EE360C: Algorithms

NP-Completeness

Summer 2019

Department of Electrical and Computer Engineering
University of Texas at Austin

Introduction

Algorithm Complexity

Recap

Up until now, we've talked mainly about polynomial time algorithms. But not all problems can be solved in polynomial time.

Coming Up

But there's also a class of problems *for which we don't know*. We know we can verify that a solution is correct in polynomial time, but we don't know if we can solve them in polynomial time.

These are the "NP-complete" problems. This has led to the $P \neq NP$ question of computer science lore

- But the kicker is... if you can solve one of the NP-Complete problems in polynomial time, you can solve them all!

NP Complete Problems

OK. The NP-Complete problems are clearly hard, so they should *look* hard, right?

Shortest vs. Longest Simple Paths

We know how to solve the shortest path problem in polynomial time. Finding the *longest* simple path is much more difficult. In fact, just writing an algorithm to determine if a graph has a path of k edges is an NP-Complete problem.

Euler tour vs. Hamiltonian tour

A Euler tour contains every edge in a graph; a Hamiltonian tour contains every vertex in the graph. We can determine whether a graph has a Euler tour in $O(E)$ time; determining whether a graph has a Hamiltonian tour is NP-Complete.

The Classes P, NP, and NPC

The Class P

The class *P* contains problems that are solvable in polynomial time (specifically, in $O(n^k)$, for instance size n and some constant k).

The Class *NP*

The class *NP* contains problems whose solutions can be verified in polynomial time (e.g., given a purported 3-coloring of a graph, we can determine that it is or is not correct using DFS)

A problem is in NPC if it is in NP and is at least as hard as any other problem in NPC (it's unlikely that the problem could be solved in polynomial time   If, as an algorithm designer, you encounter a problem that is NP-complete, it's likely better to focus your attention on an approximation algorithm...

A Summary

- **P**: problems that can be solved in polynomial time.
- **NP**: problems that can be verified in polynomial time (*nondeterministic polynomial*). Clearly $P \subseteq$ NP.
- **NP-Complete**: decision problems (those whose answers are "yes" or "no") that can be verified in polynomial time, but for which no known polynomial time solution is known. NP-Complete $\subseteq$ NP.
- **NP-Hard**: problems that are at least as hard as the hardest problem in NP
  - A problem H is in NP-Hard iff there exists a problem $L \in$ NP-complete that is polynomial time reducible to H.
  - That is, L can be solved in polynomial time given an oracle that can solve H.

Showing a Problem to be NP-Complete

We're turning the tables here a little bit... In showing a problem is NP-complete, we're not trying to design an efficient implementation but to show that one is unlikely to exist

Decision Problems vs. Optimization Problems

We're going to focus on decision problems (with a yes/no answer) instead of optimization problems

- decision problems are closely related to optimization problems (e.g., given a graph G, vertices u, v, and integer k, does there exist a path of length $\leq k$ from u to v?)
- the optimization version is at least as hard as the decision version; if the decision problem is hard, it implies that the optimization problem is hard

Showing a Problem to be NP-Complete (cont.)

To demonstrate a problem is NP-Complete, the key ingredient is a *reduction* that shows that the new problem is at least as hard as a problem known to be NP-complete.

- consider an instance of a decision problem *A* and another decision problem *B* for which we have a polynomial time algorithm
- suppose we have a procedure to transform any instance α of *A* into an instance β of *B* such that
  - the transformation takes polynomial time
  - the answer to β is yes if and only if the answer to α is yes
- this gives us a polynomial time algorithm for solving *A*

Suppose there ends up not to be a polynomial time algorithm for *A*, and *A* polynomial-time reduces to *B*, then there is no polynomial time procedure for *B*.

Polynomial-time reducibility

Polynomial-time reducibility

Intuitively, a problem Y can be reduced to another problem X if any instance of Y can be easily rephrased as an instance of X, and the solution to X provides a solution to the instance of Y.

- consider the problem Y of solving linear equations (e.g., $ax + b = 0$). We can easily rephrase this as the problem X of solving quadratic equations (e.g., $0x^2 + ax + b = 0$), where if we solve the latter, we have a solution to the former.

Claim

Problem X is at least as hard as problem Y and write $Y \leq_P X$ (formally, Y is polynomial-time reducible to X) if arbitrary instances of problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to a black-box that solves problem X.

Corollary

If X, Y are problems such that $Y \leq_p X$, then $X \in P$ implies $Y \in P$.

Example of Polynomial-time Reducibility

Independent Set

An **independent set** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ of vertices with no edges between them. The **size** of an independent set is the number of vertices it includes.

The **independent set problem** is the optimization problem of finding an independent set of maximum size in a graph. As a decision problem, we ask if an independent set of a given size at least k exists in the graph.

Example of Polynomial-time Reducibility

Vertex Cover

A **vertex-cover** of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ or $v \in V'$ (or both). The **size** of a vertex cover is the number of vertices in it.

The vertex-cover problem is to find a vertex cover of minimum size in a given graph. The decision version of this problem is to determine whether a graph has a vertex cover of a given size at most k.

Example of Polynomial-time Reducibility

Let $G = (V, E)$ be a graph. Then S is an independent set if and only if it's complement $V - S$ is a vertex cover

Proof

Suppose S is an independent set, consider any edge $e = (u, v)$. Since, S is independent both u and v cannot be in S; one of them must be in $V - S$. It follows that every edge has at least one end in $V - S$, and so $V - S$ is a vertex cover.

Conversely, suppose $V - S$ is a vertex cover. Consider any two nodes u and v in S. If there were joined by an edge e, then neither end of e would lie in $V - S$, contradicting our assumption that $V - S$ is a vertex cover. It follows that no two nodes in S are joined by an edge, so S is an independent set.

Example of Polynomial-time Reducibility

Independent Set $\leq_P$ Vertex Set

Proof: If we have a black box to solve Vertex Set, then we can decide whether G has an independent set of size at least k by asking the black box whether G has a vertex cover of size at most $n - k$.

Vertex Set $\leq_P$ Independent Set

Proof: If we have a black box to solve Independent Set, then we can decide whether G has an vertex cover of size at least k by asking the black box whether G has a independent set of size at most $n - k$.

Showing a Problem to be NP-Complete (cont.)

To show a problem is NP-Complete, we'll reduce a known NP-Complete problem to it.

- We also have to show that it's in NP, but that's usually pretty easy.

We're also going to need a "first" NP-complete problem to start the reductions. We'll see that in a little bit...

Polynomial-time verification (NP)

Showing Membership in NP

What did it mean to be a member of NP? That a certificate to a *true instance* (i.e., a “yes” instance) of a problem can be verified in polynomial time.

- For example, consider the problem of determining whether a graph has a path of length $\leq k$.
- Given a purported solution path p, we can easily “walk” p and count its edges in polynomial time.

Hamiltonian Cycle

A **Hamiltonian cycle** of an undirected graph is a simple cycle that contains every vertex in V.

- What’s the certificate for the Hamiltonian cycle?
- just the cycle itself!
- it’s easy to efficiently verify that it touches all vertices...

NP or not NP?

Problems in NP

- anything in P (e.g., does there exist a subsequence of size k for sequences X and Y?)
- does there exist a 3-coloring of graph G?
- is there a partition of a set of integers S into two subsets S_0 and S_1 such that the sum of elements in S_0 is equal to the sum of elements in S_1?

Problems Unlikely to be in NP

- counting problems
- problems that involve $\forall x . \exists y$
- provably intractable/undecidable problems

Nothing is known about the relationships between P, NP, co-NP, NP-complete, etc.

Questions