CS 577: Introduction to Algorithms — Sample Midterm Solutions

1. To show that the solitaire game is NP-hard, it suffices to reduce a known NP-hard problem to it. We will reduce from 3-SAT. Given a 3-CNF formula ϕ, we will construct an instance M of the solitaire game so that M has a solution if and only if ϕ has a satisfying assignment.

   Let $C_1, C_2, \ldots, C_k$ be the clauses in ϕ, and let $x_1, x_2, \ldots, x_n$ be the variables in ϕ. We say a literal is *positive* if it is of the form x_i for some i, and we say a literal is *negative* if it is of the form $\overline{x_i}$ for some i. We assume without loss of generality that, for every clause C_j, no variable appears as both positively and negatively in C_j.

   We construct M to be the game with one row for each clause, and one column for each variable. For each clause C_j and each variable x_i, we place a blue/circular stone in the corresponding cell of M if x_i appears positively in C_j, and we place a red/cross stone in the corresponding cell of M if x_i appears negatively in C_j. (By our assumption that no clause of ϕ has both the positive and negative form of a literal, at most one stone is placed in any particular cell.) This completes the reduction.

   We now argue that M has a solution if and only if ϕ has a satisfying assignment.

   In one direction, suppose that $x_i \leftarrow a_i : i = 1, \ldots, n$ is a satisfying assignment to ϕ. We give the following rule for removing stones from M: for each variable x_i, if $a_i = 1$, remove the red/cross stones from x_i's column; if $a_i = 0$, remove the blue/circular stones from x_i's column. Clearly, this leaves every column with at most one type of stone in it. To see that every row has at least one stone left in it, consider the row for the clause C_j. Since $x \leftarrow a$ is a satisfying assignment for ϕ, some literal of C_j is set to 1. Accordingly, there is a variable x_i so that the stone placed in the C_j row and x_i column is left in by the above rule. Thus every row of M has a stone left in it, and hence M is solvable.

   In the other direction, suppose that M is solvable. We take this to mean that for each column, there is a type of stone (blue/circular or red/cross) so that every stone remaining in that column has that type. (This allows for the possibility that some column has no stones, in which case a stone-type is chosen arbitrarily.) Moreover, for each row, there is a stone remaining in that row. We now use this solution of M to construct a satisfying assignment $x \leftarrow a$ to ϕ. For each variable x_i, if the solution of M leaves blue/circular stones, set $x_i \leftarrow 1$ (ie $a_i = 1$), and if the solution of M leaves red/cross stones, set $x_i \leftarrow 0$ ($a_i = 0$). For any clause C_j, we know that in the solution to M, there is a stone in some column in C_j's row. It follows that the corresponding literal is set to true in the assignment $x \leftarrow a$. Thus every clause of ϕ is satisfied, and so ϕ is satisfied. This completes the proof.

2. Our solution will reformulate the escape problem as a network flow problem. We can think of vertex-disjoint paths through the grid as distinct paths carrying flow through a network. Since we want to determine whether there are m disjoint paths from the terminals to the boundaries, we will try to push m units of flow through the terminals to the boundaries, bounding the flow along each path to 1 unit.

   Some caution is required, though. If we just turn the grid into a network in the obvious way (by connecting a source to the terminals, the boundaries to a sink, and setting directed edge capacities of 1 throughout), it is possible that we could achieve a flow value of m even if there aren't m vertex-disjoint paths. This is because the flow paths need not be vertex-disjoint: two units of flow could enter the same vertex and then continue along different out-edges. We need to modify the "obvious" network to make sure that each vertex has at most one unit of flow sent through it.

   One way to accomplish this is by adding a single "internal" edge of capacity 1 inside each vertex. This way, we can't send more than one unit of flow through any vertex because the internal edge will be saturated. While internal edges aren't one of the basic building blocks of graphs or networks, we can achieve the same effect by splitting each vertex v into two vertices v_{in} and v_{out} and including a directed edge (v_{in}, v_{out}) of capacity 1. The intuition here is that v_{in} is where flow enters v, while v_{out} is where it exits, and (v_{in}, v_{out}) plays the part of the desired internal edge. With this construct in mind, we are now ready to present our algorithm.

   Escape(G):

1. Construct a flow network G' as follows:
   (i) Create a source s and a sink t.
   (ii) For each vertex v in G, create a pair of vertices v_{in} and v_{out}, with a directed edge (v_{in}, v_{out}) of capacity 1.
   (iii) For each pair v, w of adjacent vertices in G, create (in G') directed edges of capacity 1 from v_{out} to w_{in} and from w_{out} to v_{in}.
   (iv) For each terminal vertex v in G, create (in G') a directed edge of capacity 1 from s to v_{in}.
   (v) For each boundary vertex w in G, create a directed edge of capacity 1 from w_{out} to t.
2. Run Ford-Fulkerson on G' to determine the value of a maximum flow.
3. If that max flow is m, return true (escape problem can be solved); else, return false.

Runtime Analysis: There are $2n^2 + 2$ vertices in the network G'. (Two for each vertex of G, and also s and t.) Each of s and t has $O(n^2)$ edges, and each other vertex has at most 5 edges (as many as 4 neighbors, plus the internal edge). The number of edges in G' is therefore $O(n^2) + O(n^2) + 5O(n^2) = O(n^2)$. So the time required for network construction is $O(|V| + |E|) = O(n^2) + O(n^2) = O(n^2)$.

The runtime of Ford-Fulkerson is $O(|E| \cdot |f^*|)$. We have $|E| = O(n^2)$, and the maximum flow value cannot be more than m, so this runtime is $O(mn^2)$. This dominates the $O(n^2)$ term for network construction, so we conclude that our overall runtime is $O(mn^2)$.

Proof of Correctness: It suffices to show that the escape problem is solvable if and only if G' has a flow of total weight m.

First suppose the escape problem is solvable, so that there exist m vertex-disjoint paths from all of the terminals to boundary vertices. Since s has an edge to each terminal and each boundary vertex has an edge to t, this means that there are m vertex-disjoint paths from s to t in G'. All edges in all of these paths have capacity 1, so by sending a single unit of flow through each path, we obtain a flow of total weight m.

Now suppose that G' has a flow of weight m. Since all edge capacities are integers, we know that there exists a flow f^* with only integral flow values such that $|f^*| = m$. The flow f^* is obtainable (via Ford-Fulkerson) by sequentially sending one more unit of flow along an augmenting path. All edge capacities are 1, so no single path can carry more than a single unit of flow; to have a total flow of m, there must therefore be m different s, t paths in G'. These paths are, furthermore, edge-disjoint—if any two of them shared an edge, then the shared edge would carry at least two units of flow, violating the edge capacity of 1.

Each of the m paths in G' can be specified by listing the visited vertices in order, and each path will have the form $(s, v^1_{in}, v^1_{out}, \ldots, v^k_{in}, v^k_{out}, t)$, where (v^1_{in}, v^1_{out}) corresponds to a terminal vertex v^1 and (v^k_{in}, v^k_{out}) corresponds to a boundary vertex v^k. We can "collapse" this path into the terminal-boundary path $(v^1, \ldots, v^k)$ in G by eliminating s, t, and the internal edges. No two of the paths obtained in this way can share any vertices. If two of the paths shared a vertex v, then the internal edge (v_{in}, v_{out}) would be used by both corresponding paths in G', and so that edge would be filled over capacity in f^*. We conclude that there are m vertex-disjoint paths in G from terminal vertices to boundary vertices, so the escape problem is solvable.

3. (a) For $i > 1$, the probability that i is a leaf at the end of the process is the probability that no node $j > i$ connects to it. For a particular j, this probability is $1 - 1/(j-1)$, since there are $j-1$ potential nodes that j could connect to, of which i is one. Since these events are independent for all j, the probability is

$$\left(1 - \frac{1}{i}\right) \cdot \left(1 - \frac{1}{i+1}\right) \cdot \cdots \cdot \left(1 - \frac{1}{n-1}\right) = \frac{i-1}{i} \cdot \frac{i}{i+1} \cdot \cdots \cdot \frac{n-2}{n-1}$$
$$= \frac{i-1}{n-1}$$

For $i = 1$, the calculation is slightly different. In order to be a leaf, *two* later vertices need to select it. However, the second vertex always chooses vertex 1. The net result is

$$\left(1 - \frac{1}{i+1}\right) \cdot \left(1 - \frac{1}{i+2}\right) \cdot \cdots \cdot \left(1 - \frac{1}{n-1}\right) = \frac{i}{n-1}$$

where we use $i = 1$ to contrast with the previous formula. This expression simplifies to $1/(n-1)$.

(b) To get the answer, we sum over the above probabilities for all i in $\{1, \ldots, n\}$. We get

$$\frac{1}{n-1}(1 + 1 + 2 + \cdots + n - 1) = \frac{n}{2} + \frac{1}{n-1}$$

4. With a little initial processing, it is possible to make a greedy approach work here. Consider the jogger i with minimal b_i—that is, the jogger who exits the path at the earliest point. In order for jogger i to see one of the advertisements, we must place an ad on one of the billboards $a_i, \ldots, b_i$. We might as well choose board b_i (the last one that jogger i sees). Each other jogger j has $b_j \geq b_i$, so if jogger j sees a given board $b \in \{a_i, \ldots, b_i\}$, then they will also see board b_i before exiting the path. After selecting board b_i, we can repeat this selection strategy for the other joggers, adding a new billboard only when a jogger's path does not already contain a billboard. Our algorithm makes use of two facts: for each jogger i, at least one of the boards $\{a_i, \ldots, b_i\}$ must be chosen; and in each such case, board b_i is at least as good a choice as any other. Our initial processing will consist of sorting the joggers in increasing order of b_i. The joggers who exit the path earliest come first, and those who exit latest come last.

BillboardSelection:

1. Sort the joggers' board intervals in increasing order of b_i. For convenience, permute the indices so that $b_1 \leq b_2 \leq \cdots \leq b_m$.
2. Keep track of the last billboard selected. Initialize prevBoard $= 0$.
3. for $i = 1, \ldots, m$:
   (a) If $a_i \leq$ prevBoard, do nothing. (The jogger will have already seen the previous board, so we don't need to add another for jogger i.)
   (b) Else, add board b_i, and update prevBoard $\leftarrow b_i$.
4. Return the set of selected boards.

Runtime Analysis: Steps 2 and 4 take constant time each. Step 3 takes constant time for each of m joggers, for a total of $O(m)$ work. Step 1 is a sort over m elements, so it takes time $O(m \log m)$. This last term is the dominant one, so our overall runtime is $O(m \log m)$.

Proof of Correctness: It is clear that each jogger passes a board in our algorithm's solution. Step 3 considers each jogger and, if necessary, adds a board that that jogger will see.

To prove optimality, let $B = \{b_{i_1}, \ldots, b_{i_k}\}$ be the set of billboards chosen by our algorithm. Also let $J_{i_1}, \ldots, J_{i_k}$ be the joggers considered by the algorithm when $b_{i_1}, \ldots, b_{i_k}$ are selected. We claim that any two of these joggers must have disjoint board intervals.

To prove that, consider joggers J_{i_1} and J_{i_2}, and suppose without loss of generality that $b_{i_1} < b_{i_2}$. (This reasoning will extend to any pair of the J_{i_r}.) Suppose, by way of contradiction, that the intervals $\{a_{i_1}, \ldots, b_{i_1}\}$ and $\{a_{i_2}, \ldots, b_{i_2}\}$ overlap. Since $b_{i_1} < b_{i_2}$, this implies that $a_{i_2} \leq b_{i_1}$. The joggers are considered in ascending order of b_i, so jogger J_{i_1} is considered before J_{i_2} in the algorithm. When considering J_{i_1}, the algorithm adds board b_{i_1}. We already noted that $a_{i_2} \leq b_{i_1}$, so board b_{i_1} is seen by jogger J_{i_2}, too. Thus, when jogger J_{i_2} is

considered in a later iteration of step 3, no new boards will be added. But by assumption, a new board—namely, b_{i_2}—is added when J_{i_2} was considered. This gives us our contradiction, and so we conclude that the intervals for J_{i_1} and J_{i_2} are disjoint.

Joggers $J_{i_1}, \ldots, J_{i_k}$ must each pass by a board, and since their intervals are disjoint, we need a separate board for each of them. This establishes k as a lower bound on the number of boards in an optimal solution. By assumption, our algorithm only selects k boards, so our algorithm is optimal.