1 Overview

In these notes, we tie together major themes in both CS and math via a very conceptual topic. We will provide a roadmap for most of mathematical CS itself. We introduce the notion of Turing reducibility via a connection to the Halting Problem and its variants. Then, we generalize the Turing reducibility notion to a larger class of problems including NP-completeness.

2 Introduction

The purpose of these notes are to tie together major themes of both CS and math. The topics covered provide a roadmap for most of mathematical CS, including

- What CS people do.
- What you’ll see in courses ahead.
- A system for organizing and harnessing that mass of facts and techniques,

Note that you cannot be a computer scientist without a reasonable understanding of reductions. Also, you cannot pass CS330 (a course many of you will take soon!) without a reasonable understanding of reductions.

Up until this point in class, we have had various assignments involving variants of the classical Halting Theorem. Let us refer to the Halting Problem as \( HP \). It’s not always easy to show these undecidability results, and it would be nice if there exited some \textit{recipe} for such proofs (almost like induction). This lecture is designed to show you such a recipe.

For many of the \( HP \) proofs, we showed the results directly via a direct contradiction argument. This recipe for proving undecidability has many advantages, namely that it is:

- Very easy.
- Very powerful.
- Uses our newfound understanding of sets and functions.

This recipe is called “Turing Reducibility”. Some of you call it a “wrapper function” and may even have discovered it yourself while doing the assignments.

3 Halting Problem Review

Recall the lecture on \( HP \). Our first notion is to separate out a \textit{problem} from the \textit{procedure} to solve the problem. To continue with the overall theme courses, we will refer to algorithms by their Scheme procedures
and names (if they were able to be computed!). For example, we termed the supposed procedure to solve
HP by safe?. In this case, HP is the problem and safe? is the proposed procedure to solve it. Similarly, Equiv?

is the procedure corresponding to the problem of “function equivalence”, and simple-safe? is the
procedure corresponding to the problem of “HP for thunks[1]. Let’s take a look at simple-safe?. Suppose
such a function exists. That is, given f, a thunk, then

\[(\text{simple-safe? } f) \rightarrow \begin{cases} 
\#t & \text{if } (f) \text{ halts with an answer} \\
\#f & \text{otherwise}
\end{cases}\]

You have already shown that simple-safe cannot exist. Maybe, you showed this via a direct contradiction.
Or maybe, you were able to show that, if simple-safe? exists, then we could construct a valid procedure
safe? for the original HP.

If safe? existed, we would say it decided the HP. Specifically, this means, given any pair \((f, a)\), safe?
would return \#t or \#f after deciding whether \((f a)\) halts with an answer. Similarly, Equiv? would decide
the problem of “function equivalence”, or simple-safe? would decide the problem of “HP for thunks”. But,
we knew a priori to working on simple-safe? that safe? could not exist via our safify argument. As such,
this method of writing a “wrapper” function that directly used simple-safe? to construct safe? showed that
simple-safe? also cannot exist. This notion has a name in computability theory and complexity theory,
which is reduction[2].

4 Specific Turing Reductions

4.1 Simple-Safe?

Let us now discuss a recipe to solve the question about the existence of simple-safe?. Consider the following
flowchart:

\[(f, a) \rightarrow \text{Turing-Reduce} \rightarrow \text{Thunk} \rightarrow \text{simple-safe?} \rightarrow \#t/#f\]

What exactly does this ‘Turing-Reduce’ step entail? For the simple-safe? example, consider the follow-
ing:

\[(\text{define Turing-Reduce1}
(\lambda (f a)
 (\text{list (lambda () f a))))))\]

Given this somewhat strange looking function, we can now write safe? as:

\[(\text{define safe?}
(\lambda (f a)
 (apply simple-safe? (Turing-Reduce1 f a))))\]

---

1 Thunks are just functions that take no arguments.
2 It’s an unfortunate naming collision, but this has absolutely nothing to do with the higher order procedure reduce = accumulate = foldr.
Upon showing the validity of this reduction, we would have the same proof as shown in class; however, we have a most important difference in that we have this somewhat strange looking ‘converter’. We see that Turing-Reduce is a higher order function.

Note a technical detail when writing this Turing-Reduce function in general. Such a function must be ‘Turing-computable’, a discussion for which a knowledge of Turing machines is necessary.\footnote{The wikipedia article gives a cursory discussion: \url{https://en.wikipedia.org/wiki/Turing_machine}} Given that this is beyond the scope of the course, let us rephrase our definition of the technical requirement. Let us say that our Turing-Reductions are valid (the procedures are computable) if we can write them in Scheme code. As we will discuss later, this isn’t really a simplification of the details, but it really is an equivalent formulation.

4.2 Equiv?

Let’s try another example. Consider Equiv?, a function that for any two functions \(g, h\), returns

\[
\text{(Equiv? } g \ h) \rightarrow \begin{cases} 
\#t & \text{if } g = h \\
\#f & \text{otherwise}
\end{cases}
\]

where \(g = h\) means that for all inputs \(x\), \((g \ x) = (h \ x)\). Let us suppose that such a Scheme procedure Equiv? exists. If Equiv? exists, then we can construct:

\[
\text{(define safe?}
\text{(lambda ((f <fcn>) (a <arg>))}
\text{(let ((g (lambda (x) (f a) 1)))}
\text{(h (lambda (x) 1)))}
\text{(Equiv? g h)))}
\]

Since safe? cannot exist, and all we assumed was the existence of Equiv?, we have contradicted the assumption which means Equiv? cannot exist. However, this proof is not quite complete because we once again have yet to show the validity of the reduction. What exactly does this mean? We have two if and only if (iff) claims to prove in this case. Specifically, we would need to show that

Claim 1: \(f\) halts on input \(a\) iff \(g = h\).

Claim 2: \(f\) does not halt on input \(a\) iff \(g \neq h\).

Using the substitution model, claims are easy to show. As a side note, while these details may seem a bit tedious, they are entirely necessary for a complete proof. Also, we are not entirely subject to showing these tedious details. In general, we claim that Claim 1 actually implies Claim 2 and vice versa. If this is true, then we actually only have to show one of the iff statements and we are done.

Why is this the case? Let’s say that we successfully show Claim 1. If we split this up into two subclaims, we have Claim 1a: if \(f\) halts on input \(a\) then \(g = h\), and Claim 1b: if \(g = h\) then \(f\) halts on input \(a\). For completeness, split Claim 2 similarly into Claim 2a: if \(f\) does not halt on input \(a\) then \(g \neq h\) and Claim 2b: if \(g \neq h\) then \(f\) does not halt on input \(a\). Notice that these claims are related in another way. Specifically, the contrapositive of Claim 1a is Claim 2b. Similarly, the contrapositive of Claim 1b is Claim 2a. In general, we know that if a statement is true, then the contrapositive must also be true. As such, this means that if we prove Claim 1a and Claim 1b, then by this contrapositive argument we get that Claims 2a and 2b are also true for free. This means we really only have to show two statements and we get all 4 of the subclaims.

\footnote{The contrapositive of \((P \rightarrow Q)\) is \((\neg Q \rightarrow \neg P)\).}
by extension. Of course, there’s no reason that we have to specifically show Claims 1a and 1b. We can equivalently show Claims 2a and 2b, and the proof would be complete as well. The claims we choose are completely a matter of style and elegance that is specific to the problem. Pretty useful, no?

But, similar to what we did for simple-safe?, let’s consider this alternative approach. Consider this flowchart for Equiv?:

\[
(f, a) \xrightarrow{\text{Turing-Reduce}} (g, h) \xrightarrow{\text{Equiv?}} \#t/#f
\]

Now, we write the necessary Scheme functions as well:

\[
(\text{define Turing-Reduce2} \lambda (f, a) (\text{list} \lambda (x) (f a 1)) (\lambda (x) 1))
\]

\[
(\text{define safe?} \lambda (f, a) (\text{apply Equiv?} (\text{Turing-Reduce2} f a)))
\]

If we use the substitution model to show the validity of the reduction, then once again we get the same proof. But once again, we have this mysterious ‘Turing-Reduce’ higher order function. Let us generalize this notion, and we will see that this higher order function is absolutely essential to understanding reducibility.

4.3 A practical note

You may wonder why I have split up the code for safe? into two parts, using a helper function like Turing-Reduce2 (or Turing-Reduce1). There are two reasons.

First, of course, it it not possible to actually write and test safe? since it is uncomputable. But you can easily write Turing-Reduce2, prove it is correct using the substitution model, and even run it to test whether it correctly transforms instances of \(HP\) into instances of \(Y\), the function-equivalence problem.

Second, in more advanced classes it will not strictly be required to write safe?. Instead, you will only need to write and prove Turing-Reduce2.

In this class, however, you are required to write both Turing-Reduce2 and safe?. Note that for a correct proof you must also prove Claims 1 and 2 (above) using the substitution model.

5 General Turing Reducibility

5.1 Deriving the Recipe

Let’s take a step back from these specific examples and look at these problems in a more general sense. Let \(E^*\) be the set of all possible Scheme expressions. Note that \(E^*\) contains, e.g., all possible Scheme programs and all possible inputs to those programs. In this way, we can redefine \(HP\) with set notation:

\[
HP = \{(f, a) \in E^* | f \text{ halts on input } a\}
\]
Similarly, for the problem corresponding to simple-safe?, we can define

\[ HP_{Th} = \{ f \in E^* \mid f \text{ halts on no arguments} \} \]

This abstraction gives us an elegant way to express all of these problems that we have been discussing thus far. Now, we define:

**Definition 1. Reducibility:** For a problem \( Y \), we write \( HP \leq_T Y \) if there exists a Scheme-computable function \( R : E^* \rightarrow E^* \) s.t. \( e \in HP \iff R(e) \in Y \).

In this case, we say “\( HP \) is Turing-Reducible to \( Y \)” and we call \( R \) a reduction. The following diagram illustrates this idea:

\[ \begin{array}{ccc}
E^* & \xrightarrow{R} & E^* \\
\text{HP} & & \text{Y}
\end{array} \]

We see that our function \( R \) stays within \( E^* \) itself, but we are not necessarily staying within the domain of \( HP \). Consider the case of reducing \( HP \) to \( HP_{Th} \). These problems may have different domains, but these inputs must stay within the bounds of \( E^* \).

As an aside, two of the most influential figures in this field are Alonzo Church and Alan Turing. These two proved that, \( g \) is a Scheme function iff \( g \) is Turing-computable. This shows that we can keep our discussion entirely in Scheme and lose none of the overarching generality.

Now, consider once again the problem \( Y \). How do we use this to garner information about \( HP \)? The crux of this method relies on the following claim:

Suppose \( HP \leq_T Y \), and \( Y \) is decidable. Then, \( HP \) is decidable.

**Proof:** If \( Y \) is decidable, let \( S \) be the scheme function deciding it. If \( HP \leq_T Y \), let \( R \) be the scheme function reducing \( HP \) to \( Y \). Given an instance \( (p, a) \in E^* \), run the expression

\[ (\text{apply } S \ (R \ (p, a))) \]

to decide whether \( (p, a) \in HP \). The latter follows from the substitution model.

The preceding claim is crucial in establishing our recipe. Suppose by way of contradiction that some procedure \( S \) decides \( Y \). Then, \( S \circ R \) decides \( HP \), as shown by our claim. Since we can write \( R \) and \( S \) as Scheme functions, we can easily write the composition \( S \circ R \) as a Scheme function as well. In general, this means that composing 2 Turing computable functions yields a Turing computable function. Next, consider some \( e = (p, a) \in HP \). Since, by assumption, \( S \) can decide any element of \( Y \), we can input \( R(e) \) as a specific example, as shown by the proof. By definition of reduction, this means that \( e \in HP \iff (S \ (R \ e)) \rightarrow \#t \).

This is a contradiction, since we know that \( HP \) is undecidable. Since our only assumption was that \( S \) exists, we conclude that \( S \) does not exist.

Now that we have generalized our proof technique from earlier, we finally arrive at our long sought after recipe to show that the procedure for a problem \( Y \) is uncomputable. In particular, we need to reduce \( HP \) to \( Y \), which involves:

1. Construct a Turing computable function \( R : E^* \rightarrow E^* \).
2. Show $e \in HP \iff R(e) \in Y$.

The beauty of this recipe is that, if we do these two things, then we’re done! However, we do see that the choice of $R$ depends on the problem itself - there is no one size fits all. As such, it is fair to say $R_S$ rather than just $R$.

As an aside, for future reference note that in theory classes, $HP$ is actually called $K_{TM}$ where $TM$ = “Turing Machine”. Similarly, $HP_{Th}$ is called $K_{BT}$.

In general, this general recipe gives us a way of classifying decidability using mappings:

\[
\begin{array}{c}
E^* \\
\downarrow \quad \downarrow \quad \downarrow \\
HP & Y_1 & Y_2 \\
\quad R_1 \quad R_2 \quad S_1 \quad S_2 \\
\quad \{\#t, \#f\} \\
\vdots
\end{array}
\]

In principal, we can do this for all uncomputable problems of interest. In terms of an uncomputability hierarchy, we can derive a slightly different yet related diagram:

\[
\begin{array}{c}
K_{BT} \\
\downarrow \quad \downarrow \quad \downarrow \\
K_{\pi,r} & K_{1} & K_{Equiv} \\
\downarrow \quad \downarrow \quad \downarrow \\
\cdots & \cdots & \cdots \\
\vdots
\end{array}
\]

In this diagram, the arrows indicate Turing reducibility. This entire set represents problems can be reached and are at least as hard as $K_{TM}$ and are therefore uncomputable. Now, consider the following question. The way this diagram is drawn, there’s a nascent implication there: what if we took a problem like $K_{BT}$ and reduced $K_{BT}$ to some problem $Y$? Would this allow us to make the same implications about the undecidability of $Y$? The answer to this question is actually yes. Roughly speaking, if we can reduce $K_{BT}$ to $Y$, then via composition, we can reduce $K_{TM}$ to $K_{BT}$ and then reduce $K_{BT}$ to $Y$, and this gives the same implication about the undecidability of $Y$.

5.2 Direction of the Reduction

In general, if we have the relationship

$$K_{TM} \preceq_T Y$$
then we are reducing $K_{TM}$ to $Y$. This is a general theme across all reductions. When you want to discover something about a problem, you reduce the problem that you already know something about to the problem that you want to know something about. Another point to notice about reduction as per Definition 1 is that the definition itself has nothing to do with undecidability. Rather, undecidability ends up being a conclusion because we started with $K_{TM}$ as our ‘base’ problem of sorts. But, the fact that undecidability is not a core detail to reductions implies that the technique is even more general. As we’ll see soon in section 6, this actually does turn out to be the case.

6 NP-Completeness

In order to understand the more general uses of reduction, we first need a slight digression into an understanding of complexity classes. In particular, rather than thinking about problems that are impossible to procedurally compute, what if we change our question to problems that are “very hard” to compute? To formalize this notion, we impose a new restriction on our reductions:

Definition 2. Polynomial time Turing reduction: $A \leq_{P} B$ is a polynomial time Turing reduction if there exists a Scheme-computable function $R : E^{*} \rightarrow E^{*}$ s.t. $e \in A \iff R(e) \in B$ and $R$ has run time polynomial in the input size. Note the subscript $P$ to specify that the reduction must take polynomial time.

We impose this restriction because we are now dealing with problems that are difficult to solve, and in general we tease out this distinction of being “easy to solve” as having a polynomial time solution. As such, if we allow our reductions to take exponential time, then we cannot garner useful information about the problems because the reductions themselves are “very hard” to compute. So, in general, from here on out, when we talk about reductions, we implicitly mean polynomial time Turing reductions. Now, let us actually formalize the notion of what makes a problem “very hard” to solve.

6.1 P vs. NP

First, for purpose of contrast, let us consider what problems are “easy to solve”, namely:

Definition 3. Complexity class P: The class of all decision problems that can be solved by a deterministic Turing machine in time polynomial in the binary representation of the input size.\(^5\)

First, a “decision problem” is simply a problem for which the output is a binary true or false. As per the earlier discussion, we can represent programs on Turing machines by Scheme programs. Deterministic just means that we do not allow randomness in our algorithms. Lastly, polynomial time just means that, for a problem with an input size $n$, that the run time to solve the problem is $O(n^k)$ for some constant $k$. In terms of complexity theory, if we have a decision problem $A$ and input $x$, then if $(A \ x) \rightarrow \#t$ then we call $x$ a YES instance. Similarly, $(A \ x) \rightarrow \#f$ would be called a NO instance.

Definition 4. Complexity class NP: The class of all decision problems where solutions can be verified by a deterministic Turing machine in time polynomial in the binary representation of the input size.\(^6\)

Here, we see the most key difference between $P$ and $NP$. Essentially, a problem is in $NP$ if, given a candidate solution to the problem instance, can we verify whether or not that specific instance of the problem actually returns true? What does such a “candidate solution” look like? Well, it depends on the specific problem at hand.

\(^{5}\)https://en.wikipedia.org/wiki/P_(complexity)

\(^{6}\)https://en.wikipedia.org/wiki/NP_(complexity)
Remark 1. A common misconception about NP is that it stands for “non-polynomial”. This is NOT true - the name comes from an equivalent formulation: NP is the class of decision problems that can be solved by a nondeterministic Turing machine in polynomial time.

Given these definitions, we get the following Lemma:

Lemma 1. \( P \subset NP \)

Proof: Consider some problem \( A \in P \). Now, let’s say that we are given a candidate solution to an instance of \( A \). Since \( A \in P \), we can ignore the candidate solution and directly solve the problem in polynomial time. Then, using the direct solution, we can verify whether or not the given instance is a YES instance. Since we can verify if we have a YES instance in polynomial time on a deterministic Turing machine, \( A \in NP \). This implies \( P \subset NP \).

Now, the million dollar question\(^7\) is the opposite case, specifically, is \( NP \subset P \)? This is the much sought after \( P \) vs. \( NP \) problem\(^8\). This is because \( NP \subset P \) would imply that \( P = NP \), which means that the set of solvable and verifiable problems are actually the same. As of now, the problem is still open.

6.2 NP-Hardness

Given these hierarchies of complexity classes, we can now define our notion of hardness.

Definition 5. NP-Hard: A problem \( X \) is NP-Hard if, for all problems \( A \in NP \), \( A \leq_P X \). Note that this reduction must specifically be polynomial time.

Definition 6. NP-Complete: A problem \( X \) is NP-Complete if \( X \in NP \) and \( X \) is NP-Hard.

In this sense, the class of NP-Hard problems contain all problems that are “at least as hard” as all problems in NP. Figure\(^9\) gives us a nice illustration of this.

---

\(^7\)Quite literally: [https://en.wikipedia.org/wiki/Millennium_Prize_Problems](https://en.wikipedia.org/wiki/Millennium_Prize_Problems)

\(^8\)https://en.wikipedia.org/wiki/P_versus_NP_problem

These definitions also tie together the overarching ideas of computational hardness and reduction. In an
intuitive sense, if \( A \leq_p B \), then \( B \) must be “at least as hard” as \( A \). This is because, if we can solve \( B \), then by
extension we can solve \( A \) by transforming our instance of \( A \) to an instance of \( B \), solving this new instance,
then transforming our answer back to an answer for \( A \). In the case of decision problems, this transformation
back is trivial because we are simply returning \#t or \#f.

Note that, since the question of \( P \) vs. \( NP \) is still an open question, we cannot as of yet definitively say that
\( NP \)-complete problems can or cannot be solved in polynomial time. Given all of the research that theoretical
computer scientists and mathematicians alike have poured many years of research into the problem to no
avail, the general consensus is that \( P \neq NP \). But of course, we cannot be certain. Regardless, if this general
conjecture is true, then it is very unlikely that \( NP \)-complete problems can be solved in polynomial time. Let
us discuss a specific example.

### 6.3 Hamiltonian Cycle

To complete this notion of \( NP \)-hardness, let us define a specific hard problem. In particular, let us consider
the Hamiltonian Cycle problem, which we denote by \( HAM \).

**Definition 7.** Hamiltonian Cycle: Given a graph \( G = (V, E) \), a Hamiltonian Cycle is a path that, aside from
the starting vertex, includes each vertex exactly once and returns to the starting vertex.

Note some specifics. The Hamiltonian Path problem is closely related, with the caveat that the path does
not need to return to the starting vertex. Also, for sake of simplicity, let us assume the graph \( G \) is undirected.
Now, under these assumptions, we can use our set notation and say

\[
HAM = \{(V, E) \in E^* \mid (V, E) \text{ has a Hamiltonian Cycle}\}
\]

Consider the following example graphs:

We see that there exists no Hamiltonian cycle. Roughly speaking, we can see this because once the cycle
hits \( v_3 \), there’s no way for it to hit other vertices without having to come back to \( v_3 \). On the other hand, what
about the following graph?
Here, we do have a Hamiltonian cycle. Specifically, such a cycle is \((v_1, v_2, v_3, v_4, v_5, v_1)\). We see that all vertices are reached, no intermediate vertices are repeated, and we return to the starting vertex. In terms of size, we say that the size of \(G\), denoted by \(|G|\) is given by \(|G| = |V| + |E|\). Note that if \(|V| = n\), then \(|E| = O(n^2)\).

Now, let’s consider some other problem \(Y\). Such that \(HAM \leq_p Y\). We get a very similar looking diagram to before:

Suppose by way of contradiction that \(f\) decides \(Y\) in polynomial time. Then, \(f \circ R\) decides \(HAM\) in polynomial time.

**Proof:** First, by definitions of \(f\) and \(R\), \(f \circ R\) is Turing computable. Second, since each function runs in polynomial time, the composition does as well. Lastly, via our construction, \(e \in HAM \iff f(R(e)) = \#t\). So, if such an \(f\) exists, then we can solve \(HAM\) in polynomial time. From this, we conclude that it is very unlikely that \(f\) exists. We then write that \(Y\) is \(NP\)-Hard.

Note the extreme similarity to our discussion in Section 5.1. In fact, the recipe is basically the exact same to show \(HAM \leq_p Y\) and therefore conclude that \(Y\) is \(NP\)-Hard.

1. Construct a Turing computable function \(R : E^* \rightarrow E^*\).
2. Show \(e \in HAM \iff R(e) \in Y\).
3. Show \(R\) runs in polynomial time.

In fact, the only difference in the recipe itself is this extra step of showing that \(R\) runs in polynomial time. Otherwise, the conclusion is also different, but this shows the generality and power of reduction as a technique.

### 6.4 A Satisfactory Bit of History

In Section 6.3 we showed that if we can reduce \(HAM\) to another problem \(Y\), then \(Y\) is \(NP\)-Hard. However, this was predicated on the assumption that \(HAM\) itself is \(NP\)-Hard. Well, how would we show this? Recall the original definition of \(NP\)-Hardness in definition 5 in section 6.2. A problem \(X\) is \(NP\)-Hard if for all problems \(A \in NP, A \leq_p X\). In general, this is a very difficult thing to show. It requires such a large amount of abstraction that it ends up being difficult.

Another thing to consider is that, \(HAM\) is in \(NP\) itself. Why? If we are given a potential candidate solution to \(HAM\), what would this look like? It could be a potential sequence of vertices of \(G\). In polynomial time, we can trace out this path and verify that we hit every intermediate vertex exactly once and return to the starting vertex. Since we can verify this candidate in polynomial time, \(HAM\) is in \(NP\).

Consider another problem \(NP\)-Complete problem \(Y\). By definition, this means all problems in \(NP\) can be reduced to \(Y\), including \(HAM\). But, we also knew a priori that \(HAM\) is \(NP\)-Hard, so it must be the case that \(Y \leq_p HAM\) as well. This gives rise to the following definition:

**Definition 8.** \(X \equiv_p Y\) if \(X \leq_p Y\) and \(Y \leq_p X\).
So, in some sense, this means that the set of all $NP$-Complete problems are “as hard as each other”. So, computer scientists have collected many hard problems $\{X_i\}$ such that $X_i =_P X_j$. For this entire class of problems, no one up until now has been able to efficiently solve them, so we’ve essentially gotten everyone to agree that they are hard.

One such way to do this is to reduce $HAM$ to show that $HAM \leq_P X_i$ for all $i$. This still does not quite answer the original question of how to show that $HAM$ itself is $NP$-Hard. In order to use this reduction argument, we need a starting point of sorts, so a problem that we know is $NP$-Hard that we can use to reduce to other problems. Was this $HAM$? Not quite - in 1971, the Cook-Levin Theorem\(^{[10]}\) was published, which didn’t work with $HAM$; rather, they showed that the problem of boolean satisfiability is $NP$-Complete. They showed this via the original definition of $NP$-Hardness by reducing any general problem in $NP$ to boolean satisfiability. Once they established this, people finally had a base starting point. Now that they knew boolean satisfiability was $NP$-Complete, they could reduce it to a wide range of other problems. Now, to show that a problem is $NP$-Hard, there’s no need to show the highly abstract definition; we can use our general recipe of reduction which is much, much simpler. A year after the Cook-Levin Theorem was published, Richard Karp used the theorem to show a set of 21 $NP$-Complete problems, all in one paper\(^{[11]}\). $HAM$ was among these, which finally gives us our long sought after answer of why we can reduce it to other problems. Note how amazing this is: Cook’s original paper was entirely devoted to showing that one problem was $NP$-Complete, while Karp’s paper, by using reduction\(^{[12]}\), showed that twenty-one different problems were also $NP$-Complete. This really goes to show both the power and simplicity of reduction.

### 6.5 Traveling Salesman Problem

To finish our discussion on $NP$ reductions, let’s show a specific example. Let us consider the Traveling Salesman Problem, which we will now call $TSP$. The problem is stated as, given a set of $N$ cities and pairwise distances between them, find a travel route of minimum distance for the salesman, given that he must visit each city exactly once and return to where we started. As an abstract representation, we can think of the cities as represented by vertices in a graph and the edges between vertices carry weights corresponding to the distances. For simplicity, let us say the distances are symmetric, so the graph can remain undirected. Since all cities have some distance between them, the graph is also complete - that is, there is an edge between each pair of vertices.

Now, note one small caveat to the problem that makes it different from $HAM$ - it’s not a decision problem! The output should be a cycle of minimum weight. For our discussion on $NP$ reductions, we need a decision version for the problem. This is easy to do, by artificially introducing another parameter to the problem $k$. The role of $k$ essentially serves as a budget for our salesman. Rather than saying he must find the shortest possible cycle, he has to figure out if there exists any cycle that has total weight at most $k$. This has turned the optimization variant of the TSP to a decision variant. Let us denote this new problem by $TSPD$, which, given an instance of the traveling salesman problem and an extra parameter $k$, now asks if there exists any cycle for the salesman that has total weight at most $k$. Now that we have set up the problem, we have the following theorem:

**Theorem 2.** $TSPD$ is $NP$-Complete.

---

\(^{[10]}\)https://en.wikipedia.org/wiki/Cook%E2%80%93Levin_theorem

\(^{[11]}\)https://en.wikipedia.org/wiki/Karp%27s_21_NP-complete_problems

\(^{[12]}\)Technically speaking, Karp used a stronger form of Turing Reductions called a “many-one reduction”. But, the overall point still stands about the power of reduction.
Proof. We need to show both that \( TSPD \) is in \( NP \) and that it is \( NP \)-Hard. Let us show the former statement first. What would a candidate solution to the problem look like? Such a solution would be a candidate sequence of vertices. In polynomial time, we can simply trace this candidate solution throughout the graph, check that we do not repeat intermediate vertices, and make sure we end up at the starting vertex. Also, as we trace the path, keep a running count of the sum of the weights on the edges the path takes. At the end, check if this running sum is \( \leq k \). If all conditions are true, return that the input is a YES instance. Otherwise, return that it is a NO instance.

For the more difficult step, we need to show that \( TSPD \) is \( NP \)-Hard. We will do this by reducing \( HAM \) to \( TSPD \). Consider a general instance of \( HAM \), which would just be an undirected graph \( G = (V, E) \). On the other hand, \( TSPD \) requires a complete graph, a weight function on the edges, and another parameter \( k \). But, the instance \( G \) for \( HAM \) does not need to be complete, does not have edge weights, and no such parameter \( k \). What does this mean for our reduction? It means we have to \textit{construct} a new graph \( G' \), a weight function \( w \), and choose \( k \) to be some specific value such that \( G \in HAM \iff (G', w, k) \in TSPD \).

One such construction works as follows:

\[
\text{(define HAM-graph-to-TSPD-graph}
\begin{align*}
\lambda (G) \\
\text{(let* ((v-names (name-vertices (vertices G))) (make-pairs (lambda (l) (letrec ((loop (lambda (curr pairs) (if (null? curr) pairs (loop (cdr curr) (append pairs (map (lambda (x) (list (car curr) x)) (cdr curr)))))))))) (loop 1 '())))) (e-w-list (map (lambda (x) (list (first x) (second x) (if (edge-in-graph? (make-edge-from-names (first x) (second x)) G) 1 2)))))
\end{align*}
\]

Essentially, to construct \( G' \), we take the vertices to be the same as the vertices in \( G \). For the weight function \( w \), on the edges: if, for vertices \( v_i, v_j \), there exists the edge \((v_i, v_j) \in G\), then put \((v_i, v_j) \in G' \) with weight 1. Otherwise, put \((v_i, v_j) \in G' \) with weight 2. Then, the reduction becomes

\[
\text{(define hamiltonian-cycle?) (lambda (G) (decision-tsp? (HAM-graph-to-TSPD-graph G) weight (length (vertices G))))}
\]

Also note that if \( G \) has \( n \) vertices, then we choose the parameter \( k = n \). To complete the proof, we need to show that \( G \in HAM \iff (G', w, n) \in TSPD \).

For the forward direction, assume that \( G \in HAM \), namely that \( G \) has a Hamiltonian cycle. By definition, this means there exists a sequence of vertices that hits every intermediate vertex exactly once and returns to the starting vertex. Well, let us trace this same sequence of vertices in \( G' \). Since the sequence hit every
vertex exactly once in \( G \), it must do the same in \( G' \) since the vertex sets are the same, and \( G' \) has all possible edges. As for the path weight, since \( G' \) traces the same path as in \( G \), every edge that the sequence takes in \( G' \) must also exist in \( G \). Since both graphs have \( n \) vertices, the sequence must also take \( n \) edges. But, each of these edges in \( G' \) also exists in \( G \), so they all have weight 1. This means the total weight of the sequence is \( n \), and our chosen parameter was also \( k = n \). So, \( (G', w, n) \in TSPD \).

For the reverse direction, assume that \( (G', w, n) \in TSPD \). Since we have a YES instance, there exists a sequence of vertices that hits every intermediate vertex exactly once, returns to the starting vertex, and has total weight \( n \). Since \( G' \) has \( n \) vertices, this sequence must take exactly \( n \) edges, but each edge has weight \( \geq 1 \). This means that any such sequence has total weight \( \geq n \). But, since we have a YES instance of \( TSPD \), the sequence must have weight \( \leq n \), and the two of these together means the total weight is exactly \( n \). The only way this can happen in \( G' \) is, if in each step, the sequence takes a weight 1 edge. Taking any weight 2 edges would give a total weight of at least \( n + 1 \). But, via our construction, every weight 1 edge also exists in \( G \). So, we can trace this sequence of vertices in \( G \), which by definition, forms a Hamiltonian cycle in \( G \). So, \( G \in HAM \).

Both of these steps together show the validity of the reduction, which means that we have reduced \( HAM \) to \( TSPD \). The final step requires us to show that the reduction takes polynomial time. Looking at the Scheme procedure, we see that making all pairs of vertices takes \( O(n^2) \) time. Then, we map the weight function onto each of these pairs. For each pair, the weight function takes at worst \( O(m) \) time, when \( G \) has \( m \) edges. We know \( m = O(n^2) \). Even the most trivial checking function would loop over all edges in \( G \) per each pair of vertices to check for existence in \( G \). So, even the most naive estimate would say the algorithm runs an \( O(n^2) \) check for each of \( O(n^2) \) pairs, giving an \( O(n^4) \) run time, which is still polynomial.

So, \( HAM \leq_p TSPD \), but we knew a priori that \( HAM \) is \( NP \)-Hard. Therefore, \( TSPD \) is also \( NP \)-Hard.

### 6.6 Closing Remarks

Notice a few things about our \( HAM \) to \( TSPD \) reduction. First, this construction was in no way unique. The choice of weights 1 and 2 is not imperative for the reduction to work; rather, we just needed to pick some values such that we could make the argument about the reduction’s validity, and these values presented an elegant choice. When showing the validity, we once again implicitly made use of the contrapositive argument, which meant that we only had to show that \( G \in HAM \iff (G', w, n) \in TSPD \). Also, our analysis of the reduction’s run time was not particularly clever or trenchant. All we needed to show was that it is polynomial, which we successfully did. The final lingering question may have to do with \( NP \) reductions in general, because this reduction seems kind of special to the problem at hand. Well, unfortunately, there is no general schema for actually writing the reduction, it really is constructed on a case by case basis.

### 7 Summary

In summary, we reviewed the Halting Problem and used it to introduce the concept of reduction. By reducing the Halting Problem to other problems, we could show undecidability for a large class of problems. Then, we generalized the concept of reduction and showed that it can be used not only for showing undecidability but hardness among problems as well.

#### 7.1 A note on terminology

The reductions in this lecture are called \textit{many-one reductions}. Many-one reductions are a special case and stronger form of Turing reductions. The many-one reduction is more effective at separating problems into
distinct complexity classes. However, the increased restrictions on many-one reductions make them slightly more difficult to find. The difference in difficulty is negligible for this class, and only arises for more advanced problems.

Many-one reductions are pedagogically better for teaching in CS 230 because of our study of functions using set theory. And remember that every many-one reduction is a Turing reduction!

In future classes, the reductions we have taught you will likely be called “Many-one Reductions.”