Name: Solution

UW NetID:

CSE 332 Summer 2018: Final Exam Part 2

(closed book, closed notes, no calculators)

Instructions: Read the directions for each question carefully. We can only give partial credit based on the work you write down, so show your work.

For questions where you are drawing pictures, please circle your final answer.

Unless otherwise noted, any algorithms or code you write should be as efficient as possible, both in $O()$ terms and with respect to constant factors.

Take a deep breath.
Every tree is a forest.
You got this.

Good Luck!

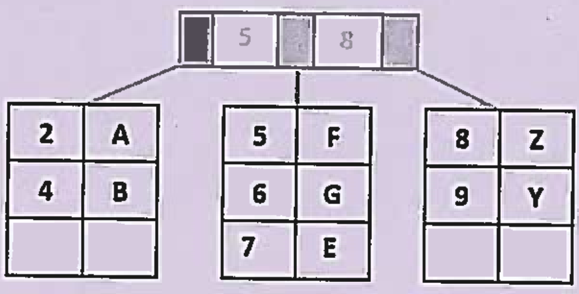
Total: 80 points. Time: 60 minutes.

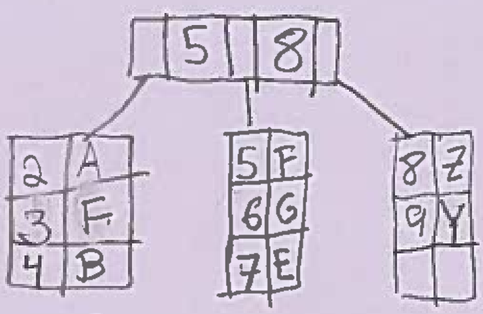
Question	**Max Points**	**Score**
7. B-Trees	6	
8. Hash Tables	13	
9. Minimum Spanning Trees	14	
10. Shortest Paths	7	
11. Graph Representations	14	
12. Using Graphs	12	
13. P vs. NP	14	
Day 2 Total	80	

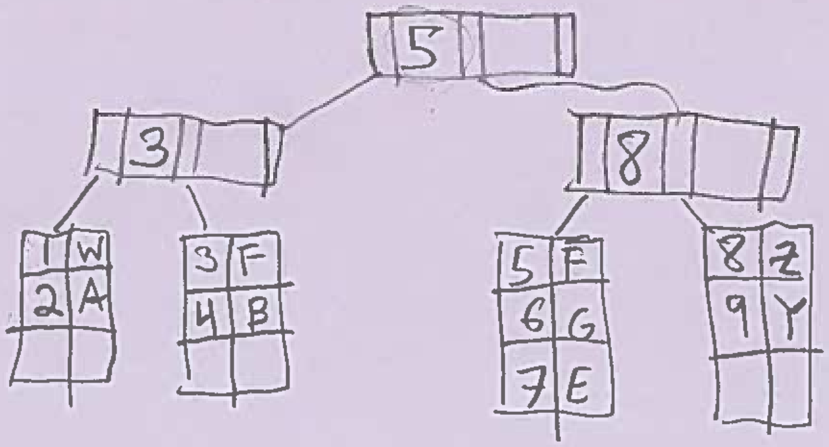
7 B-Trees

[6 points]

Below is a B-Tree storing integer keys with character values. Insert the (key, value) pairs (3,F) and (1,W) in that order into the B-Tree below. You should draw two B-trees: one for after (3,F) was inserted and one after both were inserted.







8 Hashing

[13 points]

Insert the following keys into the hash tables below, where the hash function is just `%TableSize`. In the top table, resolve collisions with quadratic probing. In the bottom table resolve by separate chaining into a sorted linked list (with the smallest element at the head of the list). If an insertion fails, record which key failed, but attempt to insert any later keys in the list. Do not resize the tables. [8 points]

Insert these keys: 22, 14, 32, 42, 37, 13

0	1	2	3	4	5	6	7	8	9
		22	32	14		42	37		13

0	1	2	3	4	5	6	7	8	9
		22	13	14			37		
		↓							
		32							
		↓							
		42							

What are the load factors of the top and bottom hash tables? [1 point]

$5/10$ for both

Under what conditions can we guarantee that insertion will succeed in a hash table using quadratic probing? [2 points]

When the table size is prime and the load factor is less than ½, we are guaranteed a successful insert.

What is the worst case running time to insert into each of these hash tables? Justify your answer (with 1-2 sentences) in each case. [2 points]

In both cases, the worst case is $O(n)$.

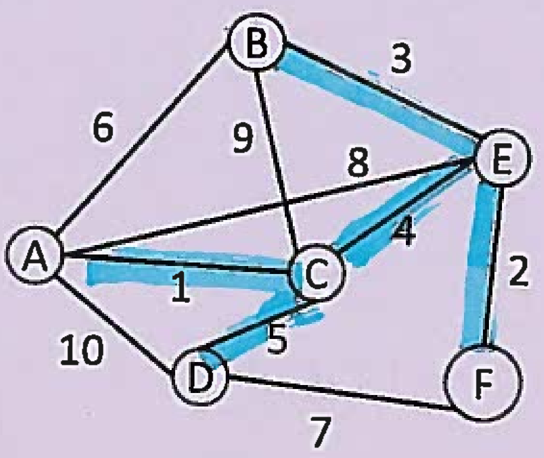
For separate chaining, if all elements are in the same chain, we require $O(n)$ to insert (since it's a linked list and we're keeping it sorted).

For quadratic probing, we may need to probe to $\Omega(n)$ locations before finding an empty one, so time is $\Theta(n)$ in the worst case.

9 MSTs

[14 points; 7 points per algorithm]

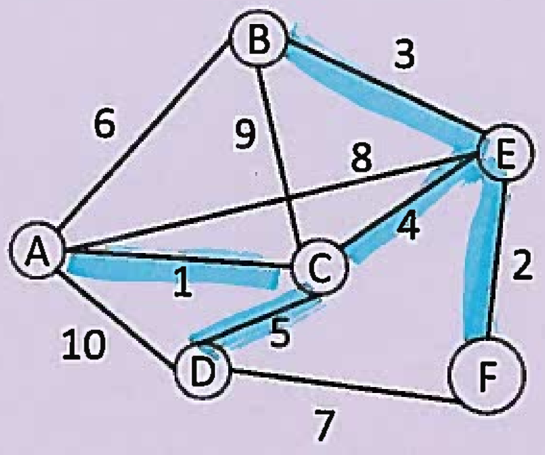
In class, we discussed two algorithms for finding a minimum spanning tree of a graph. Execute both algorithms for the graph below. We have provided space to show your work. You must show your work, crossing out old variable values when they are overwritten (as shown in lecture and section). Highlight the edges that end up in the final spanning tree.



Vertex	Distance	Best Edge	Processed
A	0	—	✓
B	~~6~~ 3	~~(A,B)~~ (B,E)	✓
C	1	(A,C)	✓
D	~~10~~ 5	~~(A,D)~~ (C,D)	✓
E	~~8~~ 4	~~(A,E)~~ (C,E)	✓
F	2	(E,F)	✓

Which algorithm did you execute?

Prim's Algorithm.



Edge	Include?	Reason
(A,C)	yes	
(E,F)	yes	
(B,E)	yes	
(C,E)	yes	
(C,D)	yes	
(A,B)	no	cycle A, B, E, C
(D,F)	no	cycle F, E, C, D
(A,E)	no	cycle A, C, E
(B,C)	no	cycle B, E, C
(A,D)	no	cycle A, C, D

Which algorithm did you execute?

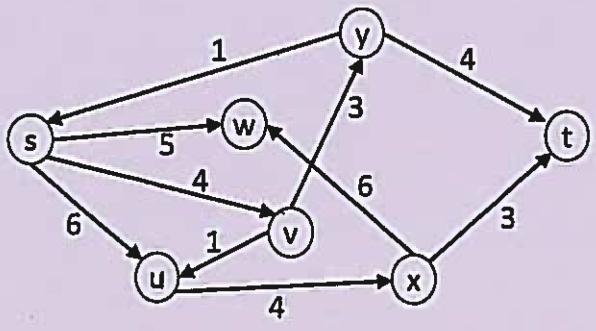
Kruskal's

10 Shortest Paths

[7 points]

Run Dijkstra's algorithm on the graph below with starting point s. Fill out the table. As you're performing the algorithm cross out previous values as shown in lecture and section.

If you ever need to break a tie between two vertices in your algorithm, choose the one which is first alphabetically.



Vertex	Distance	Predecessor	Processed
s	0	—	✓
t	~~∞~~ 11	y	✓
u	~~∞~~ ~~6~~ 5	~~s~~ v	✓
v	~~∞~~ 4	s	✓
w	~~∞~~ 5	s	✓
x	~~∞~~ 9	u	✓
y	~~∞~~ 7	v	✓

What is the shortest path from s to t? Describe how you found it using the results of the algorithm.

s, v, y, t

Starting from t, follow the predecessor's back to s.

11 Graph Representation

[14 points; 7 points each]

1. Hank Levy and the rest of the CSE leadership team need your help. A very important donor wants to tour CSE2 tomorrow, but there's a problem: they haven't started carpeting the hallways, and it would be embarrassing for the donor to step on uncarpeted floors. Hank's goal is to carpet enough hallways to allow him to give a tour (using only carpeted hallways), and to get the carpeting done as quickly as possible. Hank shows you a graph representation of the building. He has a vertex for each room the donor *simply must see* and edges representing the unfinished hallways connecting them. Each edge is also labeled with the time it would take to carpet that hallway. Hank has a (single) carpeting team ready to work until the donors arrive, but they need to know which hallways to carpet and in what order to allow for the tour to be fully-carpeted.

   Will you be able to tell Hank how to direct the carpeting team? If so, describe an efficient algorithm to run, and briefly explain why it produces the right answer. You may call any of the graph algorithms we've discussed in class. If not, informally argue why you shouldn't be expected to find the optimal carpeting scheme.

What does Hank need in order to show the donor all the rooms? He needs the set of carpetted hallways to connect every vertex of the graph to every other.
Subject to that, we should choose the minimum weight set of hallways (to give us the best chance of finishing before the donor arrives).
The minimum weight set of edges to connect all the vertices is the definition of a minimum spanning tree.

So our algorithm is to run Prim's (or Kruskal's) on Hank's graph, and tell the carpeting team to carpet the hallways in the MST. (in whatever order they want).

Despite the mention of tours, Hank doesn't say he cares how long it takes to do the tour, so this is not a travelling salesperson problem.

2. Puppy Dubs's handlers have decided he doesn't follow enough accounts on Twitter. They tell you they'd like to follow any account that:

   - Follows Puppy Dubs or
   - Follows someone who follows Puppy Dubs.

   You are given a graph representation of Twitter, stored as an adjacency list (see class definitions below). In the graph there is an edge from user1 to user2 if and only if user1 follows user2. Recall that "follows" is not symmetric, so the graph is directed.

   Give pseudocode for an efficient algorithm to find everyone Puppy Dubs should follow. You may call any of the graph algorithms we've discussed in class. You may assume all classes have appropriate constructors and iterators.

   HINT: the most efficient algorithm begins by altering the graph.

```
public class Graph{
    Vertex[] Vertices;
    //methods and constructors omitted.
}

public class Vertex{
     HashTable<V> outNeighbors;
     //any other fields required to run your algorithms.
}
```

```
//Step 1: Reverse the graph
Let G be the given graph.
HashTable<Vertex> Verts = new HashTable<Vertex>();
for(int i=0; i< G.Vertices.length; i++){
    Verts.insert( new Vertex(G.Vertices[i]));
        //copy constructor will create Vertex that .equals()
        //the original, but which has its own, empty outNeighbors.
}
for(Vertex u: G.Vertices){
    for(Vertex v: u.outNeighbors){
        (Verts.get(v)).outNeighbors.insert(u);
    }
}
Graph reversed = new Graph(Verts);
```

Continue your algorithm on the next page if you need more space.

Continue your algorithm description here, if necessary

//Now that we have a reversed graph,
// we just want all vertices at distance ≤2 from Puppy Dubs.
// The graph is unweighted, so we have an algorithm for that.

```
BFSShortestPaths(reversed, starting at Puppy Dubs)
List vertsToFollow = new List();
for (Vertex v: reversed.Vertices){
    if(v.dist ≤ 2 && !v.equals (Puppy Dubs))
        vertsToFollow.insert(v)
}

return vertsToFollow;
```

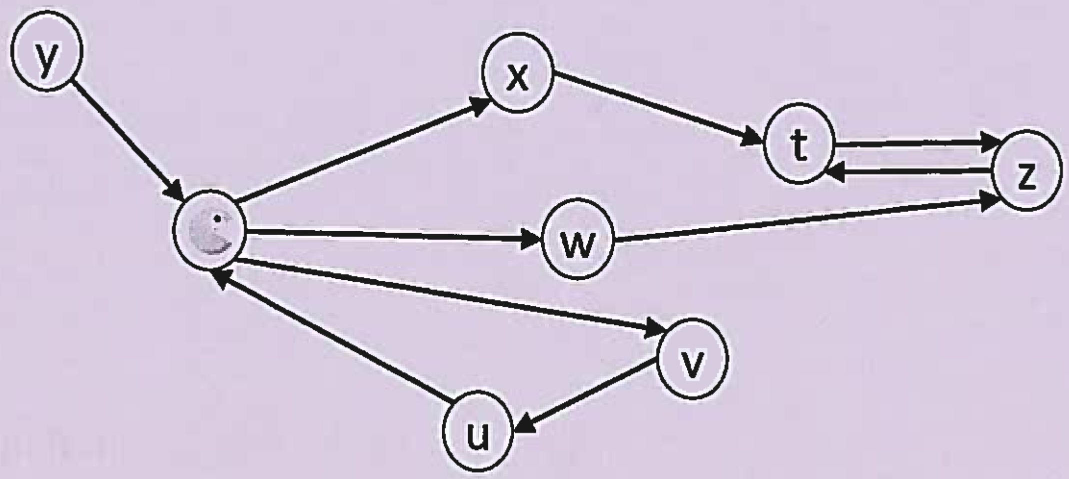
What is the running time of your algorithm (briefly justify your answer)?

We need to use an iterator over a hash table, and insert/get from a hash table. If we do average case, all these operations are $O(1)$. We need to do $O(|E|+|V|)$ of those operations. BFS is $O(|E|+|V|)$ as well so it's $O(|V|+|E|)$ on average.

In the worst case, depending on the exact implementation, we'd expect $O(|V|^2)$ from the first loop, $O(|V||E|)$ for the second loop. Those dominate the BFS time so we get $O(|V|^2+|V||E|)$.

12 Using Graphs
[12 points]

Pac-Man just arrived in a new world, represented as a directed graph. Each vertex represents a location with a single Pac-Dot (the food that Pac-Man eats), and there is an edge from vertex u to vertex v if Pac-Man can move from u to v. Pac-Man would like to eat all of the Pac-Dots, but he is worried that the graph may not be connected enough.



What are the strongly connected components of the graph above? The name of the vertex Pac-Man is sitting on is s. [3 points]

$\{y\}, \{s,v,u\}, \{x\}, \{w\}, \{t,z\}.$

On this particular graph (with Pac-Man at his current location) can he eat all the Pac-dots? Justify your answer [2 points]

No. Pac-Man definitely can't eat the Pac-Dot at y.

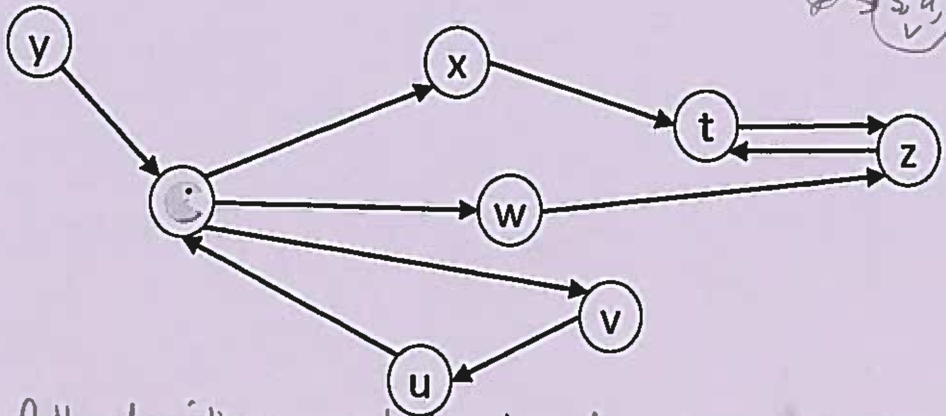
He also won't be able to eat the Pac-Dots at both x and w.

[7 points] There is a general strategy (i.e. an efficient algortihm) for Pac-Man to tell whether he can eat all the Pac-Dots. Describe such an algorithm. You may use any algorithms discussed in class as black boxes.

This algorithm is trickier than previous questions. We will award partial credit for answers which handle special cases. We recommend you think about the example graph:

$$\begin{tikzcd}
y \arrow[r] & s,u,v \arrow[r] \arrow[rr, bend left] & w \arrow[rr, bend right] & x \arrow[r] & t,z
\end{tikzcd}$$

Graph of SCC



The full algorithm must handle two issues, both shown in the above graph. The first issue is that Pac-Man must be able to reach every vertex. For example, here y is not reachable.
But even if Pac-Man can reach every vertex, he still might not be able to eat every Pac-Dot. Here, he can reach one of x, w. But once he enters one he cannot get back.

We can handle this issue as follows:
Find the SCCs of the graph, and build H the graph of SCCs.
Find a topological sort of H.
Now check that consecutive vertices of topo sort have an edge between them

```
for (int i=0; i< H.vertices.length; i++){
    if(i^th vertex of H in sort does not have edge to vertex i+1)
        return PacMan can't eat all Pac-Dots
}
return Pac-Man can eat all Pac-Dots
```

What is the running time of your algorithm (justify your answer)?

The full list of steps is:

Run DFS, find if a vertex is unvisited	$O(\|V\|+\|E\|)$	
Find SCCs	$O(\|V\|+\|E\|)$	
Topo Sort	$O(\|V\|+\|E\|)$	H has no more vertices & edges than G.
Adjacent check	$O(\|V\|+\|E\|)$	

Total $O(|V|+|E|)$.

13 P, NP
[14 points]

1. What do the letters NP stand for? [2 points]

2. For the following problems, circle all the classes to which the problem is **known** to belong [2 points each]

Determine if the minimum spanning tree of a graph has a total weight of at least k	[✓ P]	[✓ NP]	NP-complete	NP-hard
Given a graph, determine if there is a path of length at least k from s to t.	P	[✓ NP]	[✓ NP-complete]	[✓ NP-hard]
Given a graph, determine if there is a pair of vertices at distance at least k from each other	[✓ P]	[✓ NP]	NP-complete	NP-hard

Briefly justify why each of these statements are true or false. [2 points each]

3. True or false: If we discover an efficient (i.e. polynomial time) algorithm for any NP problem, then we will have an efficient algorithm for all problems in NP.

   False, we have efficient algorithms for lots of problems in NP - all the ones we know to be in P.

   Replacing the first NP with NP-complete or NP-hard fixes the statement

4. True or false: If P = NP, then all decision problems can be solved in polynomial time.

   False, only the ones in NP will immediately get efficient algorithms.

   There are decision problems not in NP, like the halting problem and $n \times n$ chess, which are known to not have efficient algorithms even if P = NP.

5. True or false: NP-complete problems are rare – only a few dozen are known.

   False, there are literally thousands known.

   Dozens are discovered every year.

Blank page for scratch work.

Some Useful Facts

When we're using the tree method to solve a recurrence, we usually use the following steps:

0. Draw a few levels of the tree.
1. Let the root node be at level 0. Give a formula for the size of the input at level i.
2. What is the number of nodes at level i?
3. What is the work done at the i^{th} recursive level?
4. What is the last level of the tree?
5. What is the work done at the base case?
6. Write an expression for the total work done.
7. Simplify until you have a "closed form" (i.e. no summations or recursion).

Geometric series identities:

$$\sum_{i=0}^{k} c^i = \frac{c^{k+1}-1}{c-1} \qquad \sum_{i=0}^{\infty} c^i = \frac{1}{1-c} \text{ if } |c| < 1$$

Common Summations:

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \qquad \sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \qquad \sum_{i=0}^{n} i^3 = \frac{n^2(n+1)^2}{4}$$

Log identities:

$$a^{\log_b(c)} = c^{\log_b(a)} \qquad \log_b(a) = \frac{\log_d(a)}{\log_d(b)}$$

Master Theorem:
Given a recurrence of the following form:

$$T(n) = \begin{cases} d & \text{if } n \leq \text{ some constant} \\ aT(n/b) + n^c & \text{otherwise} \end{cases}$$

with a, b, c are constants.
If $\log_b(a) < c$ then $T(n)$ is $\Theta(n^c)$
If $\log_b(a) = c$ then $T(n)$ is $\Theta(n^c \log n)$
If $\log_b(a) > c$ then $T(n)$ is $\Theta\left(n^{\log_b(a)}\right)$