University of Illinois at Urbana-Champaign
Department of Computer Science

Third Examination

CS 225 Data Structures and Software Principles
Sample Exam 2
75 minutes permitted

Print your name, netID, and lab section day/time neatly in the space provided below; print your name at the upper right corner of every page.

Name: SOLUTION
NetID:
Lab Section (Day/Time):

- This is a **closed book** and **closed notes** exam. In addition, you are not allowed to use any electronic aides of any kind.
- Do all 5 problems in this booklet. Read each question very carefully.
- You should have 7 sheets total (the cover sheet, plus numbered pages 1-12). The last sheet is scratch paper; you may detach it while taking the exam, but must turn it in with the exam when you leave.
- Unless otherwise stated in a problem, assume the best possible design of a particular implementation is being used.
- Unless the problem specifically says otherwise, (1) assume the code compiles, and thus any compiler error is an exam typo (though hopefully there are not any typos), and (2) assume you are NOT allowed to write any helper methods to help solve the problem, nor are you allowed to use additional arrays, lists, or other collection data structures unless we have said you can.

Problem	Points	Score	Grader
1	12		
2	30		
3	18		
4	15		
5	15		
Total	90		

1. **[Short Answer – 12 points (4 points each)].**

   (a) What two data structures are used to implement Kruskal's Algorithm?

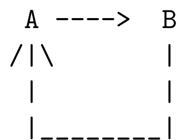
   Heaps and Disjoint Sets

   (b) Using big-$\mathcal{O}$ notation, indicate the running time of breadth-first search on a graph with `V` vertices and *no* edges.

   $\mathcal{O}(V)$.

   (c) Draw a directed graph that does not have a legal topological sort.

   There are many. Here is the simplest one:



   The presence of a cycle means that there is no topological sort. Please note that the *absence* of edges means there *is* a topological sort – in fact, in a graph with no edges, every possible ordering of the vertices is a topological sort, since no ordering places a target before its source.

2. **[Algorithms – 30 points (6 points each)].**

(a) Under what conditions will Dijkstra's Algorithm be more efficiently implemented using a heap than with a table, if your graph is implemented using an adjacency list? Justify your answer.

If your graph implemented with an adjacency list is sparse (i.e. has relatively few edges out of the number of edges it could have, for that number of vertices), then the heap implementation will be better. The heap implementation improves the time to search for a new vertex, but processing each neighbor of a given vertex is more expensive, since now, if the neighbor's distance is changed, you don't just change a table entry ($\mathcal{O}(1)$), but you also must (potentially) percolate the vertex upward in a heap ($\mathcal{O}(\lg V)$, to reflect its lowered priority (i.e. lowered distance, since distance equals priority). The more edges we have, the more neighbor updates we potentially have, and thus the more times we run the part of the heap implementation that is more expensive than the corresponding table implementation of the operation. With enough edges, this cost makes the overall cost more expensive than the table implementation, despite the faster search time. For sparse graphs, we don't have quite so many edges, so the improved search time outweighs the worsened update costs.

(b) Justify the correctness of the depth-first-search version of our topological sort algorithm. You don't need to justify the correctness of depth-first search – just explain why our use of it to perform topological sort *must* assign numbers to the vertices in such a way that the vertices are numbered in topological-sort order.

Our algorithm assigned integers from 1...V in reverse order – that is, V was assigned first, then V-1, and so on. In addition, our modification of depth-first search assigned a number to a vertex only *after* all the neighbors had been explored with depth-first-search (and thus numbered, since you number a vertex before returning from the depth-first-search call on that vertex). So, we know that for any vertex, its neighbors all get numbers before that vertex gets a number. And since we number in reverse order, that must mean the neighbors of a vertex always get higher numbers than the vertex itself, since the higher numbers get assigned first and the neighbors receive numbers first. And, the definition of topological sort is that every vertex appears before its neighbors in the ordering – so we are done, since now every vertex has a lower number than its neighbors.

(c) In Prim's Algorithm, we said you choose an edge at each step, from among all edges that go from set `S` to set `N`. What do these two sets represent, and why would it be a problem to pick an edge that goes between two vertices in `S`?

Set S is the set of all vertices that have been added to the tree; set N is the set of all vertices not yet added to the tree. If we chose an edge between two vertices in S, that means we are connecting two vertices that are both already in the tree – that is, we are connecting two vertices that are both already connected to each other. This means we are connecting them a second way – which means we have created a cycle.

(d) Under what conditions would running breadth-first search on an undirected graph, result in a breadth-first spanning forest of more than one tree, rather than a breadth-first spanning tree? That is, what kinds of undirected graphs would result in that kind of answer? Justify your answer.

If the undirected graph is NOT connected, then there will not be a path from the start vertex, to every other vertex in the graph, and thus we will not be able to reach every other vertex once we begin searching from our start vertex. This means once we finish our first tree in the breadth-first search, there will still be unmarked vertices, meaning we won't have a single breadth-first spanning tree, but rather, will have to start one or more additional trees to get those other vertices.

(e) For the given graph, run Dijkstra's algorithm, indicating in the table below the distances at each vertex at the end of each step (d_v), and whether or not the vertex has been marked known yet at the end of each step (k_v). The initialization has already been done for you.

```
    A   B   C   D   E   F   G
A   0   8   3   0   0   0   0

B   0   0   0   0   0   16  18

C   0   3   0   0   0   0   20

D   0   0   0   0   0   0   19

E   5   0   12  2   0   0   0

F   0   0   0   0   0   0   0

G   0   0   0   0   0   10  0
```

V	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v	d_v	k_v
A	∞	0	5	0	5	0	5	1	5	1	5	1	5	1	5	1
B	∞	0	∞	0	∞	0	13	0	11	0	11	1	11	1	11	1
C	∞	0	12	0	12	0	8	0	8	1	8	1	8	1	8	1
D	∞	0	2	0	2	1	2	1	2	1	2	1	2	1	2	1
E	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1
F	∞	0	∞	0	∞	0	∞	0	∞	0	27	0	27	0	27	1
G	∞	0	∞	0	21	0	21	0	21	0	21	0	21	1	21	1
-	Start		Step 1		Step 2		Step 3		Step 4		Step 5		Step 6		Step 7	

3. **[Analysis – 18 points (9 points each)].**

(a) What is the order of growth of the running time of finding the complement of a graph (the graph with the exact same vertices and the exact opposite set of edges), if you have a graph implemented with an adjaceny matrix? Express your answer in big-$\mathcal{O}$ notation and justify your answer. Assume your adjaceny matrix implementation *does* have a one-dimensional array of vertex information, and that your adjacency matrix itself does not have any "edge info" records, but rather, merely tells you if an edge exists or not.

The complement of a graph implemented by an adjacency matrix, would be obtained simply by "removing" every edge that does exist and then "adding" every edge that did not exist. That is, you flip every `1` in the matrix to `0` and flip every cell that used to be `0`, to `1`. Traversing the adjacency array to do this – or to copy the values into a new adjacency array and then do this operation to that array – will take $\mathcal{O}(V^2)$ time. Copying the one-dimensional array of vertex information will take $\mathcal{O}(V)$ time. Thus, the total is $\mathcal{O}(V^2)$.

(b) What is the order of growth of the running time of depth-first search on a graph with `V` vertices and `E` edges? Express your answer in big-$\mathcal{O}$ notation and justify your answer.

Each call to depth-first search:

- marks a vertex as "encountered"
- performs constant-time work on each neighbor of that vertex (checking the neighbor to see if it has been marked)
- performs constant-time work on all unmarked neighbors (starting up a depth-first-search stackframe for that vertex neighbor, and eventually returning from that call)

And, since you only call depth-first-search on unmarked neighbors, you make one call to depth-first-search for every vertex. So, there is constant-time marking work for each vertex, or $\mathcal{O}(V)$ total. There is constant-time work inspecting a neighbor and possibly starting (and later returning from) a depth-first-search call, for each neighbor, or, $\mathcal{O}(E)$ total (since the total number of neighbors equals the total number of targets of departing edges equals the total number of edges). Thus, the running time is $\mathcal{O}(V + E)$.

4. **[Breadth-First Spanning Tree – 15 points].**

   You are given the following class:

```
class TreeNode {
public:
   int vertexNumber;
   list<TreeNode*> subtrees; // initially an empty list; the function
                             //   push_back(item) will add item to
                             //   end of list
};
```

   In addition, you are given an adjacency matrix implementation of an unweighted, undirected, connected graph – such a graph, when you run breadth-first search on it, would produce a single spanning tree, rather than a spanning forest, no matter which vertex you start at. In this adjacency matrix implementation, the graph has vertices labelled with indices `0` through `n-1`, and you are given the value `n` and a two-dimensional array with `n` rows and `n` columns, both indexed from `0` through `n-1`. (In C++, a two-dimensional array is accessed via expressions such as `arr[i][j]` where `arr` is of type `int**`.) You want to write a method that takes those two values – the array and the `n` – as parameters, and returns a pointer to the root of the breadth-first-spanning-tree of the graph. You are allowed to use as many `Queues` as you like, as well as being allowed to create other one-dimensional arrays as well.

```
TreeNode* BreadthFirstSpanningTree(int** graph, int n) {
   // your code goes here
   SOLUTION ON NEXT PAGE
```

(Breadth-First Spanning Tree, continued)

```
TreeNode* BreadthFirstSpanningTree(int** graph, int n) {
   Queue<TreeNode*> nodes;
   int marks[n];
   for (int i = 0; i < n; i++)
      marks[i] = 0;
   TreeNode* root = new TreeNode();

   // we will take 0 as our start vertex
   root->vertexNumber = 0;
   nodes.Enqueue(root);
   marks[0] = 1;

   while (!Q.IsEmpty()) {
      TreeNode* temp = Q.Dequeue();
      int index = temp->vertexNumber;
      for (int col = 0; col < n; col++) {
         if graph[index][col] == 1) {
            if (marks[col] == 0) {
               TreeNode* latest = new TreeNode();
               latest->vertexNumber = col;
               nodes.Enqueue(latest);
               marks[col] = 1;
               (temp->subtrees).push_back(latest);
            }
         }
      }
   }
   return root;
}
```

5. **[Converting to undirected - 15 points].**

Suppose you have a directed weighted graph of n vertices, where the vertex numbers are 1 through n, and the graph implementation is an adjacency list. The adjacency list is represented with an Array, indexed from 1 to n, of pointers to the following type:

```
class EdgeNode {
public:
  int index;   //index of target vertex
  int weight;  //weight of edge
  EdgeNode* next;   // ptr to next edge
};
```

We want to convert this graph to an undirected weighted graph, by adding for each existent edge from u to v, the edge in the opposite direction, that is from v to u, with the same weight. That is, if the directed version had:

```
               5
 U ---------------------------> V
```

the undirected vertion will have:

```
               5
 U ---------------------------> V
               5
 V ---------------------------> U
```

You can assume that, initially, if there is an edge `<i, j>` in the graph, there is not an edge `<j, i>` of any weight. Write the function ConvertToUndirected, which receives as a parameter an Array of EdgeNode pointers – i.e. our adjacency list, as described above – by reference. The function will perform the conversion discussed above. Do not assume any edge list is specifically sorted in any way.

```
void ConvertToUndirected(Array<EdgeNode*>& graph) {
   // your code goes here
   for (int i = 1; i <= graph.Size(); i++) {
      EdgeNode* ptr = graph[i];
      while (ptr != NULL) {
         EdgeNode* travSearch = graph[ptr->index];
         while ((travSearch != NULL) && (travSearch->index != i))
            travSearch = travSearch->next;
         if (travSearch == NULL) {
            EdgeNode* temp = new EdgeNode();
            temp->index = i;
            temp->weight = ptr->weight;
            temp->next = graph[ptr->index];
            graph[ptr->index] = temp;
         }
         ptr = ptr->next;
      }
   }
```

(Converting to undirected, continued)

.

(scratch paper)