

Programming Assignment 5
Tuesday August 15, 10:00 pm

In this assignment you will create a Graph module in C to implement both Dijkstra's algorithm and Bellman-Ford's algorithm for solving the single source shortest paths problem in a weighted directed graph. You may use your graph module from pa4 as a starting point for this project. However, since extensive changes to the design will be necessary, you may choose to start from scratch. In any case it is not required that your graph module in this project be backward compatible with pa4, or that the functionality from that project be maintained in any way.

One significant difference between this project and pa4 is that you will be working with weighted graphs. Each edge weight should be stored as type `double`. These weights should be maintained by including a two dimensional array of doubles as part of your Graph struct. If the graph contains a directed edge from vertex u to vertex v , then the u^{th} row v^{th} column of this array will contain the weight of that edge. If there is no edge from u to v , then the u^{th} row v^{th} column will contain infinity. As usual it is recommended that you avoid index 0, and therefore this array will be of size $(n+1) \times (n+1)$, where n is the number of vertices in the graph.

As in the last two projects vertices will be labeled 1 through n . Both Dijkstra and Bellman-Ford require that vertices possess the attributes *parent* and *distance*. It is recommended that these attributes be included in your Graph struct as a pair of parallel arrays of types `int` and `double` respectively. It is possible to encapsulate parent and distance fields into a private inner struct called Vertex, then include an array of Vertex in your Graph struct, but this design is not ideal, as we'll see below.

Your project will include a separate ADT module called PriorityQueue, which will be used by Dijkstra's algorithm to manage vertices. You are required to implement PriorityQueue as a heap, as described in chapter 6 (pages 127-144) of the text. There are some subtle differences however between the priority queue in this project and the one discussed in the text and in lecture. To begin with, Dijkstra requires a min-priority queue, so the underlying array will be arranged as a min-heap, not a max-heap. This array will consist of the integers from 1 to n , each of which is understood (at the level of the client Graph module) to be the label of some vertex. For purposes of forming a heap however, the 'value' of an array element u is not the integer u , but the distance field of the corresponding vertex. Inside the PriorityQueue module `distance[u]` will be known as `key[u]`. Thus if the underlying array is A , the min-heap property would read:

$$(*) \quad \text{key}[A[\text{parent}(i)]] \leq \text{key}[A[i]] \quad \text{for} \quad 1 \leq i \leq \text{heap_size}$$

The constructor `newPriorityQueue()` must build a heap out of the integer array $A=(1, 2, \dots, n)$ using values in the array of keys to establish this min-heap property. (This is why it is better not to encapsulate the distance and parent attributes of a vertex in your Graph ADT, and instead maintain `distance[]` as a separate array which can then be passed as a parameter to `newPriorityQueue()`.) It is recommended that `heapify()` and `buildHeap()` from chapter 6 be implemented as private functions to facilitate this heap building process. You may also wish to include functions `left()`, `right()`, and `parent()` from 6.1 to navigate the array. Function `newPriorityQueue()` will allocate memory for the PriorityQueue struct, then call `buildHeap()` to arrange the underlying array A as a heap according to property (*) above. When this is

done the values ($\text{key}[A[1]]$, $\text{key}[A[2]]$, ..., $\text{key}[A[n]]$) form a min-heap. Thus the elements of A may be considered to be pointers to keys, since they are actually indices to the key array. It is also very helpful to think of A as being a permutation or re-arrangement of the keys. Note that at no time are elements in the key array swapped, only the elements of A are.

There is another subtle point concerning the underlying array A . Certain of the required functions listed below, such as `inQueue()` and `decreaseKey()`, require that given u in the range $1 \leq u \leq n$, you determine the integer i in the range $1 \leq i \leq n$ such that $A[i] = u$. Now this could be done by doing a linear search of A , but such a search would be inefficient. So inefficient in fact, as to defeat the whole purpose of implementing `PriorityQueue` as a heap. The better (required) approach is to maintain another array I , also containing the integers from 1 to n , that is the inverse permutation of A in the sense that $A[I[u]] = u$ for all u , and $I[A[i]] = i$ for all i . A call to `inQueue(Q, u)` simply checks the inequality $1 \leq I[u] \leq \text{heap_size}$. To implement function `decreaseKey(Q, u, k)`, refer to the algorithm `Heap-Increase-Key` on p. 140 of the text, adapt it for a min-heap by reversing certain inequalities, and note that as stated, it requires an array index i as input, not an array element u . This is remedied by setting $i = I[u]$ then translating the pseudo-code into C. Creating and maintaining array I is not difficult. Both A and I are initialized by the constructor `newPriorityQueue()` to be the ‘identity’ permutation (1, 2, ..., n). Certain functions will perform swaps of the elements of A . These include the `PriorityQueue` operations `deleteMin()` and `decreaseKey()`, and the private function `heapify()`. To maintain I use the following rule: whenever a swap of the form $A[i] \leftrightarrow A[j]$ is performed in any of these functions, perform the inverse swap $I[A[i]] \leftrightarrow I[A[j]]$ as well. This rule guarantees that array I always represents the inverse permutation of A . (Confirm this on a few small examples if necessary to convince yourself.) A private function with the following signature may be useful in this regard:

```
void swap(int* myArray, int i, int j)
```

One additional point concerns the array of keys. The above discussion suggests that your `PriorityQueue` struct should include fields of type `int*` for the array A and its inverse I . Memory for these arrays should of course be allocated by the constructor `newPriorityQueue()`. Another field of type `double*` should also be included for the array of keys. However, memory for this array *should not* be allocated by the `PriorityQueue` constructor, since it belongs to the client (`Graph`) module, and will already have been allocated by its constructor. Thus `newPriorityQueue()` will take `Graph`’s array of vertex distances as input, then simply set the `double*` key field in the `PriorityQueue` struct to point to this array. In other words, both `Graph` and `PriorityQueue` contain fields of type `double*` which point to the very same array. This gives `PriorityQueue` the ability to reach in and alter the internal state of its client `Graph` via the function `decreaseKey()`. Note that this does not constitute a breach of the data hiding doctrine however, since that principle only goes one way: a client module may not see or affect what goes on inside its ADT service module except through exported operations.

Your `PriorityQueue` module will export the following functions:

```
/* Constructors-Destructors */
PriorityQueueRef newPriorityQueue(int n, double* key);
void freePriorityQueue(PriorityQueueRef* pQ);

/* Access functions */
int getNumElements(PriorityQueueRef Q);
int getMin(PriorityQueueRef Q);
int inQueue(PriorityQueueRef Q, int u);
```

```

/* Manipulation procedures */
void deleteMin(PriorityQueueRef Q);
void decreaseKey(PriorityQueueRef Q, int u, double k);

/* Other Functions */
void printPriorityQueue(FILE* out, PriorityQueueRef Q);

```

Function `newPriorityQueue()` creates a new min priority queue object consisting of the integers from 1 to n , and ordered according to the values in the array `key[1..n]`. `freePriorityQueue()` frees all dynamically allocated memory associated with the `PriorityQueueRef *pQ`, and sets `*pQ` to NULL. Function `getNumElements()` returns the number of integers remaining in `Q`. `getMin()` returns the integer (i.e. vertex) u for which `key[u]` (i.e. `distance[u]`) is smallest. It has the precondition `getNumElements(Q) ≥ 1`. `inQueue()` returns 1 (true) if its argument u belongs to `Q`, and returns 0 (false) otherwise. Function `deleteMin()` deletes the element in `Q` with smallest key. It has the precondition: `getNumElements(Q) ≥ 1`. `decreaseKey()` sets `key[u] = k` if $k < \text{key}[u]$, and does nothing otherwise. It has the precondition `inQueue(Q, u)`. `printPriorityQueue()` prints the state of `Q` to the file handle `out`. This function is used only for diagnostic purposes, so the format in which the internal state of `Q` is expressed is not specified, and is therefore up to you.

Your Graph ADT interface file `Graph.h` should define constant macros to stand for nil and infinity called `NIL` and `INF` respectively. Your Graph module will export the following functions.

```

/* Constructors-Destructors */
GraphRef newGraph(int n);
void freeGraph(GraphRef *pG);

/* Access functions */
int getOrder(GraphRef G);
int getParent(GraphRef G, int u);
double getDistance(GraphRef G, int u);
int getSource(GraphRef G);
void getPath(GraphRef G, int s, int u, ListRef P);

/* Manipulation procedures */
void addDirectedEdge(GraphRef G, int u, int v, double w);
void Dijkstra(GraphRef G, int s);
int BellmanFord(GraphRef G, int s);

/* Other Functions */
GraphRef copyGraph(GraphRef G);
void printGraph(FILE* out, GraphRef G);

```

Function `newGraph()` returns a reference to a new Graph structure containing n vertices and no edges. `freeGraph()` frees all dynamically allocated memory associated with the `GraphRef *pG` and sets `*pG` to NULL. `getOrder()` returns the number of vertices in `G`. `getParent()` returns the parent of vertex u in `G`. Function `getDistance()` returns the distance from the most recent source to u . `getSource()` returns the most recent source vertex to either `Dijkstra` or `BellmanFord`. `getPath()` assembles a List `P` consisting of the vertices in a minimum weight path from source s to destination u . It has the preconditions `isEmpty(P)` and `getSource(G) == s`. Function `addDirectedEdge()` adds v to the adjacency list of u , and sets the u^{th} row, v^{th} column of the weight array to w , establishing a directed edge from u to v of weight w . `Dijkstra()` performs `Dijkstra's` algorithm with source s on a directed graph `G` which contains no negative weight

edges. `BellmanFord()` performs the Bellman-Ford algorithm with source s on a directed graph G . It returns 1 (true) if no negative weight cycle is reachable from s , and 0 (false) otherwise. `copyGraph()` returns a copy of Graph G . Source, distance, and parent fields of the copy are initialized to NIL, INF, and NIL respectively, regardless of their values in the G . In all other respects, the state of the copy matches that of G . `printGraph()` prints the adjacency list representation of G to the file handle `out`, formatted according to the specifications below. Functions `getParent()`, `getDistance()`, `getPath()`, `addDirectedEdge()`, `Dijkstra()`, and `BellmanFord()` all have preconditions asserting that their various vertex parameters are in the range 1 to n , where $n = \text{getOrder}(G)$.

It is recommended that your `Graph.c` file contain private functions `Initialize()` and `Relax()` as described in chapter 24 (pp. 585-586) of the text. Recall however that when Dijkstra relaxes an edge, it must call `decreaseKey()` on a `PriorityQueue` of vertices, while Bellman-Ford does not use a `PriorityQueue`. Therefore it is recommended that `Graph.c` contain two different versions of `Relax()`, called say `Relax1()` and `Relax2()` with prototypes

```
void Relax1(GraphRef G, int u, int v, PriorityQueueRef Q);
void Relax2(GraphRef G, int u, int v);
```

to be used by `Dijkstra()` and `BellmanFord()` respectively.

The top level client module in this project will be called `FindPath`, and will be invoked at the command line with arguments giving the input and output files: `%FindPath infile outfile`. The input file will begin with a line containing a single integer giving the number of vertices in the graph, followed by some lines of the form: “ $u \ v \ w$ ” which specify the existence of an edge from vertex u to vertex v of weight w . While reading this first section of the file, your program will record whether or not any of the edge weights are negative. This section will be terminated by the dummy line “0 0 0”. After the first section is read, your program will print the adjacency list representation of G to the output file, formatted as in the examples below. The second part of the input file will consist of a number of lines of the form “ $s \ u$ ” which specify a source vertex s and destination vertex u . This section of the input file will be terminated by the dummy line “0 0”. For each source-destination pair, your program will attempt to find a shortest s - u path. If the graph contains no negative weight edges your program will use Dijkstra’s algorithm, otherwise it will use Bellman-Ford. The output file format is illustrated in the following examples.

Input File:

```
7
1 2 1
1 4 4
2 5 1
3 7 2
5 4 1
6 2 2
6 3 1
6 5 4
7 6 1
0 0 0
1 4
3 4
1 7
0 0
```

Output File:

```
1: (2, 1.0) (4, 4.0)
2: (5, 1.0)
3: (7, 2.0)
4:
5: (4, 1.0)
6: (2, 2.0) (3, 1.0) (5, 4.0)
7: (6, 1.0)

A shortest path from 1 to 4 of length 3.0 is: 1 2 5 4
A shortest path from 3 to 4 of length 7.0 is: 3 7 6 2 5 4
No path from 1 to 7 exists
```

Observe that the adjacency list representation given above includes the weights of each edge. Your `printGraph()` function should be written to conform to this format. Notice also that shortest paths are specified by just listing their vertices (with no arrows “->” in between as in pa3.) The next example includes some negative weight edges.

Input File: 7 1 2 1 1 4 4 2 5 -1 3 7 -2 5 4 1 6 2 2 6 3 -1 6 5 4 7 6 1 0 0 0 1 4 3 4 1 7 0 0	Output File: 1: (2, 1.0) (4, 4.0) 2: (5, -1.0) 3: (7, -2.0) 4: 5: (4, 1.0) 6: (2, 2.0) (3, -1.0) (5, 4.0) 7: (6, 1.0) A shortest path from 1 to 4 of length 1.0 is: 1 2 5 4 The SSSG problem is not solvable from source 3 No path from 1 to 7 exists
--	--

Notice that when Bellman-Ford detects a negative weight cycle reachable from the source, the program prints a message to the effect that the SSSG problem is not solvable.

Your project will be tested on directed graphs with no more than 1,000 vertices and 100,000 edges in which no edge weight exceeds 1,000. Therefore an adequate value to represent infinity is 100,000,001. One problem arises however when a large number is used to represent infinity. Infinity has the property that if any number is added or subtracted from it, the result is still infinity, while no large number standing in for infinity has the same property. This could cause problems with the operation of Bellman-Ford in particular, and possibly with Dijkstra also. You need to find a way to work around these problems when they occur.

You are required to submit the following files: `README`, `Makefile`, `PriorityQueue.c`, `PriorityQueue.h`, `PriorityQueueClient.c`, `List.c`, `List.h`, `ListClient.c`, `Graph.h`, `Graph.c`, `GraphClient.c`, `FindPath.c`. As usual `README` contains a catalog of submitted files and any special notes to the grader. `Makefile` should be capable of making the executables `PriorityQueueClient`, `ListClient`, `GraphClient`, `FindPath`, and should contain a clean utility which removes all object files. Let me say it once again: do not submit extra files, especially binary files. You may alter the following `Makefile` as you see fit:

```
# Makefile for Graph ADT and related modules.
# make                               makes FindPath
# make GraphClient                   makes GraphClient
# make PriorityQueueClient           makes PriorityQueueClient
# make ListClient                     makes ListClient
# make clean                          removes all object and executable files

FindPath : FindPath.o Graph.o PriorityQueue.o List.o
           gcc -o FindPath FindPath.o Graph.o PriorityQueue.o List.o

GraphClient : GraphClient.o Graph.o PriorityQueue.o List.o
```

```
gcc -o GraphClient GraphClient.o Graph.o PriorityQueue.o List.o
PriorityQueueClient : PriorityQueueClient.o PriorityQueue.o
gcc -o PriorityQueueClient PriorityQueueClient.o PriorityQueue.o

ListClient : ListClient.o List.o
gcc -o ListClient ListClient.o List.o

FindPath.o : FindPath.c Graph.h
gcc -c -ansi -Wall FindPath.c

GraphClient.o : GraphClient.c Graph.h
gcc -c -ansi -Wall GraphClient.c

PriorityQueueClient.o : PriorityQueueClient.c PriorityQueue.h
gcc -c -ansi -Wall PriorityQueueClient.c

ListClient.o : ListClient.c List.h
gcc -c -ansi -Wall ListClient.c

Graph.o : Graph.c Graph.h PriorityQueue.h List.h
gcc -c -ansi -Wall Graph.c

PriorityQueue.o : PriorityQueue.c PriorityQueue.h
gcc -c -ansi -Wall PriorityQueue.c

List.o : List.c List.h
gcc -c -ansi -Wall List.c

clean :
rm -f FindPath GraphClient PriorityQueueClient ListClient\
FindPath.o GraphClient.o PriorityQueueClient.o ListClient.o\
Graph.o PriorityQueue.o List.o
```