A GENTLE INTRODUCTION TO COMPUTATIONAL COMPLEXITY THEORY, AND A LITTLE BIT MORE

SEAN HOGAN

Abstract. We give the interested reader a gentle introduction to computational complexity theory, by providing and looking at the background leading up to a discussion of the complexity classes P and NP. We also introduce NP-complete problems, and prove the Cook-Levin theorem, which shows such problems exist. We hope to show that study of NP-complete problems is vital to countless algorithms that help form a bridge between theoretical and applied areas of computer science and mathematics.

Contents

1. Introduction 1
2. Some basic notions of algorithmic analysis 2
3. The boolean satisfiability problem, SAT 4
4. The complexity classes P and NP, and reductions 8
5. The Cook-Levin Theorem (NP-completeness) 10
6. Valiant's Algebraic Complexity Classes 13
Appendix A. Propositional logic 17
Appendix B. Graph theory 17
Acknowledgments 18
References 18

1. Introduction

In "computational complexity theory", intuitively the "computational" part means problems that can be modeled and solved by a computer. The "complexity" part means that this area studies how much of some resource (time, space, etc.) a problem takes up when being solved. We will focus on the resource of time for the majority of the paper.

The motivation of this paper comes from the author noticing that concepts from computational complexity theory seem to be thrown around often in casual discussions, though poorly understood. We set out to clearly explain the fundamental concepts in the field, hoping to both enlighten the audience and spark interest in further study in the subject.

We assume some background in propositional logic and graph theory (provided in the appendix). In Section 2, we introduce some basic notions and examples of algorithmic analysis in order to provided an intuition for talking about computational difficulty. In Section 3, we talk about the boolean satisfiability problem

Date: Last Revised September 10th, 2011.

SAT, and investigate an efficient algorithm for 2-SAT in detail. In Section 4, we discuss reductions and the complexity classes P and NP, as well as their relation to NP-completeness. In Section 5, we go through a common proof of the Cook-Levin theorem, a vital result in complexity theory. In Section 6, we quickly look at a small part of Valiant's algebraic complexity theory.

2. Some basic notions of algorithmic analysis

An algorithm can be thought of as a process that takes in some input and produces a given output within a known time. An example input to an algorithm could be a list of n elements, the output being the sorted list. We can think of the number of steps it takes an algorithm to finish as a time metric. To flesh this idea out, we now introduce notation to describe an algorithm's running time based on the size of the algorithm's input.

Definition 2.1. Let $f, g : \mathbb{N}^+ \to \mathbb{R}^+$, and suppose $g(n)$ is the running time of an algorithm on an input of size n. We denote the asymptotic running time of an algorithm by $\mathrm{O}(f(n))$. This is called **Big-O notation**, meaning there exists some $c \in \mathbb{R}^+$ such that for all $n \in \mathbb{N}^+$, $g(n) \leq c \cdot f(n)$, i.e., $c \cdot f(n)$ bounds $g(n)$ from above.

Since Big-O notation measures asymptotic (large values of n) growth, one only considers the fastest growing term of some given $g(n)$ - e.g. if $g(n) = 3n^2 + \log(n)$[1], then $g(n) = O(n^2)$, because as $n \to \infty$, $\log(n)$ becomes negligible. Note that for two functions on the same domain, if $h \geq f$, then $g = O(f)$ implies $g = O(h)$. Note that there exist analogs of $O(f)$ for other inequalities, for example, if $g = \Omega(f)$, then Definition 2.1 follows exactly, except for the last sentence, where $g(n) \geq c \cdot f(n)$.

Example 2.2. Let's consider a very basic algorithm to illustrate this notation. The algorithm is Quicksort.[2] It orders an array of numbers. The pseudocode is shown below. We assume some familiarity with general control flow (if, then, else, for, return, etc.), as well as the general idea of recursion.[3]

The input is an array of n numbers. Quicksort picks at random an element v from the array, then compares it with every other element in the array. The array is split into three subarrays - for elements less than, equal to, and greater than v. Quicksort then recurses (runs itself within itself) on the "less than" and "greater than" arrays. Finally, Quicksort appends those sorted arrays to the equal array, returning the sorted result.

An example would be the input array {3,1,2}.
If $v = 1$, then our return statement would read
return quicksort$(\{\})$:$\{1\}$:*quicksort*$(\{3, 2\})$,
which after the two calls to quicksort finish, would end up as
return {}:{*1*}:{*2,3*} = {*1,2,3*}.

---

[1]Throughout this paper, $\log(n)$ refers to the base two logarithm of n.

[2]This is a slightly modified Quicksort - it also keeps track of an "equal" array, where as a textbook Quicksort usually only keeps a "less than or equal" and "greater than" array. For convenience we'll refer to it as Quicksort. On arrays with some equal elements, this Quicksort runs a little faster, but we'll analyze it assuming that all elements are distinct.

[3]The array need not contain numbers. As long as the objects in the array have some way of being ordered, the sorting algorithm works.

```
//Input: Array aIn, with sortable elements
//Output: The sorted array.
quicksort(Array aIn):

if |aIn| <= 1, return aIn; //Check size

aL,aE,aR = []; //Subarrays start empty
let v = random element of aIn;

for every p in aIn:
        if p < v, aL::v; //Prepend v to aL
        else if p = v, aE::v;
        else aR::v;

if aL and aR = [], then return aE;
return quicksort(aL)::aE::quicksort(aR);
```

Let's look at the average running time for Quicksort, on an array of n distinct numbers. Note that overhead from a computer - initializing variables, handling recursive calls, etc., are generally ignored from the analysis of the algorithm. On average, every recursive call will split the input array into subarrays of equal size (within one element). We can model the running time as a recurrence relation, and then solve it to get a closed form solution that gives us an $O(-)$ time bound. Our recurrence relation will be

$$T(n) = 2T(\frac{n}{2}) + O(n) \tag{2.3}$$

What (2.3) means is that the time $T(n)$ to run the algorithm on an input of size n is at most twice the time $T(\frac{n}{2})$ it takes to run Quicksort on a half-sized array, plus a multiple of a linear order term $O(n)$.

Let's find a closed form formula for the running time. Using the recurrence relation, we can write

$$T(\frac{n}{2}) = 2T(\frac{n}{4}) + O(\frac{n}{2}).$$

By substituting the expression for $T(\frac{n}{2})$ into (2.3), we have

$$T(n) = 2(2T(\frac{n}{4}) + O(\frac{n}{2})) + O(n) = 4T(\frac{n}{4}) + 2O(n)$$

After substituting for $T(\frac{n}{4}), T(\frac{n}{8})$, etc., we can just write the expression for $T(n)$ as

$$T(n) = 2^k T(\frac{n}{2^k}) + kcn \tag{2.4}$$

where k is how many levels deep the recursion goes (intuitively, this is just one more than how many times we make a substitution for some $T(\frac{n}{2^i})$. Recall that we can make the simplification of the $O(n)$ terms by thinking of $O(n)$ as cn, for some $c \in \mathbb{R}$. In this sense, $2O(\frac{n}{2}) = 2 \cdot \frac{cn}{2} = cn$, which is how we eventually obtain kcn. For convenience and sake of clarity, we treat n as a power of 2, so if we let $k = \log(n)$, then (2.4) simplifies to $nT(1) + cn\log(n) = n + cn\log(n)$, and since

we only worry about asymptotic behavior, we ignore the linear n term and the constant c and say that $T(n) = O(n \log(n))$.[4]

3. The boolean satisfiability problem, SAT

Let's now move our attention to a problem, called 2-Satisfiability (2-SAT). This will give us a good base for later when we talk about the class of NP problems.

Definition 3.1.
A **boolean literal** x is an object which takes on the value of either true or false. We write the negation of x as $\overline{x}$.
A **clause** is the disjunction (Applying the logical operator $\vee$) of a finite set of distinct literals (we consider some x and $\overline{x}$ to be distinct literals).
A **conjunctive normal form (CNF) formula** is the conjunction (Applying the logical operator $\wedge$) of a finite set of clauses.

We say a **2-SAT** problem is finding an assignment of truth values that will evaluate an entire CNF formula to true when every clause has at most two literals.

Example 3.2. Consider the formula $(x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2)$. A satisfying assignment would be $x_1 = T, x_2 = T$. The formula would evaluate as follows: $(T \vee F) \wedge (F \vee T) = (T) \wedge (T) = T$.

It turns out that we can solve any instance of the 2-SAT problem in linear time (there is an algorithm that is $O(n)$). To solve it, we introduce a **depth-first search**-based algorithm (DFS) on a directed graph. Intuitively, DFS visits every node of a graph by walking along a branch and backtracking when necessary. DFS first marks all vertices as a graph as "not visited". A vertex is "visited" if we have *started* to iterate through its set of neighbors. DFS also initializes a counter, val, which is used to assign pre-visit and post-visit values to each vertex.

DFS then picks a vertex v_0 from the graph, and calls *explore* on it, which does the following. It marks v_0 as visited, assigns it a pre-visit value val, increments val, then iterates through v_0's neighbors, calling *explore* on each neighbor that is *not visited*. After all the calls to *explore* return, DFS assigns v_0 a post-visit value, and increments val.

Since DFS has a way of checking if a vertex has been visited, it looks at all vertices's neighbors exactly once, and since the iteration in the dfs function checks that every vertex is visited, we are guaranteed to have $pre/post$ values for each vertex.

An example of DFS is pictured below. Just follow the numbers from 1 to 12 to understand the search.

[4] There exists a theorem for solving recurrence relations of the form $T(n) = aT(\frac{n}{b}) + O(n^d)$, called the Master Theorem. We omit the theorem since the paper's focus is not entirely on algorithmic analysis, but the interested reader can easily find many good proofs.

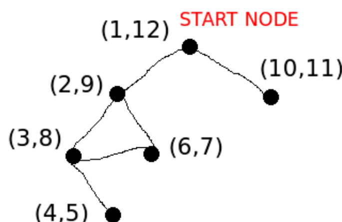

FIGURE 1

The pseudocode of this depth-first search algorithm is below. *pre* and *post* are arrays with integer values who are indexed by vertices.

```
//Input: Directed graph G = (V,E)
//Output: Arrays pre and post
dfs(Graph G = (V,E)) {
pre, post = [];
val = 1;
for all v in V: visited(v) = false;
for all v in V: if (visited(v) = false), then explore(G,v);

return (pre,post);
}

//Input: Directed graph G = (V,E) and a vertex v in V.
//Output: pre/post values of v are set
explore(G,v) {
visited(v) = true;
pre[v] = val;
val++; //Increment val
for all (v,u) in E:
        if (visited(u) = false), then explore(G,u);
post[v] = val;
val++;
}
```

This algorithm's runtime is based on the size of V and E - it is $O(|V| + |E|)$ because each vertex and edge is visited exactly once. We now prove the following that will help us solve 2-SAT:

Claim. *Given the pre and post arrays of a graph G, in linear time we can determine the structure of G's meta-graph of strongly connected components (SCCs)*[1].

Proof. Recall that a **Meta-graph** refers to the graph made if we think of every SCC as its own node. These nodes might be **sinks** - SCCs with no outgoing arrows - or **sources** - the opposite. Note that an SCC can be both a source and sink, if it is isolated. Convince yourself that calling *explore* on a node in a sink SCC will *only* mark all nodes in the sink as visited. Now consider G^R, the **reverse graph**,

[1]See appendix for definitions of meta-graph and SCCs if unfamiliar

which is just G, but with all edges reversed - meaning that all sources and sinks switch roles! If we run our DFS on G^R, then there will be some node v with the highest post number. v must lie in a source of G^R, meaning v must lie in a sink of G.

Now, if we call *explore* on v in G, we can identify every vertex in this sink SCC. If we delete both the sink from G and source from G^R, then we can look at the next highest post number of G^R, and use this to find another sink SCC of G, and so forth.

Eventually our working G and G^R will be empty, and we'll be done. At that point we can just run a linear time (in the number of edges) search through the edges to figure out the connectivity of the meta-graph.

Note we avoid discussion of data structures since we're not as interested in the implementation. However, the sum of all these steps takes only linear time!

□

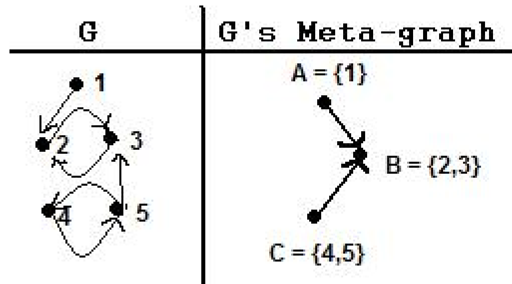

FIGURE 2. A directed graph G and its meta-graph. The SCCs are $A = \{1\}, B = \{2,3\}, C = \{4,5\}$. A and C are sources, B is a sink. If we were to run depth-first search on this graph, starting at vertex 1 and assuming we iterate through the ordered list of vertices, the (pre,post) numbers would be: 1 = (1,6), 2 = (2,5), 3 = (3,4), 4 = (7,10), 5 = (8,9).

We can now examine how DFS relates to 2-SAT. For every clause $(a \vee b)$ in the formula, draw the directed edges $(\overline{a}, b), (\overline{b}, a)$ (note these edges correspond to implications $\overline{a} \to b$ and $\overline{b} \to a$, both logically equivalent to $a \vee b$). We now have a directed graph that we can assign truth values to. A satisfactory assignment to the graph will have all paths evaluate to true. We can use this structure to help solve 2-SAT, but first a claim:

Claim 3.3. *In our constructed implication graph, if any strongly connected component contains some literal x and its negation $\overline{x}$, then there is no satisfying assignment to the proposed formula.*

Proof. Well, suppose that some SCC S contains x and $\overline{x}$. Then there is a path $(x, ..., \overline{x})$ and $(\overline{x}, ..., x)$ consisting of nodes only in S. We either assign $x = T$ or $x = F$ - if we assign $x = T$, then at some point in the path $(x, ..., \overline{x})$, we must have some edge (a, b), where $a = T$ and $b = F$, because $\overline{x} = F$. By construction, (a, b) corresponds to the implication $\overline{y} \to z$ or $\overline{z} \to y$. Recall these implications are logically equivalent to $y \vee z$, so our assignment results in a clause in our original

formula that is not satisfied (see Figure 2). Now if we assign $x = F$, then $\overline{x} = T$, and the same argument follows (but with the path from $\overline{x}$ to x). $\square$

Note that checking for a literal and its disjunction in a SCC can be checked in linear time.

We can actually strengthen the previous claim into being necessary. To do so, we prove the converse, and along the way, pick up an algorithm that will be our final piece in the algorithm for solving any instance of 2-SAT.

Claim. *In our constructed implication graph, if no strongly connected componnts contain some literal x and its negation $\overline{x}$, then there is a satisfying assignment to the proposed formula.*

Proof. In this proof, it will help to have an example to work through in order to gain understanding - Figure 2 (Right) can be a helpful example.

Suppose we have an implication graph G constructed from a boolean formula C, and no literal and its negation lie in the same SCC of G. Recall that the SCC generating algorithm pulls off sink SCCs from the working graph in order to determine the structure of the meta-graph. Assign all variables in the first such sink to true. There cannot be any implications that would falsify the formula, because all implications within the sink would be $T \to T$, and all implications coming into the sink would either be $T \to F$ or $F \to T$.

Note that due to a nice symmetry our construction of G gives us, the negations of the variables in the sink must lie in a source SCC of G. Also, the total number of SCCs must be even. Now, assign the negations in the source to false. Now delete the assigned sink and source from G. We can safely repeat this process till we are finished, because no literals and their negations lie in the same SCC, making it impossible to have an implication $T \to F$. Such an implication would imply there was an edge from a source SCC to a sink SCC.

Eventually we run out of SCCs, and we have a satisfying assignment to our formula.

$\square$

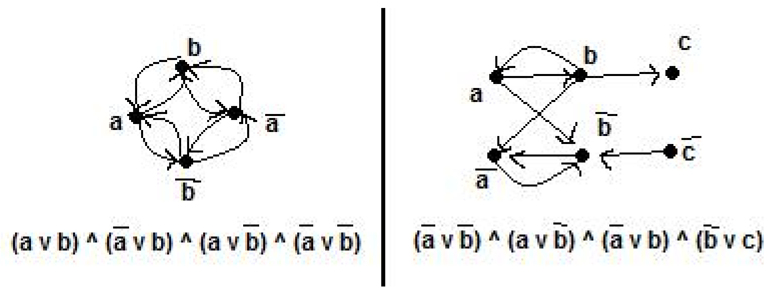

FIGURE 3. Left: An unsatisfiable formula and its implication graph. Right: A satisfiable formula and its implication graph. Notice the symmetry - where are the sink/source SCCs?

This shows that an implication graph has a satisfactory assignment iff none of its SCCs contain a variable and a negation! And our algorithm for solving 2-SAT is complete, and is entirely in linear time, once one actually implements it and manages the data.

Try to see this process from a high level - it's useful to try and abstract from all the details. We have built this up part by part - the depth first search, the SCC generating algorithm, and then the algorithm that applies claims 3.4 and 3.5. It turns out that 2-SAT is just a special case of a very important problem - SAT, which is the same problem as 2-SAT, except that clauses can have any finite number of literals. You can imagine how hard solving SAT is (in fact, 3-SAT is just as hard), as we will see.

Definition 3.4. We say an algorithm is **polynomial-time** or **efficient** if it is $O(n^k)$, for some constant k. A problem is **tractable** iff there exists a polynomial-time algorithm that solves all instances of it. A problem is **intractable** iff there is no known polynomial-time algorithm for solving all instances of it. It is possible that a currently intractable problem may have an efficient algorithm that solves it - discovering such an algorithm would effectively mean the problem is no longer intractable.

So how would you go about solving SAT? There's the naive brute force, $O(2^n)$ approach of trying every possible combination for the literal assignments. However, this means our possible set of solutions would grow exponentially as our number of different literals increased - namely, for some formula with $\{x_1, x_2, ..., x_n\}$, there would be 2^n possible combinations for the assignments! Even though you can *check* if any given solution is correct in polynomial time, it is very inefficient to *find* the solution!

SAT falls into a bucket with many other seemingly different, but related problems, as we will see.

4. The complexity classes P and NP, and reductions

Definition 4.1. A **decision problem** is any problem to which a solution can either be correct or incorrect, e.g. SAT: "Is this a satisfying assignment?", or some other possibly familiar problems: Euler Circuit: "Is this an Eulerian circuit?", Traveling Salesman Problem: "Is this a route that visits all cities under cost c?", and so forth.

Definition 4.2. We define **NP** (Nondeterministic polynomial time) as the set of all decision problems of which the validity of any proposed solution S to an instance I of the problem can be checked efficiently.[1]

Here are a few more decision problems:

Set Cover is the problem, given a budget b, and a collection of subsets C of some set S, to find a collection of at most b sets in C whose union is S. It usually deals with finite sets.

Dominating Set is the problem of, given an undirected graph G and a budget b, find a set C of at most b vertices such that all vertices in G are neighbors or members of vertices in C.

Definition 4.3. A problem is in **P** (Polynomial time) if it is in NP and is tractable. This definition implies P $\subset$ NP.

[1]The reader will be introduced to Turing machines, but NP's formal (not necessary for this paper, but perhaps interesting) definition is any decision problem that is efficiently checked by a deterministic Turing machine, and is efficiently solved by a nondeterministic Turing machine.

Where do problems that are intractable and in NP lay? To discuss such problems, it will be helpful to introduce the notion of *reduction*, a process that can sometimes be used to convert instances of one problem into instances another.

Definition 4.4. A problem A reduces to B if an instance I_A of A can be efficiently mapped to an instance I_B of B, and solutions of I_A and I_B can be efficiently transformed into solutions of the other. We sometimes write this as $A \leq_p B$.

Definition 4.5. A problem is **NP-complete** (NP-C) if it is in NP, and every problem in NP reduces to it in polynomial time. A problem is **NP-hard** (NP-H) if an NP-complete problem reduces to it.

This notation is an unfortunate consequence of history and admittedly can be confusing, but just keep the following relations in mind: P $\cup$ NP-C $\subset$ NP and NP-C $\subset$ NP-H. For example, Traveling Salesman Problem (TSP) with no budget is *not* an NP-complete problem (We can't efficiently verify if some proposed solution is the "best" solution), but it is an NP-hard problem. Another important idea to take away is that there are problems in NP $\setminus$ (P $\cup$ NP-C) (to be discussed briefly later), and that not all NP-H problems are in NP.

Reducibility need not be commutative. For example, 2-SAT reduces to SAT, but SAT does not reduce to 2-SAT (or else we'd have an efficient way of solving the intractable SAT, since all instances of SAT could be mapped to 2-SAT, which we know is efficiently solvable) In the case of NP-Complete problems, both problems have to reduce to each other - here is a worked out example.

Example 4.6. *NP-completeness of Dominating Set via reduction from NP-complete Set Cover*

When proving NP-completeness based off of a specific NP-complete problem (in this case, Set Cover), the reduction needs to only be performed from the NP-complete problem to the NP problem. However, one can take that a step further and prove equivalence (each instance of problem A maps to an instance of problem B, vice versa) of the problems by also doing the reduction in the other direction. This amounts to 6 small proofs, 2 for converting instances of the problems, and 4 for mapping solutions from one problem to another. We must also make sure both problems are decision problems, but that much is given in this example.

Moreover, in this proof, Set Cover is assumed to be NP-complete. To show Set Cover is NP-complete, we need the Cook-Levin theorem, which we will prove later.

Proof. (Reduction of Set Cover to Dominating Set) Consider an instance of Set Cover. Let $S = \{x_1, x_2, ..., x_n\}$, $C = \{S_1, S_2, ..., S_k\}$, let our budget be b. We want to convert our instance into a graph, so create a vertex s_i for each S_i, and a vertex p_j for each x_j. Draw edges between s_i and p_j if $x_j \in S_i$. Draw all possible edges between the s_i (k-clique for the graph theory familiar). If it exists, a dominating set of the new graph G will be a subset of the s_i (we say this because if for some reason a dominating set contains a p_j, we can just replace it with some s_i that is its neighbor, which will not affect the validity of the solution).

Now given some dominating set C, where $|C| \leq b$, each $s_i \in C$ corresponds to choosing some subset of S. And since C is dominating, the union of the subsets corresponding to the $s_i \in C$ must equal S, as required. If we are given some set cover, then we easily know how to pick the correct vertices in our dominating set graph, based on the indices of the set cover subsets. $\square$

Furthermore, we still need to transform instances of dominating set to set cover - this is because we are proving NP-completeness using specific problems. Doing this reduction will be good practice, and is easier than the direction we've shown. (Hint: What's a natural way you could think of a vertex and its neighbors in a graph?)

Also note, as of this time of writing, P $\cap$ NP-C $= \emptyset$. This introduces the question: does P = NP? The general consensus is no, but proving one way or the other will take considerable work. To prove P = NP, we would have to show an NP-complete problem is efficiently solvable. The NP-complete problems could all be proved if only one is shown to be efficient (why?). In short, the question "P = NP?" is still open, and is fundamental to computational complexity theory.

5. The Cook-Levin Theorem (NP-completeness)

You may have noticed that in proving the NP-completeness of a problem, we had to know a previous problem was NP-complete. How was the first problem proved to be NP-C if there were no known specific NP-C problems beforehand? The Cook-Levin Theorem proves this for us, by abstracting what an NP-C problem is. In order to understand the proof, we need an understanding of the Turing machine model and some first-order logic nuances.

Theorem 5.1. *(Cook-Levin Theorem): SAT is NP-Complete.*

Proof. Recall the definition of NP-complete. To show SAT is NP-C, first we need to show SAT is in NP, and second, that every problem in NP reduces to it in polynomial time.

Showing that SAT is in NP is relatively quick, since the question SAT poses is "Is there a satisfying assignment to this CNF formula?", and the time to check if a solution is valid is linear in the number of literals in the formula, as we need just check each clause for validity. Thus, SAT $\in$ NP.

To show the second part, we need to introduce a more general view of NP problems. Consider a problem P. Let the set of all instances of P have two subsets, A_T and A_F such that $A_F \sqcup A_T = A$, which are respectively the sets containing all instances of the problem to which there is an valid answer and those to where there is not. For example, if P is SAT, then A could be all SAT instances, A_T and A_F would respectively be all instances of SAT with/without a solution.

All NP problems can be generalized thusly: "Given an instance w of a problem A, is $w \in A_T$?". If we efficiently reduce this to SAT, we are done.

We can now introduce the idea of a Turing machine; a theoretical computing device.

Definition 5.2. *Turing Machine.* Imagine a possibly infinite row of squares. We refer to the row as the **tape**, and the squares as **cells**. Every cell contains one **symbol**, which could be a letter, a number, etc.

An **alphabet** Σ is a finite set of symbols. Every Turing machine has its own alphabet; one symbol from its alphabet is designated the **blank symbol** b, which is the only symbol allowed to appear in infinitely many cells.

A Turing machine has one **head**, which is positioned over exactly one cell. The head reads the symbol in the cell.

A **rule** always tells the head to rewrite the cell it is over, and to move to the left or right. A **state** q is a set of rules the Turing machine follows based on what

symbol its head reads. A Turing machine can have finitely many states, we call the **state set** Q. Each Turing machine starts in its special **start state**, q_0. Turing machines can end in any of the accept states F, which are states where the Turing machine stops its computation. Rules can have the Turing machine switch between states.

Each Turing machine has a function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R\}$, called the **transition function**, which takes in M's current state q and input symbol s, then returns a triple $(q', s', \{L, R\}^1)$, which M interprets as what symbol to overwrite the input symbol with, what state to change to, and what direction to move the head.

A Turing machine is **nondeterministic** (NDTM) if some input symbol can cause more than one possible outcome with some unknown chance. NDTMs can also solve intractable problems efficiently. This would technically mean δ isn't a function for an NDTM - but we ignore this. Otherwise, it is **deterministic**.

It is helpful to think of the state set as a directed graph. The Turing machine starts at vertex q_0, and has outgoing edges corresponding to input symbols the head reads. Each edge specifies what symbol to rewrite the current cell with, whether to move the head left or right, and the edge points to the next state to transition to.

We can now succinctly write a **Turing machine** M as a 3-tuple (Σ, Q, δ). The sets Σ, Q are finite, and there uniquely exists $b \in \Sigma$, $q_0 \in Q$, and there is some finite $F \subset Q$.

Definition 5.3. A **word** is concatenation of finitely many symbols from an alphabet Σ, possibly with repeats. The set of all possible words is denoted Σ^n. A **language** L is a subset of Σ^n.

For Cook-Levin, let $\Sigma^n = \{0, 1\}^n$. (An example word would be 011001). Let us define a language A_T to be the set of all satisfiable instances of some problem P in NP. Take an instance of this problem, w. By modeling the action of a NDTM on w as a boolean formula, if there is an accepting state, the formula will be satisfiable.

Suppose the input size is n. We want to show there is a polynomial-sized computation by a NDTM M, that we can represent as a boolean formula. By size, we mean number of steps the NDTM takes to accept or reject w. So we say the size is $p(n)$, and our tape will be $O(p(n))$ cells long.

We only need three types of variables to do this. Because we have an NDTM, we must account for differences in the computation with many variables. The types of variables must be enough to represent all possible computations, so we need variables to represent the symbols in cells, the position of the head, and the state of the Turing Machine. We introduce the following variables to do so: T_{ijk}, which is true if cell i contains symbol j at step k, H_{ik}, true if the head is over cell i at step k, and Q_{qk}, true if M is in state q at step k. Note that $-p(n) \leq i \leq p(n)$, $0 \leq k \leq p(n)$.

We use Big-O here to showing an upper bound on the order of magnitude for how many terms we have. We have about $2p(n)$ cells, at $p(n)$ possible times, and finitely many symbols, hence about $O(p(n)^2)$ of T_{ijk} - a similar argument follows for H_{ik}. With a finite number of states, and $p(n)$ steps, we have $O(p(n))$ of Q_{qk}.

We need a formula that encodes a lot of conditions! In encoding these formulas, for clarity we will use any logical connectives. This is okay, because conversion from any combination of variables and connectives can efficiently be transformed into CNF. We now go through the formulas one by one.

A valid start state should have M's head over the first symbol $w_1 \in \Sigma$ of the size n input. Moreover, M should be in the start state $q_0 \in Q$, and all other symbols on the tape should have the blank symbol $b \in \Sigma$. The below equation takes care of this. First, it takes a conjunction over all cells that are not part of the input word, to make sure they contain b. Then it forces the cells 0 to $n-1$ to be the respective symbols from the input word. The H and Q variables guarantee that M starts with its head over cell 0 and starts in the start state q_0.

$$\phi_{start} = (\bigwedge_{i \in \{-p(n)..p(n)\} \setminus \{0..n-1\}} T_{i,b,0}) \wedge T_{0,w_1,0} \wedge T_{1,w_2,0} \wedge ... \wedge T_{n-1,w_n,0} \wedge H_{0,0} \wedge Q_{q_0,0} \tag{5.4}$$

We need to make sure each cell has only one symbol, M is only in one state at any time, and the head is only in one position at a time. For (5.5), the key is understanding that the only way to invalidate the formula is if some $T_{i,s,j}$ and $T_{i,s',j}$ were both true, because we would have the implication $T \rightarrow F = F$. The same idea holds for (5.6) and (5.7), but we take these conjunctions over different ranges to account for the context the variables come from.

$$\phi_{cell} = \bigwedge_{0 \le |i|, j \le p(n)} [\bigwedge_{s,s' \in \Sigma, s \neq s'} (T_{i,s,j} \rightarrow \overline{T_{i,s',j}})] \tag{5.5}$$

$$\phi_{state} = \bigwedge_{0 \le t \le p(n)} [\bigwedge_{q,q' \in Q, q \neq q'} (Q_{q,t} \rightarrow \overline{Q_{q',t}})] \tag{5.6}$$

$$\phi_{head} = \bigwedge_{0 \le t \le p(n)} [\bigwedge_{0 \le |i|,|j| \le p(n), i \neq j} (H_{i,t} \rightarrow \overline{H_{j,t}})] \tag{5.7}$$

We also need to make sure that all of M's possible transitions are legal. That is, if the head is over cell i, then in the next stage of computation, only cell i's symbol may change, and the head must be over cell $i-1$ or $i+1$. Moreover, this change must be a legal move from some input to a state, we say R represents all valid conditions of the tape from time i to $i+1$ (for a given i, j), including the position of the head only moving one cell. It helps to think that if (5.8) is true, then "In the entire computation, every move made is valid, and thus the move is in R".

$$\phi_{move} = \bigwedge_{0 < |i|, j < p(n)} [\bigvee_{R} (T_{i-1,a_1,j} \wedge T_{i,a_2,j} \wedge T_{i+1,a_3,j} \wedge T_{i-1,a_4,j+1} \wedge T_{i,a_5,j+1} \wedge T_{i+1,a_6,j+1}] \tag{5.8}$$

For example, suppose that there is only one state, and the rule for reading an A is to write a B and move left. Let the apostrophe represent the head position. Then the left table is valid (and in R), and the right table is invalid (and thus not in R, and not checked for by our disjunction over R).

A	A'	A
A'	B	A

A	A'	A
A'	A	A

Also, we need to make sure that a cell j changes its contents from time i to $i+1$ only when the head was at j at time i.

$$\phi_{write} = \bigwedge_{0 \leq |i| \leq p(n), 0 \leq j < p(n), (s, s' \in \Sigma), s \neq s'} [(T_{i,s,j} \wedge T_{i,s',j+1}) \rightarrow H_{i,j}] \tag{5.9}$$

Lastly, the formula should only be satisfied if we end in an accepting state. The following will only be true if we are in an accepting state at time $p(n)$.

$$\phi_{accept} = \bigvee_{f \in F} Q_{f,p(n)} \tag{5.10}$$

Finally, we can just take the conjunction of the above formulas to be ϕ. If ϕ is satisfiable, then its solution maps to some sequence of computations that ends in an accepting state for w, so $w \in A_T$. If the computation ends in an accepting state, we only need to set the variables of the formula as the computation proceeded to find a satisfying assignment.

We just need to check this construction takes polynomial time. Note that as shown earlier, the size of our sets of variables is polynomial. The constructions of our formulas are also polynomial - (5.8) is constant in size, (5.9) has a constant sized inner conjunction with a $p(n)^2$ outer conjunction (to account for all possible positions and times), so the number of clauses is $O(p(n)^2)$. A similar idea works for (5.10), but this time the outer conjunction is only over a set of size $p(n)$, so the number of clauses in (5.10) is $O(p(n))$. (5.11) ends up as $O(p(n)^3)$. (5.12) is $O(p(n)^2)$ because we have to compare every $O(p(n)^2)$ 2×3 grid against a finite table of states. (5.13) is $O(p(n)^2)$ for similar reasons to others, and (5.14) is constant.

The encoding of our variables only takes $O(\log(n))$ time because we write them in binary. Thus the whole encoding takes $O(\log(n)) \cdot O(p(n)^3)$, or $O(\log(n)p(n)^3)$. And that proves Cook-Levin. Phew!

□

This proof gave the reason why we are able to find other NP-complete problems, and shows the power of polynomial-time reductions.

6. Valiant's Algebraic Complexity Classes

Valiant introduced an algebraic theory of complexity, related to the computation of polynomials. It has some relation to P and NP (and is interesting), so we discuss it briefly. For this section, we consider the natural numbers to include 0.

Definition 6.1. In this context, we say a function f is **p-bounded** if $f = O(n^k)$, where $k \in \mathbb{N}$.

For some intuition, the algorithms for verifying proposed solutions to NP problems are p-bounded, while the algorithms for solving proposed instances are not necessarily p-bounded.

Consider a set of indeterminates $X = \{X_1, X_2, ..., X_m\}$. An indeterminate can be thought of as an object which helps to represent a structure in a polynomial, but does not hold any value - e.g., x is an indeterminate in the polynomial x^2. A polynomial is multivariate if it consists of more than one indeterminate, e.g., xyz.

We denote the set of polynomials over the field k with indeterminates X as $k[X]$. We can talk about a few functions relating to polynomials in our field, so take some $f \in k[X]$. The function $v : k[X] \rightarrow \mathbb{N}$ takes a polynomial and returns the number of indeterminates in it. So $v(f) \leq m$, because $|X| = m$. The function

$deg : k[X] \to \mathbb{N}$ takes a polynomial and returns the highest degree of all of its terms, so $deg(x^2y + yz) = 3$.

Definition 6.2. Consider some sequence of polynomials $F = (f_n)_{n\geq 1}, n \in \mathbb{N}.$, where $f_n \in k[X]$. We say F is a p**-family** if the functions v and deg are p-bounded with domain F.

An example of v not being p-bounded is if $f_n = X_1 + X_1X_2 + ... + X_1...X_{2^{n-1}}$. An example of deg not being p-bounded is if $f_n = X_1^{2^n}$.

Consider the operations $k, +, \cdot$ - scalar multiplication, addition, and multiplication. For any $f \in k[X]$, we can obtain f by applying the three aforementioned operators to indeterminates in X. For example, $X_1^3 + 3X_1X_2$ is obtained with five operations: $(X_1 \cdot X_1) \cdot X_1 + (3 \cdot X_1) \cdot X_2$. Every operation has a cost of 1, so the cost of "computing" this polynomial is 5. An important thing to understand is that you can save on the cost by composing. That is, we can efficiently compute X_1^8 by the following steps: $X_2 = X_1 * X_1$, $X_3 = X_2 * X_2$, and finally $X_4 = X_3 * X_3 = X_2^4 = X_1^8$. Rather than multiplying X_1 by itself seven times, we finish in three steps.

The function $L : k[X] \to \mathbb{N}$ is called a complexity measure, and it outputs the minimum cost of computing some polynomial. So, $L(X_1^3 + 3X_1X_2) = 5$. It is worth knowing that each L pertains to a specific field k over a specific set of polynomials X. In this case, we call L a straight-line program.

Definition 6.3. If for some p-family F, the complexity measure L is p-bounded for domain F, then we say F is p**-computable**.

Note that being a p-family does not imply p-computability.

Here's an example of a p-computable family. Let us define the family of polynomials $MMSUM$ (multivariate monomial sum) where

$$MMSUM_n = \sum_{(i_1, i_2, ... i_n) \in \{1,..,n\}^n} X_{i_1} \cdot X_{i_2} \cdot ... \cdot X_{i_n}$$

$MMSUM_1 = X_1, MMSUM_2 = X_1^2 + 2X_1X_2 + X_2^2$, etc. The function v is p-bounded - $v(n) \leq n$, since every term in the sum for $MMSUM_n$ has at most n indeterminates. Also, $deg(n) \leq n$, in the case where a term of $MMSUM_n$ is some X_i^n. We just need to show that L is bounded for the polynomials in $MMSUM_n$. Observe that $MMSUM_n$ is equal to $\Pi_{j=1}^n (X_1 + X_2 + ... + X_n)$. So, we require $n-1$ additions to create what we will be taking the product of, and then $n-1$ multiplications to obtain $MMSUM_n$. Obviously this complexity is linear in the size of n, so $MMSUM$ is p-computable.

We introduce a definition to describe such families.

Definition 6.4. VP is defined as Valiant's class of p-computable p-families over the field k.

There are p-families that are not in VP. Consider the family of polynomials $INTSEQ$, where $INTSEQ_n$ is the sum of monomials which represent nondecreasing, positive integers of length n, where the integers are between 1 and n inclusive. So, $INTSEQ_2 = X_1X_2 + X_1X_1 + X_2X_2$. The indeterminates representing integers do not necessarily have to be in order. Also, note that $INTSEQ$ and $MMSUM$ are very similar families, but $MMSUM_n$ contains scalar multiples of its terms (which are also terms in $INTSEQ_n$).

Now, if there exists no p-bounded algorithm to calculate all such nondecreasing integer sequences, then there is something we can say about the family $INTSEQ$.

Claim 6.5. *$INTSEQ \notin VP$.*

Proof. To prove this, we will frame computing $INTSEQ_n$ in a combinatorial sense, and use this to show that a superpolynomial number of additions (the number of additions can't be represented as a polynomial function) is needed to compute $INTSEQ_n$. This will mean that the complexity measure L is not p-bounded, implying that $INTSEQ$ is not p-computable, and thus not in VP.

The unique terms in the polynomial $INTSEQ_n$ can be represented as nondecreasing integer sequences - for example, in $INTSEQ_2$, the term $2X_1X_2$ corresponds to $(1, 2)$, X_1^2 corresponds to (1,1). Increasing sequences can also be seen as placing 0 to $n-1$ balls in n boxes, where each ball corresponds to an increase. For example, if $n = 3$, then placing 2 balls in the first box corresponds to the sequence (3,3,3), placing 1 balls in the second box corresponds to (1,2,2), etc.

For example, in $INTSEQ_3$, the sequence (3,3,3) corresponds to the term X_3^3, the sequences (1,2,2) corresponds to the term (within a multiplicative constant) $X_1X_2^2$.

Coming back to our balls in boxes, let's now look at the general case of n boxes and k balls, where $0 \leq k < n$. Place these k balls in a row, and in between the balls, drop the $n-1$ "walls" needed to form n boxes. Since we can now view our balls in boxes as a sequence of walls and balls (e.g. $X_1X_2^2$ or (1,2,2), or WBW), the possible assignments of k indistinguishable balls to n distinct boxes can be seen as the orderings of k balls and $n-1$ walls, or taking the binomial coefficient $\binom{(n-1)+k}{k}$.

Now the sum

$$\sum_{k=0}^{n-1} \binom{(n-1)+k}{k}$$

is just the number of nondecreasing integer sequences of length n, with entries from 1 to n inclusive. This number equals the number of distinct terms in $INTSEQ_n$ (For intuition, verify this is true for $n = 1, 2, 3$.) . We still need to show this number of terms is not p-bounded based on n.

Consider the term of the sum when $k = n-1$, $\binom{2n-2}{k}$, or $\frac{(2n-2)!}{(n-1)!0!}$. This equals $(2n-2)(2n-3)...(n)$. The maximum degree of this polynomial increases as n increases, which means the number of terms in $INTSEQ_n$ is not p-bounded. This means we need a superpolynomial number of additions to construct $INTSEQ_n$, which means that L is not p-bounded for $INTSEQ$, and we are done. □

One of the foremost examples of a p-family not in VP is the Hamiltonian cycle polynomials, HC. We define HC_n as

$$HC_n = \sum_{\sigma n-cycle} \prod_{i=1}^{n} X_{i\sigma(i)}$$

which is the sum of all n-term monomials whose subscripts correspond to the mappings of a permutation σ on the symmetric group S_n that results in an n-cycle, e.g. $X_{1,2}X_{2,3}X_{3,1}$ would be in HC_3, but not $X_{1,1}X_{2,2}X_{3,3}$. If we treat the indeterminate $X_{i,j}$ as an entry of an adjacency matrix for a graph G, where $a_{i,j} = 1$ iff $(i, j) \in E(G)$, $a_{i,j} = 0$ otherwise, then the entire sum equals the number of

Hamiltonian cycles in the graph. For example, the 3-cycle, C_3 corresponds to the adjacency matrix,

$$\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

which will evaluate to 2, as expected - 1 for going around the 3-cycle, 1 for going around in the other direction. The family HC is said to be the **enumerator** of the NP-complete Hamiltonian cycle problem.

This is just the surface of Valiant's algebraic complexity classes - there are rough analogs of other concepts, such as reducability, NP-completeness, NP, and so on. We hope we have at least piqued an interest in some form of complexity theory - algebraic, or computational.

Appendix A. Propositional logic

In propositional logic, we call $\rightarrow$ the symbol for **implication**, $\wedge$ the symbol for **conjunction** ("and"-ing - "is everything true?") of literals, $\vee$ the symbol for **disjunction** (inclusive "or"-ing - "is at least one thing true?"). A **truth table** is a way to determine the truth values for a given expression, and below is the truth table.

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$
T	T	T	T	T
T	F	F	T	F
F	F	F	F	T
F	T	F	T	T

Some intuition behind the implication is that only $T \rightarrow F$ evaluates to F because both $T \rightarrow T$ and $T \rightarrow F$ having the same truth value would be a contradiction (just put this in in the context of your favorite if-then statement). We say $F \rightarrow T$ and $F \rightarrow F$ are vacuously true, meaning that since the antecedent p is false, we cannot infer anything about the consequent q, so we might as well evaluate the expression to T, or else the truth tables for implication and conjunction would be the same! (and that's no good.)

Hopefully this sheds light on how the construction of the graph for solving 2-SAT works, as well as evaluation of boolean formulas.

Appendix B. Graph theory

We need not delve very deep into graph theory, but for the unfamiliar, a **graph** G is defined as a tuple (V, E), where V is a set of **vertices** (often called **nodes**) and E is a set of **edges**, where an edge is just a tuple of vertices in V.

In the below graph G, $V = \{1, 2, 3, 4, 5\}, E = \{(1, 4), (4, 2), (2, 5), (5, 4), (3, 5)\}$ In particular, G is **undirected**, meaning the edges $(1, 4)$ and $(4, 1)$ are the edge and we draw it once. **Directed** graphs distinguish the two - in a directed graph, $(1, 4)$ would be a **directed edge** (arrow) pointing from 1 to 4. Luckily, intuition works well in graph theory - a **path** is a sequence of vertices constructed by visiting adjacent edges without repeating edges. An example would be (1,4,2,5,4). If there is a path between a vertex u and v, we say $u \sim v$. If a path starts and ends at the same vertex, the path is a **cycle**. If a cycle visits every edge exactly once, it is called an **Eulerian cycle** (EC). G has no EC (why?). A cycle that visits every vertex exactly once is a **Hamiltonian cycle**.

Now imagine the edges listed above are directed edges. Then G is directed, and has three **strongly connected components** (SCCs), where $\{\text{SCC}\} = \{S \subset V | (\forall u, v \in S)(u \sim v)\}$. So for our graph, $\{\text{SCC}\} = \{\{4, 2, 5\}, \{1\}, \{3\}\}$. A **meta-graph** treats each SCC as its own node, and has an edge between an SCC A and B if some vertex in A shares an edge with a vertex in B.

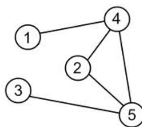

Figure 4. A graph

Acknowledgments. Special thanks to Yiwei She, my mentor, who in addition to supporting this endeavor, put up with my paper not ending up having much to do with abstract algebra or topology at all. Of course, special thanks to Peter May and other organizers of UChicago's 2011 REU, it's been a blast.

References

[1] Sanjoy Dasgupta, Christos Papadimitriou, Umesh Vazirani. Algorithms, 1st ed. McGraw-Hill, 2008
[2] Arijit Bishnu. "Lecture 6: Cook Levin Theorem" http://www.isical.ac.in/ arijit/cours-es/spring2010/slides/complexitylec6.pdf
[3] Wikipedia. Various articles on complexity theory for cross checking with other sources. http://en.wikipedia.org/
[4] Burgisser, Clausen, Shokrollahi. Algebraic complexity theory Springer, 1997