PS7

A – Cops and Robbers

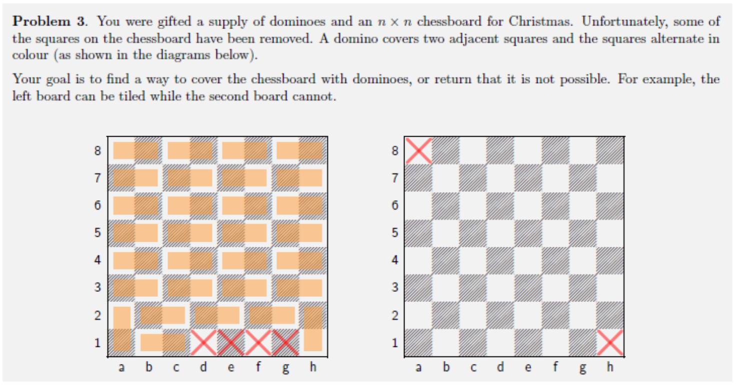
- This question is solved using the max-flow min-cut theorem, where the I need to somehow construct a flow network where each unit of flow represents the cost to barricade the cell, and the max flow is the min capacity, which is the minimum price to barricade all roads that prevents the thief from escaping
- My initial construction of graph is as follow:
    - Source: cell B
    - Sink: arbitrary node t
    - Vertices: each cell is a node
    - Edges:
        - There is an edge from every cell to adjacent cells (i.e. if u and v are adjacent cells, then there is an edge from u to v and another edge from v to u)
            - If the cell is on the boarder, then there exists an edge from the cell to t with infinite capacity
            - If cell v is next to B, the edge from v to B has infinite capacity
            - If there is an edge from u to v where v is a dot (cannot be barricaded), then the edge has infinite capacity
            - Otherwise, the capacity of the edge is the cost to barricade the destination cell (i.e. if the edge is u->v, then the cost of the edge)
- However, the setup above fails on test case 4, which got me stuck for a while. Under the hint given by the tutor, I found that the following case would break my code:
    - 5 5 1
    - . a a a .
    - a . . B a
    - a . a . a
    - a . a . a
    - . a . a a
    - 1
- The correct answer would be 12, but my code was returning 14. The reason for this is if cell a has capacity 1 which is on the boarder, and cell b and c are both next to cell a, then cell b and c are both able to send 1 unit of flow to a, which is not supposed to happen.
- I then took another approach using vertex capacity, that is decomposing each cell into 2 nodes, v_in and v_out, which gives me the following setup:
    - Source: B
    - Sink: arbitrary sink t
    - Vertices: each cell is decomposed into 2 nodes, in node and out node
    - Edges:
        - There exist an edge for every cell v between v_in and v_out with capacity being the cost of the cell
        - If cell u is next to cell v, then there exists an edge from u_out to v_in and another edge v_out to u_in, both have infinite capacity
        - If cell v is on the boarder, then there exists an edge from v_out to t with infinite capacity
        - If the cell v is next to B, then the edge from v_out to B also has infinite capacity.

- After setting up the graph above, run dinic would give the solution.

B – Magic Potion

- This question reminds me of 23T3 COMP3121 Assigment2 maxflow question, where each person can walk from one room to another using stair, which would collapse after one use, or if the room has a portal, that person can teleport to any other room with a portal at a cost of 1 coin, and the person starts with n coins.
    - The approach to this problem is to have an extra node to represent the portal, where the capacity from the source to the portal is the number of coins the person have, and the portal has an edge to all other rooms with portal
- This question is quite similar, which is a bipartite matching question with an additional node being the magic potion. The network is constructed as follow:
    - Source: Arbitrary source s
    - Sink: Arbitrary sink t
    - Vertices:
        - Each warrior is a node
        - Each monster is a node
        - Potion itself is a node
    - Edges:
        - An edge from source s to each warrior u of capacity 1 representing each warrior can only kill 1 monster
        - An edge from warrior u to monster v of capacity 1 exists if monster belongs to M_u, the set of monsters that warrior u can kill
        - An edge from each monster to sink of capacity 1 representing each monster can only be killed once
        - An edge from source to potion of capacity equals to the numbers it can be used
        - An edge from potion to each warrior of capacity 1
- Then running dinic to the network above would yield the solution

C – Oil Skimming

- The grid is a NxN board, and each oil covers 2 cells. This description reminds me of Problem3 of COMP3121 tutorial 5, the chess-board covering problem
    - 

      Problem 3. You were gifted a supply of dominoes and an $n \times n$ chessboard for Christmas. Unfortunately, some of the squares on the chessboard have been removed. A domino covers two adjacent squares and the squares alternate in colour (as shown in the diagrams below).

      Your goal is to find a way to cover the chessboard with dominoes, or return that it is not possible. For example, the left board can be tiled while the second board cannot.

    - Similarly, in this question, we can also define each cell on the grid as "black", then the adjacent cells of a "black" cells are all "white" cells, and vice versa. Each oil block has to cover a "black" cell and a "white" cell. Now the question becomes a bipartite question. We construct the following flow network:
        - Source and Sink: Arbitrary nodes s and t

        - Nodes: each cell is a node, either black or white. All adjacent cells of a black node are white nodes, all adjacent cells of a white node are black nodes.
        - Edges:
            - Source to each white node of capacity 1
            - Each black node to sink of capacity 1
            - Each white node to its adjacent black nodes of capacity 1
    - Then we run dinic, the result is the solution

D – Delivery Bears

- Intuitively, this looks like a binary search question, so I started thinking about what is the monotonic behaviour
- I then realized that letting multiple bears walking on one path each representing a flow of 0.smth is equivalent to letting multiple bears walking on one path each representing 1 unit of flow with the capacity of the path enlarged by a certain factor. With this logic, if a larger multiplier would let all the bears through, then all the smaller multipliers would also do, vice versa.
- Hence my initial approach is to keep a counter for the multiplier for path capacity starting from 1, then use binary to search the range [1, 1000000]. Until I find the multiplier, then 1/multiplier is the weight that each bear should carry
- However, I then realized the weight that each bear can carry is not necessarily 1/something, it could be 2/something or any integer/any integer, where the possible solution could be any decimal number. Hence I modified the binary search to make lo and hi both double, and removed the "+1" when searching in the upper half of the array.
    - The difficult part is for terminating condition. I initially started by setting the terminating condition to be when hi-lo < 10^-6, but that did not have enough precision for test case:
        - 2 1 100000
        - 1 2 1
            - Expected: 1
            - Output: 0.9095037967
    - Hence I increase the precision all the way up to 10^-10 for enough precision. However, now the code exceeds the time limit.
    - I then realized using binary search is essentially reducing the search range by power of 2, hence I can set the number of loops to be a large enough constant to ensure my precision is within a certain digit. I experimented with a few numbers, then decided to loop 100 times, which gave me the correct solution.