NP1: NP, Reductions

Notes for CS-8803-GA: Introduction to Graduate Algorithms

Georgia Tech (Dr. Eric Vigoda), Fall 2017

as recorded by Brent Wagenseller

Outline

* A problem is **NP-complete** if the problem cannot be solved efficiently for all inputs
* We will look at what it means for a problem to be NP
* We will detail what a reduction is
  + For example, reduce the 2-SAT problem to the strongly connected components problem
* We will look at what it means for a problem to be NP-complete, and how to prove it is NP-complete
* Generally, we are studying **computational complexity**; we will be asking
  + What does NP-completeness mean, and how do we prove it?
  + What does P=NP or P ≠ NP mean?
  + How do we show a problem is intractable?
    - **Intractable** is unlikely to be solved efficiently
      * **Efficient** means its polynomial in the input size; can we solve this problem in time polynomial in the input size?
    - We will be learning how to prove if a problem is NP-complete

Complexity Classes

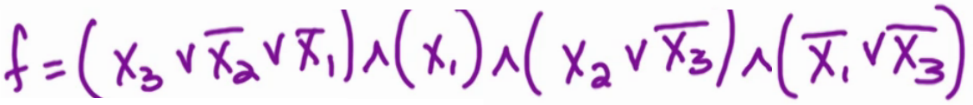
* **NP** is the class of all search problems
  + Some definitions use decision problems; that is NOT how it is defined in this course
  + A rough explanation of a **search problem** is: a problem where we can efficiently verify solutions
    - Solutions can be checked in polynomial time
    - Note the time it takes to *generate* the solution has nothing to do with it; all that matters is the *verification* of the solution in polynomial time
      * We don’t need to find a solution, we just have to be able to verify a solution is correct
  + NP stands for non-deterministic polynomial-time (more on this later)
  + Note NP does not say the solution can be generated in polynomial time
* What do we use if we want to look at the time it takes to *generate* a solution? We look at class P!
  + **P** is a class of search problems that are solvable in polynomial time
* If we can generate a solution in polynomial time, we can also check it in polynomial time
  + P ⊂ NP
  + This means P is a subset of NP
    - BRENTS NOTE: a way to remember this is the letter P is a subset of NP
  + Any problem that can be solved in P can also be solved in NP
    - BRENTS NOTE: proving P seems to be critical for proving NP
  + To recap: NP is a class of search problems where we can verify them in polynomial time, while P is a subset of those where we can solve them in polynomial time
* What does P = NP mean?
  + **P = NP** means its as difficult to solve a problem as it is to verify a solution
    - NP claims given the proof of a theorem, we can check it in polynomial time
    - P can be thought of as generating the proof of the theorem
    - In a sense, P = NP is asking if generating the proof of a theorem is as hard as verifying the proof of a theorem
      * Intuitively, its far harder to generate a proof than simply verifying it
  + If we can show that P = NP, it implies if we can verify a solution in polynomial time, we can also solve the problem in polynomial time
  + Note that P =? NP is roughly asking whether constructing a solution as hard/easy as verifying a solution

Search problems

* We will refine our understanding of a search problem
* A search problem has this form:
  + We are given
    - A search instance ‘I’
      * ‘I’ is the input to the problem
        + As an example ‘I’ could be a graph or a formula; it’s a bit general, but just know its input to a problem
      * If ‘I’ has a solution ‘S’ for this problem, we want to output the solution IF it exists
  + We Output
    - The actual solution ‘S’ IF it exists; otherwise, the word ‘NO’
  + Other requirements
    - If given an instance ‘I’ and solution ‘S’ then we can verify that ‘S’ is a solution to ‘I’ in polynomial time
      * That is to say, polynomial in |I|
      * In a nutshell, this means that if we are given a solution S – even if it wasn’t necessarily generated by our algorithm to find the solution on a particular run of the algorithm – we still need to be able to verify S in polynomial time
      * If there is no solution in polynomial time this is not required
* How do we show a particular problem is a search problem?
  + We can show an algorithm that can verify solutions in polynomial time
    - We could show an algorithm that would take as input ‘I’ and proposed solution ‘S’ and would output if S is an appropriate solution to ‘I’ in polynomial time
    - This algorithm is usually easy to construct

Satisfiability Problem

* K-SAT was one of the original NP-complete problems
* Recall k-SAT
  + Input: Boolean formula f in **conjunctive normal form** (CNF) with n variables (x1 … xn) and has m clauses
    - This is a bunch of Booleans joined with ‘OR’ in clauses, and the clauses are joined with ‘AND’
  + Output: Satisfying assignment if one exists
    - Assigning a particular sequence of True or False to the variables will evaluate to True for the overall formula f
    - If there is no satisfying combination of Booleans, we simply output ‘NO’
  + Here is an example where n (number of x’s)=3 and m (clauses) = 4:



* + Satisfying the above
    - Simplify
      * x1 MUST evaluate to True as the second clause demands it
      * Based on x1 = True, the last clause demands x3 = False
      * Based on x1 = True and x3 = False, the first clause demands x2 = False
    - Since x3 is forced to be False, the only clause left is the third clause and in this instance x3 = False makes that clause True
    - Since we can make every clause True, f can be satisfied with (x1, x2, x3) = (T, F, F)
  + What is the runtime of SAT (as a function of n and m)?
    - This is a key step in proving SAT is in NP
      * That is to say, SAT ∈ NP
    - Formally proving SAT ∈ NP
      * First, is it in the correct form?
        + YES, because we output a solution if it exists, NO otherwise
      * Now show we can verify solutions in polynomial time
        + If we are given f and assignments (x1 … xn) (collectively this assignment is referred to as little sigma σ), how long does it take to check this?
        + Answer

For a particular clause, it takes O(n) time to check if its satisfied

Overall, it takes O(nm) time to check all clauses

It takes a total of O(nm) time to check that σ satisfies f.

This proves we can verify solutions in polynomial time, so SAT is NP

Normally, showing a problem is in NP only takes a few sentences

k-Coloring Problem

* In the coloring problem we have *k* colors and we want to color the vertices so that adjacent vertices get different colors (i.e., no monochromatic edges).
  + Monochromatic means ‘no two colors are next to each other’
* Input
  + It’s a graph, so G=(V, E)
  + integer k >= 0
    - k is the number of colors in our pallet
* Output
  + K-coloring if one exists, NO otherwise
* Objective: Assign each vertex a color in {1, 2, …, k} so that adjacent / incident vertices get different colors
  + If this is not possible, output NO
* K-Colorings ∈ NP
  + This is because
    - Its in the proper form (output answer or NO)
    - We can show that we can verify solutions (although we must still do this)
* K-Colorings has a solution
  + Given G and a coloring, we can go through each edge and check to make sure its not monochromatic
  + In O(m) time (where m = number of edges), we can check that for (v, w) ∈ E, color of v ≠ w
    - That’s it – we are done showing k-Coloring is NP! We simply have to show how an algorithm can do this in polynomial time.

MST

* MST is both P and NP (otherwise, we would not have an algorithm to solve)
* Input
  + G=(V, E)
  + Positive edge lengths
* Output
  + Tree T with minimum weight
* Is MST in the correct form?
  + Yes, even though we never output a NO
    - Inherently there is ALWAYS a solution; technically there is never a NO
    - We are always outputting a solution
* How do we verify to show NP?
  + First show T is a tree
    - Run BFS/DFS to check that T is a tree
  + Now check if T is a minimum weight
    - Run Kruskal’s or Prim’s algorithm to check that T has min weight
    - This is NOT guaranteed to be the same MST that we are checking, but it *is* guaranteed to have the same weight; so we can show that the weight for T is minimal
  + This runs in O(m long n) time (same as Prim’s or Kruskal’s algorithms)
* How do we show that the MST problem is P?
  + First, we have to show that MST is a search problem
    - We just did this to show that MST is NP
  + Second, we need to show that we can find a solution in polynomial time
    - Easy – use the algorithms we learned in class, namely Kruskal’s or Prim’s
    - BRENTS NOTE: It seems to be the case that if you can stuff a problem into an algorithm in class – provided the algorithm is polynomial in time in terms of the input size – it is NP and P; otherwise, it may not be possible (Ford-Fulkerson, Knapsack are examples of this)

Knapsack

* Recall the Knapsack problem
* Input
  + N objects with integer weights w1, …, wn
  + Integer values v1, …, vn
  + Capacity B
* Output
  + Subset S of objects
    - That fit in the backpack: in other words, a total weight <= B
      * That is to say, Σ wi <= B

i ∈ S

* + - The subset must *also* maximize the total value
      * That is to say, Σ vi

i ∈ S

* Recall there were two versions, with or without repetitions
* As it turns out, you cannot verify the results of knapsack, so therefore it is unknown if it is NP and by extension of that it is ALSO unknown in P!
  + We have to be able to verify solutions, and this cannot be done in Knapsack
  + We have to take a particular input to a problem (set of weights and values as well as B) and then we need a solution S; then, we need to verify that S is the solution so we need to check
    - If the weights sum up to be less than B
      * This is easy to do and takes O(n) time
    - We need to check that the subset S has a maximum value
      * It needs to be better – or, at least, just as good – as other solutions
      * Unfortunately, the algorithm needs to run in polynomial time in terms of the input size; since Knapsack runs in O(nB) time, it is NOT polynomial in terms of the input size and thus cannot be NP (and, by extension, it cannot be P either)
        + B is represented by B bits

Therefore, its represented by (log B) bits

In order to be polynomial in the input size, we need an algorithm to run in poly(n, log B)

Our dynamic programming approach is actually exponential with respect to the input size

Therefore, we have no way of verifying that our proposed solution is optimal

* + We say knapsack is ‘unknown’ in NP and we do not say its not NP; this is because we cant actually show this one way or another
    - If we could show that Knapsack was not in NP, that would mean P ≠ NP
    - By extension, it is unknown if Knapsack is in P
* There is a variation of Knapsack that is NP
  + We drop the MAX portion of the Knapsack requirements
  + We add in another parameter which is a goal
    - So its not a MAX, it’s a goal that needs to be either met or exceeded
      * This is easier to show NP
  + This is known as the Knapsack-Search problem
* Knapsack-Search Problem
  + Input
    - N objects with integer weights w1, …, wn
    - Integer values v1, …, vn
    - Capacity B
    - Goal g (this is new)
  + Output
    - Subset S of objects
      * That fit in the backpack: in other words, a total weight <= B
        + That is to say, Σ wi <= B

i ∈ S

* + - * The subset must *also* meet the goal g
        + That is to say, Σ vi >= g

i ∈ S

* + - * + This is different from the original Knapsack problem
    - We must also add an output NO if there is no solution that meets goal g
  + To solve this, we can simply do a binary search over g
  + By doing a binary search over g, we can find the maximum g which has a solution, which tells us the max total value we can achieve
    - How many rounds in a binary search must we run?
      * How many knapsack-search problems must we run in order to solve the problem?
      * The sum of all the values definitely have a max; so sum all values {v1, …, vn}; let V be the sum of these values
      * At most, the answer will be V, so the answer is bounded by V
        + Therefore, we do a binary search on g between 1 and V
        + The total number of rounds in a binary search algorithm will be at most O(log V)

This is polynomial in the input size

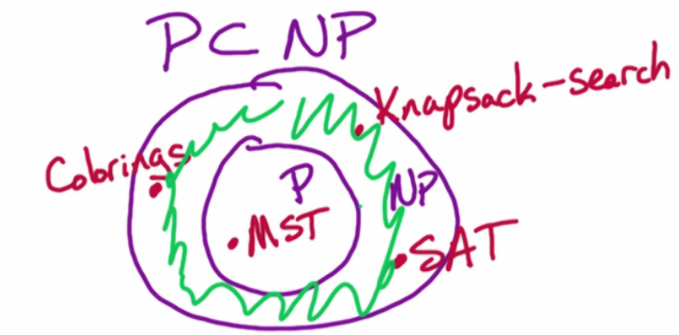
* + - * + Since we can solve this in polynomial time, we can solve the optimization in polynomial time
  + Knapsack-Search exists in NP
    - Consider the input and a solution S
    - In order to check that S is a solution we need to
      * Check if the total summed weight =< B
      * Check if the total summed value >= g
    - This can be checked in O(n) time!

Terminology

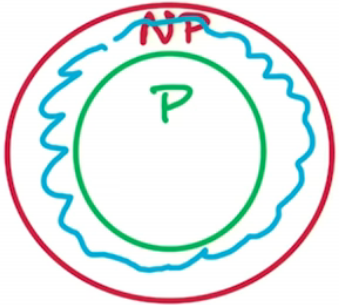
* **P** is short for polynomial time
* **NP** stands for **nondeterministic polynomial time**
  + NP DOES NOT STAND FOR NON-POLYNOMIAL TIME
  + By **nondeterministic**, it is meant the problems that can be solved in polynomial time on a nondeterministic machine
    - By **nondeterministic machine**, it is meant that we are allowed to guess at each step
      * There is a series of choices that lead to each state
      * Visual example of nondeterminism



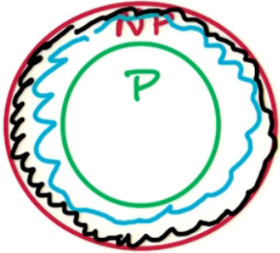
* + - * + In the above graph, we are not sure which ending state we will land in, but we are sure that we WILL land in an ending state with some probability p (which are the values of the edges).
* P VS NP
  + Recall NP is all search problems
  + Recall P are search problems that can be solved in polynomial time
  + Every search problem lies in NP, and P are those that are solvable in polynomial time
    - This means P ⊂ NP
  + A visual



* + - The above shows where certain problems lie
    - There is a philosophical argument that there may actually be no difference between P and NP; the problem is, no one can definitively prove or disprove that
      * Furthermore, we are never certain there is no P for the problems currently outside of P; its possible we just haven’t found them yet.
      * Its also possible that there are problems that are in P but not in NP (although none have been found as of yet)
  + P ≠ NP
    - If P ≠ NP, this means there are some search problems that cannot be solved in polynomial time
    - These are NP-Complete problems; **NP-Complete** means the problem is intractable and cannot be solved in polynomial time



* + - * NP-Complete problems are guaranteed to lie in the ‘doughnut’ circling P (but are not in P)
        + BRENTS NOTE: It seems there are actually problems that are not in P, are in NP, but are not NP-Complete; Dr. Vigoda described the NP-Complete problems as the outer scribbled black marker, then the nebulous NP problems that are NP but not P in between, then the problems that are both NP and P in the following:



* + - * NP-Complete problems are the hardest problems in the class NP
        + What do we mean by the hardest in the class?

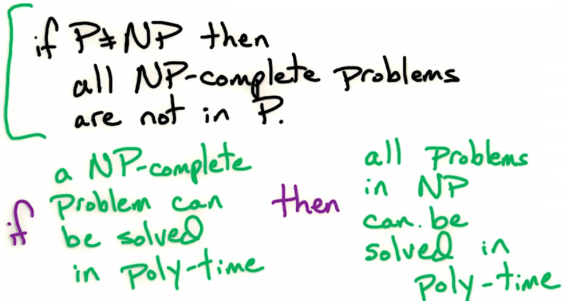
If P ≠ NP, then all NP-Complete problems are not in P

BRENTS NOTE: I think this is referencing the philosophical argument on whether or not P and NP are different; I think this is saying if P IS nested in NP, then surely, all NP-Complete problems cannot be solved in polynomial time

An equivalent is the contrapositive statement: there exists a NP-Complete problem that can be solved in polynomial time

IF this holds, then the complement holds, which is: P = NP

If P = NP, then all problems in the class NP can be solved in polynomial time



In other words, if there is a polynomial time algorithm for SAT, then we can use that methodology on ALL other NP-Complete problems and ALL NP-Complete problems would be P.

* + - To sum up the theoretical argument:
      * P = NP: If we can verify solutions efficiently, then we can construct solutions efficiently. In other words, verifying a proof is not harder than constructing a proof.
      * P ≠ NP: Some search problems cannot be solved in poly time.  These problems are NP-Complete problems, they are the hardest problems in NP in the sense that they are the most difficult search problems to solve.
        + If P ≠ NP, then there is no polynomial time algorithm for some problems in NP, these are the “hardest” problems in the class NP and hence are called NP-complete problems.
        + The problems Knapsack-search, SAT, k-Colorings, TSP are all NP-complete, and there a ton of other NP-complete problems. If we can solve one of these NP-Complete problems efficiently, then we can solve all of the rest efficiently.

SAT is NP-Complete

* We are going to see what it means for a problem to be NP-Complete, using SAT as an example
* Requirements
  + Firstly, SAT ∈ NP
    - SAT must be in NP
    - For SAT, this is easy – just plug in the proposed answers to the formula
  + Secondly, SAT is the hardest problem in this class
    - What does THAT mean?
      * For SAT to be in this class, it must mean its least likely to have an efficient solution
        + Said differently, if we have a solution for SAT, we have a solution for every other problem
      * If we can solve SAT in polynomial time, then we can solve EVERY problem in NP in polynomial time.
        + This would mean for all other NP problems – Colorings, Knapsack, Travelling Salesman, and even MST (although MST is P, still…) – all of them could be reduced to the SAT problem

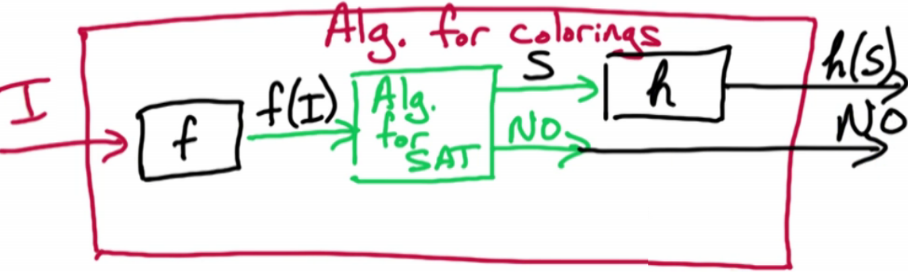
The way this would work would be we would take inputs to any other NP problem, reduce it to SAT, then run it through SAT and we would get a solution to the original problem

This would mean we could reduce ALL problems in NP to SAT

* + - SAT has to be in the most computationally difficult class in NP
* If P ≠ NP, then there are some problems that cannot be solved in polynomial time and thus SAT ∉ P
  + Alternatively, if no one knows how to prove P = NP, then no one knows how to prove that SAT is P
* It’s a reasonably fair assumption that there is no polynomial time algorithm for SAT
  + If someone did, then they could prove P = NP
* How can we prove that if there is a polynomial time algorithm for SAT, every other NP-Complete problem is solvable in P?
  + To do so lets look at reductions

Reductions

* Consider problems A and B and we want to consider a reduction from A to B: A → B
  + For argument’s sake, A = Colorings problem, B = SAT
* Note we will say A → B, but there is other notation: A<= B
  + Read: “A at most B”
  + This means when we are showing a reduction from A to B, B is at least as hard computationally as A
  + If we are trying to make an algorithm, it will be easier to make one for A
    - If we made an algorithm for B, we solve both B and A
    - If we just solved A, we may not actually solve B
* A **reduction** A → B means reducing A to B
  + Formally: If we had an algorithm which solves B in polynomial time, we can use that to solve A in polynomial time
* How to do a reduction
  + We are going to reduce Coloring → SAT
  + Suppose there is a polynomial time algorithm for SAT and we use it to get a polynomial time algorithm for Colorings
  + We are going to treat this algorithm like a black box
    - We don’t know how this black box works, but we can use it as a subroutine to get a polynomial time algorithm for the Colorings problem



* + - * We can give it some input, the input is transformed to something SAT can process, and then we either get some output or NO
      * The transformation f transform the input from a Colorings problem to a SAT problem
      * The output S is a solution from the SAT problem
        + OR, we could get a NO
      * The transformation h transforms the SAT solution S to a solution to the Colorings problem
      * In short
        + S is a solution to f(I)
        + h(S) is a solution for I
  + More on reductions
    - We need to define functions f and h
      * f takes in input for the Colorings problems (G, k) and creates input for the SAT problem f(G,k)
      * h takes the solution from the SAT problem f(G, k) and form a solution for the Colorings problem
    - What do we need to prove to prove the reduction is valid?
      * We need to prove that if S was a solution to f(G, k), then h(S) is a solution to the original (G, k)
      * We ALSO need that if there is no solution to f, then there is no solution to the Colorings problem
      * In short, we need to prove: S is a solution to f(G, k) ⇔ h(S) is a solution to (G, k)
        + We need to know if the solutions map AND vice versa
        + prove that if there is no solution for f(I) then there is no solution for I

NP-Completeness Proof

* We will use the Independent Sets problem as an example of how to show something is NP-Complete
  + We will go over the Independent Sets (IS) problem later on in the course, but for now its not important what it is
* First Step: Show that IS ∈ NP
  + Show that we can verify solutions in polynomial time
* Second Step: Show that for all A ∈ NP, A → IS
  + For every problem A in the class NP, there is a reduction of A to the Independent Sets
    - If we have a polynomial time algorithm for the IS problem, then we can use that as a subroutine to solve problem A in polynomial time
      * Since this can be done for every problem A in NP, then if a polynomial time algorithm exists for IS, then there is a polynomial time algorithm for every A in NP
* HOW does this work, given we need to show EVERY problem in NP can be reduced to this alleged algorithm?
  + Suppose we know SAT is NP-Complete
  + This also means ∀ A ∈ NP, we know the reduction A → SAT has already been shown
    - For all A that exists in NP, we know the reduction holds
  + Now, suppose we can show SAT → IS
    - We know that A → SAT
      * As this is part of the definition of NP-Complete
    - Therefore, A → SAT → IS
      * This holds for every A, therefore for every A in NP we have a reduction from A to the Independent Sets problem
      * If we have a known NP-Complete problem (such as SAT), then, to show that any other problem is NP-Complete all we need to do is
        + Show the problem lies in the class of NP

Show we can verify solutions in polynomial time

* + - * + Show there is a reduction from a known NP-Complete problem (in our case SAT) to the problem we are trying to prove is NP-Complete

This works because every known NP-Complete problem reduces to SAT

This step often confuses students and they accidentally show the reduction in the *other* direction which is meaningless

* Now we can show if a problem is computationally difficult (NP-Complete)

Practice Problems

* 8.1: Opt vs Search
* 8.2: Search vs Decision