Fall 2026 COP 3503H Exam #2 Solutions
Date: 11/5/2026

1) (15 pts) To dissuade students from flippantly signing up for the Bar Exam, the National Conference of Bar Examiners is instituting an interesting fee. For one's n^{th} attempt at passing the bar, the cost to take the exam is $(\$1000)n^2$. Sheila is determined to pass the Bar Exam and will continue taking it until she passes. Her probability of passing the exam on a single attempt is p. (Unfortunately, extra studying does not change the probability she passes the exam and each exam attempt is independent of the others.) **What is the expected amount of money Sheila will have to pay, in terms of p, to pass the bar?**

Let S be the value we desire to compute. Sheila's probability taking at least 1 exam is p. Her probability of taking at least 2 exams is 1-p. Her probability of taking at least 3 exams is $(1-p)^2$. More generally, her probability of taking at least k exams is $(1-p)^{k-1}$, since this represents failing the first k – 1 exams. Thus, with probability 1 she'll pay \$1000 (for the first exam, with probability (1 – p) she'll pay \$4000 for the second exam, ..., with probability $(1-p)^{k-1}$ she will pay $(\$1000)k^2$ for the kth exam. Using the definition of expectation, we can compute S as follows:

$S = \$1000 + (1-p) \times \$4000 + (1-p)^2 \times \$9000 + \ldots$

Multiply this equation through by (1 – p), then subtract the equation below from the one above

$(1-p)S = (1-p) \times \$1000 + (1-p)^2 \times \$4000 + \ldots$

$S - (1-p)S = \$1000 + (1-p)\$3000 + (1-p)^2 \times \$5000 + \ldots$

$pS = \$1000 + (1-p)\$3000 + (1-p)^2 \times \$5000 + \ldots$, repeat the idea

$(1-p)pS = (1-p)\$1000 + (1-p)^2 \times \$3000 + \ldots$

$pS(1-(1-p)) = \$1000 + (1-p)\$2000 + (1-p)^2 \times \$2000 + \ldots$

$p^2S = \$1000 + \$2000(1-p)(1 + (1-p) + (1-p)^2 + \ldots)$

The last sum in the parentheses is an infinite geometric sum, plug in the formula and we get

$p^2S = \$1000 + (1-p)\$2000\left(\frac{1}{1-(1-p)}\right)$

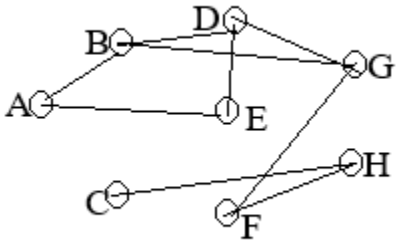
$p^2S = \$1000 + \$2000\left(\frac{1-p}{p}\right)$

Solving for S we get

$$S = \$1000\left(\frac{p}{p^3} + \frac{2(1-p)}{p^3}\right) = \$\mathbf{1000}\left(\frac{\mathbf{2-p}}{\boldsymbol{p^3}}\right)$$

Grading: 5 pts for just listing the correct original infinite sum (either formally or using ... notation), 3 pts for one application of multiplication and subtraction, 3 pts for the second application of it, 4 pts to finish it up.

2) (8 pts) Show the result of running both a Depth First Search and Breadth First Search on the graph drawn below, starting at vertex A. Whenever either recursing or adding items to a queue, do so in alphabetical order.



DFS: A , B , D , E , G , F , H , C
BFS: A , B , E , D , G , F , H , C

Grading: 1/2 pt per slot, round down to integer

3) (7 pts) Tommy must do the following five items today:

a) Deliver two packages to his mom.
b) Pick up a package from his dad.
c) Take an exam.
d) Study for an exam.
e) Pick up a package from his sister.

He has the following restrictions on the order in which he completes these tasks:
1. He must complete b before a.
2. He must complete e before a.
3. He must complete d before c.
In how many different orders can Tommy complete these five tasks within the set of restrictions? Please show work to justify your answer.

We have five slots to fill and need to count how many ways to do it satisfying the restrictions above. Note that the restriction on c and d is independent of the restrictions on a, b and e. We can choose 2 slots for c and d out of 5 in $\binom{5}{2} = 10$ ways. (Note: this was taught in COT 3100, a pre-requisite course for COP 3503.) Once we choose the slots, the order of c and d is fixed as d must go in the first of the two slots and c must go in the second of the two slots.

We have 3 remaining slots to place a, b and e. Notice that a must go in the last of these slots, but that b and c can be placed in either of 2 orders.

This means that the total number of valid orderings = $2 \times \binom{5}{2} = 2 \times 10 = \mathbf{20}$ ways.
Grading: 4 pts for 10, 2 pts for 2, 1 pt to multiply to get final answer. (Note different conceptual ways to get the 10...)

4) (5 pts) The Ford-Fulkerson algorithm uses DFS to find augmenting paths while the Edmunds-Karp Algorithm uses BFS to do so. How does Dinic's (or Dinitz's) algorithm work to find augmenting paths? Intuitively, why does this work faster in practice?

Dinic's runs a nested loop structure. The outer loop runs a series of BFS's, each guaranteed to produce augmenting paths of a different number of edges. For each single BFS the outer loop runs, the inner loop runs (potentially) multiple DFSs capped at the length of the number of edges of the BFS. A typical run of Dinic's might have one BFS with 3 edges, followed by several DFSs of augmenting paths that are each 3 edges long, followed by one BFS with 5 edges, followed by a few DFSs of augmenting paths that are each 5 edges long, followed by one BFS with 7 edges, folllowed by several DFS's of length 7. Then, the BFS fails to find any augmenting path and the algorithm ends. This works faster in practice because for standard layer graphs, a BFS explores nearly all of the edges before getting to the sink while a DFS usually explores a small fraction of the edges in the graph. Thus, each of theruns of DFS tend to save time over the corresponding run of BFS in Edmunds-Karp. Limiting the DFS length allows the algorithm to avoid the potentially bad behavior of Ford-Fulkerson in the worst case while gaining the advantage of the typically faster DFS time.
Grading: Grader's discretion!

5) (14 pts) Show the full trace of Dijkstra's Algorithm on the graph with the edges shown below with starting vertex A. When you fill out the table, in the leftmost column, indicate the vertex that gets added into the set of vertices for which correct shortest distances have been computed, followed by the shortest distance (after considering that vertex) to all of the other vertices. The vertices are labeled with the letters A through H, inclusive and the graph is given as 8 lists of directed edges, each list storing the outgoing edges from a particular vertex. Each edge is listed as (u, v, w), where u is the starting vertex, v is the ending vertex and w is the weight of the edge.

A: (A, B, 4), (A, C, 10), (A, D, 5)
B: (B, E, 12), (B, F, 7)
C: (C, E, 9), (C, F, 2)
D: (D, C, 3), (D, F, 13), (D, G, 2)
E: (E, H, 3), (E, F, 4)
F: (F, H, 10)
G: (G, H, 19)
H: None

Grading: 2 pts per row, 2 pts if 100% correct, 1 pt if a row is > 50% correct.

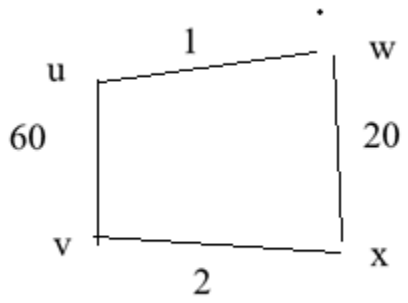
Vertex	A	B	C	D	E	F	G	H
Add to Set	0	∞	∞	∞	∞	∞	∞	∞
A	0	4	10	5	∞	∞	∞	∞
B	0	4	10	5	16	11	∞	∞
D	0	4	8	5	16	11	7	∞
G	0	4	8	5	16	11	7	26
C	0	4	8	5	16	10	7	26
F	0	4	8	5	16	10	7	20
E	0	4	8	5	16	10	7	19

6) (10 pts) Using the crucial fact about minimum spanning trees taught in class, Elise has come up with an alternate algorithm to solve the minimum spanning tree problem:

1. Split the input graph vertices into two sets A and B of (nearly) equal size, if the graph has 3 or more vertices. (For 1 vertex, the MST is empty, for 2 vertices, it's the edge between the two vertices.)
2. Recursively find the MST of the vertices in set A (using only the edges between vertices in set A and do the same for set B.
3. Consider all edges between set A and set B. Of these, add the minimum edges to the two recursively formed MSTs and return this as the MST for the whole graph.

This algorithm doesn't work. Generate a small counter-example showing a graph for which it fails. Explain why the failure of this algorithm doesn't contradict the crucial fact about minimum spanning trees taught in class.

Consider the example below with vertices u, v, w and x with set A = {u,v} and set B = {w,x}:



The MST of {u,v} is 60 and the MST of {w,x} is 20. The algorithm described would add the edge uw to these two MSTs to get an MST of weight 60 + 20 + 1 = 81 for the full graph. Unfortunately, the real MST is the set of edges {uw, wx, vx} which has total sum of weights 1 + 20 + 2 = 23.

The reason this counter-example doesn't contradict the crucial fact about MSTs is that fact just said it there exists some MST with the minimum edge in between set A and set B of any partition of the vertices. It never said anything about the non-minimum edges across this cut. As this example shows, on some occasions, non-minimum edge across cuts can ALSO be part of MSTs.

Grading: 6 points for counter-example and clear explanation as to why it's a counter-example, 4 points for explanation of why the crucial fact isn't contradicted by the counter-example.

7) (15 pts) Consider a 50 x 50 grid with red pieces and black pieces, representing two teams. Each square can be empty, have one red piece on it, or have one black piece on it. On a single move, a piece can move up, down, left or right by one square. If a red piece moves to a square with a black piece on it, the red piece captures the black piece. The red square is not allowed to move to a square with another red piece on it. A **parallel move** is when any subset of the red pieces all move at the same time. During this move, each red piece may move to either a square that is currently open or a square that contains a black piece. (Thus, if two red pieces are adjacent to each other, neither is allowed to move to the other piece's square.) Describe an algorithm to determine the maximum number of black pieces that the red pieces can capture on a single parallel move.

Notice that the graph induced by the movement of the pieces is bipartite. Namely, when moving we always change the parity of our sum of row and column. For example, if I am at (2, 3), I can move to (2, 2), (2, 4), (1, 3) or (3, 3). The sum 2 +3 = 5 is odd while the four sums 2+2, 2+4, 1+3, and 3+3 are all even. It should be clear why this is the case. Every move either adds or subtracts 1 from the total sum of row and column, guaranteeing a change in parity. Thus, we can think of the cells in the grid partitioned into two sets: squares with row+col is even parity and squares with row+col is odd parity.

We can think of an edge being drawn between squares if the two squares are adjacent and have pieces of opposing colors. Furthermore, a flow of 1 through this edge essentially represents a capture of a black piece by a red piece. Once we see this connection, we can come up with a network flow graph to solve the problem:

The graph will have 2502 vertices, 1 for each square in the, as well as a source and sink.

1. Connect the source to each vertex representing a square of even parity with capacity 1.
2. Connect each vertex representing a square of odd parity to the sink with capacity 1.
3. Connect each edge between two squares of opposite parity if and only if those two squares each have a piece and the pieces are of opposite colors. (So in our graph, it might look like some red pieces are capturing black pieces and some black pieces are capturing red pieces, but we can just interpret each of the black capture red moves in the other direction.)

The maximum flow through this graph will equal the maximum number of captures the red pieces can have on a single parallel move. Each unit of flow that goes through the graph represents a specific capture. When no more flow can go through, then no more captures can happen. The flow graph captures (sorry pun intended) the restrictions inherent in the problem: each red piece can only capture one black piece, and each piece can only move up, down, left or right.

Grading: 3 points for recognizing this is Network Flow, automatic 0 points otherwise
5 points for explaining the bipartite nature of the induced graph
6 points for the detained description of building the graph (2 pts for each type of edges)
1 point for the justification of correctness

8) (15 pts) Given a 2^n by 2^n grid of cells, with the top left corner having the position (s_r, s_c), the top right corner having position (s_r, $s_c + 2^n - 1$) and the bottom right corner having the position ($s_r + 2^n - 1$, $s_c + 2^n - 1$), we can describe a "path" to any cell on the grid as follows:

For n > 1, break the grid up into four quadrants, each of size 2^{n-1} x 2^{n-1}. If the cell is in the top left quadrant, then the path starts with the character '0'. If the cell is in the top right quadrant, the path starts with the character '1'. If the cell is in the bottom left quadrant, the path starts with the character '2'. Finally, if the cell is in the botttom right quadrant, the path starts with '3'. For example, here is an 8 by 8 grid with the path for each cell written into it.

000	001	010	011	100	101	110	111
002	003	012	013	102	103	112	113
020	021	030	031	120	121	130	131
022	023	032	033	122	123	132	133
200	201	210	211	300	301	310	311
202	203	212	213	302	303	312	313
220	221	230	231	320	321	330	331
222	223	232	233	322	323	332	333

Write a method (using Divide and Conquer, so not iteratively), that takes in the value of n (guaranteed to be positive), the values of s_r and s_c, as well as a cell location on row r, column c, and returns a String storing the path to that cell location. (Please use base case n = 1.)

```
public static String path(int n, int sR, int sC, int r, int c) {

    if (n == 1)
        return new String(""+(2*(r-sR)+(c-sC)));

    int div = 1<<(n-1);
    int thisMove = 2*((r-sR)/div) + (c-sC)/div;
    int offR = div*((r-sR)/div);
    int offC = div*((c-sC)/div);

    String rest = path(n-1, sR + offR , sC + offC, r, c);
    return thisMove + rest;
}
```

Grading: Base case – 4 pts

Separating out which quadrant we are in and/or arithmetic to handle all cases - 5 pts

Recursive call - 4 pts

Final return - 2 pts

9) (10 pts) Let the integer array, path, be such that path[i][j] stores the last vertex visited in a graph on the shortest path from vertex i to vertex j. Complete the static method below so that it takes in the path array (note: the length of this array is the number of vertices in the graph it corresponds to), a start vertex number s, and an end vertex number, e, and returns an ArrayList<Integer> storing the shortest path from vertex number s to vertex number e. You may assume that s and e are distinct and that the list your code returns must start with s and end with e and that such a path exists.

```
public static ArrayList<Integer> getPath(int[][] path, int s, int e) {

    ArrayList<Integer> res = new ArrayList<Integer>();      // 1 pt
    res.add(e);                                             // 1 pt
    while (path[s][e] != s) {                               // 1 pt
        res.add(path[s][e]);                                // 2 pts
        e = path[s][e];                                     // 2 pts
    }
    res.add(s);                                             // 1 pt
    Collections.reverse(res);                               // 1 pt
    return res;                                             // 1 pt
}
```

10) (1 pt) One of the examples of a BFS application shown in class was the 8-Puzzle. In this game, how many tiles are arranged on the game board?

Eight (give to all)