

1 Feasible Computation

So far, we have been looking at designing algorithms that are as efficient as possible (preferably linear in the size of the input, or within a constant factor of a lower bound for the computation time). We will work with a more relaxed notion of efficiency where any algorithm whose running time is *polynomial* in the size of the input is an efficient algorithm or a *feasible computation*.

Some of the reasons for equating polynomial time with feasible computation are the following.

- Growth rate of a polynomial does not have the explosive growth rate of an exponential function.
- Although some polynomials, e.g., n^{100} do not represent a running time that is in any way feasible, *in practice*, virtually every natural problem for which a polynomial-time algorithm has been developed has a running time that is a small power of the input size.
- Polynomials have nice closure properties – the addition or multiplication of two polynomials is a polynomial, as is functional composition.
- Polynomial time *remains invariant* across machine models, e.g., any computation running in polynomial time on a RAM (the underlying machine model for our pseudocodes) will also run in polynomial time on the Turing machine.

1.1 Decision Problems

A *decision problem* is a problem with yes/no answers. Hence in a decision problem, we can equivalently talk of the *language* associated with the decision problem, namely, the set of inputs for which the answer is yes.

Typically, we assume that the input is coded in binary, so the set of all possible inputs is $\{0, 1\}^*$ and the *language associated with a decision problem* Q is

$$L(Q) = \{x \in \{0, 1\}^* \mid \text{the answer is yes for problem } Q \text{ on input } x\}$$

2 The Classes \mathbf{P} and \mathbf{NP}

We define \mathbf{P} as the class of decision problems that are solvable by algorithms that run in time polynomial in the length of the input.

We will next define the class \mathbf{NP} , or *Nondeterministic Polynomial Time*. Before we can define this class, we need some definitions.

Verification Algorithm.

A *verification algorithm* is an algorithm A , that takes two inputs: an ordinary input x (coded in binary), and a *certificate* y , and outputs a 1 on certain combinations of x and y .

Verification algorithm A verifies an input string x if *there exists a certificate* y such that $A(x, y) = 1$.

The language verified by verification algorithm A is

$$L = \{\text{input strings } x \mid \text{there exists certificate string } y \text{ such that } A(x, y) = 1\}$$

Note that for an x that is not in L , *for every certificate* y , $A(x, y) \neq 1$.

Polynomial-time Verification Algorithm.

A verification algorithm A for a language L is a *polynomial-time verification algorithm* for L if

- For each $x \in L$, there is a certificate y of size polynomial in the size of x such that $A(x, y) = 1$, and $A(x, y)$ returns 1 in time polynomial in x .
- Since A is a verification algorithm for L , for every x not in L there is no certificate y for which $A(x, y) = 1$.

The Class \mathbf{NP}

A language $L \in \mathbf{NP}$ if and only if there exists a polynomial-time verification algorithm A for L .

Note that $\mathbf{P} \subseteq \mathbf{NP}$.

At this time we do not know if $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \subset \mathbf{NP}$.

2.1 Examples of Languages in NP

See Chapter 34 in your textbook for the definitions of the following decision problems.

Here, $\langle x \rangle$ represents the binary encoding of x .

- HAM-CYCLE = $\{\langle G \rangle \mid G \text{ is a Hamiltonian graph}\}$
- CIRCUIT-SAT = $\{\langle C \rangle \mid C \text{ is a satisfiable Boolean circuit}\}$
- SAT = $\{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$
- CNF-SAT = $\{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula in CNF}\}$
- 3-CNF-SAT = $\{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula in 3-CNF}\}$
- CLIQUE = $\{\langle G, k \rangle \mid G \text{ is an undirected graph with a clique of size } k\}$
- IS = $\{\langle G, k \rangle \mid G \text{ is an undirected graph with an independent set of size } k\}$
- VERTEX-COVER = $\{\langle G, k \rangle \mid \text{undirected graph } G \text{ has a vertex cover of size } k\}$
- TSP = $\{\langle G, c, k \rangle \mid G = (V, E) \text{ is a complete graph, } c : V \times V \rightarrow Z \text{ is a cost function, } k \in Z, \text{ and } G \text{ has a traveling salesman tour with cost at most } k\}$
- SUBSET-SUM = $\{\langle S, t \rangle \mid \text{there is a subset } S' \subseteq S \text{ such that } t = \sum_{s \in S'} s\}$

It is quite straightforward to come up with a polynomial-time verification algorithm for each of these problems, i.e., it is quite easy to show that each of these problems is in **NP**.

3 NP-completeness

3.1 Polynomial-time reductions

Let L_1 and L_2 be languages that are subsets of $\{0, 1\}^*$.

We say that L_1 is *polynomial-time reducible* to L_2 if there exists a function f

$$f : \{0, 1\}^* \rightarrow \{0, 1\}^*$$

such that for each $x \in \{0, 1\}^*$

- $x \in L_1$ if and only if $f(x) \in L_2$, and $f(x)$ is computable by an algorithm A_f that runs in time polynomial in $|x|$, where $|x|$ is the size of x .

We say that f is the *reduction function* and A_f is the *reduction algorithm* for the polynomial-time reduction of L_1 to L_2 .

We use the notation

$$L_1 \leq_P L_2$$

to denote that L_1 is polynomial-time reducible to L_2 .

Lemma 3.1 *Let $L_1 \leq_P L_2$. Then, $L_2 \in \mathbf{P}$ implies $L_1 \in \mathbf{P}$.*

3.2 NP-completeness definition

Definition. A language $L \subseteq \{0, 1\}^*$ is **NP-complete** if it satisfies the following two properties.

- $L \in \mathbf{NP}$; and
- for every $L' \in \mathbf{NP}$, $L' \leq_P L$.

We use the notation $L \in \mathbf{NPC}$ to denote that L is **NP-complete**.

Theorem 3.1 $\mathbf{P} = \mathbf{NP}$ if and only if an **NP-complete** problem is solvable in polynomial time.

4 Cook's Theorem

Theorem 4.1 (*Cook's Theorem.*) **SAT** is **NP-complete**.

5 NP-completeness proofs

By changing the proof of Cook's theorem slightly, we can also show that CNF-SAT and 3-CNF-SAT are **NP-complete**, which gives us the following theorem.

Theorem 5.1 SAT, CNF-SAT and 3-CNF-SAT are **NP-complete**.

The next theorem allows us to show additional problems in **NP** to be **NP-complete** without having to go through a proof similar to Cook's theorem.

Theorem 5.2 Let $L, L' \in \text{NP}$. If

- L' is **NP-complete**,
- and
- $L' \leq_P L$,

then L is **NP-complete**.

We now apply Theorems 5.1 and 5.2 to show some **NP-completeness** results.

Theorem 5.3 CLIQUE is **NP-complete**.

Proof: We already know that CLIQUE is in **NP**.

By Theorem 5.2 it suffices to find a *known NP-complete* language L for which we can show $L \leq_P \text{CLIQUE}$. We will use 3-CNF-SAT for L .

Let Φ be an instance of 3-CNF-SAT, and let

$$\Phi = C_1 \wedge \cdots \wedge C_k \quad \text{where } C_r \text{ is given by } C_r = l_1^r \vee l_2^r \vee l_3^r, \quad 1 \leq r \leq k.$$

Here each l_j^i is a *literal* which is an occurrence of a variable or its negation.

Given Φ , we construct an input to CLIQUE, which consists of an undirected graph $G = (V, E)$ and an integer k as follows:

- There is a vertex v_i^r in V for each literal l_i^r in a clause in Φ .
- There is an edge (v_i^r, v_j^s) in E if and only if
 - (i) $r \neq s$, and
 - (ii) $l_i^r \neq \bar{l}_j^s$.

The input to CLIQUE is $\langle G = (V, E), k \rangle$, and clearly this can be constructed in time polynomial in the size of Φ .

To complete the proof we show that Φ is satisfiable if and only if G has a clique of size k . \square

Theorem 5.4 IS *is* NP-complete.

Theorem 5.5 VERTEX-COVER *is* NP-complete.

Proof: We know already that VERTEX-COVER is in NP.

We show that IS \leq_P VERTEX-COVER to establish the Theorem.

Given an instance $\langle G = (V, E), k \rangle$ for IS we construct an instance for VERTEX-COVER as $\langle G = (V, E), |V| - k \rangle$, which can be clearly constructed in polynomial time. We then show that this represents a reduction from CLIQUE to VERTEX-COVER. \square

6 Approximation Algorithms

For most **NP-complete** problems, we usually wish to solve the optimization version of the problem, not the decision version, e.g, *maximum* clique, *minimum* vertex cover, *minimum-cost* traveling salesman tour, etc. Clearly we will not be able to solve the optimization version of **NP-complete** problems in polynomial time unless **P=NP**. So, in the meantime we look for *approximation algorithms*. We will refer to an optimization problem as being **NP-hard** if there is an underlying **NP-complete** decision problem that can be trivially solved by solving this optimization problem.

An algorithm A provides an *approximation ratio* $\rho(n)$ for an optimization problem Q if, for every input of size n , A returns a feasible solution of cost C , which is within a factor $\rho(n)$ of the cost C^* of an optimal solution for that input. In other words,

$$\frac{C}{C^*} \leq \rho(n) \quad \text{for a minimization problem,} \quad \frac{C^*}{C} \leq \rho(n) \quad \text{for a maximization problem}$$

We say such an algorithm is a $\rho(n)$ *approximation algorithm* for optimization problem Q . If the approximation ratio does not depend on the input size n we will denote it as simply ρ .

We will now study some *polynomial-time* approximation algorithms with small approximation ratio for some **NP-hard** optimization problems.

7 Vertex Cover

A natural heuristic for constructing a small vertex cover in an undirected graph $G = (V, E)$ is to repeatedly pick a vertex v of maximum degree in the current graph, add it to the vertex cover, and delete it and all edges incident on it from the graph, until no edge remains in the graph (*the natural greedy strategy*). It can be shown that this strategy gives an $O(\log n)$ -approximation ratio for the Minimum Vertex Cover problem.

We now study another, less intuitive, but equally simple greedy strategy that gives approximation ratio 2.

Approx-VC($G = (V, E)$)

Input. Undirected graph G .

Output. A set $C \subseteq V(G)$ that is a vertex cover of G , and contains no more than twice the number of vertices in a vertex cover of minimum size.

1. *Initialize.* $C := \Phi$; $H(V, E) = G(V, E)$
 2. **repeat**
 - pick an edge $(u, v) \in E$
 - $C := C \cup \{u, v\}$
 - Remove u and v from V and all edges incident on u and v from E
- until** H has no edge

Theorem 7.1 *APPROX-VC is a polynomial-time approximation algorithm for the minimum vertex cover problem with approximation ratio 2.*

Note. Although we saw a simple reduction $\text{CLIQUE} \leq_P \text{VERTEX-COVER}$, that reduction does not imply a polynomial-time approximation algorithm with $\rho = 2$ for CLIQUE .

(In fact, it can be proved that there is no polynomial-time approximation algorithm for CLIQUE with $\rho = n^r$ for an $r > 0$ unless $\mathbf{P}=\mathbf{NP}$.)

8 Δ -TSP

Δ -TSP is the Traveling Salesman Problem in which the costs on edges satisfy the triangle inequality, i.e., for any u, v, w ,

$$c(u, w) \leq c(u, v) + c(v, w)$$

Consider the following approximation algorithm for Δ -TSP.

Approx- Δ -TSP($G = (V, E), c$)

Input. Undirected complete graph $G = (V, E)$ with an integer-valued weight function c on the edges, which satisfy the triangle inequality.

Output. A permutation of the vertices that specifies the sequence of vertices on a TSP tour (i.e., a Hamiltonian cycle in G) whose cost is no more than twice the cost of an optimal TSP tour for G .

1. Find an MST T in G and root it at a vertex $r \in V$
2. Perform a depth-first search of T rooted at r
3. List vertices in order of their preorder numbering as the TSP tour order.

Theorem 8.1 *Algorithm **Approx- Δ -TSP** is a polynomial-time approximation algorithm with approximation ratio 2 for the problem of finding a minimum-cost TSP tour in a graph in which the edge weights satisfy the triangle inequality.*

Proof:

Let W be the cost of the tour returned by **Approx- Δ -TSP**, and let OPT be the cost of an optimal TSP tour. If $c(T)$ the the cost of the MST T , then $W = 2 \cdot c(T)$. But,

$$c(T) \leq OPT. \quad \text{Hence,} \quad W \leq 2 \cdot OPT.$$

□

9 General TSP

Theorem 9.1 *If $P \neq NP$, there is no polynomial-time algorithm with constant ρ that can find a minimum-cost TSP tour in a graph in which edge weights are arbitrary.*

Proof: Let A be a polynomial-time approximation algorithm for minimum-cost TSP with approximation ratio ρ . We claim that A can be used to solve **HAM-CYCLE** in polynomial time as follows: Given an input $G = (V, E)$ to **HAM-CYCLE**, we construct an input to algorithm A as $H = (V, E')$, where H is the complete graph on V , together with cost function c on E' with $c(e) = 1$ if $e \in E$ and $c(e) = \rho \cdot n + 1$ if e is not in E . Then graph G has a Hamiltonian cycle if and only if algorithm A returns a solution with cost at most $\rho \cdot n$. □