IE 511: Integer Programming, Spring 2021 **11 Mar, 2021**

Lecture 14: Efficiency of algorithms

Lecturer: Karthik Chandrasekaran *Scribe: Karthik*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications.*

Until now, we have been working under the assumption that linear programs are solvable *efficiently* and hence, we have been trying to reduce integer programs to linear programs. However, some of the LP relaxations (with integral optimum solutions) that we saw in the process have exponentially many constraints, so it is not immediately clear how to design efficient algorithms for such linear programs. So, going forward, we will aim to design *efficient* algorithms for such integer programs directly. But, what is an *efficient algorithm*?

In the next couple of lectures, we will formalize the notion of *efficient* algorithms. We will set up some notation, understand the notion of efficient algorithms, define and understand the complexity classes P and NP, and ultimately, conclude that IP is a *difficult* problem (i.e., an NP-complete problem) with no known efficient algorithms.

14.1 Problem vs Problem Instances

We begin by emphasizing the distinction between problems and problem instances. We will be interested in understanding problems as opposed to problem instances.

Examples:

(1) IP problem: $\max\{c^T x : Ax \leq b, x \in \mathbb{Z}^n\}$.
    Instance of IP: $\max\{3x_1 + 5x_2 : 4x_1 + 2x_2 \leq 3, 3x_1 - 7x_2 \leq -5, x \in \mathbb{Z}^2\}$.

(2) Max weight matching problem: $\max\{\sum_{e \in M} w_e : M \text{ is a matching in } G\}$.
    Instance of max weight matching: particular graph shown in Figure 14.1.

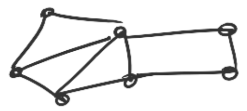


Figure 14.1: A graph with all edge weights being one.

14.2 Decision Problems

Recall that we are interested in optimization problems of the following form.

$$\text{Optimization Problem: } \max\{c^T x : x \in \chi\}.$$

We will rephrase optimization problems as decision problems:

$$\text{Decision Problem: Does there exist } x \in \chi \text{ with } c^T x > k?$$

For instance, in Max weight matching, instead of asking for the weight of the max weight matching, the decision problem asks whether there is a matching whose weight is at least k.

Why study decision problems as opposed to optimization problems? Note that the output of a decision problem is a single bit while that of the optimization problem could need several bits to express. Restricting the output domain to a single bit will allow us to focus on computation time without having to worry about the time needed to represent the output itself.

14.3 Input Size

Computation time is related to the size of the problem instance. For instance, the size of an IP instance depends on (1) the number of variables, (2) the number of constraints, and (3) the magnitude of the entries in the constraint matrix A, the RHS vector b, and the objective vector c. Similarly, the size of an optimization problem instance defined over graphs (e.g., max weight matching or max weight forest) depends on (1) the number of nodes, (2) the number of edges, and (3) the magnitude of the weights.

We emphasize that the size of an input instance also depends on the magnitude of the numerical data in the instance. We frequently assume that simple operations like addition and subtraction can be done in constant time but it is unreasonable to assume that huge numbers (like factorials of two numbers) can be added in constant time. So, we take the magnitude of the numerical data into account while determining the size of the input.

Definition 1. For a problem instance P, the *length of the input*, denoted $L(P)$, is the length of the binary representation of a "standard representation" of the instance.

"Standard representation" is what is considered reasonable. For instance,

- Graphs should be represented by adjacency matrices or edge lists.
- Weights on the edges of a graph should be represented in binary and not in unary.
- Rational numbers p/q should be represented as (binary (p), binary (q)) such that $\gcd(p,q) = 1$.

Examples:

(1) IP (Decision Version):

Given:	A, b, c, k
Goal:	Does there exist $x \in \mathbb{Z}^n : c^T x \geq k, Ax \leq b$?
Size of input:	size(A)+size(b)+size(c)+ $\log k$

(2) Max cardinality matching (Decision Version):

Given:	Graph G, Integer k
Goal:	Does exist a matching of size k?
Input:	Representation of G and k
Size of Input:	$2\|E\| \log \|V\| + \log k$

14.4 Computation Time

We will measure the computation time to solve a problem as a function of the size of the input instance.

Definition 2. Given a problem π, an algorithm A, an instance P of π, let $r_A(P)$ be the number of elementary calculations required to run the algorithm A on instance P. The *running time* of algorithm A, denoted $r_A^*(n)$, is the supremum of the running time over all input instances of size n, i.e., $r_A^*(n) := \sup_P\{r_A(P) : L(P) = n\}$.

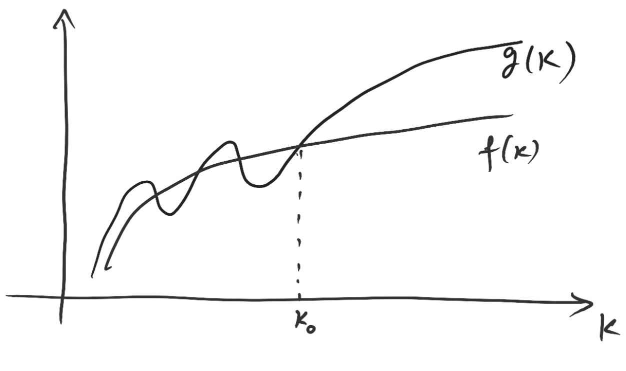
We associate the running time with all instances of size n and take the maximum. This approach is known as *worst-case analysis*. Worst-case analysis has its advantages:

1. It gives an absolute guarantee on the runtime that depends only on the size of the input instance.
2. The guarantee is independent of other features of the input instance.

In typical applications, we only care about the rate of growth of the run-time as a function of the input size. So we use the following notation.

Definition 3 (Big-O notation)**.** For functions $g, f : \mathbb{R}_+ \to \mathbb{R}$, we say that $f(k) = O(g(k))$ if there exists a positive constant c and a positive integer k_0 such that

$$f(k) \le cg(k) \ \forall k \ge k_0.$$



Example:

1. Say $f(x) = 5x^3$. Then $f(x) = O(x^5)$. Also, $f(x) = O(x^3)$.
2. Suppose $f(x) = c_d x^d + c_{d-1}x^{d-1} + \cdots + c_1 x_1 + c_0$. Then $f(x) = O(x^d)$.

The advantage of the $O(\cdot)$ notation is that it reveals the behavior of the function as $k \to \infty$. This is enough information for the purpose of run-time since we care about run-time for large inputs, i.e., when n is large (large graphs, large matrices, big data).

With these notations, we now define polynomial-time algorithms.

Definition 4. An algorithm A for problem π is a *polynomial-time algorithm* if $r^*_A(n) = O(n^d)$ for some fixed positive integer d.

Polynomial-time algorithms are deemed to be fast algorithms or efficient algorithms. Problems which have polynomial-time algorithms are problems that can be solved efficiently. We have a name for the family of such problems.

Definition 5. Let $\mathbb{P}$ be the family of decision problems that can be solved in polynomial time, i.e., problem π is in $\mathbb{P}$ iff there exists a polynomial time algorithm for solving π.

Example:

1) **Shortest path problem.** Dijkstra's algorithm can be implemented to run in polynomial-time.

2) **Solving linear equations.**

Given:	$A \in \mathbb{R}^{m\times n}, b \in \mathbb{R}^m, A$ is non-singular
Goal:	Does there exist $x \in \mathbb{R}^n$ such that $Ax = b$?
Algorithm:	Gaussian Elimination

- Gaussian Elimination runs in $O(n^3)$ calculations.
- Note that Gaussian Elimination involves multiplication and division operations, so it could potentially end up with numbers that are exponential in the magnitude of the inputs. We need to ensure that the implementation does not end up with such large intermediate numbers as it would be unreasonable to expect the addition of factorials to be done using an elementary operation. Indeed, there exists an implementation of Gaussian Elimination that ensures that the magnitude of the intermediate numbers are small. The implementation ensures that the magnitude of the intermediate numbers is at most the maximum determinant of square submatrices of $[A \;\; b]$. Note that the maximum determinant of square submatrices of $[A \;\; b]$ is at most

$$n!\left(\max_{i,j}\{A_{ij}, b_i\}\right)^n \le \left(n\max_{i,j}\{A_{ij}, b_i\}\right)^n.$$

  Therefore, the representation of these intermediate numbers uses at most

$$n\log n + n\log\left(\max_{i,j}\{A_{ij}, b_i\}\right)$$

  bits which is polynomial in the size of the input.

  The above reasoning also implies that we can verify if $\text{rank}(A) = k$ for a given matrix A and an integer k in polynomial-time. We can also verify if a given matrix A is invertible in polynomial-time.

The family of decision problems which have polynomial time algorithms also have polynomial-sized *certificate* for YES and NO instances. If a decision problem has polynomial-sized *certificate* for a YES instance, then it is said to be in NP.

Definition 6. NP is the family of decision problems with the property that an instance for which the answer is yes, there is a *polynomial-sized certificate* of YES.

- By certificate, we are referring to a proof certifying that the answer is YES.
- By polynomial-sized certificate, we mean that there exists a constant d such that the size of the proof for input instances of length n is $O(n^d)$.

Note: $\mathbb{P} \subseteq NP$.

Examples of problems in NP.

1) **Max Cardinality Matching.**

   Given: Graph $G = (V, E), k$
   Goal: Does there exist a matching in G of size $\geq k$?

   If a graph G has a matching M of size at least k, then M is a certificate and the size of M is at most $|E|$ which is polynomial in the input size.

2) **Linear Programs.**

   Given: $A \in \mathbb{Z}^{m\times n}, b \in \mathbb{Z}^m, c \in \mathbb{Z}^n, k$
   Goal: Does there exist $x : Ax \leq b, c^T x \geq k$?

   Does it have a polynomial-sized certificate of YES instances?

   Suppose x satisfies the linear system, then x could be certificate for YES instances provided that the magnitudes of x can be represented using at most polynomial(input-size) bits.

   Exercise: Prove that such an x exists using the characterization of extreme points that we learnt before.

In fact, both problems above are also in $\mathbb{P}$.

14.5 Reductions

Next we will understand the difficulty of problems relative to each other. Is one problem more difficult (i.e., needs more run-time) than another? For this, we define the notion of polynomial-time reductions.

Definition 7. A polynomial-time reduction from a problem π to a problem π' is a *polynomial-time computable* function f : instances(π) $\rightarrow$ instances(π') such that S is a YES instance of problem π iff $f(S)$ is a YES instance of problem π'.

Note that f is a mapping from instances of π to instances of π'.

Notation: π is *polynomial-time reducible* to π' if there exists a polynomial-time reduction from π to π'.

One of the advantages of polynomial-time reductions is the following: if we have a polynomial-time reduction from π to π' and a polynomial-time algorithm to solve π', then we have a polynomial-time algorithm to solve π.

Example: We will see an example of polynomial-time reductions to get familiar with this notion.

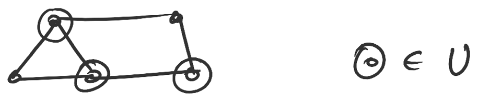
π: Min Cardinality Vertex Cover

Given: A graph $G = (V, E), k \in \mathbb{Z}_+$
Goal: Does there exist a vertex cover in G of size $\leq k$

Recap

Definition. $U \subseteq V$ is a *vertex cover* in G if every edge in G has at least one end-vertex in U.

Example:

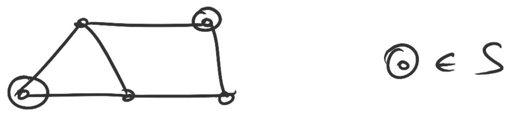


π': Max Cardinality Stable set

Given: A graph $G = (V, E), r \in \mathbb{Z}_+$
Goal: Does there exist a stable set in G of size $\geq r$?

Definition 8. A set $S \subseteq V$ is a *stable set* in G if no edge of G has both its end vertices in S.

Example:



We will show the following reduction in the next lecture.

Proposition 9. *There exists a polynomial-time reduction from π to π' and vice-versa.*