

## Final Project

---

This final project draws together concepts from across the quarter: graph algorithms, divide-and-conquer algorithms, randomized algorithms, greedy algorithms, dynamic programming, and intractability. The problems here are designed to combine these topics together in new ways so that you can appreciate how versatile a skillset you've acquired this quarter.

**Answer any two of the three problems given here.** Each problem combines two different techniques from the course, so by answering two of the three problems you will have demonstrated a mastery of four of the six topics we have covered. You will not improve your overall score by submitting answers to multiple problems – if you do, we will only grade the first two – but you are welcome to answer all three problems and submit your answers to the two you are most comfortable with.

Each problem consists of four parts, each of which builds off of the previous parts. If at any point you get stuck on a part of a problem, you can skip that part and continue onto the next parts as if you had completed it. For example, if part (ii) of a question asks you to find an algorithm for solving problem  $X$  in time  $O(f(n))$ , you can use the fact that such an algorithm exists in parts (iii) and (iv) even if you cannot solve part (ii).

**This project must be completed on your own.** While we permit collaboration on the problem sets, you **must not** collaborate with anyone else on this project. You can ask the course staff clarifying questions about the problems if you are unsure what they are asking, but we will not provide hints, check your work, etc.

**You must not consult any outside resources when completing this project.** You may only refer to materials on the course website, the book *Algorithm Design* by Kleinberg and Tardos, notes that you yourself have taken over the course of the class, lecture videos, and your own graded problem sets. For example, you **must not** use a search engine to look up anything related to any of the problems in this project, nor should you look at any other students' notes.

You can submit your answers in one of two ways. First, you can submit electronically by emailing the submissions list at [cs161-sum1213-submissions@lists.stanford.edu](mailto:cs161-sum1213-submissions@lists.stanford.edu) and attaching your project as a single PDF. Second, you can submit a hard copy by dropping off your answers in the submissions filing cabinet in the Gates building located inside the entrance marked “Stanford Engineering Venture Fund Laboratories.” The filing cabinet is dark gray and the top drawer is marked as the CS161 drop-off- box.

This final project is worth 30% of your grade in this course. Each problem is worth 24 points, and since you will complete two of the three problems, this final project is out of 48 points.

**It has been a pleasure teaching CS161 this quarter. Best of luck on the final project!**

**Due Saturday, August 17 at 12:15 PM  
No Late Submissions Accepted**

**Problem One: Multicolored Spanning Trees**

Suppose that you have a connected, undirected graph  $G = (V, E)$  where each edge is colored either red or blue. Given a number  $k$ , you are interested in determining whether there is some spanning tree of  $G$  that contains exactly  $k$  blue edges.

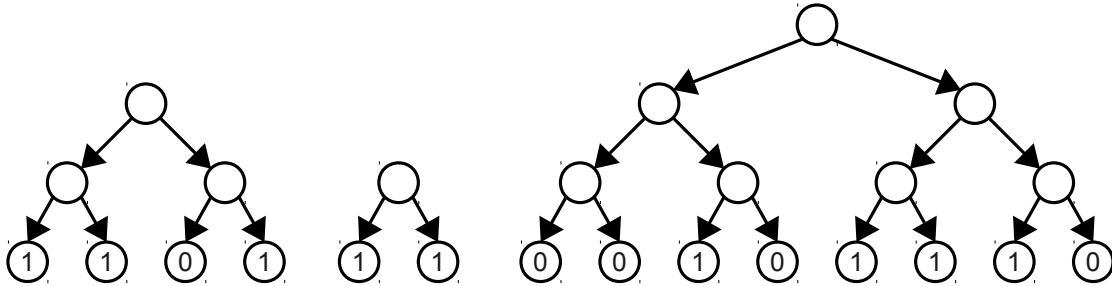
- i. **(5 Points)** Design a polynomial-time algorithm that finds a spanning tree of  $G$  containing the *minimum* possible number of blue edges. Then:
  - Describe your algorithm.
  - Prove that your algorithm finds a spanning tree of  $G$  containing the minimum possible number of blue edges.
  - Prove that your algorithm runs in polynomial time.
- ii. **(3 Points)** Design an algorithm that finds a spanning tree of  $G$  containing the *maximum* possible number of blue edges. Then:
  - Describe your algorithm.
  - Prove that your algorithm finds a spanning tree of  $G$  containing the maximum possible number of blue edges.
  - Prove that your algorithm runs in polynomial time.
- iii. **(12 Points)** Suppose  $T_1$  and  $T_2$  are spanning trees of  $G$  where  $T_1$  contains  $k_1$  blue edges and  $T_2$  contains  $k_2 > k_1$  blue edges. Prove there must be some spanning tree  $T$  of  $G$  containing exactly  $k_1 + 1$  blue edges.
- iv. **(4 Points)** Design an algorithm that, given a number  $k$ , determines whether there is a spanning tree of  $G$  that contains exactly  $k$  blue edges. Note that you don't need to *find* such a spanning tree; you just need to determine whether one exists. Your algorithm should run in time polynomial in  $n$  and  $m$  (the number of nodes and edges in  $G$ ), but not in  $k$ . Then:
  - Describe your algorithm.
  - Briefly justify why your algorithm determines whether there is a spanning tree of  $G$  containing exactly  $k$  blue edges. You don't need to write a formal proof here, but should give a one-paragraph justification as to why your algorithm works.
  - Briefly justify why your algorithm runs in time polynomial in  $n$  and  $m$ .

## Problem Two: Evaluating NAND Trees

A *NAND tree* is a complete binary tree with the following properties:

- Each leaf node is labeled either 0 or 1.
- All internal nodes are *NAND gates*. A NAND gate is a logic gate that takes in two inputs and evaluates to 0 if both its inputs are 1 and to 1 if either input is 0.

We can *evaluate* a NAND tree by computing the value of the top-level NAND gate in the tree, which will evaluate either to 0 or to 1. (If the tree is a single leaf, the tree evaluates to the value of that leaf.) For example, the left and right trees below evaluate to 1; the middle tree evaluates to 0:



Here is a simple recursive algorithm for evaluating a NAND tree:

- If the tree is a single leaf node, return the value of that node.
- Otherwise, recursively evaluate the left and right subtrees, then apply the NAND operator to both of those values.

This algorithm takes  $\Theta(n)$  time to evaluate a NAND tree with  $n$  leaf nodes. We can improve this algorithm using *short-circuiting*. If one subtree of node  $v$  evaluates to 0, then  $v$  must evaluate to 1 because  $0 \text{ NAND } 0 = 1$  and  $0 \text{ NAND } 1 = 1$ . Therefore, we don't need to evaluate  $v$ 's other subtree. This gives the following algorithm, which we'll call the *left-first algorithm*:

- If the tree is a single leaf node, return the value of that node.
- Otherwise:
  - Recursively evaluate the left subtree.
  - If it evaluates to 0, return 1.
  - Otherwise, recursively evaluate the right subtree.
  - If it evaluates to 0, return 1; otherwise return 0.

In many cases, the left-first algorithm runs faster than the  $\Theta(n)$ -time naïve algorithm. However, it is possible to construct NAND trees for which the left-first algorithm runs in time  $\Theta(n)$ .

- (8 Points)** Design an algorithm that creates a NAND tree  $T$  with  $n = 2^k$  leaf nodes such that the left-first algorithm never short-circuits when evaluating  $T$ . Your algorithm should run in time polynomial in  $n$ . Then:
  - Describe your algorithm.
  - Prove that your algorithm produces a tree  $T$  with  $n$  leaves such that the left-first algorithm never short-circuits when evaluating  $T$ .
  - Prove your algorithm runs in time polynomial in  $n$ .

Since the left-first algorithm never short-circuits on inputs produced by your algorithm, the left-first algorithm has a worst-case runtime of  $\Theta(n)$ .

More generally, *any* deterministic algorithm for evaluating a NAND tree will have at least one input that causes it to run in  $\Theta(n)$  time, but you don't need to prove this.

Despite the  $\Theta(n)$  worst-case for deterministic evaluation algorithms, there is a simple *randomized* algorithm for evaluating NAND trees that, on expectation, does less than  $\Theta(n)$  work. The idea is simple: use the same algorithm as above, but choose which subtree to evaluate first uniformly at random. We'll call this the *random-first algorithm*. More concretely:

- If the tree is a single leaf node, return the value of that node.
- Otherwise:
  - Choose one of the subtrees of the root at random and evaluate it.
  - If the value is 0, return 1.
  - Otherwise, recursively evaluate the other subtree.
  - If the value is 0, return 1; otherwise return 0.

To determine the runtime of the random-first algorithm, we will introduce *two* recurrence relations. Let  $T_0(n)$  be the *expected* runtime of the random-first algorithm on a tree with  $n$  leaf nodes assuming the root evaluates to 0. Let  $T_1(n)$  be the *expected* runtime of the random-first algorithm on a tree with  $n$  leaf nodes assuming the root evaluates to 1.

- ii. **(6 Points)** Prove that the following recurrence relations for  $T_0(n)$  and  $T_1(n)$  are correct:

$$\begin{aligned} T_0(1) &\leq \Theta(1) \\ T_0(n) &\leq 2T_1(n/2) + \Theta(1) \\ T_1(1) &\leq \Theta(1) \\ T_1(n) &\leq \frac{1}{2}T_1(n/2) + T_0(n/2) + \Theta(1) \end{aligned}$$

- iii. **(6 Points)** It turns out that  $T_1(n) \leq T_0(n)$ , though it's somewhat difficult to formally establish this. Using this fact, prove that  $T_0(n) = O(n^\epsilon)$  for some  $\epsilon < 1$ . You can assume  $n = 4^k$  for some natural number  $k$ . (*Hint: Write  $T_0(n)$  in terms of itself.*)

Your result from (iii) proves that the random-first algorithm has expected sublinear runtime on *all* inputs, since  $T_1(n) \leq T_0(n) = O(n^\epsilon) = o(n)$ . This is one of a few known problems where the best randomized algorithm is more efficient on expectation than the best deterministic algorithm in the worst case.

The last part of this problem explores this question: what happens if you try to evaluate a randomly-chosen NAND tree? The result is surprising.

Let's say a *random NAND tree* with  $n = 2^k$  leaves is a NAND tree where each leaf is independently assigned a value of 0 or 1 uniformly at random.

- iv. **(4 Points)** Let  $P_0(n)$  denote the probability that a random NAND tree with  $n$  leaves evaluates to 0 and  $P_1(n)$  denote the probability that a random NAND tree with  $n$  leaves evaluates to 1. Write recurrence relations for  $P_0(n)$  and  $P_1(n)$  and briefly explain why your recurrences are correct.

The recurrence relations you came up with in (iv) can't be solved using the techniques we've developed in this course, but you can easily write a short computer program to determine their values by writing out  $n = 2^k$  and evaluating the recurrence for increasing values of  $k$ . If you do, you'll find that when  $k \geq 15$ ,  $P_0(n)$  is extremely close to 1 if  $k$  is even and  $P_1(n)$  is extremely close to 1 if  $k$  is odd. Consequently, the algorithm "return the height of the tree modulo 2" returns the right answer with high probability in time  $\Theta(\log n)$ , even though it never actually evaluates the tree!

### Problem Three: Constrained Scheduling

Suppose you have a supercomputer that can run jobs one at a time. You have a set of jobs  $J$  that you need to run and want to determine the best order in which to run them. Not all jobs take the same amount of time to complete; specifically, job  $j_k$  takes time  $t_k$  to complete. Each job must run to completion once started, so you can't pause or stop a job after starting it.

Certain jobs depend on results computed by other jobs, so you cannot run the jobs in a completely arbitrary order. Specifically, you have a DAG  $G = (J, E)$  whose nodes are the jobs  $J$  and where each edge  $(j_i, j_k)$  indicates that job  $j_i$  must be run before job  $j_k$ .

Under these restrictions, it's easy to schedule all the jobs as efficiently as possible: just topologically sort the DAG and run the jobs in that order. Of course, there's a catch. Associated with each job  $j_k$  is a cost function  $c_k(t)$  denoting the cost of completing job  $j_k$  at time  $t$ . These functions are monotonically increasing, so for any job  $j_k$  and any  $\epsilon > 0$ , we have  $c_k(t) < c_k(t + \epsilon)$ . Your task is to find a way of ordering all of the jobs on the supercomputer so that all constraints are satisfied and the total cost is as low as possible. Specifically, you want to minimize

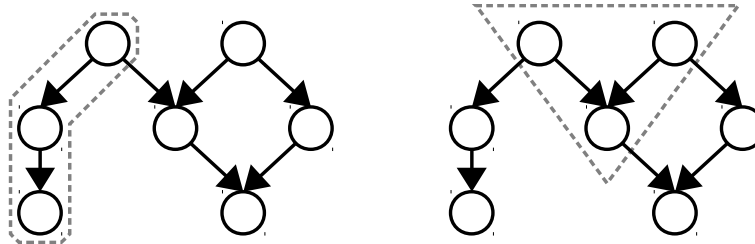
$$\sum_{j_k \in J} c_k(f(j_k))$$

Where  $f(j_k)$  denotes the time at which job  $j_k$  finishes. This problem is known to be **NP-hard**.

A naïve algorithm for this problem is try out every possible topological ordering of the DAG and find the ordering with the least total cost, but this algorithm can be incredibly slow.

- i. **(4 Points)** Prove that for all  $n \geq 0$ , there is a DAG with  $n$  nodes and  $\Omega(n!)$  topological orderings. This shows the naïve algorithm has worst-case runtime  $\Omega(n!)$ .

Fortunately, we can improve upon the naïve algorithm using dynamic programming. Let's call a set  $S \subseteq J$  a *feasible set* iff for every  $j_k \in S$ , if there is a path from  $j_i$  to  $j_k$  in  $G$  (i.e.  $j_k$  depends on  $j_i$ ), then  $j_i \in S$ . Intuitively, a feasible set is a set of jobs that can be scheduled without missing any prerequisites. For example, in the following DAGs, the indicated nodes are feasible sets:



For any feasible set  $S$ , let  $LAST(S)$  denote the set of all jobs  $j_k \in S$  such that  $(j_k, j_i) \notin E$  for any  $j_i \in S$ . In other words,  $LAST(S)$  consists of all jobs in  $S$  that no other jobs depend on.

- ii. **(3 Points)** Prove that if  $S$  is a feasible set, then  $S - \{j\}$  is feasible for any  $j \in LAST(S)$ .

For any feasible set  $S$ , let  $OPT(S)$  denote the optimal cost of scheduling the jobs in set  $S$ .

- iii. **(5 Points)** Prove that in any optimal schedule for the jobs in  $S$ , the supercomputer is never idle before all jobs have completed (i.e. until all jobs have finished executing, the supercomputer is always executing some job.)
- iv. **(12 Points)** Write a recurrence relation for  $OPT(S)$ , then prove that your recurrence relation is correct.

Given a recurrence relation for  $\text{OPT}(S)$ , it's possible to find the cost of an optimal schedule by using the following dynamic programming algorithm:

- Let DP be a table of size  $2^n$ .
- For each subset  $S \subseteq J$ , in an appropriate order:
  - If  $S$  is feasible, fill in  $\text{DP}[S]$  based on the recurrence from (iv).
- Return  $\text{DP}[J]$ .

If we assume each function  $c_k$  can be evaluated in time  $O(1)$ , then (with the right recurrence relation for  $\text{OPT}(S)$ ) it's possible to fill each entry of DP in time  $O(n + m)$ . It's also possible to check whether a set is feasible in time  $O(n + m)$ . This means that the overall runtime for this algorithm is  $O(2^n(n + m))$ , which is significantly better than the  $\Omega(n!)$  worst-case of the naïve algorithm!