# CME 305: Discrete Mathematics and Algorithms

**Instructor: Professor Aaron Sidford (sidford@stanford.edu)**

**February 27, 2018**

## Lecture 14 - P v.s. NP[1]

In this lecture we start Unit 3 on NP-hardness and approximation algorithms. Up to this point in the class our focus has been on developing fast algorithms for precisely solving combinatorial problems. At times we have touched upon problems which we said were more computationally difficult or provided fast algorithms for approximating the answer to a problem. However, we did this without any formal justification of the difficulty of these problems. In this unit we discuss techniques for identifying problems that are difficult to solve exactly and approximately solving them when exact solutions seem computationally infeasible to come by.

In this lecture, we start the unit by introducing a notion of problem hardness known as NP-hardness and a broad prevalent class of hard problems known as NP-complete problems. In the next lecture we build upon this to classify the complexity of a number of problems by showing that they are NP-complete.

## 1 Problem Difficulty

We begin this unit by discussing how we might hope to establish the difficulty of a problem and motivating the notion of difficulty we will work with. Up to now our focus has been on "easy" problems, i.e. problems that we can obtain fast running times for. Some examples include the following:

- **Global minimum cut** - we saw that we can compute this in polynomial time deterministicaly by solving for the maximum $s$-$t$ flow between any $s$ and every $t$ and that faster algorithms could be achieved by employing randomized algorithms.
- **Minimum weight spanning tree (MST)** - we saw how to compute this in nearly linear time by using the greedy algorithm.
- **Bipartiteness** - we saw how to check if a graph is bipartite using breadth first search (BFS).
- **Shortest path** - we saw how to compute the shortest path in a graph in nearly linear time by variants of the reachability algorithm.

Here our notion of "easy" was that of being able to solve the problem in polynomial time. While each problem had a trivial exponential time algorithm we showed that we could obtain a much faster polynomial time algorithm and in some cases closer to nearly linear time. As we discussed in the beginning of the course, while certain polynomial running times may be prohibitively expensive in practice for large enough instances the classification of a problem as polynomial time solvable is in many cases a good indicator that fast practical algorithms may be achievable.

In contrast to these "easy" problems, there are is a very broad class of combinatorial problems for which we have exponential time algorithms and yet, not significantly faster or polynomial time algorithm is known. For example, even slight modifications to the problems above induce problems for which we do not know of polynomial time algorithms. A few examples include the following.

[1] These lecture notes are a work in progress and there may be typos, awkward language, omitted proof details, etc. Moreover, content from lectures might be missing. These notes are intended to converge to a polished superset of the material covered in class so if you would like anything clarified, please do not hesitate to post to Piazza and ask.

- **Maximum cut** - in this problem we have an undirected graph and we wish to compute the set of vertices such that the number of edges cut is largest. This the problem induced by flipping form minimization to maximization in the minimum cut problem.
- **Minimum spanning $k$ connected graph** - in this problem we have a weighted graph undirected graph and wish to compute the subgraph of minimum weight that is $k$ connected, i.e. at least $k$ edges have to be removed to disconnect this graph. For $k = 1$ this problem is exactly the MST problem, however for $k > 1$ this problem becomes hard
- **$k$-colorability** - in this problem we have a undirected unweighted graph $G = (V, E)$ and wish to assign every vertex one of $k$ values, i.e. find $l : V \to [k]$ such that for $\{i, j\} \in E$ it is the case that $l(i) \neq l(j)$, i.e. for every edge each endpoint is assigned a different color. Note that for $k = 2$ this problem is exactly the problem of checking if a graph is bipartite, however for $k > 2$ this problem becomes hard.
- **Longest path** - in this problem we have a directed graph $G = (V, E)$ and for two vertices $s, t \in V$ we wish to compute the simple path from $s$-$t$ with the most edges (i.e. the longest simple path).

Now how would we show these problems are hard? For each of these problems the best known running time is exponential and therefore we could simply try to prove that these problems actually require exponential time to solve. Even easier, if we are classifying problems as easy if they are polynomial time solveable, we could more simply just try to prove that these problems are not solvable in polynomial time.

While this is a fine and natural approach. Unfortunately, the problem of proving such polynomial time lower bounds on these sorts of problems is incredibly difficult. There are numerous problems like the ones listed above for which no polynomial time algorithm is known and there has been extensive research for decades attempting to provide lower bounds on the running time need to solve these problems. Unfortunately, after significant effort the current best time lower bound results are not even able to rule out that these problems are solvable in nearly linear time, much less polynomial time.

So what should we do in the face of this difficulty? We would still like machinery to identify problems as difficult or in some way possibly separate from easy polynomial time algorithms we know. Ideally, such machinery so when faced with a new problem we can quickly and easily build some evidence for whether it falls in the class of easy problems or hard problems are something else.

The classic approach towards solving this problem is that rather than directly attempt to show the hard problems are difficult, instead we might hope to show that they are equivalent to each other or equally difficult in some way. This would allow us to at least focus our study of these difficult problems on a few canonical problems rather than each individually. It would also allow us to better identify these hard problems should we encounter them in theory and practice and such equivalence machinery could even possibly allow us to use algorithms for one problem to inform the design of algorithms for another. This is the approach we will take in this class and in the next section we formally introduce the notion of equivalence we will consider.

## 2 Polynomial Time Reducibility

In light of the discussion in the previous section to reason about possibly computationally difficult problems we wish to have a means of comparing different problems and possibly showing that they are equivalent. While there are many such notions one could consider the one we will focus on in this class is that of *polynomial time reducibility* formally defined below.

**Definition 1** (Polynomial Time Reducibility)**.** We say problem $X$ is polynomial time reducible to problem $Y$, denoted $X \leq_p Y$ if and only if there exists an algorithm for solving $X$ in in polynomial time plus the

time needed to solve $Y$ a polynomial number of times on polynomially sized input. We define $X =_p Y$ to denote the condition that $X \leq_p Y$ and $Y \leq_p X$.

Intuitively $X \leq_p Y$ should be interpreted as roughly saying that $Y$ is at least as hard $X$ up to polynomial time. Since our goal is to reason about whether or not problems are possibly polynomial time solveable, this definition is natural. The statement $X \leq_p Y$ is saying that ignoring the impact of polynomial factors $X$ is easier than $Y$ and $X =_p Y$ is saying that these problems are equivalent. To gain further intuition on this notion consider the following easy lemmas.

**Lemma 2.** *If $X \leq_p Y$ and $Y$ can be solved in polynomial time then so can $X$.*

*Proof.* Since $X \leq_p Y$ there is an algorithm that solves $X$ in polynomial time plus the time needed to solve $Y$ a polynomial number of times on polynomial sized input. Simply running this algorithm using a polynomial time algorithm for $Y$ yields a polynomial time algorithm for $X$ since a polynomial of a polynomial is a polynomial. □

**Lemma 3.** *If $X \leq_p Y$ and $X$ cannot be solved in polynomial time, then neither can $Y$.*

*Proof.* This is simply the contrapositive of the previous lemma. □

These lemmas further motivate the utility of polynomial time reductions in understanding the class of polynomial time solvable problems. The lemmas imply that if $X =_p Y$ then either both $X$ and $Y$ are solveable in polynomial time or neither of them are. Consequently for the sake of understanding whether or not problems are equivalent up to polynomial time, we simply need to provide polynomial time reductions between them. Moreover, as we show in the following lemma, polynomial time reducibility is transitive so we by providing polynomial time reductions between problems we can hope to identify a broad class of problems that are polynomial time reducible to each other. Therefore, while it might be difficult to formally separate the hard problems we have discussed from polynomial time solvable problems, we can at least possibly show that they are either all polynomial time solveable or none are.

**Lemma 4.** *If $X \leq_p Y$ and $Y \leq_p Z$ then $X \leq_p Z$.*

*Proof.* By assumption we can solve $X$ in polynomial time plus the time needed to solve $Y$ a polynomial number of times on polynomial sized input and in turn we can solve $Y$ in polynomial time plus the time needed to solve $Z$ a polynomial number of times on polynomial sized input. By simply running this algorithm for $X$ implementing each solve of $Y$ with the algorithm that reduced $Y$ to $Z$ yields an algorithm for solving $X$ in polynomial time plus the time needed to solve $Z$ a polynomial number of times on polynomial sized input, where here we are leveraging the fact that a polynomial of a polynomial is a polynomial. □

### 2.1 Example - Vertex Cover and Independent Set

As a quick example of polynomial time reducibility consider the following natural problems on an undirected graph.

**Definition 5** (Vertex Cover)**.** For undirected $G = (V, E)$ a set $S \subseteq V$ is a *vertex cover* if and only if every edge $e = \{i, j\} \in E$ has at least one of its endpoints in $S$, i.e. $|e \cap S| \geq 1$. The *minimum vertex cover problem* asks to find a vertex cover with a minimum number of vertices.

Note that clearly $V$ is a vertex cover of a graph, so finding a vertex cover is always trivial. Furthermore, the minimum vertex cover can always be found in exponential time simply by enumerating over all sets and

checking if they constitute a vertex cover and then outputting the one of minimum size. However, there is no polynomial time algorithm known for this problem.

**Definition 6** (Independent Set)**.** For undirected $G = (V, E)$ a set $S \subseteq V$ is an *independent set* if and only if every edge $e = \{i, j\} \in E$ it is not the case that both its endpoints are in $S$, i.e. at least one of its endpoints is not in $S$ or equivalently, $|e \cap S| \leq 1$. The *maximum independent set problem* asks to find an independent set with a maximum number of vertices.

Similar to vertex cover, note that clearly $\emptyset$ is an independent set in a graph, so finding an independent set is always trivial. Furthermore, the maximum independent set can always be found in exponential time simply by enumerating over all sets and checking if they constitute an independent set and then outputting the one of maximum size. However, there is no polynomial time algorithm known for this problem.

Even though both of these problems are suspected to be hard we can easily show that they are polynomial time reducible to each other.

**Lemma 7.** *Minimum Vertex Cover* $=_p$ *Maximum Independent Set.*

*Proof.* Given $G = (V, E)$ we claim that $S \subseteq V$ is a vertex cover if and only if $V \setminus S$ is an independent set. To see this, let $e \in E$ be arbitrary. Note that trivially $|e \cap S| + |e \cap (V \setminus S)| = 2$. Consequently, $|e \cap S| \geq 1$ if and only if $|e \cap (V \setminus S)| \leq 1$ and $S$ is an independent set, i.e. $|e \cap S| \geq 1$ for all $e \in E$, if and only if $|e \cap (V \setminus S)| \leq 1$ for all $e \in E$, i.e. $V \setminus S$ is an independent set.

Consequently, to compute the minimum vertex cover we can simply compute the maximum independent set and return its complement as the minimum vertex cover and therefore Minimum Vertex Cover $\leq_p$ Maximum Independent Set. Similarly, compute the maximum independent set we can simply compute the minimum vertex cover and return its complement as the maximum independent set and therefore Maximum Independent Set $\leq_p$ Minimum Vertex Cover. □

## 3 NP-Completeness

Now, given this notion of polynomial time reducibility, what we would like to do is show that a broad class of commonly occurring problems for which we do not have polynomial time algorithms are polynomial time reducible to each other. However, to do this what we would like to have is a slightly clearer characterization of this class of hard problems, rather than just those reducible from some particular problem. Here we formally introduce this class of hard problems known as NP-complete problems.

### 3.1 Decision Problems

Before introducing this class of problems we wish to simplify the discussion of algorithm problems more broadly. Up to this point our focus has been on the discussion of *optimization problems*, i.e. problems where we wish to minimize or maximize some quantity, and on *search problems*, where we wish to output the best answer like a subgraph or a cut or a coloring. We will find it more convenient to instead focus on *decision problems*, i.e. problems where the answer is just "yes" or "no" or "true" or "false."

For example, rather than considering the minimum vertex cover or maximum independent set problems we will consider the decision problem variants of "does $G$ have a vertex cover of size $\leq k$" or "does $G$ have an independent set of size $\geq k$." Furthermore, rather than considering the problem of finding the maximum cut in a graph or finding the 3-coloring of a graph we will focus on the decision variants, "does $G$ have a cut of size $\geq k$" or "does $G$ have a 3-coloring?"

Note that we will not lose much typically by focusing on decision problems rather than search or optimization problems as the latter are often polynomial time reducible to the former. For example if we want to know the size of the maximum independent set or maximum cut we can simply binary search on $k$ to find it with only a $O(\log n)$ loss in running time. Similarly, if we want to find the actual maximum cut or maximum independent set we can simply make a a guess at whether or not a vertex is in the independent set or two vertices are in the same side of the maximum cut and see if the value of the problem changes, similar as we did to reduce checking if a graph has a perfect matching to finding a maximum matching. This causes only a polynomial loss in running time.

### 3.2 Decision Problem Formalism

We focus on decision problems as it allows us to simplify some of the reduction theory and notation regarding the specification of a problem. Moreover, as we discussed in the previous subsection we can do this without loss of generality.

Formally, we can think of an instance of a decision problem as simply a sequence of ones and zeros, that we call a *binary string*, i.e. $s \in \{0,1\}^k$ where we use $|s| = k$ to denote the length of the string. This binary string encodes the input to the problem, for instance the graph and $k$ for maximum independent set. Now since the output of a decision problem is simply "yes" or "no" we can simply denote the set of inputs corresponding to a "yes" output as some $X$ that is a subset of all strings. Note that this $X$ fully characterizes a decision problem. For input binary string $s$ we have $s \in X$ if and only if the output on $s$ should be "yes" and $s \notin X$ if and only if the output on $s$ should be "no." Consequently, we can think of a decision problem as just determining or *deciding* whether or not an input binary string is in $X$ and we can simply define the problem as this set $X$.

With this formalism in place we can formally define $P$, the set of decision problems solvable in polynomial time as follows:

**Definition 8** ($P$)**.** A decision problem $X$ is polynomial time solvable, denoted $X \in P$, if and only if there is an algorithm that given any binary string $s$ can compute or decide if $s \in X$ in time polynomial in $|s|$.

Note that how the input is represented is important for the definition of $P$. For example, if we encoded an instance of maximum cut by writing down both the graph and every possible cut as the string, then this string would have length exponential in the size of the graph and deciding if this graph has a large cut would be trivially linear time. Consequently, for the problems we consider we will typically assume they have the natural representation of specifying things like a graph in $\tilde{O}(n+m)$ bits or a number in $\tilde{O}(1)$ bits.

### 3.3 NP-Hard Problems

We now have everything we need to define the problem class which characterizes the class of hard problems we have considered so far. To motivate this class, note that a common property across all the hard decision problems we have consider so far. For each of these problems, in the case when the answer is "yes" there is a proof that we can use to verify in polynomial time that is the case. For example, if the size of the maximum cut is $\geq k$ then given the maxcut we can easily verify that it has this property. Similarly, if a graph is 3-colorable or it does have an independent set of size $\geq k$ we can check that a coloring is a valid coloring or that an independent set is of size $\geq k$ each in polynomial time. Formally, this class of decision problems for which the "yes" instances are polynomial time verifiable is known as $NP$ the class of algorithms solvable in non-deterministic polynomial time. In the following we formally define this notion of verification and $NP$.

**Definition 9** ($NP$)**.** A decision problem $X$ is solvable in nondeterministic polynomial time, denoted $X \in NP$, if and only if there is a polynomial time verifier for $X$, that is there is an algorithm $B$ that given binary

strings $s$ and $t$ output $B(s,t) \in \{"yes","no"\}$ in time polynomial in $|s|+|t|$ with the property that if binary string $s \notin X$ then $B(s,t) = "no"$ for all $t$ and if $s \in X$ then there exists $t$ with $|t| = O(\text{poly}(|s|))$ such that $B(s,t) = "yes"$.

We call $B$ a verifier as we can use its output to verify whether or not a binary string $s \in X$ as we can simply search over all $t$ to see if $B(s,t) = "yes"$ for all $t$. This happens if and only if $s \in X$. We call $B$ efficient as it runs in polynomial time and will run in polynomial time for the $t$ with $|t| = O(\text{poly}(|s|))$ for the $t$ such that $B(s,t) = "yes"$ in the case when $s \in X$. Consequently, we can think of $t$ as a candidate proof of the fact that $s \in X$ and $B$ as the algorithm that verifies such proofs.

The class $NP$ is called non-deterministic polynomial time as given an algorithm which can "guess" $t$ we can solve the problem in polynomial time. This "guessing" is known as non-determinism. For further information, the curious reader should consult one of the many wonderful texts on complexity and automata theory.

Now, note that the decision version of each of the "hard" problems we have considered so far is in $NP$ as we have argued. Furthermore, we can trivially show the following.

**Lemma 10.** *$P \subseteq NP$.*

*Proof.* If $X \in P$ then the algorithm which given binary string $s$ and $t$ simply uses a polynomial time algorithm to determine $s \in X$ and output "$yes$" if $s \in X$ and "$no$" otherwise is a polynomial time verifier for $X$. □

Consequently, the running times for solving problems in $P$ are in the worst case no larger than those for solving problems in $NP$. However, as we have suggested it is not known actually known that the class $NP$ are actually harder than those in $P$. In other words whether or not $NP \subseteq P$ and therefore $P = NP$ is open. It is one of the largest open problems in theoretical computer science and one of the Millenium Prize problems for which a resolution would earn you at least a million dollars.

Given this issue, as we have discussed our goal instead is to use NP to classify the computational complexity of our hard problems. To do this, we will argue that the problems we have consider are not only in NP, but they are (up to polynomial time reductions) the hardest problems in $NP$. Formally we define the following complexity classes.

**Definition 11** ($NP$-Hardness and Completeness)**.** A decision problem $X$ is said to be *$NP$-hard* if for all $Y \in NP$ it is the case that $Y \leq_p X$. If additional $X \in NP$ then $X$ is said to be $NP$-complete.

In other words, we say a problem is $NP$-hard if it is at least as hard (in the sense of polynomial time reductions) as every problem in $NP$ and it is complete if it is $NP$ as well. Consequently, a problem is $NP$-complete if it is of the hardest problems in $NP$. One of the nice things about this definition is that if we had one $NP$-complete problem then we would have an easy recipe for proving that some other problem is $NP$-complete as well, simply prove it is $NP$ and reduce the $NP$-complete problem to it.

**Lemma 12.** *If $X$ is $NP$-complete and $X \leq_p Y$ for $Y \in NP$ then $Y$ is $NP$-complete as well.*

*Proof.* Since $X \leq_p Y$ by the transitivity of polynomial time reductions and the definition of $NP$-completeness we have that for all $Z \in NP$ it is the case that $Z \leq Y$ and therefore $Y$ is NP-hard. Since $Y$ is assumed to be $NP$-complete as well the result follows by the definition of $NP$-completeness. □

Consequently, the notion of the $NP$-completeness has the hope of meeting our objectives. It has a nice interpretation that seems not too adhoc or tied to a particular problem and it comes with a recipe for showing that new problems are $NP$-complete as well. Thus we might hope to easily show that many

problems are $NP$-complete if only we could show that one problem is $NP$-complete. In fact, this can be shown, that all the hard problems we have considered so far are $NP$-complete.

However, there is a key missing step in following this program of developing the class of $NP$-complete problems. In particular, it is not clear any such problem exists. How do we know it isn't the case that there is a hierarchy of increasingly hard problems in $NP$ and none that is harder than all? If $P = NP$ then clearly this would not be the case (as every problem in $P$ would be $NP$-complete trivially) however as we have discussed we are far away from doing this.

Fortunately, there were breakthrough results in the early 1971 independently by Cook and Levin that showed in fact this is not the case and that there are $NP$-complete problems. Since there breakthrough a broad class of problems has been shown to be $NP$-complete and this has been a major intellectual export of theoretical computer science to many communities in this ability to give evidence of problem difficulty.

In the remainder of this lecture we will provide a sketch of the proof that a problem is $NP$-complete and in the next lecture we will use the fact that this problem is $NP$-complete to show that many more problems are $NP$-complete.

## 4 Circuit Satisfiability is $NP$-complete

Here we define the problem of *circuit satisfiability*, or *circuit-SAT*, and sketch the proof that is $NP$-complete. This problem will serve as the basis for showing that a broader class of problems are $NP$-complete.

Note that the definition of a problem being $NP$-complete is tied only to the idea of a verifier running in polynomial time. Consequently, to show that circuit-SAT is $NP$-complete we will end up showing that we can encode the computation of this verifier in some other problem. To do this formally involves being very formal about the notion of computation we are using and typically involves defining things like Turing Machines. This formal notion is a bit outside the scope of this class and therefore the proof that we provide that circuit-SAT is $NP$-complete will only be a sketch. If you would like to see a more formal proof of the fact that circuit-SAT is NP-complete I would encourage you to take a course in complexity theory or automata theory or consult one of the wonderful texts on this subject.

Now to formally define the circuit-SAT problem we need to formally define a Boolean circuit.

**Definition 13** (Boolean Circuit)**.** A *Boolean circuit* is defined by Boolean variables $x_1, ..., x_k$ and a connected tree with edges oriented from the leaves towards a root with the following properties:

- Each leaf node is labeled with one of $x_i$, $T$ for "true," or $F$ for "false"
- Each internal node is labeled with one of $\vee$ for Boolean "or", $\wedge$ for Boolean "and," or $\neg$ for Boolean "not"
- Each $\vee$ and $\wedge$ node has in-degree 2 and each $\neg$ node has in-degree node

Given an assignment of the variables $x_i$, i.e. setting each $x_i$ to either $T$ for "true" or $F$ for "false" the outgoing edge of each internal node is labeled with the result of applying the Boolean operation of that node to each of the input and the *evaluation* of the circuit is said to be the Boolean value corresponding to applying the Boolean operation of the root node to its input. If the output of the circuit is $T$ the circuit is said to evaluate to true and if the output of the circuit is $F$ the circuit is said to evaluate to false.

Note that a Boolean circuit is simply a particular way of representing a Boolean formula and that the evaluation of the Boolean circuit as defined above is simply the evaluation of the Boolean formula in the standard sense. Given this definition of a Boolean Circuit the Circuit-SAT problem is defined as follows.

**Definition 14** (Circuit-SAT)**.** The *circuit satisfiability problem*, or *circuit-SAT*, is the problem of given a Boolean circuit determining if their is an assignment of the variables that causes the circuit to evaluate to true.

Consequently, the Circuit-SAT problem is simply the problem of determining if the associated Boolean formula is identically false or not. In the remainder of this lecture we simply sketch the proof that Circuit-SAT is NP-complete.

**Theorem 15.** *Circuit-SAT is $NP$-complete.*

*Proof Sketch.* As with the proof that any problem is $NP$-complete, the proof usually starts with the easy part of proving that the problem is in $NP$. For Circuit-SAT as with many problems, this is quite simple. Given an assignment of the Boolean variables, we can clearly evaluate the circuit in linear time and therefore check if it is a satisfying assignment in linear time. Consequently, there is a polynomial time verifier which simply considers the proof string $t$ as a candidate satisfying assignment.

Now the much harder part of the proof is to prove that Circuit-SAT is $NP$-complete. Since we have not yet established that any problem is $NP$-complete we will need to prove this by appealing directly to the definition of $NP$. Let $X \in NP$ be arbitrary. By definition, we know that $X$ has a polynomial time verifier, that is $B$ which takes as input binary strings $s$ and $t$, runs in time polynomial in $|s|+|t|$ and always outputs $F$ if $s \notin X$ and outputs $T$ for some $t$ with $|t| = O(\text{poly}(|s|))$ when $s \in X$.

Now what we wish to do is in polynomial time encode the execution of $B$ as a Boolean circuit which is polynomial size $|s|$ and has a satisfying assignment if and only if there exists a $t$ with $|t| = O(\text{poly}(|s|))$ such that $B(s,t) = T$. If we could do this, then by solving circuit-SAT we could check if $s \in X$ in polynomial time plus the time to solve circuit-SAT on this polynomial size input. Consequently, if we can do this, then $X \leq_p Circuit - SAT$ and Circuit-SAT is $NP$-complete as desired.

To do this encoding, let's fix the input length of $s$ as $|s| = k$. Note that $B$ runs in time at most $c$ where $c = O(\text{poly}(k))$. Consequently, $B$ uses space at most $m$ for $m = O(\text{poly}(k))$ and thus we can encode the state for $B$ at each time step $i \in [m]$ as some Boolean string $s_i \in \{0,1\}^m$. Now we can assume that $s_1$ is simply $s$ concatenated with the string $t$, i.e. the input to $B$. Furthermore, we can assume without loss of generality that $B$ in the case when it outputs true sets its final state to some string $s^T$ and when it outputs false sets its final state to some string $s^F$. Now we wish to check when $B$ outputs $F$ and thus to our circuit we can simply set $s_c = F$.

Now we set our Boolean circuit to check to output $T$ if and only if it is the case that

$$\wedge_{i \in [k-1]} (s_{i+1} \text{ is the state of } B \text{ if the previous state is } s_i)\,.$$

In, other words, we construct the circuit that evaluates to true if and only if the $s_i$ correspond to the actual states of $B$ starting from $s_1$. This is where we need to use a formal definition of computation and where our proof only becomes a sketch. We claim that it is possible to write a polynomial size Boolean circuit that is true if and only if "$s_{i+1}$ is the state of $B$ if the previous state is $s_i$" and therefore the circuit can be written as just the and of the circuits which check this. The idea is that the algorithm simply applies fixed logical rules to go from one state to the next and we can just write down a Boolean formula encapsulating this. This is tedious to do formally, but I believe makes sense intuitively.

Consequently, we have constructed a polynomial size circuit that is satisfiable if and only if there is a sequence of states that correspond to a valid execution of $B$ that go from some input $(s,t)$ to a state corresponding to outputting true. Therefore, by checking if this circuit is satisfiable we can check if an input string $s \in X$ in time polynomial in $|s|$ plus the time to solve circuit-SAT once on a problem of size $O(\text{poly}(|s|))$. □

This proof should hopefully also convey or suggest why it might seem that $NP$ is harder than the problems we have seen in $P$. We have essentially shown that $NP$-complete problems allow us to brute force over the input to polynomial size computation which seems incredibly powerful - though we do not know how to show formally that this is actually more powerful than anything in $P$.