University of Illinois at Urbana-Champaign
Department of Computer Science

Third Examination

CS 225 Data Structures and Software Principles
Sample Exam 1
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: SOLUTIONS
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.
- You should have 6 sheets total (the cover sheet, plus numbered pages 1-10). 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	20		
2	20		
3	20		
4	30		
Total	90		

1. **[Ghost of Euler – 20 points].**

   You have the following `EdgeNode` class:

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

   Furthermore, you have a variable of type `Array<EdgeNode*>` that is indexed from `1` to the size of the array (given by the `Size()` method in the `Array` class). The array is an adjacency list implementation of a graph, of the kind we first discussed in lecture; the vertices have indices from `1` to `Size()`.

   You want to write a method `numOddDegree` which has one parameter, a reference to an `Array<EdgeNode*>` as described above. You can assume this adjacency list represents an undirected graph. You want to return the number of vertices with odd degree. (The degree of a vertex in an undirected graph, is the number of edges which that vertex is an endpoint for. So, you want to know how many vertices are endpoint to an odd number of edges.) You can assume the graph has no self-loops (i.e. no vertex has an edge to itself.)

```
int numOddDegree(Array<EdgeNode*>& graph) {
   // your code goes here
   int numOdd = 0;
   int currentVert = 1;
   while (currentVert <= graph.Size()) {
      EdgeNode* trav = graph[currentVert];
      int numEdges = 0;
      while (trav != NULL) {
         trav = trav->next;
         numEdges++;
      }
      if (numEdges % 2 == 1)
         numOdd++;
      currentVert++;
   }
   return numOdd;
}
```

(Ghost of Euler, continued)

2. **[Vertices Nearby – 20 points].**

You are given an adjacency matrix implementation of an unweighted, directed graph – 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`. You want to write a method that takes those two values (the integer `n` and an `int**` to the two-dimensional array) and returns a `1` if the distance from any vertex to any other vertex is always `2` or less, and returns a `0` if there is at least one vertex for which the minimum distance to some other vertex is `3` or more. You are allowed to create additional one-dimensional arrays if you need to.

```
int DistanceCheck(int** graph, int n) {
   // your code goes here
   int thereIsNotABadVertexYet = 1;
   int currVertex = 0;
   int* marks = new int[n];
   while ((thereIsNotABadVertexYet) && (currVert < n)) {
      for (int i = 0; i < n; i++)
         marks[i] = 0;
      marks[currVert] = 1;
      for (int i = 0; i < n; i++) {
         if (graph[currVert][i] == 1) {
            marks[i] = 1;                // mark everything one away from currVert
            for (int j = 0; j < n; j++)
               if (graph[i][j] == 1)
                  marks[j] = 1;  // mark everything one away from what is
                                 //     one away from currVert
         }
      }
      int i = 0;
      while ((thereIsNotABadVertexYet) && (i < n))
         if (marks[i] == 0)
            thereIsNotABadVertexYet = 0;
         else
            i++;
      }
      if (thereIsNotABadVertexYet == 1)
         currVert++;
   }
   delete[] marks;
   return (currVert == n);
}
```

(Vertices Nearby, continued)

3. **[Analysis – 20 points].**

(a) You are given a graph which is implemented by an adjacency matrix. This matrix has `V` rows and `V` columns, and represents a graph with `V` vertices and `E` edges. What is the order of growth of the worst-case running time for adding a vertex to the graph? Express your answer in Big-$\mathcal{O}$ notation and explain convincingly why your answer is correct.

You would need to create a new 2-D array of `V+1` rows and `V+1` columns, copy the V^2 values from the old array into the new array, and then initialize the new column and row to `0`. Copying the V^2 values will be constant-time each, since we are reading and writing to arrays; that means the copying costs are $\mathcal{O}(V^2)$ total. writing `0` into every cell in a row of length `V+1` is $\mathcal{O}(V)$ time, as is writing `0` into every cell of a column of length `V+1`. Therefore the total time is $\mathcal{O}(V^2 + V + V)$ which is $\mathcal{O}(V^2)$.

(b) You are given a graph which is implemented via an adjacency list. This adjacency list is of the kind we first discussed in class – i.e. an array of vertices, with each cell of the array pointing to a linked list of edge nodes. You want to find the vertex with the greatest number of departing edges. What will be the worst-case running time of this algorithm, in terms of `V` and `E`? Express your answer in big-$\mathcal{O}$ notation and explain convincingly why your answer is correct.

The linked list of edge nodes that each cell of the array points to, *is* the list of departing edges for the vertex that cell represents. So another way to ask this question is, "Which array cell has the longest linked list?" That can be done by simply calculating the length of each linked list by traversing down the lists one by one – and the sum of the lengths of the lists is the sum of all departing edges in the graph (or, if it is an undirected graph, the sum of the lengths of the list is double the number of edges in the graph). So there's $\mathcal{O}(E)$ work there. It's also possible some vertices have no departing edges and yet we'd need to inspect those vertices anyway, so you've got the $\mathcal{O}(V)$ traversal down the array to add in as well, since even though *usually* `V < E`, that won't always be the case. So, the total run time involves looking at each array cell and each list node, once – or $\mathcal{O}(V+E)$.

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

(a) Give an example of a undirected, weighted graph that has two edges of equal weight, for which there is still a unique minimum spanning tree.

There are many examples of such a graph. As a trivial example, any undirected, weighted graph that is really a tree, and has two equal-weight edges, has a unique minimum spanning tree – the entire graph itself is the only possible spanning tree, since the graph itself is a tree.

(b) In the depth-first-search-based implementation of topological sort, if we moved the "post-visit" code to instead be "pre-visit" code (i.e. if we took the code we run after we explore a vertex's neighbors, and instead run it *before* we explore a vertex's neighbors), the algorithm will no longer work. Explain why.

The algorithm relied on the idea that, since you were assigning numbers to vertices from highest number to lowest number, as long as you assigned a number to a vertex *after* you assigned to all its neighbors, you guaranteed that your vertex got a lower number than its neighbors. If you instead assign to the vertex *before* exploring all its neighbors, perhaps there is a neighbor that has not been reached yet, that will now be assigned a number *after* your vertex gets a number...and thus will get a lower number than its prerequisite vertex, which is not allowed.

(c) When we implement breadth-first-search, it is important that we mark a vertex as "encountered" *before* it is enqueued, rather than *after* it is *dequeued*. Explain why it is important – i.e. explain why we couldn't instead choose to mark a vertex "encountered" after dequeueing it.

When we dequeue a vertex, we will proceed to inspect its neighbors. If one of those neighbors is on the queue already, but not marked "encountered", we would add it to the queue a second time. And therefore, we would ultimately dequeue that neighbor twice and process it twice. Marking vertices as "encountered" once we have *first reached them* prevents us from adding a vertex to the queue multiple times, since every time we reach that vertex after the first time we reach it, it will already have the "encountered" flag set and we will therefore ignore that vertex.

(d) Explain cycle detection for Kruskal's algorithm. That is, explain why disjoint sets are used for this part of the algorithm (i.e. explain what the equivalence relation is that we are trying to model with our disjoint sets structure), and explain how the disjoint sets structure gets used to figure out whether to accept or reject an edge that is being considered.

The equivalence relation we are trying to model is "two vertices are in the same connected component of the MST we are in the midddle of building". Certainly every vertex is in the same connected component as itself, and if A and B are in the same connected component, then it means B and A are as well. Futhermore, by the very defintion of connection, if A and B are in the same connected component and B and C are, then A and C must be. So we have reflexivity, symmetry, and transitivity.

As a result, we can use the disjoint set structure to determine if two vertices are in the same connected component – if they are in the same set, they are in the same connected component, and if they are in different sets, then they are in different connected components. Every time we add an edge between two connected components, we are merging them into one connected component, and likewise we should union the disjoint sets that represent them. And, if an edge is going to be added between two vertices that are already conected (i.e. already in the same connected component), that would be a cycle, so saying "these two vertices are in the same connected component" (which is saying "these two vertices are in the same set") is the same as saying "these two vertices are already connected" which is the same as saying, "there would be a cycle created if these two vertices are connected again, by putting an edge between them".

(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).

	A	B	C	D	E	F	G
A	0	12	0	0	0	0	18
B	0	0	0	0	5	0	1
C	0	17	0	0	4	0	12
D	0	0	3	0	0	13	0
E	9	0	0	5	0	10	0
F	3	8	0	0	0	0	0
G	2	0	0	0	0	0	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	9	0	9	0	9	0	9	1	9	1	9	1	9	1
B	∞	0	∞	0	∞	0	25	0	21	0	18	0	18	1	18	1
C	∞	0	∞	0	8	0	8	1	8	1	8	1	8	1	8	1
D	∞	0	5	0	5	1	5	1	5	1	5	1	5	1	5	1
E	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1
F	∞	0	10	0	10	0	10	0	10	0	10	1	10	1	10	1
G	∞	0	∞	0	∞	0	20	0	20	0	20	0	19	0	19	1
-	Start		Step 1		Step 2		Step 3		Step 4		Step 5		Step 6		Step 7	

(scratch paper)