The consumer threads take jobs off of the job queue. To simulate the time it takes to execute the job, each thread sleeps (halts it processing) for the duration of the job. Once the consumer is done with its "job" (more like a refreshing nap), it tries to take another job from the queue.

Compile the program with *make mapqueue* and run it with no command-line arguments to simulate the system with one map server (one consumer thread). You should get output that looks something like:

```
0 created a job of length 3. Number of jobs left: 1
1 took a job of length 3. Number of jobs left: 0
0 created a job of length 5. Number of jobs left: 1
0 created a job of length 4. Number of jobs left: 2
1 finished its job.
1 took a job of length 5. Number of jobs left: 1
0 created a job of length 4. Number of jobs left: 2
0 created a job of length 1. Number of jobs left: 3
0 created a job of length 3. Number of jobs left: 4
...
```

Of course it's quite clear that one server is not enough to handle all the jobs at this rate. If the server takes a 5s job, 5 more jobs get submitted in the meantime!

You can also pass the number of servers as a command-line argument. Try it with 2. The output should look something like this:

```
0 created a job of length 3. Number of jobs left: 1
1 took a job of length 3. Number of jobs left: 0
0 created a job of length 5. Number of jobs left: 1
2 took a job of length 5. Number of jobs left: 0
0 created a job of length 4. Number of jobs left: 1
1 finished its job.
1 took a job of length 4. Number of jobs left: 0
```

```
0 created a job of length 4. Number of jobs left: 1
0 created a job of length 1. Number of jobs left: 2
0 created a job of length 3. Number of jobs left: 3
2 finished its job.
2 took a job of length 4. Number of jobs left: 2
0 created a job of length 5. Number of jobs left: 3
1 finished its job.
1 took a job of length 3. Number of jobs left: 2
0 created a job of length 4. Number of jobs left: 4
...
```

Notice how the two consumer threads work in parallel, each taking a job whenever it is ready to do so. Notice too that while one server is processing a job, more jobs continue to be farmed out. All seems fine and dandy until you try it with more servers. Run the program with 3, 4, and 5 servers and look carefully at the output. It ranges from occasionally strange to catastrophically broken.

Your task: fix the simulation using mutex. In *mapqueuefix.txt* clearly explain what is wrong with the program as-is and how you addressed it. Submit your corrected *mapqueue.cpp* with the problem fixed.

When running correctly, the output with 5 printers should look something like:

```
0 created a job of length 4. Number of jobs left: 1
1 took a job of length 4. Number of jobs left: 0
0 created a job of length 2. Number of jobs left: 1
2 took a job of length 2. Number of jobs left: 0
0 created a job of length 3. Number of jobs left: 1
5 took a job of length 3. Number of jobs left: 0
0 created a job of length 1. Number of jobs left: 1
2 took a job of length 1. Number of jobs left: 0
0 created a job of length 4. Number of jobs left: 1
1 took a job of length 4. Number of jobs left: 0
...
```

When there are enough servers to handle the load, the queue stays small (or empty!). Also, jobs are produced and consumed in an orderly fashion.

Some notes:

- It may be a good idea to test with ridiculous numbers of servers (20, 100, more?). Sometimes large numbers of threads can cause problems that would be rare with only a small number.
- Sensible behavior is a good sign, but it is not a guarantee of correctness. Recall that errors involving concurrency can be intermittent and capricious. Take care to think through the *logic* of your solution to be sure that you have ruled out the chance of the error resurfacing (and not caused new problems!).
- As you are used to by now, the C++ library does not have safety checks when you break the rules of mutex (most importantly that the mutex should only be unlocked by the thread that locked it). The behavior when you break the rules is “undefined”, meaning that it will often appear to work fine...until it doesn't. Again, be sure the logic of your solution is valid.

---You have reached Checkpoint 3---

Files to turn in by 11:59, Monday 12/3:

readGraph.hpp/cpp, shortestPath.hpp/cpp, BinaryHeap.hpp/cpp, parallelDijkstra.cpp, mapqueue.cpp, shortestPath_TEST.cpp, Makefile, analysis.pdf, mapqueuefix.txt

and, as always, don't forget to fill out and sign your cover sheet!

Part 4: Clean up

You now have a little bit of time before the final submission deadline. Use it wisely! Tie up any loose ends that were still dangling at a checkpoint. Carefully test your code and fix any bugs you find. Write your *readme.txt.* Look over your code for opportunities to improve clarity, elegance, and efficiency.

---Final Submission Deadline---
Files to turn in by 11:59pm, Monday 12/10:
readGraph.hpp/cpp, shortestPath.hpp/cpp, BinaryHeap.hpp/cpp, matrixDijkstra.cpp, listDijkstra.cpp, bellmanFord.cpp, currency.cpp, parallelDijkstra.cpp, mapqueue.cpp, readGraph_TEST.cpp, shortestPath_TEST.cpp, BinaryHeap_TEST.cpp, Makefile, analysis.pdf, mapqueuefix.txt
and, as always, don't forget to fill out and sign your cover sheet!

The grade breakdown for the final submission is as follows:
`readGraph` (adj. matrix): 5
`readGraph` (adj. list): 5
`readGraph` (edge list): 5
`dijkstra` (adj. matrix): 10
`dijkstra` (adj. list): 10
`BinaryHeap`: 15
`bellmanFord`: 10
`dijkstra` (parallel): 15
`getPath/getCycle`: 5
matrixDijkstra.cpp, listDijkstra.cpp, bellmanFord.cpp, parallelDijkstra.cpp, currency.cpp: 10
mapqueue.cpp: 5
Test programs: 5 (get 85% coverage for full credit)
Makefile: 5 pts
mapqueuefix.txt: 5
analysis.pdf: 5
readme.txt: 5 pts
Total: 120 pts
Your final submission grade will comprise 95% of the project grade.

Warning: any file that does not compile on a lab machine will receive 0 credit! Make sure everything compiles and runs before submitting.