CS 170 Algorithms
Spring 2009 David Wagner

Final Soln

Final exam solutions

Please— do not read or discuss these solutions in the exam room while others are still taking the exam.

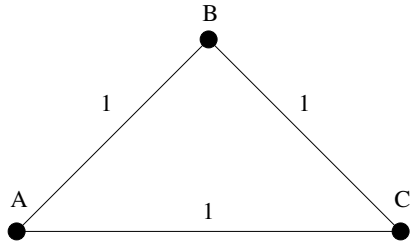
Problem 1. [True or false] (8 points)

(a) [TRUE] or FALSE: Representing a graph in adjacency matrix representation takes $\Theta(|V|^2)$ space.

Justification: It's a $|V| \times |V|$ matrix, which has $|V|^2$ elements.

(b) TRUE or [FALSE]: The minimum spanning tree in a weighted graph G is always unique, no matter what the graph G or the weights are.

Justification: In the following graph, both $A-B-C$ and $B-C-A$ are MSTs of weight 2.



Comment: If the edge weights are all different, then the MST is guaranteed to be unique.

(c) [TRUE] or FALSE: Suppose G is a graph with n vertices and $n^{1.5}$ edges, represented in adjacency list representation. Then depth-first search in G runs in $O(n^{1.5})$ time.

Justification: $|V| = n$, $|E| = n^{1.5}$. The running time of DFS is $O(|V| + |E|) = O(n + n^{1.5}) = O(n^{1.5})$.

Problem 2. [Recurrence relations] (12 points)

Consider the following code for driving a robot in a spiral-like pattern:

SPIRAL(n):
1. If $n \leq 1$, return (without doing any driving).
2. Drive north n miles.
3. Drive west n miles.
4. Drive south n miles.
5. Drive east $n-1$ miles.
6. Drive north 1 mile.
7. Call SPIRAL($n-2$).

Let $M(n)$ denote the number of miles that the robot drives (in total) if we call SPIRAL(n).

(a) $M(1) =$ [0].

(b) $M(2) =$ [8].

(c) Write a recurrence relation for $M(n)$:

$M(n) =$ [$M(n-2)+4n$].

(d) Solve for $M(n)$, as a function of n. Write your answer using $O(\cdot)$ notation.

$M(n) =$ [$O(n^2)$].

Problem 3. [Best guess] (21 points)

(a) [TRUE], SUSPECTED TRUE, SUSPECTED FALSE, or FALSE: 3SAT is in **NP**.

Justification: It is a search problem: we can verify in polynomial time whether a particular assignment to the variables satisfies the 3CNF formula.

(b) [TRUE], SUSPECTED TRUE, SUSPECTED FALSE, or FALSE: 3SAT is NP-complete.

Justification: This follows from the Cook-Levin theorem.

(c) TRUE, SUSPECTED TRUE, [SUSPECTED FALSE], or FALSE: There is a polynomial-time algorithm for 3SAT.

Justification: Not unless $\mathbf{P} = \mathbf{NP}$. Since 3SAT is NP-complete, if there is a polynomial-time algorithm for 3SAT, there is a polynomial-time algorithm for all of **NP**.

(d) TRUE, SUSPECTED TRUE, [SUSPECTED FALSE], or FALSE: The search problem "Given a graph G, vertices s,t, and a threshold h, find a flow from s to t whose value exceeds h, or report that none exists" is NP-complete.

Justification: Not unless $\mathbf{P} = \mathbf{NP}$. There is a polynomial-time algorithm for this problem, so if it is NP-complete, there is a polynomial-time algorithm for all of **NP**.

(e) [TRUE], SUSPECTED TRUE, SUSPECTED FALSE, or FALSE: Given a graph where all edge lengths are integers (possibly negative), there is a polynomial-time algorithm to test whether the graph contains a negative cycle (i.e., a cycle where the sum of the lengths of the edges in the cycle is negative).

Justification: See HW6.

(f) [TRUE], SUSPECTED TRUE, SUSPECTED FALSE, or FALSE: The search problem "Given a graph G with n vertices, find a way to color the graph with $n-1$ colors, or report that no such coloring exists" is in **P**.

Justification: There are only $\binom{n}{2} = O(n^2)$ possible colorings: we choose 2 vertices to receive the same color, and the rest of the vertices each receive their own color. We can try all such candidate colorings in polynomial time and see whether any of them is valid.

Alternative answer: If there exists any pair of vertices u,v with no edge between them, then we can give them the same color and color the rest of the vertices uniquely. So the graph can be colored with $n-1$ colors if and only if it is not a complete graph; and the latter can be checked in polynomial time.

(g) [TRUE], SUSPECTED TRUE, SUSPECTED FALSE, or FALSE: Linear programming is in **P**.

Justification: See the book.

Problem 4. [Updating distances] (10 points)

We have a directed graph $G = (V,E)$, where each edge (u,v) has a length $\ell(u,v)$ that is a positive integer. Let n denote the number of vertices in G. In other words, $n = |V|$. Suppose we have previously computed a $n \times n$ matrix $d[\cdot,\cdot]$, where for each pair of vertices $u,v \in V$, $d[u,v]$ stores the length of the shortest path from u to v in G. The $d[\cdot,\cdot]$ matrix is provided to you.

Now we add a single edge (a,b) to get the graph $G' = (V,E')$, where $E' = E \cup \{(a,b)\}$. Let $\ell(a,b)$ denote the the length of the new edge. Your job is to compute a new distance matrix $d'[\cdot,\cdot]$, which should be filled in so that $d'[u,v]$ holds the length of the shortest path from u to v in G', for each $u,v \in V$.

(a) Write a concise and efficient algorithm to fill in the $d'[\cdot,\cdot]$ matrix. You should not need more than about 3 lines of pseudocode.

Answer:

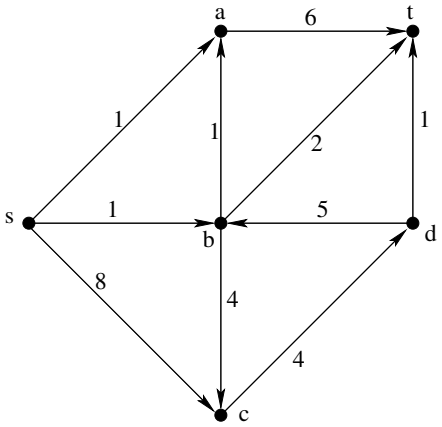
1. For each pair of vertices $u, v \in V$:
2. Set $d'[u,v] := \min(d[u,v], d[u,a] + \ell(a,b) + d[b,v])$.

(b) What is the asymptotic worst-case running time of your algorithm? Express your answer in $O(\cdot)$ notation, as a function of n and $|E|$.

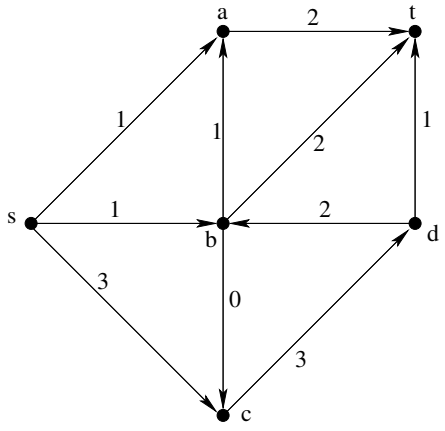
Answer: $O(n^2)$.

Problem 5. [Network flow] (15 points)

Consider the following directed graph. Each edge is labelled with the capacity of that edge.



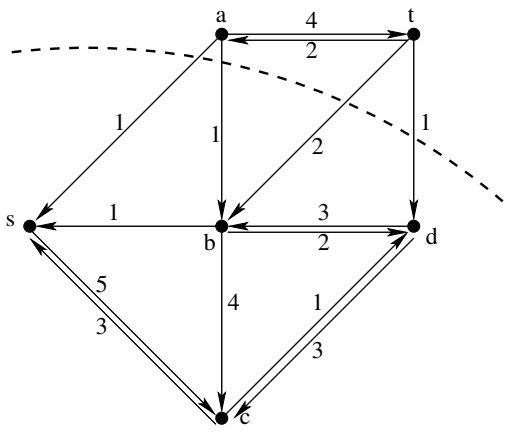
(a) Find the maximum flow from s to t in this graph. Fill in the graph below with your flow: label each edge with the amount of flow you are sending along that edge.



(b) What is the value of your flow?

Answer: 5.

(c) Draw the residual graph corresponding to your flow from part (a). I've provided the verticies; you fill in the edges of the residual graph. Label each edge in your residual graph with its capacity.



(d) Find the minimum-capacity cut (L,R) between s and t in this graph. Show your answer by drawing a circle around the vertices of L, in the picture above.

Answer: $L = \{s,b,c,d\}$, $R = \{a,t\}$, as shown above.

(e) What is the capacity of the cut you identified in part (d)?

Answer: 5.

(f) How could you check you didn't make any mistakes in part (a)? In other words, how could you use your answers to these questions to convince someone else that the flow you drew in part (a) truly is a maximum flow in the graph (without asking them to find the maximum flow themselves from scratch)? Explain, in one or two sentences.

Answer: Check that the value of the flow is equal to the capacity of the cut. Given any cut, the value of any flow must be at most the capacity of the cut. The cut shown above has capacity 5, which proves that no flow can have value greater than 5. So the flow shown in part (a) must be optimal.

Problem 6. [A reduction] (7 points)

GRAPH 3-EQUICOLORING is the following search problem:

Given: An undirected graph G with $3n$ vertices, for some integer n
Find: A 3-equicoloring of G, or report that none exists

Recall that a 3-coloring is a way to color each vertex of G with one of 3 colors, say Blue, Gold, and Tan, so that no pair of adjacent vertices share the same color. A *3-equicoloring* is a 3-coloring where each color is used on the same number of vertices (so in a graph with $3n$ vertices there are exactly n Blue vertices, n Gold vertices, and n Tan vertices). You may assume that GRAPH 3-EQUICOLORING is NP-complete.

CLIQUE is the following search problem:

Given: An undirected graph G and a positive integer k
Find: A clique of size k in G, or report that none exists

A *clique* of size k is a set of k vertices such that every pair of vertices in the clique are connected by an edge. In other words, vertices $v_1, \ldots, v_k$ form a clique of size k if G includes the edge (v_i, v_j) for every i, j.

Professor Lilac proposes to prove that CLIQUE is NP-complete, by reducing GRAPH 3-EQUICOLORING to CLIQUE. She suggests the following argument:

> Suppose we have an algorithm B that solves the CLIQUE problem. I will show how to build an algorithm A to solve GRAPH 3-EQUICOLORING, using B as a subroutine. The algorithm A is given an undirected graph $G = (V, E)$ with $3n$ vertices. A will construct a new undirected graph $G^* = (V^*, E^*)$, as follows. Each vertex of G^* corresponds to a set of n vertices from G that have no edges between them. In other words,
>
> $$V^* = \{S : S \subseteq V, |S| = n, \text{ and } \forall u, v \in S \,.\, (u, v) \notin E\}.$$
>
> The graph G^* will have an edge (S, T) between the sets S, T if these two sets are disjoint. In other words,
>
> $$E^* = \{(S, T) : S, T \in V^* \text{ and } S \cap T = \emptyset\}.$$
>
> We run algorithm B on input G^* and with $k = 3$, to determine whether there exists a clique of size 3 in G^*. If B finds a clique of size 3 in G^*, then this corresponds to a 3-equicoloring of G (each vertex $S \in V^*$ in the 3-clique corresponds to a set of vertices in V that all receive the same color). Moreover, if there is no clique of size 3 in G^*, then there is no 3-equicoloring of G. This proves that CLIQUE is at least as hard as GRAPH 3-EQUICOLORING. Since we know that GRAPH 3-EQUICOLORING is NP-complete, and since CLIQUE is in NP, it follows that CLIQUE is NP-complete, too.

Is Professor Lilac's reasoning correct? Circle YES or NO below. If you circle NO, briefly explain why in the space below (one or two sentences is plenty).

YES NO

Answer: The graph G^* could be exponentially larger than G. Even if B runs in polynomial time, A will not necessarily run in polynomial time.

Further explanation: There are $\binom{3n}{n}$ subsets of V of size n. And

$$\binom{3n}{n} = \frac{3n \times (3n-1) \times \cdots \times n}{n \times (n-1) \times \cdots \times 1} \geq \frac{3}{2} \times \frac{3}{2} \times \cdots \times \frac{3}{2} = 1.5^{2n} = 2.25^n \geq 2^n,$$

which is exponential in n. If G has no edges, then there are this many vertices in G^*, so G^* is exponentially large.

Comment: Another "bad smell" that might have made you suspicious: If Prof. Lilac's proof were valid, it would prove not just that CLIQUE is NP-complete, but that 3-CLIQUE (the problem of determining whether a clique of size 3 exists) is NP-complete. But that can't be right. There is a straightforward polynomial-time algorithm to test whether a graph contains a clique of size 3: just try all $\binom{|V^*|}{3} = O(|V^*|^3)$ possible triples of vertices to see whether they form a clique. A proof that 3-CLIQUE is NP-complete would thus amount to a proof that $\mathbf{P} = \mathbf{NP}$, and we should be suspicious of any claims to have an easy proof of this fact.

However, it turns out it is true that CLIQUE *is* NP-complete. So Prof. Lilac's conclusion happens to be correct; it was just the line of reasoning she used that was faulty.

Problem 7. [A game] (13 points)

Consider the following two-player game, which is *very fun*. We start with a $n \times n$ matrix M filled with positive integers. When it is her turn, a player can delete the last row or last column of the matrix, but only if the sum of the numbers in that row/column is even. If the sum of numbers in the last row is odd, and the sum of numbers in the last column is odd, then the player has no legal move and loses the game.

Your task: given the $n \times n$ matrix $M[\cdot,\cdot]$, design a polynomial-time algorithm to classify the game as either W (first player wins) or L (first player loses). W means that the first player has a strategy to ensure she will win, no matter how the second player plays the game; L means that no matter what the first player does, the second player has a strategy that will ensure him the ultimate victory. (For instance, the 3×3 matrix given above would be classified as W, since there is a strategy Alice can use to ensure she will win.)

In other words, your algorithm should accept $M[1..n, 1..n]$ as input and return either W or L. The running time of your algorithm should be polynomial in n.

(a) In 5 words or less, what would be the best hint you could give to another CS170 student, that would give away how to approach this problem?

Answer: dynamic programming.

(b) Show the main idea and pseudocode for your algorithm.

Main idea: We'll use a table $T[0..n, 1..n]$.
$T[i,j] = \mathsf{W}$ if $M[1..i, 1..j]$ is a first-player win; L otherwise.
We have the relation

$$T[i,j] = \begin{cases} \mathsf{W} & \text{if } M[i,1] + M[i,2] + \cdots + M[i,j] \text{ is even and } T[i-1,j] = \mathsf{L}, \\ \mathsf{W} & \text{if } M[1,j] + M[2,j] + \cdots + M[i,j] \text{ is even and } T[i,j-1] = \mathsf{L}, \\ \mathsf{L} & \text{otherwise.} \end{cases}$$

Pseudocode:

1. Set $T[i,0] := \mathsf{L}$ and $T[0,i] := \mathsf{L}$ for $i := 1, \ldots, n$.
2. For $j := 1, \ldots, n$:
3. For $i := 1, \ldots, n$:
4. If $T[i-1,j] = \mathsf{L}$ and $M[i,1] + \cdots + M[i,j]$ is even, set $T[i,j] := \mathsf{W}$.
5. If $T[i,j-1] = \mathsf{L}$ and $M[1,j] + \cdots + M[i,j]$ is even, set $T[i,j] := \mathsf{W}$.
6. Otherwise, if neither of these conditions attain, set $T[i,j] := \mathsf{L}$.

Comment: The pseudocode above runs in $O(n^3)$ time. It's possible to solve this in $O(n^2)$ time. We first precompute the sums $M[1,j] + \cdots + M[i,j]$ and $M[i,1] + \cdots + M[i,j]$ for every i, j, storing them in a $n \times n$ matrix; this can be done in $O(n^2)$ time. Then steps 4–6 of the code above take $O(1)$ time, so the whole things finishes in $O(n^2)$ time. But you didn't need to do anything nearly so fancy.

Problem 8. [Running time analysis] (14 points)

Professor Wagner has n topics he'd like to lecture on. As he is preparing his lectures, from time to time he notices a dependency: topic i will need to come before topic j. Let's build a data structure to record these dependencies. The data structure has two operations:

- BEFORE(i,j) records that topic i must come before topic j.
- QUERY(i,j) returns true if the previous calls to BEFORE imply that i must come before topic j, or false otherwise.

Note that "comes before" is a transitive relation: if we know that i must come before j, and we know that j must come before k, then it follows that i will have to come before k.

Alice's Algorithm. Alice proposes the following approach. She will maintain a directed graph $G = (V,E)$ with vertices $V = \{1,2,\ldots,n\}$ corresponding to the n topics, represented in memory using adjacency list representation. At any point in time, the graph G will have an edge (i,j) if and only if Professor Wagner has previously called BEFORE(i,j). Alice suggests to implement QUERY using depth-first search in this graph. In pseudocode:

BEFORE(i,j):
1. Add the edge (i,j) to the graph, if it is not already present.

QUERY(i,j):
1. Perform a depth-first search, starting from vertex i (without restarts).
2. If vertex j was reached, return true; otherwise, return false.

Answer the following questions.

(a) Suppose Professor Wagner performs m calls to BEFORE, followed by a single call to QUERY, using Alice's algorithm. What is the worst-case running time of the last call to QUERY? Express your answer as a function of n and m, using $O(\cdot)$ notation. You may assume $0 \leq m \leq n^2$.

Answer: $O(m)$.

Comment: There will be only m edges in the graph, and at most m vertices reachable from i, so DFS (without restarts) will take $O(m)$ time.

(b) Suppose Professor Wagner performs a sequence of $n^{1.5}$ calls to BEFORE and $n^{1.8}$ calls to QUERY, interleaved in some order. As in part (a), he uses Alice's algorithm. What is the total running time of all of these calls, in the worst case? Express your answer as a function of n, using $O(\cdot)$ notation.

Answer: $O(n^{3.3})$.

Comment: The worst case is if Professor Wagner makes all his BEFORE calls before any of the QUERY calls. The BEFORE calls will take $O(n^{1.5})$ time. At that point, there will be $m = n^{1.5}$ edges in the graph, so each of the $n^{1.8}$ QUERY calls could take $O(n^{1.5})$ time. The total running time is $O(n^{1.8} \times n^{1.5}) = O(n^{3.3})$.

Joe's algorithm. Joe has a different idea. He likes the idea of maintaining a directed graph $G = (V,E)$, but he prefers to store his graph in an adjacency matrix representation. Also, Joe's graph is a bit different: at any point in time, his graph will have an edge (i,j) if and only if QUERY(i,j) would return true at that point in time. Joe scribbles quickly on the back of an envelope, and you look over his shoulder and see the following pseudocode:

BEFORE(i,j):
1. If the edge (i,j) is already present in the graph, return.
2. Add the edge (i,j) to the graph.

3. For each k such that (j,k) is an edge in the graph, call BEFORE(i,k).
4. For each h such that (h,i) is an edge in the graph, call BEFORE$(h,\boxed{i})$.

QUERY(i,j):
1. If (i,j) is an edge in the graph, return true; otherwise, return false.

(c) Joe's pseudocode has a small bug in it: sometimes it gives a wrong answer. Circle the error in the pseudocode above. Then, show a simple way to fix the bug using a very small change to the pseudocode.

Answer: BEFORE(h,i) (on line 4) should have been BEFORE(h,j).

(d) Suppose Professor Wagner performs m calls to BEFORE, followed by a single call to QUERY, using Joe's algorithm (with your fix). What is the worst-case running time of the last call to QUERY? Express your answer as a function of n and m, using $O(\cdot)$ notation. You may assume $0 \leq m \leq n$.

Answer: $O(1)$.

(e) Suppose Professor Wagner performs m calls to BEFORE, followed by one more call to BEFORE, using Joe's algorithm (and the fix). How many edges could Professor Wagner's last call to BEFORE have added to the graph, at most (in the worst case)? In other words, if k_{before} denotes the number of edges in the graph just before Professor Wagner's last call to BEFORE, and k_{after} denotes the number of edges in the graph after this call to BEFORE returns, how large could $k_{\text{after}} - k_{\text{before}}$ be? Express your answer as a function of n and m, using $O(\cdot)$ notation. You may assume $0 \leq m \leq n$.

Answer: $O(m^2)$.

Comment: Suppose Prof. Wagner's last call was to BEFORE(a,b). Just before that call, there could be $m/2$ vertices in the graph that can reach vertex a; and there could be $m/2$ vertices reachable from vertex b. Then Joe's algorithm will add an edge from every vertex in the former set, to every vertex in the latter set, for a total of $(m/2)^2 = O(m^2)$ edges.

(f) Suppose Professor Wagner performs m calls to BEFORE, followed by one more call to BEFORE, again using Joe's algorithm (and the fix). What is the worst-case running time of Professor Wagner's last call to BEFORE? Express your answer as a function of n and m, using $O(\cdot)$ notation. You may assume $0 \leq m \leq n$.

Answer: $O(m^2 n)$.

Comment: This code performs $O(n)$ work per edge added to the graph (in line 1 of BEFORE), plus $O(1)$ work per call to BEFORE that does not add any edges. That's because lines 3 and 4 of Joe's algorithm involve iterating over n entries in the adjacency matrix. Since there could be up to $O(m^2)$ edges added to the graph in Professor Wagner's last call to BEFORE, the total amount of work could be as large as $O(m^2 n)$—but no larger.

(g) Suppose Professor Wagner performs a sequence of $n^{1.5}$ calls to BEFORE and $n^{1.8}$ calls to QUERY, interleaved in some order. He continues to use Joe's algorithm (and your fix). What is the total running time of all of these calls, in the worst case? Express your answer as a function of n, using $O(\cdot)$ notation.

Answer: $O(n^3)$.

Comment: Over this entire sequence, we can only add at most $O(n^2)$ edges (we can add at most one edge between every pair of vertices; at that point, no more edges can be added). We do $O(n)$ work per edge added, so that's a total of $O(n^2 \times n) = O(n^3)$ work in all. The time to call QUERY $n^{1.8}$ times is just $O(n^{1.8})$, and $O(n^3 + n^{1.8}) = O(n^3)$.

For these parameters, Joe's algorithm beats Alice's algorithm.