
CSOR 4231: Final Practice Problems, Fall 2014

These problems are ungraded, and are intended as a study aid. Solutions will also be posted on courseworks. They are very similar to the problems that will appear on the final. The final is closed book, closed notes, no cheat sheet. It will cover up material from dynamic programming up to and including NP-completeness.

Problem 1 For each statement below, indicate whether it is true or false and justify your answer.

1. An integer N is given in binary. An algorithm that runs in time $\mathcal{O}(\sqrt{N})$ to find a prime factor of N is considered to be a polynomial-time algorithm.
2. Suppose that problems A and B are both decision problems. Suppose that f is a polynomial-time computable function that maps “Yes” instances of problem A to “Yes” instances of problem B , and “No” instances of problem A to “No” instances of problem B . Then if problem B is NP -hard, so is problem A .
3. Boolean circuit satisfiability is in the class NP .

Solution Item 1. is False. When N is given in binary, the number of bits to represent N is $\log(N)$, so to qualify as polynomial time, an algorithm should run in time $\mathcal{O}(\log^c(N))$ for some constant c .

Item 2. is False. The reduction f shows that we can use a solution to B to solve A , but there may be other, easier ways of solving A . It is not necessarily true here that A is NP -hard.

Item 3. is true. We can check a proposed satisfying assignment in polynomial time by simply computing the circuit for this assignment (a process which is polynomial in terms of the size of the circuit).

Problem 2 Recall the 0-1 Knapsack problem, where there are n items with (positive) weights w_1, \dots, w_n and (positive) values v_1, \dots, v_n . The optimization version of this problem is to maximize the sum of the values for a subset of the items while staying within a given weight constraint W . An item must be taken in whole and cannot be split into fractional pieces.

1. State the decision version of this problem.
2. State what is known about the complexity of the decision version (e.g. is in the class NP ? known to be in P ? known to be NP -complete?)
3. Suppose someone proposes a dynamic programming approach to solving the optimization problem. He suggests defining $V(k, W)$ to be the optimal value that can be obtained using only the first k items with a weight bound of W , and proposes to compute $V(k, W)$ by maximizing over $V(k-1, W)$ and $v_k + V(k-1, W - w_k)$. Will this approach yield a polynomial time algorithm to find the optimal value $V(n, W)$? Explain why or why not.

Solution 1. The decision version of the problem can be stated as follows: Given a bound $b \geq 0$, does there exist a subset S of the items such that the sum of the weights of the items in S is $\leq W$ while the sum of the values is $\geq b$?

2. This decision problem is in the class NP (a proposed subset can be checked in polynomial time). It is NP-complete (which means it is not known to be in P, and if it is discovered to be in P, then all of $NP = P$).

3. Since this problem is NP-complete, we should guess that this approach will not lead to a polynomial time algorithm. One reason is that the computation of $V(k, W)$ requires computing values of the form $V(k-1, W-w_k)$: if we try to compute and store all of the values $V(k-1, W')$ for all the $W' \leq W$ we will encounter in this dependence, it will be far too many values of W' (more than polynomially many). This is because there are 2^n different subsets of the weights to subtract from W , and we need to run in time polynomial in n and the number of bits to represent W , b , $\{v_i\}$, $\{w_i\}$ in binary.

Problem 3 Consider the undirected, connected, weighted graph pictured in the separate pdf file named FinalPracticeProblem3.pdf. Use an algorithm of your choice to find a minimum spanning tree in that graph. Briefly explain your algorithm, indicate which edges of the graph are contained in your final tree, and the order in which you add them to your tree.

Solution We will use Kruskal's algorithm, which initially considers these 8 vertices as 8 separate components and iteratively builds a minimal weight spanning tree by adding a lightest remaining edge that connects two previously disconnected components. When there is more than one lightest such edge (i.e. there is a tie for the lightest edge with this property), we can choose among these tied candidates arbitrarily.

We start by adding the light edge of weight 1 from b to e. We next add the edge of weight 1 from e to h. Next we add the edge of weight 2 from e to a, then the edge of weight 2 from b to d. Now, we add the edge of weight 3 from c to f. At this point, we have 3 connected components: one containing b,e,h,a,d, one containing c and f, and one containing just g. As a lightest edge connecting two of these components, we can next take the edge of weight 4 connecting e and c, and finally the edge of weight 4 connecting h and g.

Problem 4 If we perform depth-first-search on graph, we obtain a forest containing all the vertices of the graph and a subset of the edges. Each node u has a "discovery time" $u.d$ and a "finish time" $u.f$. We consider the possible relationships between the intervals $[u.d, u.f]$ and $[v.d, v.f]$ for two distinct vertices u and v . For each, either draw a DFS tree where the relationship holds, or explain why it is impossible.

- $u.d < v.d$ and $v.f < u.f$
- $u.f < v.d$
- $u.d < v.d < u.f < v.f$

Solution 1. This situation arises when v is a descendant of u in the DFS tree. An example is pictured in Figure 1. below.

2. This situation can arise when u is discovered before v and v is not reachable from u . An example is pictured in Figure 2. below for a directed graph.

3. This is impossible. Since u is discovered before v but not yet finished, all of the nodes reachable from v will be discovered before u is finished, hence $u.d < v.d$ implies that $v.f < u.f$.

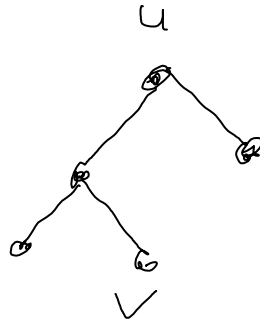


Figure 1: 1. v is a descendant of u in the DFS tree.

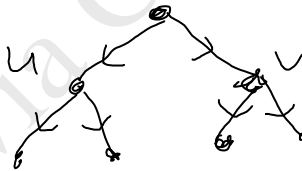


Figure 2: 2. u is discovered first and v is not reachable from u .

Problem 5

- Consider a flow network as a directed graph G where each edge (u, v) has a capacity $c(u, v) \geq 0$, and for each pair of vertices u, v , at most one of (u, v) and (v, u) is in the edge set E (so there is always only one direction for flow to travel between u and v). We designate a source node s and a sink node t . Recall that a cut (S, T) is a partition of the nodes into sets S and T with $s \in S$ and $t \in T$. The capacity of the cut (S, T) is the sum of the capacities of the edges that go from a vertex in S to a vertex in T . Is it possible to have such a G where the max flow from s to t is 13 and there exists a cut (S, T) with capacity 10? Explain your answer.
- Consider an directed graph G , containing vertices s, u, t (along with many others). Describe an efficient way to compute the number of edge disjoint paths between s and t that *do not* go through the vertex u .
- Consider a linear programming problem A whose goal is to maximize a function f subject to given constraints. Its dual is a problem B of minimizing a function g subject to other constraints. If you find a setting of the variables for problem A that satisfy the constraints and yield $f = 7$, and you also find a setting of the variables for problem B that satisfy the constraints and yield $g = 7$, what can you conclude?

Solution For the first part, recall the max-flow, min-cut theorem. This states that the maximum flow we can send from s to t is equal to the minimum capacity of a cut (S, T) such that $s \in S$ and $t \in T$. Thus, if there is a cut with capacity 10, the max flow must be at most 10, and hence cannot be 13. So no, this is not possible.

For the second part, turn G into a flow network by assigning a capacity to each edge. If the edge does not involve u , then assign it a capacity of 1. If it does involve u , assign it a capacity of 0. Thus, edges that travel through u will not contribute to the max flow. Now, if we compute the max flow from s to t in such a graph, it will count the number of edge disjoint paths, since a flow of 1 can be sent along each edge disjoint path, and no additional flow can then be sent.

For the third part, you can conclude that 7 is indeed the maximum value of f for problem A and the minimum value of g for problem B . This follows from linear programming duality, which states that every value achievable for f under the constraints of A is \leq every value achievable for g under the constraints of B , and there is a unique value achievable for both that is the desired max for A and min for B .

Problem 6

- Suppose someone discovered a polynomial algorithm for deciding whether or not a (poly-size) boolean formula is satisfiable. Describe how to derive a polynomial time algorithm for finding a satisfying assignment (whenever one exists). Could you in polynomial time find *all* satisfying assignments? Explain why or why not.
- Consider two decision problems, 0-1 Linear Programming and Graph 3-Colorability. An instance of the 0-1 Linear Programming Problem is described by variables x_1, \dots, x_n and a set of linear constraints on these variables. The answer is “Yes” if and only if there exists assignments of 0’s and 1’s to the variables x_1, \dots, x_n such that all of the constraints are satisfied. An instance of the Graph 3-Colorability problem is a description of an undirected graph G . The answer is “Yes” if and only if the vertices of G can each be assigned one of three colors such that no edge has both its endpoints assigned to the same color. Show that Graph 3-Colorability \leq_p 0-1 Linear Programming by demonstrating how to map

instances of Graph 3-Colorability into instances of 0-1 Linear Programming such that the “Yes” and “No” answers are preserved. Since Graph 3-Colorability is NP-complete, what can you conclude about the complexity of 0-1 Linear Programming?

Solution For the first part, one can proceed as follows. Suppose there are n variables in a boolean formula. If we fix the first variable to take the value 0, we can rewrite the (now simplified) formula as a new formula with $n - 1$ variables (it will be of comparable size to the original formula). We can then run the polynomial time algorithm to see if this new formula is satisfiable or not. If it is satisfiable, we are on track to find a satisfying assignment with $x_1 = 0$. If it is not, we have learned that no satisfying assignment exists with $x_1 = 0$, so we set $x_1 = 1$. Either way, we set x_1 appropriately and now continue with this new formula, next setting $x_2 = 0$, simplifying, and testing satisfiability for a formula with $n - 2$ variables. We continue in this way until we have produced a satisfying assignment or proven that none exists. Note that this will take time $\mathcal{O}(nR)$ where R is the running time of the satisfiability algorithm we are using as a subroutine. So if R is polynomial, so is our algorithm to produce a satisfying assignment. We cannot, however, hope to find all of the satisfying assignments in polynomial time for an arbitrary formula. For example, the formula could be satisfied by *all* assignments, and this would take us time 2^n just to write them all down.

For the second part, we consider a graph G with vertices $V = \{1, 2, \dots, n\}$ and edge set E . We will define a 0-1 linear program with variables $x_1, y_1, z_1, x_2, y_2, z_2, \dots, x_n, y_n, z_n$. Intuitively, we want setting $x_1 = 1$ to correspond to coloring vertex 1 with color 1, setting $y_1 = 1$ to correspond to coloring vertex 1 with color 2, and setting $z_1 = 1$ to correspond to coloring vertex 1 with color 3.

To enforce that a vertex should have exactly one color, we will add constraints

$$\begin{aligned}x_i + y_i + z_i &\leq 1, \\x_i + y_i + z_i &> 0\end{aligned}$$

for each i from 1 to n . When these variables take 0,1 values, the only way to satisfy these is to pick exactly one color per vertex.

Now we want to enforce that each edge has differently colored endpoints. Consider an edge from vertex i to j . Then we add constraints:

$$\begin{aligned}x_i + x_j &\leq 1, \\y_i + y_j &\leq 1, \\z_i + z_j &\leq 1.\end{aligned}$$

These enforce that i and j cannot be the same color, since all of the variables must be 0,1 valued.

The collection of all these constraints is a 0-1 linear program, whose size and computation time is polynomial in $n, |E|$, and if G is 3-colorable if and only if this program has a solution with 0,1 values for its variables. Hence, the fact that 3-coloring is NP-complete along with this reduction means that 0-1 linear programming is also NP-hard, and hence NP-complete since a proposed solution can be checked in polynomial time (though we already knew this was NP-complete via other methods).

This is not something you would need to say on the exam, but FYI note that this reduction *does not go both ways*. We could not use this same argument to show that NP-completeness of 3-colorability using NP-completeness of 0-1 linear programming, b/c we have not given a general way of transforming a linear program into a graph coloring problem - we have only related certain specially structured programs to 3-coloring of graphs.