CS201: Data Structures and Algorithms

Assignment 3

Version 3

Introduction

Your task is to implement Dijkstra's shortest path algorithm for undirected graphs, using a binomial heap as the basis for a priority queue.

Input

Your program will process a file containing a description of a graph. A graph description contains an arbitrary number of edge descriptions. An edge description consists of two vertices (optionally followed by a weight) followed by a semicolon. A vertex is simply a non-negative integer. If a weight is omitted, a weight of 1 should be assumed. A weight is a positive integer.

The file should be free format; whitespace may appear anywhere. Here a sample graph description:

```
1 5 ;
2
  10
  23 ;

214 33 1
;
```

which is equivalent to:

```
1 5 ;
2 10 23 ;
214 33 1 ;
```

In this example there are six vertices, named 1, 2, 5, 10, 33, and 214, and three edges, 1 to 5, 2 to 10 and 214 to 33, with weights 1, 23, and 1, respectively.

The name of your executable must be *dijkstra* and the name of the file describing the graph will be passed to your program as command line arguments, as in:

```
$ cat g1
1 2 1 ;
2 3 2 ;
3 1 3 ;
$ dijkstra g1
0: 1
1: 2(1)1 3(1)3
----
$
```

Your program should interpret the graph description as an undirected graph and should report the path weight and the path itself from the source to each of the destination vertices.
Your program should read in data into a dynamically-allocated array.

Execution

In order for your program to run on a randomly created graph, if an edge is given more than once, replace the edge weight of the vertices if the subsequent edge weight is smaller. Note: a u, v edge is the same edge as a v, u edge.

Let the smallest vertex in the graph be the source vertex. If, after running Dijkstra's algorithm and there are unreachable vertices, run the algorithm again (and again) with the smallest unreached vertex. Repeat this process as long as necessary.

If more than one vertex is eligible to be added to the shortest path tree at any given point, chose the vertex with the smaller vertex number.

Output

The output of your program should be a shortest path forest, with each tree displayed as a breadth-first traversal. Trees are presented in the order in which they are found, starting with the smallest vertex in the tree. Here is an example display:

```
$ cat g2
10 0 9 ;
6 7 11 ;
5 9 ;
4 8 7 ; 10 8 6 ;
11 12 2 ;
1 6 5 ;
0 6 10 ; 0 5 3 ; 0 4 8 ; 0 3 2 ; 0 1 4 ;
$ dijkstra g2
0 : 0
1 : 3(0)2 5(0)3 1(0)4 4(0)8 10(0)9
2 : 9(5)4 6(1)9 8(4)15
3 : 7(6)20
----
0 : 11
1 : 12(11)2
----
$
```

For each tree in the forest, you should perform a breadth-first (level-order) traversal, rooted at the origin vertex. Each level of the traversal starts with the level number (level 0 is the first level) and a colon and is followed by a list of vertex descriptions. A vertex description is the vertex followed by its predecessor (in parentheses) followed by the weight of the path from the origin to the vertex in question. The vertex descriptions in a level are to be ordered by increasing path weight with ties being broken in favor of the smaller destination vertex. You must follow the format exactly as *diff* will be used to assess your output.

The dynamic array module

You will need a generic dynamic array class. Here is a conforming dynamic array header file:

```
#include <stdio.h>

#ifndef __DARRAY_INCLUDED__
#define __DARRAY_INCLUDED__

typedef struct DArray DArray; //forward declaration of the DArray struct

extern DArray *newDArray(void (*display)(FILE *,void *));
extern void insertDArray(DArray *a,void *v);
extern void *removeDArray(DArray *a);
extern void *getDArray(DArray *a,int index);
extern void setDArray(DArray *a,int index,void *value);
extern int sizeDArray(DArray *a);
extern void displayDArray(FILE *,DArray *a);
#endif
```

Note the lack of the structure definition. We are going to make the members of the structure private (within the *darray.c* file). For a dynamic array, you will need members similar to:

- `void **array` //an array of void pointers
- `int capacity` //total number of slots
- `int size` //number of filled slots
- `void (*display)(FILE *,void *)`

The capacity of the dynamic array should start out as one. The dynamic array's capacity should double when an attempt is made to add an item to a filled array. The array should shrink by half when the size is less than 25% of the capacity.

The *setDArray* method should call *insertDArray* if the index to be set is the size. If it is less than the size, the method should just replace the current value.

Displaying the array should print a comma-separated list of values, enclosed in square brackets, immediately followed by the number of remaining empty slots, also enclosed in square brackets. For example, if the elements 5, 3, 4, 8, and 9 are inserted in the order given and then followed by one removal, the array would display as:

```
[5,3,4,8][4]
```

with no preceding white space and no white space or newline following the last set of brackets.

Your dynamic array module (*darray.c* and *darray.h*) will be replaced for testing. Your module will be tested individually, as well.

The binomial heap module

You must implement your binomial heap as a separate module named *binomial.c* and *binomial.h*. You must follow the pseudocode at `http://beastie.cs.ua.edu/cs201/binomial2.html`. Here is a conforming header file:

```
#include <stdio.h>
#include "darray.h"

#ifndef __BINOMIAL_INCLUDED__
#define __BINOMIAL_INCLUDED__

typedef struct BinomialNode BinomialNode;

BinomialNode *newBinomialNode(void (*display)(FILE *,void *),void *value);
void displayBinomialNode(FILE *fp,BinomialNode *n);

typedef struct Binomial Binomial

extern Binomial *newBinomial(
    void (*d)(FILE *,void *),        //display
    int (*c)(void *,void *),         //comparator
    void (*u)(void *,BinomialNode *) //updater
    );
extern BinomialNode *insertBinomial(Binomial *b,void *value);
extern int sizeBinomial(Binomial *b);
extern void deleteBinomial(Binomial *b,BinomialNode *n);
extern void decreaseKeyBinomial(Binomial *b,BinomialNode *n,void *value);
extern void *extractBinomial(Binomial *b);
extern void displayBinomial(FILE *fp,Binomial *b);
#endif
```

A *BinomialNode* will have members similar to:

- `void *value`
- `DArray *children`
- `struct BinomialNode *parent`
- `void (*display)(FILE *,void *)`

A *Binomial* heap will have members similar to:

- `DArray *rootlist`
- `int (*compare)(void *,void *)`
- `void (*update)(void *,BinomialNode *)`
- `BinomialNode *extreme`
- `int size`
- `void (*display)(FILE *,void *)`

The constructor *newBinomial* takes in a display function, which allows heap to display itself, a comparator function, which allows it to maintain heap ordering, and an updater function, which allows it to inform a value that the node in the heap holding that value has changed. Note: a null pointer may be passed in as the updater, if this feature is not needed by the application. In this case, the updater function is never called.

If one were to insert the numbers 4, 8, 16, 5, and 1 in the order given and then displayed the heap, the output would look like:

```
0: 1-0
----
0: 4-2
1: 8-0(4-2) 5-1(4-2)
2: 16-0(5-1)
----
```

where the *-N* indicates the degree of the node. You should use a level-order traversal to display a subtree. The children of a node should be displayed in increasing degree order.

Your binomial heap module (*binomial.c* and *binomial.h*) will be replaced for testing. Your module will be tested individually, as well.

Other details

You must implement your program in C99. Only the most foolish student would not recompile and thoroughly test the implementation on a Linux system. You must use the *test3* dropbox to make sure the individual testing of your modules goes as desired.

Makefile

You must provide a makefile which responds properly to the commands *make*, *make test*, and *make clean*. The *make* command builds the dijkstra executable. Compilation must proceed with no errors or warnings and it must compile with the highest level of error checking (the *-Wall* and *-Wextra* options). The *make test* command should run your program through some test files of your choosing. The *make clean* command should remove all intermediate files, including the executable *dijkstra*. A call to *make* should never result in unnecessary compilation.

Restrictions

You cannot use any fixed-sized data structures for storing the graph (unless you preprocess the input to determine its extent) or any other data whose size changes depending upon the input. You many not use built-in data structures. All data structure methods must run as efficiently as commonly expected.

Documentation

All code you hand in should be attributed to its author. Comment sparingly but well. Do explain the purpose of your program. Do not explain obvious code. If code is not obvious, consider rewriting the code rather than explaining what is going on through comments.

Handing in results

For preliminary testing, delete all intermediate files and executables (`make clean`). Then, send me all the files in your directory by running the command:

```
submit cs201 lusth test3
```

For your final submission, use the command:

```
submit cs201 lusth assign3
```

Again, your implementation may be developed on other hardware and operating systems, but it must also compile and run cleanly and correctly on a Linux system. You may submit as many times as you like, up to the deadline.