1 Formatting

Please use the \LaTeX{} default margins. (In other words, don’t use a margin package such as \texttt{fullpage} or \texttt{geometry}.) We’re counting on wide margins so that we have room to embed our feedback there.

In this document, I used the \texttt{amsthm} proof environment. It is one of many \LaTeX{} environments for formatting proofs. If you wish to use this environment and it’s not already part of your \LaTeX{} installation, you can search for it online and download it and its documentation. You can also format your proofs in other ways. Just make sure that they are easy to read and include an end-of-proof marker (e.g., a box or a “Q.E.D.” at the end – which most proof environments will do for you automatically). This particular environment allows you to say:

\begin{verbatim}
\usepackage{amsthm} % Defines the proof environment and more!
\begin{proof}
Blah, blah, blah
\end{proof}
\end{verbatim}

2 Writing

One of the objectives of this course is to practice good writing in the domain of algorithms. This skill is highly valued in both industry and academia. Thus, a substantial fraction of the points on each assignment will be allocated for good writing. Here are the elements that we’ll be looking for:

Executive Summary: Always begin with a short overview – or “executive summary” – of the approach that you’ll be taking to the problem. This provides the reader with the “big picture” and some context in which to read what follows. Sometimes a single sentence will suffice (e.g., “Our approach is to transform this problem into a shortest path problem, prove that our transformation is correct, and then apply Dijkstra’s
algorithm to solve the problem.”). In other cases, when the problem is more involved, you might need a few sentences.

Motivate Your Approach: Provide the reader with motivation for your approach. For example, compare the sentence “We now construct a network flow problem with source, sink, and three sets of vertices, \(X, Y, \) and \(Z \ldots\)” with “This problem is naturally represented as a network flow problem where we can think of vertices as representing teams, edges as representing matches between teams, and the capacity of an edge as representing the number of games to be played between those two teams. So, we introduce three sets of vertices \(\ldots\)”

Begin Paragraphs with an Overview: Break your prose up into relatively short paragraphs just as you would break a long program up into constituent functions. And, just as you would provide short documentation for each function, start each paragraph with a sentence explaining what you’re about to do (e.g., “Now we will prove that our algorithm is correct using a proof by induction” or “Now we analyze the running time of our function.”).

Use Intuitive Notation: Just as you use good variable names in your programs, use intuitive notation in your algorithm description and proof. While some notation is standard (e.g., \(G\) and \(H\) are common names for graphs, \(V\) and \(E\) are common names for the vertex and edge sets in graphs, and \(u, v, \) and \(w\) are widely-used generic names for vertices in graphs), in many cases you’ll need to invent your own notation and variable names. It makes sense to choose notation and names that are easy for the reader to understand and remember. For example, in an argument about cats and mice, using the variable \(c\) for a cat and the variable \(m\) for a mouse is much clearer than using \(x\) and \(y\).

Use Pseudo-Code Only When Necessary: In many cases, pseudo-code doesn’t actually make your algorithm description more clear. But, if you feel that pseudo-code is helpful, only give it after explaining your code in clear English prose. And, then use very high-level pseudo-code, abstracting away variable declarations and cryptic syntax in favor of adding clear constructs like “consider each possible vertex \(u\) adjacent to \(v\) and choose the one that minimizes the distance \(d(u, v)\).”
Read What You Wrote: Please read what you wrote and make an editorial pass before you submit your write-up. Check for spelling and grammatical errors, malformed mathematical expressions, etc. And, almost certainly, you'll find places that you can tweak your write-up to make it clearer and more concise.

Provide a self-assessment of your solution at the end: If you know that something is not quite right in your solution, explain what you understand to be less-than-completely correct. And, if you’re quite confident that your solution is correct, tell us that too. Here’s an example: “In the NP-completeness proof below, I know that there is a problem with the ‘only if’ direction in my proof. That part of the proof doesn’t quite work because . . .”

3 Examples

Here are two sample problems and their solutions, using the writing guidelines above.

3.1 First Example: A Board Game

**Problem:** We are given a $n \times n$ board in which some cells are black and some are white. Our goal is to place $n$ tokens on the board such that the tokens are only placed on white cells, every row contains exactly one token, and every column contains exactly one token. Your algorithm should find such a placement if one exists and return `False` otherwise. For example, the board in Figure 1 has a solution where the cross-marks indicate a valid placement of tokens. Describe an algorithm for this problem, prove it correct, and analyze its running time.

**Solution:** This problem can be naturally reduced to a network flow problem in which each unit of flow selects a white cell in a distinct row and column of the grid. We will thus describe the network flow formulation, prove that our transformation is correct, and then analyze its running time.

The transformation to network flow works as follows: We introduce a source vertex $s$, a set $R$ of $n$ vertices corresponding to the $n$ rows in the grid, a set $C$ of $n$ vertices corresponding to the $n$ columns of the grid, and a sink
Figure 1: An example of a $n \times n$ board and a valid placement of tokens.

$t$. We introduce an edge from $s$ to each vertex in $R$ and from each vertex of $C$ to $t$. For each white cell in row $r$ and column $c$, we introduce an edge from vertex $r \in R$ to vertex $c \in C$. Finally, we make all edges have capacity 1.

Next, we claim that there is a flow from source $s$ to sink $t$ of value $n$ if and only if there exists a valid placement of tokens on the board. Assume that there exists a flow of value $n$ in our network. By the capacity constraints, that flow comprises a collection of $n$ paths from $s$ to $t$ where each path includes a distinct vertex in $r \in R$ and a distinct vertex in $c \in C$. By construction, each such path corresponds to a unique white cell at row $r$ and column $c$. Thus, a flow of value of $n$ corresponds to a set of $n$ white cells no two of which are in the same row or column. Conversely, assume that there exists a valid placement of $n$ tokens. Then each token at row $r$ and column $c$ corresponds to a path from $s$ to $r$ to $c$ to $t$ and thus we can construct $n$ paths that do not share vertices in $R$ nor vertices in $C$, and thus comprise a flow of value $n$ in the constructed network. Thus, after constructing this network, we run the Ford-Fulkerson algorithm to find a maximum flow. If the value of that flow is less than $n$, we return FALSE to indicate that no valid token placement exists. Otherwise, we use the flow to identify a valid placement.

Finally, we analyze the running time of this algorithm. Constructing the network takes time $O(n + |W|)$ where $|W|$ denotes the number of white cells. The Ford-Fulkerson algorithm has worst-case running time $O(|f|(|V| + |E|))$ where $|f|$ denotes the value of the maximum flow, $|V|$ denotes the number of vertices, and $|E|$ denotes the number of edges. In our case, $|f|$ is bounded by $n$, $|V| \in O(n)$ and there are $2n + |W|$ edges. Therefore, the running time of
the construction and network flow algorithm is bounded by $O(n(n+|W|))$. If the value of the flow is equal to $n$, we can simply examine each edge between $r \in R$ and $c \in C$ to determine if there is a flow of 1 on that edge. If so, our solution tells us to place a token at grid cell $(r, c)$. This additional step takes time $O(n^2)$ and thus is subsumed by the $O(n(n+|W|))$ term, which is the overall running time of our algorithm.

Self Assessment: I'm confident in the correctness of my solution.

3.2 Second Example: Cat and Mouse

Problem: *Cat-and-mouse* is a strategy game between two players, named *cat* and *mouse*, played on a connected undirected graph. The input is a graph $G$ with $n$ vertices, a designated vertex $h$ called the *mouse hole*, and starting vertices $c$ and $m$ for the cat and mouse, respectively. Cat and mouse alternate play, with cat starting first. At each move, the player moves to an adjacent vertex. Cat wins if it lands on the mouse. Mouse wins if it reaches the mouse hole before cat catches it or if it can evade the cat indefinitely. The cat is said to have a winning strategy if there exists a move for cat such that for all moves for mouse there exists a move for cat, etc. (alternating between “there exists” a move for cat and “for all” moves for mouse) such that the cat eventually catches the mouse at a vertex other than the mouse hole. In other words, no matter what the mouse does at a given move, there is a response that the cat can make that leads it to eventually catch the mouse. Describe a polynomial time algorithm for determining if there is a winning strategy for the cat and analyze the running time of your algorithm. The algorithm should return True if the cat can win and False otherwise.

Solution: Our approach is to define a recursive algorithm for this problem and then use a dynamic programming approach to make it run in polynomial time. Let us call our recursive function \texttt{cat}(u, v, turn, k) where $u$ is the vertex where the cat is currently located; $v$ is the vertex where the mouse is currently located; \texttt{turn} is either \texttt{cat} or \texttt{mouse}, indicating whether it is the cat’s or mouse’s move, respectively; and $k$ is the number of moves permitted until the end of the game. We define \texttt{cat}(u, v, turn, k) to be True if the cat can win the game given that the cat is at vertex $u$, the mouse is at vertex $v$, the next player to move is given by \texttt{turn}, and we permit up to $k$ more moves to be made before the game to an end.
(Another approach is to define two separate functions, one for the cat and one for the mouse, and use mutual recursion: The cat function determines if the cat can win from a given location for the cat and mouse and, analogously, the mouse function determines if the mouse can win from a given location of the cat and mouse. The cat function calls the mouse function and the mouse function calls the cat function.)

We first note that no winning game for the cat requires more than \( M = 2n^2 \) moves because there are only \( 2n^2 \) distinct game configurations, where a configuration comprises one of \( n \) vertex locations for the cat, one of \( n \) vertex locations for the mouse, and two choices of whether it is the cat’s or mouse’s turn to move. Any game that involves more than \( 2n^2 \) moves contains a cycle that can be removed to construct a shorter game with the same outcome. Since the cat and mouse begin at vertices \( c \) and \( m \), respectively, the cat makes the first move, and no winning game for the cat can comprise more than \( M = 2n^2 \) moves, we ultimately wish to compute \( \text{cat}(c, r, \text{cat}, M) \) where \( M = 2n^2 \).

The base cases for the \( \text{cat} \) function are as follows: If \( v = h \) (the mouse has reached the mouse hole), we return \( \text{FALSE} \) since the cat has lost. Otherwise, if \( u = v \) then the cat has reached the mouse at a vertex other than the mouse hole and we return \( \text{TRUE} \). Otherwise, if \( k = 0 \), we have no more moves remaining and we must return \( \text{FALSE} \).

The general cases for \( \text{cat}(u, v, \text{turn}, k) \) are as follows: If \( \text{turn} \) is \( \text{cat} \) then for each neighbor \( w \) of \( v \) in graph \( G \), consider \( \text{cat}(w, v, \text{mouse}, k-1) \) and return \( \text{TRUE} \) if at least one such recursive call returns \( \text{TRUE} \) (because there exists a move that the cat can make that will result in a win) and otherwise return \( \text{FALSE} \) (because no move that the cat can make will result in a win). If \( \text{turn} \) is \( \text{mouse} \) then for each neighbor \( x \) of \( v \) in graph \( G \), consider \( \text{cat}(u, x, \text{cat}, k-1) \) and return \( \text{TRUE} \) if all such recursive calls return \( \text{TRUE} \) (because the cat can only be assured of a win if every move that the mouse makes will result in a win for the cat) and otherwise return \( \text{FALSE} \).

We prove the correctness of our algorithm by induction on the fourth argument, \( k \), in the function \( \text{cat}(u, v, \text{turn}, k) \). When \( k = 0 \), the base case of the algorithm is invoked and clearly gives the correct answer. Assume that the algorithm is correct for \( k - 1 \) and now consider \( k \). If the mouse has reached the mouse hole, our algorithm correctly returns \( \text{FALSE} \). If the cat has caught the mouse, the algorithm correctly returns the value \( \text{TRUE} \).

Otherwise, the analysis depends on whether it is the cat’s turn or the mouse’s turn. If it is the cat’s turn, then the cat can win from vertex \( u \) if
there is some neighboring vertex $w$ that assures that it will win when it is the mouse’s turn from its current vertex $v$, and the number of remaining moves has been reduced by one. By the induction hypothesis, this value is correctly computed by $\text{cat}(w, v, \text{mouse}, k-1)$. If it is the mouse’s turn, the cat can win only if it can win for every possible move to vertex $x$ that the mouse could make from its current location $u$, and the number of remaining moves has been reduced by one. Again, by the induction hypothesis, $\text{cat}(u, x, \text{cat}, k-1)$ correctly computes each such option.

To make this algorithm efficient, we implement it using dynamic programming. Since the function has four arguments, the DP table will be four-dimensional with dimensions $n \times n \times 2 \times 2n^2$. Since function $\text{cat}(u, v, \text{turn}, k)$ makes recursive calls where the fourth argument is $k - 1$, we can fill in the DP table for $k = 0$ (using the base cases) and then fill in the remainder of the table from $k = 1$ to $2n^2$. Since the table has size $O(n^3)$ and each cell involves $O(n)$ lookups, the total running time is $O(n^4)$.

**Self Assessment:** I’m confident in the correctness of my solution.