TTIC 31010 / CMSC 37000 - Algorithms February 19, 2019
Lecture #11 last changed: February 18, 2019

Today we will complete our discussion of *Linear Programming* and begin discussing the topic of *NP-Completeness.*

1 Linear Programming Duality Recap

Consider a "primal" maximization LP. Solve for $\mathbf{x}$ to:

$$\begin{aligned} \text{maximize } & \mathbf{c}^T\mathbf{x} \\ \text{subject to } & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}, \end{aligned} \tag{1}$$

(Recall that $\mathbf{a} \leq \mathbf{b}$ for vectors means component-wise inequality, i.e., $a_i \leq b_i$ for all i.) Since any positive linear combination of constraints is a valid constraint, we can get an upper bound on the best possible objective value by finding a positive linear combination whose left-hand-side is some $\mathbf{c}' \geq \mathbf{c}$: the right-hand-side then is a bound on how large an objective we could possibly achieve. We can solve for the *best* upper bound we can produce in this way as another linear program called the dual. The dual LP asks to minimize the right-hand-side subject to the left-hand-side being some $\mathbf{c}' \geq \mathbf{c}$. Specifically, the dual LP is to solve for weights y_i on the rows of A to:

$$\begin{aligned} \text{minimize } & \mathbf{y}^T\mathbf{b} \\ \text{subject to } & \mathbf{y}^T A \geq \mathbf{c}^T \\ & \mathbf{y} \geq \mathbf{0}, \end{aligned} \tag{2}$$

Notice that if the primal has n variables and m constraints then the dual has m variables and n constraints. And if you take the dual of (2) to try to get the best lower bound on this LP, you'll get (1). *The dual of the dual is the primal.*

One natural question is whether one can obtain a better upper-bound on the primal LP in a different way, but in fact the optimum of the dual is a tight upper bound. This is called *strong duality.*

Theorem 1 (Strong Duality) *If $\mathbf{x}^*$ is an optimal feasible solution to the primal LP (assume the primal LP is feasible and the optimal value is not infinite) then the dual LP is feasible and its optimal solution $\mathbf{y}^*$ satisfies $\mathbf{c}^T\mathbf{x}^* = \mathbf{b}^T\mathbf{y}^*$.*

2 LP Duality for Shortest Paths

Let's see how this works for the problem of finding the shortest path from s to t in a weighted graph. We'll write this problem as a linear program, and then take the dual and see what it gives us. Our initial LP will be a minimization problem, and then the dual will be a maximization problem, so I'll use y's for the variables for our initial formulation. We will see that the primal and dual correspond to the two ways of writing shortest paths as an LP that we saw in tutorial.

To write shortest paths as an LP minimization problem, consider the problem of sending one unit of flow from s to t in the cheapest way possible, where each edge (u, v) has a cost equal to its length w_{uv}. The cheapest solution will only use shortest paths, and the value of the cheapest solution will be the length of the shortest path. To make this look more like the standard form, we will allow the algorithm to send ≥ 1 total units of flow, and will allow leakage at internal nodes (flow in $\geq$ flow out) so long as at least 1 unit ends up at t. This is legal because putting extra flow on edges is only more expensive. (Assume all edge weights are non-negative).

We will have one variable y_{uv} for each edge $(u, v) \in E$. The LP then looks like this:

Minimize:

$$\sum_{(u,v)\in E} w_{uv} y_{uv}$$

Subject to:

$$\begin{array}{rcll} \displaystyle\sum_{u:(u,t)\in E} y_{ut} & \geq & 1 & \text{(at least one unit of flow into } t\text{)} \\ \forall v \notin \{s,t\}, \displaystyle\sum_{u:(u,v)\in E} y_{uv} - \sum_{u:(v,u)\in E} y_{vu} & \geq & 0 & \text{(flow in } \geq \text{ flow out)} \\ \mathbf{y} & \geq & \mathbf{0} & \end{array}$$

Now, let's write down the dual of the above LP. We will have one non-negative variable for each constraint in the LP above (except for the non-negativity constraints), which we can think of as one variable x_v per vertex $v \neq s$. Then, since we want to produce a lower bound on $\sum_{(u,v)\in E} w_{uv} y_{uv}$, we want this weighted sum of inequalities to have a left-hand-side that is $\leq w_{uv}$ in each coordinate uv. So, what does that constraint mean? Let's break this down into cases:

- For $(u', v') \in E$, such that $u', v' \notin \{s, t\}$, we need $x_{v'} - x_{u'} \leq w_{u'v'}$.
- For $(u', t) \in E$ such that $u' \neq s$ we need $x_t - x_{u'} \leq w_{u't}$.
- For $(s, v') \in E$ we need $x_{v'} \leq w_{sv'}$, or equivalently, we need $x_{v'} - x_s \leq w_{sv'}$ where we set $x_s = 0$.

And of course we need $x_v \geq 0$ for all v. Now, given these constraints, we want to maximize the right-hand-side, which is just x_t. So, our dual LP looks like:

Maximize: x_t

Subject to:

$$\begin{array}{rcll} x_v - x_u & \leq & w_{uv} & \text{for all } u, v \in E \\ x_s & = & 0 & \\ \mathbf{x} & \geq & \mathbf{0} & \end{array}$$

This is equivalent to asking: how far apart can you separate nodes s and t if every edge is is a string of length equal to its weight. Both LPs solve to the length of the shortest path (which we went into in more detail in tutorial), one as a minimization problem and one as a maximization problem.

3 NP-completeness

In the past few lectures we have looked at increasingly more expressive problems that we were able to solve using efficient algorithms. In this lecture we introduce a class of problems that are so expressive — they are able to model *any* problem in an extremely large class called **NP** — that we believe them to be *intrinsically unsolvable by polynomial-time algorithms.* These are the **NP-complete** problems. What is particularly surprising about this class is that they include many problems that at first glance appear to be quite benign. Specific topics in this lecture include:

- Reductions and expressiveness
- Informal definitions and the ESP problem
- Formal definitions: decision problems, P and NP.

4 Introduction: Reduction and Expressiveness

In the last few lectures have seen a series of increasingly more expressive problems: network flow, min cost max flow, and finally linear programming. These problems have the property that you can code up a lot of different problems in their "language". So, by solving these well, we end up with important tools we can use to solve other problems.

To talk about this a little more precisely, it is helpful to make the following definitions:

Definition 2 *We say that an algorithm runs in* **Polynomial Time** *if, for some constant c, its running time is $O(n^c)$, where n is the size of the input.*

In the above definition, "size of input" means "number of bits it takes to write the input down". So, to be precise, when defining a problem and asking whether or not a certain algorithm runs in polynomial time, it is important to say how the input is given. For instance, the basic Ford-Fulkerson algorithm is *not* a polynomial-time algorithm for network flow when edge capacities are written in binary, but both of the Edmonds-Karp algorithms *are* polynomial-time.

Definition 3 *A Problem A is* **poly-time reducible** *to problem B (written as $A \leq_p B$) if we can solve problem A in polynomial time given a polynomial time black-box algorithm for problem B. Problem A is* **poly-time equivalent** *to problem B ($A =_p B$) if $A \leq_p B$ and $B \leq_p A$.*

For instance, we gave an efficient algorithm for Bipartite Matching by showing it was poly-time reducible to Max Flow. Notice that it could be that $A \leq_p B$ and yet our fastest algorithm for solving problem A might be slower than our fastest algorithm for solving problem B (because our reduction might involve several calls to the algorithm for problem B, or might involve blowing up the input size by a polynomial but still nontrivial amount).

5 Our first NP-Complete Problem: ESP

Many of the problems we would like to solve have the property that if someone handed us a solution, we could at least check if the solution was correct. For instance the Traveling Salesman Problem asks: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?" We may not know how to find such a tour quickly, but if someone gave such a tour to us, we could easily check if it satisfied the desired conditions (visited all the vertices and had total length at most k). Similarly, for the 3-Coloring problem: "Given a graph G, can vertices be assigned colors red, blue, and green so that no two neighbors have the same color?" we don't know of any polynomial-time algorithms for solving the problem but we could easily check a proposed solution if someone gave one to us. The class of problems of this type — namely, if the answer is YES, then there exists a polynomial-length proof that can be checked in polynomial time — is called **NP**. (we define the class **NP** formally in Section 7).

Let's consider now what would be a problem *so expressive* that if we could solve it, we could solve any problem of this kind. Moreover, let's see if we can define the problem so that it is of this kind as well. Here is a natural candidate:

Definition 4 Existence of a verifiable Solution Problem (ESP)*: The input to this problem is in three parts. The first part is a program $V(I, X)$, written in some standard programming language, that has two arguments.*[1] *The second part is a string I intended as a first argument, and the third part is a bound b written in unary (a string of b 1s). Question: does there exist a string X, $|X| \leq b$, such that $V(I, X)$ halts in at most b steps and outputs YES?*

What we will show is that (a) ESP $\in$ **NP** and (b) for any problem $Q \in$ **NP** we have $Q \leq_p$ ESP. (I.e., if you "had ESP" you could solve any problem in **NP**).[2]

Let's begin with (a): why is ESP $\in$ **NP**? This is the reason for the bound b written in unary. If we didn't have b at all, then (since we can't even in general tell if a program is ever going to halt) the ESP question would not even be computable. However, with the bound b, if the answer is YES, then there is a short proof (namely the string X) that we can check in polynomial time (just run $V(I, X)$ for b steps). The reason we ask for b to be written in unary is precisely so that this check counts as being polynomial time: if b were in binary, then this check could take time exponential in the number of bits in the input (much like Ford-Fulkerson is not a polynomial-time algorithm if the capacities are written in binary).

Now, let's go to (b): why is it the case that for any problem $Q \in$ **NP** we have $Q \leq_p$ ESP? Consider some **NP** problem we might want to solve like 3-Coloring. We don't know any fast ways of solving that problem, but we can easily write a program V that given inputs $I = G$ and $X =$ an assignment of colors to the vertices, verifies whether X indeed satisfies our requirements (uses at most three colors and gives no two adjacent vertices the same color). Furthermore, this solution-verifier is linear time. So, if we had an algorithm to solve the ESP, we could feed in this V, feed in the graph G, feed in a bound b that is linear in the size of G, and solve the 3-Coloring problem. Similarly, we could do this for the Traveling Salesman Problem: program V, given inputs $I = (G, k)$ and $X =$ a description of a tour through G, just verifies that the tour indeed has length at most k and visits all the vertices. More generally, we can do this for any problem Q in **NP**. By definition of **NP**, YES-instances of Q must have short proofs that can be easily checked: i.e., they must have such a solution-verifier V that we can plug into our magic ESP algorithm.

Thus, we have shown that ESP satisfies both conditions (a) and (b) and therefore is **NP-complete**.

[1] We use "V" for the program because we will think of it as a solution-verifier.

[2] Thanks to Manuel Blum for suggesting the acronym.

6 Search versus Decision

Technically, a polynomial-time algorithm for the ESP just tells us if a solution exists, but doesn't actually produce it. How could we use an algorithm that just answers the YES/NO question of ESP to actually find a solution X? If we can do this, then we can use it to actually *find* the coloring or *find* the tour, not just smugly tell us that there is one. The problem of actually finding a solution is often called the *search* version of the problem, as opposed to the *decision* version that just asks whether or not the solution exists. That is, we are asking: can we reduce the search version of the ESP to the decision version?

It turns out that in fact we can, by essentially performing binary search. In particular, once we know that a solution X exists, we want to ask: "how about a solution whose first bit is 0?" If, say, the answer to that is YES, then we will ask: "how about a solution whose first two bits are 00?" If, say, the answer to that is NO (so there must exist a solution whose first two bits are 01) we will then ask: "how about a solution whose first three bits are 010?" And so on. The key point is that we can do this using a black-box algorithm for the decision version of ESP as follows. Given a string of bits S, we define a new program $V_S(I, X) = V(I, X_S)$ where X_S is the string X whose first $|S|$ bits are replaced by S. We then feed our magic ESP algorithm the program V_S instead of V. This way, using at most b calls to the decision algorithm, we can solve the search problem too.

So, if we had a polynomial-time algorithm for the decision version of ESP, we immediately get a polynomial-time algorithm for the search version of ESP. So, we can *find* the tour or coloring or whatever.

The ESP seems pretty stylized. But we can now show that other simpler-looking problems have the property that if you could solve them in polynomial-time, then you could solve the ESP in polynomial time as well, so they too are **NP**-complete. That is, they are so expressive that if we *could* solve them in polynomial-time, then it would mean that for any problem where we could *check* a proposed solution efficiently, we could also *find* such a solution efficiently. Now, onto formal definitions.

7 Formal definitions: P, NP, and NP-Completeness

We will formally be considering decision problems: problems whose answer is YES or NO. E.g., "Does the given network have a flow of value at least k?" or "Does the given graph have a 3-coloring?" For such problems, we can split all possible instances into two categories: YES-instances (whose correct answer is YES) and NO-instances (whose correct answer is NO). We can also put any ill-formed instances into the NO category. We now define the complexity classes **P** and **NP**.

Definition 5 **P** *is the set of decision problems solvable in polynomial time.*

E.g., the decision version of the network flow problem: "Given a network G and a flow value k, does there exist a flow $\geq k$?" belongs to **P**.

Definition 6 **NP** *is the set of decision problems that have polynomial-time* verifiers. *Specificially, problem Q is in* **NP** *if there is a polynomial-time algorithm $V(I, X)$ such that:*

- *If I is a YES-instance, then there exists X such that $V(I, X) =$ YES.*
- *If I is a NO-instance, then for all X, $V(I, X) =$ NO.*

Furthermore, X should have length polynomial in size of I (since we are really only giving V time polynomial in the size of the instance, not the combined size of the instance and solution).

The second input X to the verifier V is often called a *witness*. E.g., for 3-coloring, the witness that an answer is YES is the coloring. For factoring, the witness that N has a factor between 2 and k is a factor. For the Traveling Salesman Problem: "Given a weighted graph G and an integer k, does G have a tour that visits all the vertices and has total length at most k?" the witness is the tour. All these problems belong to **NP**. Of course, any problem in **P** is also in **NP**, since V could just ignore X and directly solve I. So, $\mathbf{P} \subseteq \mathbf{NP}$.

A huge open question in complexity theory is whether $\mathbf{P} = \mathbf{NP}$. It would be quite strange if they were equal since that would mean that any problem for which a solution can be easily *verified* also has the property that a solution can be easily *found*. So most people believe $\mathbf{P} \neq \mathbf{NP}$. But, it's very hard to prove that a fast algorithm for something does *not* exist. So, it's still an open problem.

Definition 7 *Problem Q is* **NP***-complete if:*

1. *Q is in* **NP***, and*
2. *For any other problem Q' in* **NP***, $Q' \leq_p Q$.*

So if Q is **NP**-complete and you could solve Q in polynomial time, you could solve *any* problem in **NP** in polynomial time. If Q just satisfies part (2) of the definition, then it's called **NP**-hard.

As we showed above, the ESP is **NP**-complete: it belongs to **NP** (that was the reason for including the bound b in unary, so that running the verifier for b steps counts as being polynomial-time), and we saw that we could use a polynomial-time algorithm for the ESP to solve any other problem in **NP** in polynomial-time.