I've finished the binomial and dynamic array classes and trying to start writing the dijkstra part. I understand we need to modify some portion of dijkstra's algorithm to something like this:
(from one of damccoy's posts)

for each vertex in adj. list array
  u.key = infinity
  u.predecessor = null
root/source.key = 0
build priority queue with vertex pointers
while queue is not empty
  u = extract-min
  for each vertex v in the adj. list of u
    if(v is in queue and u.key + weight(u,v) < v.key)
      v.predecessor = u
      v.key = u.key + weight(u,v)

My best idea at the moment is to create an adjacency list of Vertex objects like this:

struct Vertex {
  int visited;
  int num;    //This is the vertex's number
  int key;    //This starts at the equivalent of infinity
  BinomialNode value;
  DArray neighbors;    //I might be able to use the adjacency list instead but this seems easier
}

Where each vertex's BinomialNode has a value equal to the vertex's key, i.e. infinity, and a null parent. Then find the root vertex and do the modified dijkstras above starting with building the priority queue.

My questions are:
1. Has anyone else done it like this/Will it work?
2. How should the adjacency list be done, a 2d dynamic array or reading the file first to determine the largest value and then creating a static array?
3. Where does the update function go? I would put it in binomial.c because it needs to be called from there, but then how is it passed into the Binomial constructor called in main() from dijkstra.c?
4. How do we display the forest? The only way I can think of is to repeatedly read through the adjacency matrix testing each vertex's parent pointer, which seems wasteful.
5. Unrelated but did anyone have to modify their compare function like compareInteger() so that it would check for null before trying to get a value?
1. Not sure about your whole idea, but I don't know of anyone who hasn't needed to create a Vertex object. Although the vertex itself is what you should be inserting into the binomial heap, not just its key/distance. The compare and update function you pass in will compare the distances of the vertices in the heap and update the BinomialNode that vertex now belongs to.

2. As soon as I read from the input file, I create a Vertex object and insert them into a single-dimension DArray. No need for adjacency list or matrix. (later they are all added to the heap, but when processing the input, if a node already exists you will need to search for it in the DArray)

edit: the only issue with this is that you will need some way to store the distance between one vertex and its neighbor. In my implementation, I have a Neighbor object, which contains a vertex and its distance from whatever vertex it is a neighbor to. Each Vertex has a DArray of Neighbors, and if a is in b's neighbor list, then b is in a's neighbor list.

3. The update function should be part of your Vertex helper functions in dijkstra.c. See this thread

4. In my implementation, I make a queue to put each Vertex in after it is visited by the Dijkstra algorithm. If at any time (after the very first source is extracted) the algorithm takes a Vertex with a null parent from the heap (or the heap is empty), I immediately pass the queue of already visited nodes to a helper function called displayDijkstra, and then empty that queue to start the next graph. (Oh yeah, I keep track of steps from the source to the Vertex object as well. This is just int steps in the Vertex struct)

The displayDijkstra function takes a DArray of nodes. This DArray will be one tree in the forest (one connected graph) since the binomial heap will only extract a node with a null parent if it is a disconnected portion. The function does the following:
1) sorts all nodes by distance by inserting them into a new heap and extracting the minimum until it is empty. During some point in this sorting, the maximum number of steps should be noted
2) creates an array of new queues for each level, from 0 to maxSteps, so that each sorted node can be put into its respective level queue with enqueue(levelQueueArray[someVertex->steps], someVertex)
3) iterates through each queue in the array to display nodes according to level

5. Yup, you kinda have to.

-
Quick question, just because I missed class last week due to circumstances:

We substitute linked lists with DArrays in the binomial code, right? I figured yes, but I'm always afraid there's something that I'm not in on.

wjtreutel wrote on Mon, 27 March 2017 17:58:

Quick question, just because I missed class last week due to circumstances:

We substitute linked lists with DArrays in the binomial code, right? I figured yes, but I'm always afraid there's something that I'm not in on.

That's what I did!

Number 5 is actually a big deal; can we assume that Lusth's comparator checks for NULL? I've been testing for NULL before I run the comparator function, which is annoying but I assumed necessary.

From the binomial pseudocode:
Quote: Note: the comparator function has to be implemented so that it treats a null value as more extreme than any other value.

Will his comparator also test for the case in which they are both NULL and return 0? My extract
function runs through the root list and looks for a value equal to the extreme value (in order to find the extreme node). Allowing the comparator to test NULLs breaks my code if it cannot also test for both being NULL. But I feel uncomfortable assuming Lusth's comparator takes that into consideration...

Subject: Re: Anyone else confused on the dijkstra part
Posted by davidmccoy on Tue, 04 Apr 2017 18:02:34 GMT
View Forum Message <> Reply to Message

If you are looking through the rootlist for y, you should just check that y points to the same node as the binomialnode pointer at that index in the list (if (y == getDArray(b->rootlist,i))).

But Lusth's code handles all stuff relevant to dijkstra, no worries.