Intractability

Analysis of Algorithms
Complexity Theory

• We categorize computational problems according to how difficult they are to solve
  – “Difficult to solve” ≠ hard for you to solve
  – “Difficult to solve” = lots of computing resources

• Note that there are a huge (infinite?) number of computational problems that defy categorization
  – We don’t know how difficult they are to solve
Complexity Theory

• In theory, a computational problem to be solved is thought of as a “language”
  – Encode an instance of the problem as a string
  – If the string represents a solution to the problem, then the language “accepts” the string
  – If the string does not represent a solution to the problem, then the language “rejects” the string
  – For example, Java is a language that accepts or rejects (compiles or not) your string (your program)
Complexity Theory

- Recognizable
- Decidable
- \text{EXPSPACE}
- \text{EXPTIME}
- \text{PSPACE=NPSPACE}
- \text{NP}
- \text{coNP}
- \text{P}
- \text{CFL (regular)}
- \text{BPP}

Image from http://www.cse.psu.edu/~sofya/cmpsc464/
Complexity Theory

• The aim of complexity theory is to classify problems according to their intrinsic difficulty.
• For which problems can we find *efficient* algorithms?
• This, of course, depends on the definition of efficient, which is a subjective matter.
Complexity Theory

• In the 1960’s Jack Edmonds proposed (in a paper called “Paths, Trees, and Flowers”) that an algorithm should be called “efficient” if its running time is a polynomial function of the size of the input, and “inefficient” otherwise.
Complexity Theory

• This suggestion was adopted, and since then, a problem is considered *tractable* if it has a polynomial time solution and *intractable* if it does not.
  – The tractable problems all belong to Class P.

• It is a measure of our inadequacy that for most problems we still do not know whether they are tractable or not.
The Class P

The class P consists of those decision problems for which there is a deterministic Turing machine that when presented with a string w which is an instance of the problem, will correctly accept or reject w in time having an upper bound that is proportional to a polynomial function of |w|

According to this definition, it seems that to show that a problem is in P, we would have to actually find a Turing machine to solve that problem in polynomial time
The Class P

• Fortunately we can stick with our normal algorithm analysis in terms of elementary operations such as array accesses, loops, summations, etc., because a Turing machine can implement all of those in polynomial time

• Therefore, problems such as sorting a list of integers or finding the shortest path in a graph are clearly in P
The Class P

• For many other problems, nobody knows whether or not they are in P
• Sometimes there is an obvious naïve algorithm which consists of trying all the possibilities
• For example, one could solve the subset sum problem by trying all $2^N$ possible subsets
  – A decidable Turing machine could be designed to do this
The Class P

• Some problems (such as subset sum) have no known algorithm for solving them that are more efficient than trying all possibilities
• Why don’t we have efficient algorithms for these problems?
• Either:
  – We are not clever enough to find an efficient algorithm for these problems or
  – An efficient algorithm for these problems does not exist
The Class NP

- The **class NP** contains a collection of problems that *might* have polynomial time solutions.
- The idea is to exclude the Turing-recognizable problems such as the Halting problem which can never have a polynomial time solution.
The Class NP

The class NP consists of those decision problems for which there is a non-deterministic Turing machine that when presented with a string $w$ which is an instance of the problem, will correctly accept or reject $w$ in time having an upper bound that is proportional to a polynomial function of $|w|$

Always remember:

$P = \text{Polynomial}$

$NP = \text{Non-deterministic Polynomial}$
The Class NP

• There are two views that may help with understanding the class NP:
  – NP is the class of problems that can be solved in polynomial time by an infinite number of Turing machines running in parallel (non-determinism)
  – NP is the class of problems for which, given a potential solution to a specific problem, the solution can be verified in polynomial time
    • This is the more useful definition, for our purposes
The Class NP

• The class P is a subset of the class NP because we can verify any class P problem the same way that we verify class NP problems
  – The difference is that for class P, not only can we verify a solution in polynomial time, we can also find the solution in polynomial time
  – In summary, for a problem to belong to class NP, it is only necessary that a candidate solution is verifiable in polynomial time
  – Actually finding the solution in polynomial time may or may not be possible for problems in NP
The Class NP

- Thus, it is easy to see that class P is contained within NP, so we have the following relationship:

\[ P \subseteq NP \]
Is $P = NP$?

- Nobody has yet been able to determine if all of the problems in class NP have polynomial time verifications and polynomial time solutions
  - Just how big is that class $P$?
- After some 25-30 years of sustained effort there still remains the question:

  Is $P = NP$?
Is \( P = NP? \)

- In order to solve this question a vast body of theory has been built up.
- We shall merely scratch the surface of this by considering those problems that are known to be *NP-complete*
  - NP-complete problems are all said to be *intractable*.
  - They are considered to be the most difficult problems in NP to solve (i.e., subset sum), and seem to require (?) trying all possibilities to get a solution.
Is \( P = NP? \)

The diagram on the left depicts the situation where the problems in class \( P \) and class \( NPC \) are distinctly different (but both belong to class \( NP \)). The diagram on the right depicts the situation where class \( NPC \) is a subset of class \( P \).
Is $P = \text{NP}$?

(a) **Impossible** because the trivial decision problem whose solution is “accept” for all inputs is easy and clearly not NP-complete

(b) **Possible** – this depicts the case of $P = \text{NP}$

(c) **Unlikely** because there are problems in NP that are neither in P nor NP-complete (i.e., graph isomorphism)

(d) **Impossible** because if there is one NP-complete problem in P, then all of the NP-complete problems are in P

(e) **Possible** – this depicts the case of $P \neq \text{NP}$
Problem Reductions

• Suppose we have two computational problems, A and B, that seem very similar
  – Assume that we have an algorithm to solve B, but we don’t have an algorithm to solve A
• If we could somehow “transform” problem A into problem B, then we could use the algorithm for B to solve A
  – In other words, if we “transform” (reduce) A to B, then we could use the solution for B to get a solution for A (by un-transforming it back to A)
Problem Reductions

• Problem A **reduces to** problem B if we can use the algorithm for B to solve A

Cost of solving A = cost of solving B + cost of reducing A to B.
Problem Reductions

Question: Describe how to reduce the following problems (A) to problems (B). State the cost.

1. Problem A: Computing the median of a list of integers; Problem B: Sorting a list of integers.
2. Problem A: Detecting a cycle in a directed graph; Problem B: Depth-first search in a directed graph.
3. Problem A: All-pairs shortest paths; Problem B: single-source shortest paths.
4. Problem A: Element distinctness; Problem B: Sorting.
Problem Reductions

• If we can transform A to B in polynomial time, and B can be solved in polynomial time, we know that A must also be solvable in polynomial time
  – If B is easy to solve (it is tractable), then A must also be easy to solve (it is also tractable), since we could use B to get a solution for A

• This technique is used to prove that certain problems are NP-complete (intractable), with one difference ...
Problem Reductions

• An NP-complete proof is about showing how difficult a problem is, not how easy it is, therefore we do the reduction in the opposite direction
  – If A is not easy to solve (intractable), then B must also be not easy to solve because A is easier, or at least no harder, than B
  – In other words, we use the known intractability of A to prove the intractability of B
Intractable Problems

Let’s say we have an algorithm that solves problem B, but we don’t know how tractable it is. All we know is that we could use the algorithm for B to get a solution for A. So if B is tractable, then A must be also be tractable (because A is easier to solve than B). But what if we know for sure that A is intractable?

Then it **must** be the case that B is also intractable.
Intractable Problems

• Proof that problem B is intractable (proof by contradiction):
  – Assume the algorithm for problem B is tractable and that problem A is polynomial time reducible to B (A \leq_p B)
  – If this assumption is true, then we could use the algorithm for B to get an efficient solution to problem A
  – But if we’re certain that A is intractable, then it must be the case that B is also intractable
  – Therefore, the assumption that the algorithm for B is tractable must be false – it is intractable
Crossword Puzzles

• Consider solving a crossword puzzle (C-PUZZLE)
  – If we could correctly guess which letter goes in each square, then C-PUZZLE is solvable in O(N) time, where N is the number of words available
    • It is the “guessing” part of the algorithm that makes it non-deterministic (non-determinism = guessing correctly = NP)
    • A specific guess that represents the solution is called a qualifying certificate (also called a witness)
  – If, given a specific certificate, we can verify in polynomial time whether or not that certificate solves the puzzle, then C-PUZZLE belongs to class NP
  – But what if we do not have a certificate? Is C-PUZZLE easy or hard to solve in the absence of a certificate?
Crossword Puzzles

- Let’s assume that we have a “magic” crossword puzzle solver that always gives the correct solution to a crossword puzzle, but we don’t know how tractable such a magic puzzle solver is.
- Let’s also assume we have a problem, graph coloring (which is widely known to be intractable), for which we would like a solution.
- Reduce the graph coloring problem (a specific coloring of a graph) into a specific crossword puzzle puzzle (how?)
- Next, we could use the magic crossword puzzle solver to solve the crossword puzzle that was created from the colored graph.
- When we “un-reduce” the solution for the crossword puzzle, we will have a solution for the graph coloring problem!
- Since it is well-known that graph coloring is intractable, crossword puzzle solving must also be intractable.
Crossword Puzzles

Consider a specific instance of a graph we would like to correctly color. How could we use a “magic” crossword puzzle solver to get a correct coloring for the graph?

Crossword puzzle word list:
RRR, RRR, GGG, GGG, BBB, BBB, YYY, YYY, RRRRR, GGGGG, GGGGG, BBBB, BBBB, YYYY, YYYY, RRRRR, RRRRR, BBBB, RG, GB, BY, RY, RB

Given an instance of graph coloring, create a “special” crossword puzzle puzzle: create a pair of 3-letter words to represent each vertex in the graph, a pair of 5-letter words to represent each edge in the graph, and 2-letter words to represent the restriction that no two adjacent vertices in the graph can be connected by an edge. Note that all 3- and 5-letter words contain only the same letters, whereas the 2-letter words always contain two different letters. Now solve the puzzle using the magic crossword puzzle solver. This gives a solution to graph coloring.
The solution for the puzzle corresponds to the solution for the graph where the upper-left vertex is colored red and the upper-right vertex is colored green. The 2-letter words guarantee that adjacent vertices have different colors.

If we had an algorithm that knew how to correctly fill in the words of the crossword puzzle (using the magic puzzle solver), then we could (and would!) use that algorithm to solve graph coloring problems.

But since we know that graph coloring cannot be solved efficiently (unless P = NP), then it must be the case that crossword puzzles also cannot be solved efficiently either (unless P = NP).
NP-Complete Proofs

• Definition of NP-complete (memorize):

    Problem $B \subseteq \{0,1\}^*$ is NP-complete if:
    1. $B \in \text{NP}
    2. $A \leq_p B$ for every $A \in \text{NP}$

• If $B$ satisfies property 2 but not necessarily property 1 then $B$ is called $\textit{NP-hard}$

• Recall that NP-complete problems are considered to be the hardest problems in class NP to solve, so all of the other problems in NP (all problems $A$) are easier (or at least no harder) than problem $B$
NP-Complete Proofs

• If *any* NP-complete problem can be solved in polynomial-time, then *every* problem in NP has a polynomial-time solution, i.e., $P = NP$

• Therefore, it is very important to identify those problems that are indeed NP-complete
  – But must we consider *all* of the problems in NP to make the proof?
  – Property 2 from the definition of NP-complete seems to imply yes!

  1. $B \in NP$
  2. $A \leq_p B$ for *every* $A \in NP$
Circuit Satisfiability is NP-Complete

• The trick is to leverage the idea that all NP-complete problems are really just “variations” of one another
  – This implies that you really only need to choose one of those problems; since they are all equivalent, any one will do
• Well, that is fine, but there must have been a first problem that was proven to be NP-complete
  – This problem is known as Circuit Satisfiability (CIRCUIT-SAT)
  – A logic circuit is said to be satisfiable if there is a way to assign truth values to each of the logic elements such that the output will be true
Circuit Satisfiability is NP-Complete

- CIRCUIT-SAT Problem:
  - Given a Boolean combinational circuit, is it satisfiable?

Three basic logic gates, with binary inputs and outputs. Under each gate is the truth table that describes the gate’s operation. (a) The NOT gate. (b) The AND gate. (c) The OR gate.
Circuit Satisfiability is NP-Complete

- **Solution:**
  - Try all possible assignments to the inputs
  - For N inputs, there are $2^N$ possible ways to assign inputs, which means the efficiency of this solution is $O(2^N)$

Two instances of the circuit satisfiability problem. (a) The assignment $(x_1=1, x_2=1, x_3=0)$ to the inputs of this circuit causes the output of the circuit to be 1. This circuit is therefore satisfiable. (b) No assignment to the inputs of this circuit can cause the output of the circuit to be 1. This circuit is therefore unsatisfiable.
Circuit Satisfiability is NP-Complete

• Proof that CIRCUIT-SAT is NP-complete:
  – 2 steps:
    1. Show that CIRCUIT-SAT belongs to NP
    2. Show that every other problem in NP (they are easier) is polynomial-time reducible to CIRCUIT-SAT
  – This means that CIRCUIT-SAT is at least as hard (probably harder!) than any other problem in NP
  – For the complete direct proof, see Garey & Johnson 1979
Circuit Satisfiability is NP-Complete

1. Show that CIRCUIT-SAT belongs to NP:

Provide a p-time algorithm that can verify CIRCUIT-SAT. This means that if every logic gate in the circuit were assigned a value of 0 or 1, you could easily check whether or not the circuit is satisfied (the output wire has a value of 1). To do this, you need two things: an encoding of the circuit in the form of a graph, and a specific assignment (called a certificate).

The p-time algorithm is this: For the given assignment, check that the value provided by the certificate is the same as the value on the output wire. If the output wire has value 1, then the graph outputs true, otherwise it outputs false. This algorithm is O(n).
Circuit Satisfiability is NP-Complete

2. Show that CIRCUIT-SAT is NP-hard:

Create a graph that represents the circuit C. Map every binary string \( x \) to a circuit C such that \( x \) is a solution to the graph problem if and only if C is a solution to CIRCUIT-SAT.

To do this, generate a sequence of configurations of M, the machine that computes the output of C. A configuration represents the state of M for one step of the computation. Each configuration is mapped to the next configuration by a Boolean combinatorial circuit (M is the mapping). At each stage the output is either true or false. The final output is either true (accept) or false (reject).
Circuit Satisfiability is NP-Complete

Thus, since every (computable) problem must use a (Turing) machine to get an answer, this problem is at least as hard (probably harder) to solve than any of those other problems, i.e., every problem in NP (every $A \in \text{NP}$) is polynomial-time reducible to CIRCUIT-SAT. Proof complete!
Outline of Proof Approach

• Step 1: Showing that a problem belongs to class NP is relatively easy: just describe how to verify a candidate

• Step 2: When trying to show that a problem B is NP-hard, start by choosing a problem A that is already known to be NP-hard
  – Next, describe how to reduce A to B in polynomial time (e.g., transform a colored graph into a crossword puzzle)
    • You must be specific when describing the reduction!
  – Then explain how the solution for B gives a solution for A
  – State that since it is already known that A is NP-hard, then it must be the case that B is also NP-hard

• Conclude by stating that since B belongs to class NP and it is NP-hard, problem B is therefore NP-complete