On the Computational Hardness of Graph Coloring

Steven Rutherford

June 3, 2011

Contents

1 Introduction

This paper will introduce the ideas of the hardness of computational problems. Specifically, it will use graph coloring as a case study in the computational hardness of combinatorial problems. Although I hope to make this as accessible as possible to those who have not been exposed to the ideas of the theory of computation, familiarity with the ideas of computation will make the material much simpler to understand.

I will begin by introducing the important ideas behind the Turing Machine and how the idea of the algorithm relates to the Turing Machine. I will then introduce decision problems, computational complexity of problems, and complexity classes [2] [1].
The aim of the paper is to give an intuition for the two best understood complexity classes, **P** and **NP**, as well as an intuition for the hardness of solving problems with in **NP**.

2 Turing Machine

Definition 1. (Turing Machine) A *Turing Machine* is a 7-tuple, $(Q, \sigma, \gamma, \delta, q_0, q_{accept}, q_{reject})$, where Q, σ, γ are finite sets, and

1. Q is the set of states
2. σ is the input alphabet (which does not include the blank space)
3. γ is the tape alphabet ($\sigma \subsetneq \gamma$, and γ includes the blank space)
4. $\delta \times \gamma \to Q \times \gamma \times L, R$ [1]
5. $q_0 \in Q$ is the start state
6. $q_{accept} \in Q$ is the accept state
7. $q_{reject} \in Q$ is the reject state ($q_{accept} \in q_{reject}$ [2]

Woah! How many undefined terms can I give in a single paragraph? Indeed, this is the formal definition of the Single Tape Turing Machine. Although precise, this definition is very distracting for our purposes. As such, we will distance ourselves from the specifics of this definition, and instead use a simpler, more intuitive definition, which is equivalent for our purposes[2].

Before I introduce a less clunky intuition of the Turing Machine, I first need to introduce the most fundamental problems that Turing Machines try to solve!

Definition 2. (Language) A **language**, L, is a set of strings, not necessarily finite in cardinality, which contains only finitely many characters.

[1] L, R are the left and right directions, which the tape head moves after a given operation

[2] If you are wondering what all of this means, I highly recommend reading [2], which does a very good treatment of the details of the Turing Machine

For example, a language could be as simple as the language of integers encoded in binary, or as complex as the encodings of graphs that are 3 colorable.

Definition 3. (Decision Problem) A **decision problem** is the computational problem of deciding whether or not a string is within some language.

The fundamental ideas behind the Turing Machine (TM) are as follows:

1. TMs (try to) answer decision problems
2. TMs might or might not halt at all, but if it does halt, it gives an answer (Accept/yes or reject/no)
3. TMs follow a fixed length algorithm
4. TMs have an infinite block of memory, which initially only contains the input
5. TMs can read and write to a single space in memory in a single operation[3]

Although this is not precisely the same as a Turing Machine, it is equivalent in what it can compute, and how fast it can compute it[4]. Interestingly enough, almost all reasonable models of computation are equivalent in this sense[1][5].

One limitation is that not every decision problem can be solved. It is not necessarily the case that a TM will always stop/halt. A Turing Machine *decides* a language if it accepts every string from the language, and rejects every string not in the language. A language is **decidable**, if some TM decides it. A language is **undecideable** if no TM exists that decides it.
Many undecidable languages exist, for example, the language HALT is undecidable, where halt is the language of TM, input pairs, such that the TM halts on the given input[2]. Although the question of which decision problems are decidable or undecidable is interesting, we will only concern ourselves with decidable problems. Specifically, we will analyze the *computational complexity* of decision problems.

3 Complexity Classes

Complexity classes are useful in classifying the hardness of solving decision problems. Many complexity classes exist, but I will only introduce two in this paper: **P** and **NP**.

[3]An experienced reader will quickly realize that this simplified notion of memory access is ill-defined. The intuitive notion shared here is the idea behind the Random Access Machine. Technically, it is necessary to have a way of specifying what piece of memory will be accessed next. I will not explain the nuances of how this can be done. I point the curious reader to[1]

[4]They are equivalent in the sense that this machine solves decision problem in polynomial time, iff the classic Turing Machine can. Similarly, this machine solves decision problems in non-deterministic polynomial time, iff the classic Turing Machine can. More precisely, if this machine can answer a decision problem in $T(n)$, a typical Turing Machine can answer the problem in $T(n)^2$ time. These terms will be defined later.

[5]The only notable exception being the models of computation that rely upon quantum mechanics.

4 Polynomial Time (P)

Definition 4. (**DTIME**) Let T: $\mathbb{N} \to \mathbb{N}$ be some function. A language, L, is in the complexity class **DTIME**$(T(n))$ if and only if there is a Turing Machine that decides L in $c \cdot T(n)$ operations, for some positive constant c.

The Turing Machine defined above is deterministic; given the same input it will always proceed in the same way, and only in one way. The complexity class **DTIME** corresponds to the set of languages that can be solved by deterministic Turing Machines in the given number of operations (running time), which depends on the length of the input.

Definition 5. $\mathbf{P} = \cup_{c>0}(DTIME)(n^c)$

Intuitively, **P** corresponds to the set of languages for which membership can be checked in a number of operations, equivalently time, that proportional to a polynomial in the length of the input. Here, we will analyze the language of colored graphs and prove that the language is the complexity class P.

4.1 COLORED-GRAPH

Definition 6. Let $< V, E, C >$ be the encoding of a 3-tuple V,E,C, where V,E are the sets of vertices and edges of a graph, respectively, and $C : V \to \mathbb{N}$ defines the coloring of the vertices of the graph (The naturals being the colors). The language **COLORED-GRAPH** contains all $< V, E, C >$ such that no two adjacent edges are of the same color.

Theorem 1. *COLORED-GRAPH* $\in$ ***P.***

Proof. This proof will be by construction.

Input: Tuple $< V, E, C >$
Output: Is C a valid coloring of (V,E)
foreach *Edge* $e \in E$ **do**
  Lookup v_1 and v_2 be the two ends of e.
  if $C(v_1) = C(v_2)$ **then**
    REJECT
  end
end
ACCEPT

I claim that this algorithm rejects iff the coloring is invalid and accepts iff the coloring is valid, and does so in time polynomial in the length of the input. Indeed, the algorithm checks if the endpoints of every edge are of different colors. If any given edge has two ends of the same color it rejects, and otherwise it accepts. Also, checking the validity of an edge takes only a constant amount of time (c): it only requires looking up the corresponding end points, querying the function C twice, and checking equality. This only needs to be done at most $|E|$ times, which is at most linear in the length of the input (depending on the encoding style of the graph). As

such, the graph coloring can be verified in time $T(n)$ $c \cdot n$, which is polynomial in n. Thus, COLORED-GRAPH $\in$ **P**. □

Sure, the problem of verifying graph colorings might be easy, but what about coming up with them? Trivially, it is possible to color a graph G=(V,E) with —V— colors, but how hard is it to decide if an arbitrary graph can be colored with some fixed number of colors?
It turns out this problem is relatively easy for only two colors.

4.2 2COLOR

Definition 7. Let graph $G =< V, E >$. The language **2COLOR** is the language of such $< V, E >$ where there exists a valid 2-coloring $C : V \to \{1, 2\}$.

Theorem 2. *2COLOR* $\in$ ***P***.

Proof. Once again, I will show this by construction.

Input: Tuple $< V, E >$
Output: (V,E) has a valid coloring
Copy V into V'
while *V' is not empty* **do**
  if *All vertices in V' are not colored* **then**
    Pick some $v \in V$
    $c(v) \leftarrow 1$
  end
  while *Some edge* $v \in V'$ *is colored* **do**
    foreach *Vertex adjacent to v, v'* **do**
      if *v' is colored and* $C(v') = C(v)$ **then**
        REJECT
      end
      else if *v' is not colored* **then**
        Color v' the opposite of v
      end
    end
    Remove v from V'.
  end
end

The intuition behind this algorithm lies in the fact that if a valid 2-coloring existed, every vertex would be the opposite color of its neighbors. As such, one can simply assign one node a color, and then follow through with the implications of the color of this node on its neighbors, and its neighbors neighbors... Eventually the connected subgraph that contained that node will either have a valid coloring, or the graph as a whole does not have one. Repeating this further, with other uncolored nodes, eventually tries to color every disjoint subgraph, and thus either finds a valid coloring of the entire graph, or declares the graph not 2-colorable.
In this algorithm each node will be used to try to color its neighbors only once, and

each node has at most $|V|$ neighbors, thus the algorithm runs in at most $|V|^2$ time, which is polynomial in the length of the input.
Thus, 2COLOR is in **P**. □

5 Nondeterministic Polynomial Time (NP)

But what about the generalized problem of finding k-colorings? The case of 2-coloring was simple. If a valid 2-coloring existed, then every vertex and its neighbors were of opposite colors. In the case of generalized k-coloring, life is not so simple. Indeed, if an efficient[6] algorithm for generalized k-coloring existed, then many difficult, yet interesting, combinatorial problems could also be solved efficiently. Which problems? All of the problems within the complexity class NP.

Definition 8. (Polynomial Verifiability) Let L be a language. Membership in L is said to be **polynomially verifiable** iff there exists a language $L' \in P$ such that $L = x|\exists y(< x, y > \in L')$.

The intuition behind this definition is much more clear than the definition itself. Equivalently, and more intuitively, membership in L is polynomially verifiable iff every string, x ∈ L has some 'proof' (typically called a certificate or witness), y, that allows the membership of x to verified in polynomial time. For example, consider the problem of k-colorability, COLOR . Each graph $G =< V, E >$ in COLOR has a corresponding valid coloring function, C, which uses at most k colors. As we proved earlier, checking if a coloring, $< V, E, C >$, is valid can be done in polynomial time, and as such, membership in COLOR is polynomially verifiable by verifying the validity of the coloring, and the fact that it used at most k colors.

Definition 9. (NP) A language L is in the complexity class **NP** iff membership in L is polynomially verifiable.

Very little is known about the relationship of this complexity class with others. Resolving whether or not $P = NP$ is one of the seven Clay Millennium Prize problems. It is entirely consistent with our current knowledge of complexity theory that every problem in NP is also in P. Showing the opposite is quite simple. Almost trivially, membership in any language in P can be polynomially verified with absolutely no proof by simply running the polynomial time TM on the string in question.

Theorem 3. $P \subseteq NP$.

It is conjectured that $P \neq NP$. Why? A large set of challenging problems, the most challenging in NP, have been shown to be equivalent to each other, in the sense that each of those problems is in P iff all problems in NP are in P. These are the so-called NP-Complete problems.In order to rigorously define NP-Completeness, we need the concept of computable functions and polynomial time reductions.

[6]Polynomial time.

Definition 10. Let A and B be languages, let $f : A \to B$ be some function, and let M be a Turing Machine. M computes f, if, given $a \in A$, M runs for some time $T(n)$, where n is the length of a, writes the corresponding $f(a) \in B$ to the tape, and then halts. A function is computable if some Turing Machine computes it. A function is $T(n)$-computable if some Turing Machine computes it in time $T(n)$.

Definition 11. A language A is **polynomially reducible** to a language B if there is a polynomial time computable function f, such iff $\forall x \in A$ then $f(x) \in B$. The is denoted as $A \leq_P B$.

Definition 12. A Language L is **NP-Hard** if it is at least as hard as the hardest problem in NP. That is, that every problem in NP is polynomially reducible to L. A language L' is **NP-Complete** if it is in NP and also NP-Hard.

Essentially, a language L is NP-Hard if it can be used to solve all of the problems in NP quickly.

Theorem 4. *If L is NP-Complete and L* $\in$ ***P****, then* ***P=NP***

Proof. If L is NP-Complete, then every language in NP is polynomial reducible to L. If L is also in P, then all of the languages in NP can be decided in polynomial time, because the membership question can be polynomially reduced to the membership question of L. Thus, all languages in NP are also in P. □

With this terminology, it is now possible to introduce the hardest problems in NP.

5.1 SAT

Definition 13. A Boolean variable is a variable that can only assume the values true and false. The Boolean operations AND, OR, and NOT are represented by $\wedge$, $\vee$, and $\not$respectively. A Boolean function, f is an expression made up of Boolean variables, Boolean operations, and parentheses, such that $true, false^n \to true, false$, where n is the number of Boolean variables. A Boolean formula is satisfiable there exists an input to f, such that f evaluates to true; such an assignment of the Boolean variables is called a satisfying assignment. The language of satisfiable Boolean formulas is SAT.

Theorem 5. *(Cook-Levin theorem) SAT is NP-Complete.*

Proof Intuition 1. *It should be reasonably clear why SAT is in* ***NP****. Every satisfiable Boolean function has a satisfying assignment, and no unsatisfiable Boolean function has one. The satisfying assignment can be used as a witness to the satisfiability of the function by simply evaluating the function on the satisfying assignment, and checking if the result is true.*
The reducibility of every language in NP to SAT is clearly the more challenging piece. Despite seeming daunting, the proof is reasonably straightforward, but it requires a lot of manipulation of the definition of a Turing Machine. The main idea is to design,

in polynomial time, a Boolean function that simulates an arbitrary Nondeterministic Polynomial time machine on some input, that returns true if the simulated machine accepts, and returns false if the simulated machine rejects[2].

The exiciting thing about this proof is that it bootstraps our ability to show that problems are NP-Hard. If there exists a polynomial time reduction from SAT to another language (or for that matter from any NP-Hard problem to another problem), then that language is then NP-Hard. Next, I will cite another result from Sipser, and use this result to prove the NP-Completeness of COLOR.

Definition 14. Conjunctive normal form is format of boolean functions, such every variable within a clause is connected by the binary OR operation, and each clause is connected by the AND operation. Let **3SAT** be the problem of deciding if a boolean function in conjunctive normal form with only 3 variables per clause is satisfiable.

Proof Intuition 2. *This can be proven by polynomially reducing SAT to 3SAT. This can be done by breaking up nested clauses, converting the SAT instance to conjunctive normal form, and then reducing the size of the clauses by breaking them up into smaller clauses, which are glued together by dummy variables that do not help the satisfiability.*

5.2 3COLOR

Thus, we are ready to prove the NP-Completeness of COLOR. We will do so by first proving that the problem of deciding if there exists a valid 3-coloring, 3COLOR, is NP-Hard, and then reducing COLOR to 3COLOR. The proof will rely upon producing a *gadget*, a structure that simulates the properties of a better understood problem within a poorly understood one.

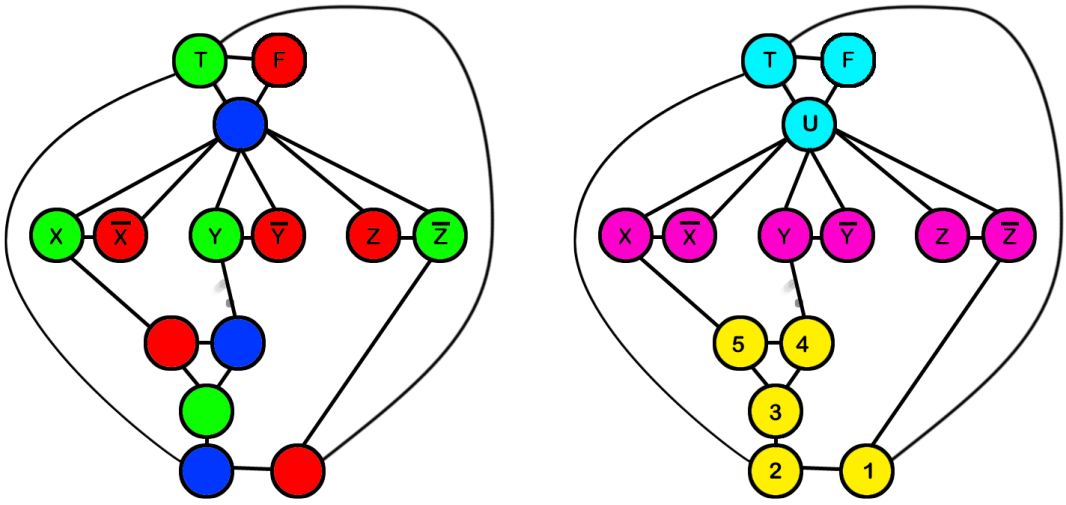

Figure 1: Caption of subfigures **??**, **??**

Theorem 6. *3COLOR is NP-Complete*

Proof. There are two main pieces to this proof. First, constructing the gadgets that allow us to create graphs that correspond to Boolean functions, and then showing that these graphs are in 3COLOR iff the given Boolean function is in 3SAT.

Figure (a) shows the basics of the construction. The three cyan nodes are the nodes that map colors back to the corresponding values of the Boolean variables. The magenta nodes represent the Boolean variables. Note that every magenta node is adjacent to the unnamed cyan node, and that each node corresponding to a variable is adjacent to its negation. As such, all of the magenta nodes can only have colors corresponding to true or false, and that each of the boolean variables has the opposite color of its negation.
The yellow nodes forms the gadget that is the crux of this proof. Every such gadget is identical, except in how it is connected to the Boolean nodes. This example of the gadget corresponds to the Boolean function $(x \vee y \vee \neg z)$.
The gadget is 3-colorable, iff the corresponding clause in the given Boolean formula is satisfied. More specifically, it is three colorable iff the Boolean vertices that are not all colored the same color as the node that corresponds to false.
If all of the Boolean vertices connected to a given gadget are all of the color corresponding to false, then that pins node five as being the color of U. Thus, 2 must be the same Color as F, and 3,4,5 are therefore adjacent to edges of the color of F. Yet, 3,4,5 are each adjacent to one another, but none of them can be the color of F, thus these three node cannot be colored.
Yet, if there is a satisfying assignment, each gadget must be adjacent to at least one Boolean vertex that is the color of T. As such, 3,4,5 will not all be adjacent to nodes of the same color, and thus three colors can be distributed across them, making 3-coloring possible.
The presence of gadgets corresponding to clauses constrains the colorability of the graph in the same way that clauses constrains the satisfiability of the corresponding Boolean function. As such, the constructed graph is 3-colorable iff the Boolean function is satisfiable.
Because this construction requires only local knowledge of the Boolean function (what variables are in each clause), and basic knowledge of the remainder of the graph, it can be created in polynomial time.

Combining the polynomial nature of the reduction with the correctness of the reduction and the fact that 3COLOR was shown to be in NP earlier in the paper, 3COLOR is thus NP-Complete.

COLOR is simply a generalization of 3COLOR, which contains 3COLOR as a special case, thus we have,

Theorem 7. *COLOR is NP-Complete.*

□

References

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[2] Michael Sipser. *Introduction to the Theory of Computation*. International Thomson Publishing, 1st edition, 1996.