Recitation 12 - NP-Completeness

Sebastian Laudenschlager

sebastian.laudenschlager@colorado.edu

April 20, 2018

Introduction

- Most algorithms we've looked at thus far has had a polynomial running time.
  - Input size $n \Rightarrow \mathcal{O}(n^k)$ for some constant k.
- Not all problems can be solved in polynomial time, however.
- Some problems can't be solved at all, regardless of how much time we're given.
- NP-complete problems: status unknown.
  - No polynomial time algorithm has been found for an NP-complete problem.
  - No one has proved that no polynomial algorithm can exist for any NP-complete problem.
- This is the famous $P = NP$ question.

Slippery slope into NP-Complete land

- Some NP-complete problems are only slight variations on problems we've seen before:
  - SSSP: no problem. Longest path is NP-complete.
  - Euler tour of a graph is a cycle that traverses each edge exactly once. This can be solved in $\mathcal{O}(E)$ time. A Hamiltonian cycle of a graph is a cycle containing each vertex. The problem of finding a Hamiltonian cycle is NP-complete.

Complexity classes

- The class P consists of problems solvable in polynomial time.
  - $\Rightarrow \mathcal{O}(n^k)$ for constant k.
- The class NP consists of problems that are *verifiable* in polynomial time.
  - Given a "certificate" of a solution, we can verify its correctness in polynomial time (as a function of problem size).
- Note that any problem in P is also in NP, since we can generate the solution of a problem in P in polynomial time without a certificate.
- The NP-complete (NPC) class refers to any problem that is as hard as any problem in NP.
  - If *any* NPC problem can be solved in polynomial time, then *all* NPC problems can be solved in polynomial time.

Complexity classes, cont.

- It is conjectured that $P \neq NP$, simply given the large number of NP-complete problems in existence. It would be remarkable if it turned out that *all* of them could be solved in polynomial time.
- From a practical perspective, why would you care about complexity classes?
  - If you could show that the problem you are trying to solve is NP-complete, you could then spend your time developing an efficient approximation algorithm instead of trying to solve the problem exactly.

Decision vs. Optimization

- NP-completeness applies to decision problems, not optimization problems.
  - Decision problems are problems to which we get either 1 or 0 (yes or no).
  - Optimization problems involve finding the best solution out of many feasible solutions, e.g. SSSP.
- So how can we apply NP-completeness to optimization problems?
- $\rightarrow$ cast an optimization problem as a decision problem.
  - Impose a bound on the value to be optimized.
  - For example, in shortest path we could ask: Given a directed graph G, source vertex s and destination vertex t, and an integer k, does a path exist from s to t consisting of at most k edges?
  - Solve shortest path, count edges in shortest path and compare to decision parameter k.
- If an optimization problem is easy, its related decision problem is easy as well.

Reductions

- We want to show that two problems are equally hard (even when both are decision problems).
- Let A be a decision problem we wish to solve (in polynomial time).
- Assume we already know how to solve some other decision problem B in polynomial time, and that we can transform any instance α of A into some instance β of B such that
  - The transformation itself takes polynomial time.
  - The answers to the two problems are the same, i.e. the answer for α is 1 iff answer to β is 1.
- This is called a reduction algorithm.
- Given a reduction algorithm, we can solve A:
  - Transform an instance α in A to an instance β in B.
  - Run the polynomial decision algorithm for B on β.
  - Use the answer for β as the answer for α.

Reductions, cont.

- Goal: Use polynomial-time reductions to show that no polynomial-time algorithm can exist for a particular problem B.
- Suppose we have a decision problem A for which we know no polynomial-time algorithm can exist.
- Also suppose that we have a polynomial-time reduction to transform an instance in A to an instance in B.
- For a contradiction, assume that B has a polynomial-time algorithm.
- Then we could use a reduction to conclude that A has a polynomial-time algorithm, which contradicts our assumption.
- Proof methodology: Can't assume no polynomial algorithm exists for A (for NP-complete). Instead, prove B is NP-complete on the assumption that A is also NP-complete.

Boolean Satisfiability

- Our methodology for proving that a problem is NP-complete requires a reduction to a problem we *know* is NP-complete.
- But at this point we don't know any such algorithms.
- The problem we choose to use is that of Boolean Satisfiability, i.e. the problem of deciding whether a string of boolean variables, chained together with *AND*, *OR*, and *NOT* operations, can return 1 (true).
- Cook-Levin Theorem states that Boolean Satisfiability is NP-complete.

Subset-sum problem

- Given a finite set S of positive integers and a target $t > 0$, does there exist a subset $S' \subseteq S$ whose elements sum to t?
- SUBSET-SUM $= \{(S, t) : \exists$ subset $S' \subseteq S$ such that $\sum_{s \in S'} s = t\}$.

Subset-sum is NP-complete

- **Proof:**
  - First show that SUBSET-SUM $\in NP$.
  - Easy to check whether a finite set of integers add up to t.
    $\rightarrow$ polynomial time verification.
  - To show that SUBSET-SUM is NP-complete, we reduce an instance of 3-CNF-SAT to one of SUBSET-SUM.
  - Given a 3-CNF formula $\phi = x_1, x_2, \ldots, x_n$ with k clauses $C_1, C_2, \ldots, C_k$ (each of which contains three literals), we aim to construct an instance (S, t) of SUBSET-SUM.
  - Want to show that ϕ is satisfiable iff there exists a subset of S whose sum is exactly t.
  - Let's assume (WLOG) that no clause contains both a variable and its negation, for that automatically satisfies the clause.
  - Also assume that each variable appears in some clause, otherwise it wouldn't matter what value we assigned to it.

Subset-sum proof, cont.

- Construct set S and target t as follows:
  - Create two numbers in S for each variable x_i and two numbers in S for each clause C_j.
  - Each number has $n + k$ digits.
  - The target t has a 1 in each variable digit and 4 in each clause digit.
  - For each variable x_i, S contains two integers v_i and v_i'. Each v_i and v_i' has a 1 in the variable digit x_i and a 0 in every other variable digit. In addition, if x_i appears in clause C_j, then the corresponding clause digit gets assigned a 1.
  - Furthermore, each clause C_j contains two integers s_j and s_j'. Each s_j and s_j' has a 0 in the variable digits. s_j gets a 1 in the C_j digit and s_j' gets a 2 in the C_j digit.

Subset-sum proof, cont.

- Note: The greatest sum of any of the digit positions is 6. Why?
- Thus, if we use, e.g. base 10 to interpret these numbers, no carry can occur from lower to higher digits.
- We can perform this reduction in polynomial time, i.e. each of the $n + k$ digits can be produced in constant time, and there are $2n + 2k$ such numbers, each of which has $n + k$ digits.
- Now we just need to show that the 3-CNF-SAT FORMULA ϕ is satisifiable iff there exists a subset $S' \subseteq S$ whose sum is t.

Subset-sum proof, cont.

- $\Rightarrow$: Suppose that ϕ has a satisfying assigment.
- For $i = 1, \ldots, n$, if $x_i = 1$ in this assignment, then include v_i in S', otherwise include v_i'.
- So we only include the v_i and v_i' numbers that correspond to literals with value 1.
- Since we include only one of v_i or v_i' and since we labeled the variable digits for s_j and s_j' with 0, the sum of any variable digit in S' must be 1 (which matches that of the target t).
- Because we have a satisfying assignment, each clause must contain a 1, meaning each clause digit has at least one 1 (could be 1, 2, or 3) contributed to its sum by v_i or v_i' in S'.
- To get the clause digit sum to add to 4, we include in S' the appropriate slack values.

Subset-sum proof, cont.

- $\Leftarrow$: Suppose that there is a subset $S' \subseteq S$ that sums to t.
- S' must include one of v_i or v_i' for each $i = 1, \ldots, n$.
- If $v_i \in S'$, set $x_i = 1$, otherwise if $v_i' \in S'$, set $x_i = 0$.
- To get any clause digit C_j to sum to 4, S' must include one v_i or v_i' that has value 1 in C_j, since the slack variables account for a value of at most 3.
- If S' includes a v_i that has a 1 in position C_j, then x_i appears in clause C_j. We have set this $x_i = 1$ when $v_i \in S'$, so clause C_j is satisfied.
- If S' includes a v_i' with a 1 in position C_j, then $\neg x_i$ appears in C_j, which we have set to $x_i = 0$ when $v_i' \in S'$, which again satisfies the clause C_j.
- That satisfies all clauses, so ϕ itself is satisfied.

Example of 3-CNF-SAT to Subset-sum reduction

- Consider the following 3-CNF-SAT formula:
- $\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$.
- $n = 3$ variables, $k = 4$ clauses.
- Let's apply the reduction to subset-sum.
- Fill out the following table using our reduction algorithm:

Example, cont.

-	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	-	-	-	-	-	-	-
v_1'	-	-	-	-	-	-	-
v_2	-	-	-	-	-	-	-
v_2'	-	-	-	-	-	-	-
v_3	-	-	-	-	-	-	-
v_3'	-	-	-	-	-	-	-
s_1	-	-	-	-	-	-	-
s_1'	-	-	-	-	-	-	-
s_2	-	-	-	-	-	-	-
s_2'	-	-	-	-	-	-	-
s_3	-	-	-	-	-	-	-
s_3'	-	-	-	-	-	-	-
s_4	-	-	-	-	-	-	-
s_4'	-	-	-	-	-	-	-
t	-	-	-	-	-	-	-

Example, cont.

-	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	1	0	0	1	0	0	1
v_1'	1	0	0	0	1	1	0
v_2	0	1	0	0	0	0	1
v_2'	0	1	0	1	1	1	0
v_3	0	0	1	0	0	1	1
v_3'	0	0	1	1	1	0	0
s_1	0	0	0	1	0	0	0
s_1'	0	0	0	2	0	0	0
s_2	0	0	0	0	1	0	0
s_2'	0	0	0	0	2	0	0
s_3	0	0	0	0	0	1	0
s_3'	0	0	0	0	0	2	0
s_4	0	0	0	0	0	0	1
s_4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

Example, cont.

- Sample satisfying assignment: $x_1 = 0, x_2 = 0, x_3 = 1$.
- We now have our S for the subset problem:

$$\begin{aligned} S &= \{v_1, v_1', v_2, v_2', v_3, v_3', s_1, s_1', s_2, s_2', s_3, s_3', s_4, s_4'\} \\ &= \{1001001, 1000110, 100001, 101110, 10011, 11100, \\ &\qquad 1000, 2000, 100, 200, 10, 20, 1, 2\} \end{aligned} \tag{1}$$

- We also have our subset S', i.e. $S' = \{v_1', v_2', v_3, s_1, s_1', s_2', s_3, s_4, s_4'\}$.
- Note that the members of S' sum to t:
- $1000110 + 101110 + 10011 + 1000 + 2000 + 200 + 10 + 1 + 2 = 1114444$.