1 NP-Completeness

For almost every problem that we've seen in this class, there's a (usually bad) naive algorithm that just considers every possible answer and returns the best one. Some examples are:

- What is the shortest path from s to t in G?
- What is the longest common subsequence?
- What is the cloest pair of points?
- Does G have a topological ordering?

Of course, for nearly every algorithm we've discussed in class thus far, we were able to come up with a clever solution that runs in polynomial time. However, it's not generally the case that such (clever) algorithms will exist.

Note that, in this class, we define an *easy* problem to be one that can be solved in polynomial time and a *hard* problem to be one that cannot be solved in polynomial time.

1.1 Nondeterministic Polynomial (NP)

Such problems are said to be in **Nondeterministic Polynomial** time (NP). To think of this in a theoretical perspective, imagine some nondeterministic computer that can try every possibility in parallel and then return the best possible solution. This type of computer can solve these problems in polynomial time. For example, it is allowed to search for the shortest path by trying every single path in *parallel* and seeing which one is the shortest. There are two types of NP problems.

- **NP-Decision** problems ask if there is some object that satisfies a polynomial time-checkable property. For example, is there a path from s to t? If you write down a path from s to t, then it is easy for the computer to check, in polynomial time, whether this is actually a valid path from s to t. So, effectively, a decision problem is one whose answer is either yes or no.
- **NP-Optimization** problems ask for the object that maximizes (or minimizes) some polynomial time-computable objective. For example, instead of asking if there is a path from s to t, you might instead ask what the shortest path from s to t is. So, effectively, an optimization problem is one whose answer is a minimum or maximum value.

Essentially, NP is the set of problems for which you can *verify* the answer in polynomial time. Alternatively, it's the set of problems that can be solved in polynomial time by a *nondeterministic* computer.

1.2 Difference Between Decision & Optimization Problems

Now, note that NP-Decision and NP-Optimization problems are not too different.

- Every decision problem can be phrased as an optimization problem.
- Every optimization problem has a decision form.

For theoretical reasons, it's a good idea to understand the distinction. But, for practical purposes, it doesn't really matter.

1.3 Examples of NP Problems

Some examples of NP problems are:

- Formula-SAT (NP-Decision).
- Traveling Salesman Problem (NP-Optimization).

- Hamiltonian Cycles (NP-Decision).
- Generalized Knapsack (NP-Optimization).
- Maximum Independent Set (NP-Optimization).

1.3.1 Formula-SAT

Given a logical formula in a number of Boolean variables, is there an assignment to the variables that causes the formula to be true?

For example, consider the following formulas:

- Formula 1: If $x =$ True, $y =$ True, and $z =$ False, then the following formula is satisfied:

$$(x \lor y) \land (y \lor z) \land (z \lor x) \land (\overline{x} \lor \overline{y} \lor \overline{z})$$

- Formula 2: No assignments of x, y, and z can satisfy the following formula:

$$(x \lor y) \land (y \lor z) \land (z \lor x) \land (\overline{x} \lor \overline{y}) \land (\overline{y} \lor \overline{z}) \land (\overline{x} \lor \overline{z})$$

There are some applications of SAT. In particular:

- Circuit Design: Given some circuit, you want to make sure it actually computes the function that it's supposed to compute. This is basically a satisfiability problem.
- Logic Puzzle: Most logic puzzle usually come down to some satisfiability problem; you want to find some settings or variables that satisfy some set of rules.

1.3.2 Hamiltonian Cycles

Given an undirected graph G, is there a cycle that can visit every vertex exactly once?

1.3.3 General Knapsack

Recall that knapsack has a number of items, each with a weight and a value. The goal is to find the set of items whose total value is as much as possible without the total weight exceeding some capacity.

The general knapsack problem essentially runs in polynomial time in the *weights*. Note that if the weights are allowed to be large (for example, written in binary), then you don't have a good algorithm.

1.4 Brute-Force Search

Every NP problem has a brute-force search algorithm (the naive algorithm). In this class, we have looked at problems with algorithms that substantially improve on these brute-force algorithms. The question, then, is: *does every NP problem have a better-than-brute-force algorithm?* Put it another way, *is it the case that every algorithm in NP has a polynomial time algorithm?*

- If this is the case, then every NP problem has a reasonably efficient solution. No matter what kind of complicated search or decision problem we want to solve, there is always a clever way to do it better than brute-force.
- If this is *not* the case, then some NP problems are fundamentally difficult. There are some problems which are easy to check but there is no straight-forward or efficient way to actually find the answer.

Note that P is the set of all problems that can be solved in polynomial time.

1.5 Reductions

In practice, at least some problems in NP appear to be hard. Despite decades of trying, people still don't know particularly good problems. Suppose you have a problem. How do you know if it's hard or not?

- You can search for an efficient algorithm to show that the problem is easy.
- You can try to prove that the problem is hard, but this is difficult.
- You can try to relate the difficulty of your problem to the difficult of other problems.

With the last bullet point, we'll introduce the notion of *reductions*.

1.5.1 What Is It?

Reductions are a method for proving that one problem is *at least as hard* as another problem. You can think of this as proving inequalities about problems. For example, problem A is as hard as problem B.

The way we do this is as follows: we can show that *if* there is an algorithm for solving A, then we can use this algorithm to solve B. Therefore, B is no harder than A.

To better understand this concept, consider the following statements:

- If problem X is no harder than problem Y, and if Y is easy, then X must also be easy.

> To get the intuition behind this, suppose you have two numbers x and y such that x is no greater than y; that is, $x \leq y$. Suppose y is negative. Then, x must also be negative (it cannot be positive).
>
> The same idea holds here. If we have two problems X and Y such that X is no harder than (or at least as easy as) Y and Y is easy, then X cannot be harder than Y; X has to be *as easy as, or easier than,* Y.

- If problem X is no harder than problem Y, and if X is hard, then Y must also be hard.

> Again, to get the intuition behind this, suppose you have two numbers x and y such that x is no greater than y; that is, $x \leq y$. Suppose x is positive. Then, y must also be positive (it cannot be negative).
>
> The same idea holds here. If we have two problems X and Y such that X is no harder than (or at least as easy as) Y and X is hard, then Y cannot be easier than X; Y has to be *as hard as, or harder than,* X.

1.5.2 Reducing Hamiltonian Cycle to Traveling Salesman Problem

We note that there is a natural reduction here. Particularly, they're related in the sense that we're visiting each vertex exactly once.

- Recall that the Traveling Salesman Problem asked: *Given a weighted (undirected) graph G with n vertices, find a cycle that visits each vertex exactly once whose total weight is as small as possible.*
- Recall that the Hamiltonian Path problem asked: *Given an undirected graph G, is there a cycle that can visit every vertex exactly once?*

To see this in action, consider the following example:

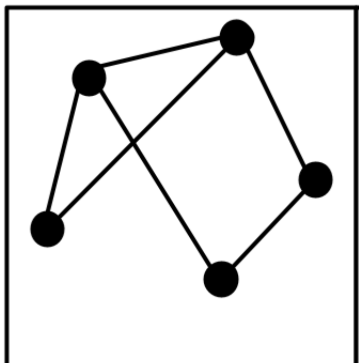


Figure 1: Graph **G**

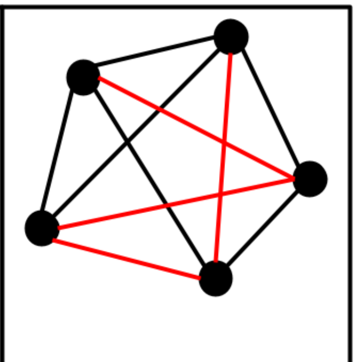


Figure 2: Graph **H**

Suppose we have an algorithm that solves the Traveling Salesman Problem efficiently. The idea is to convert a Hamiltonian Cycle problem to a Traveling Salesman problem. We do this by taking the original graph G from the Hamiltonian Cycle problem and "copying" it to the Traveling Salesman problem, which we'll call the copied graph H. For all edges in G, we can give the corresponding edges in H a weight of 1 (in Figure 1 and 2, the *black* edges). Then, any extra edges that we add to H (in Figure 2, the *red* edges) can be given a very expensive weight (say 1000).

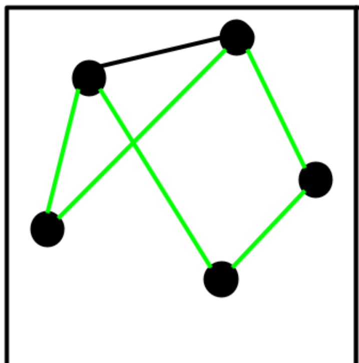


Figure 3: Graph **G**

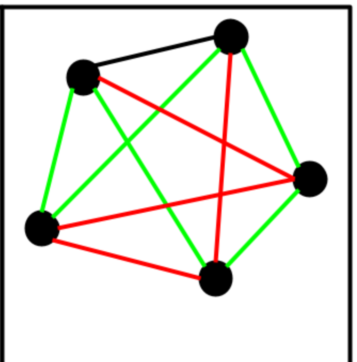


Figure 4: Graph **H**

Looking at Figure 4, we see that there is a Traveling Salesman path of cost 5. But, the path taken in the Traveling Salesman problem is the same path found in the Hamiltonian Cycle problem. So, the cheapest path we can hope for in the Traveling Salesman problem is the path consisting only of cost 1 edges. So, if we can find a path in the Traveling Salesman problem consisting only of cost 1 edges, then we have a Hamiltonian Cycle. *Otherwise*, we'll have to make use of some expensive edges, so that implies that no such Hamiltonian Cycle can exist.

To be more formal about it, suppose you have a Hamiltonian Cycle instance G with n vertices. Suppose that you have an efficient algorithm A that solves the Traveling Salesman Problem[1] Now, we create a Traveling Salesman problem instance H where:

- The edges in G have cost 1.
- The edges not in G have cost 1000 (or some expensive weight).

Then, using this Traveling Salesman Problem algorithm, we can solve the Traveling Salesman instance. We note that we have a path of cost n in H if and only if there is a Hamiltonian cycle in G. Therefore, we can use this answer to solve the initial problem.

[1] We don't need to know how this algorithm works, only that you *have* one.

- If there is a Hamiltonian Cycle in G, then the Traveling Salesman Problem will have a weight of n. If there is *no* Hamiltonian Cycle, then the best cost will be greater than n.
- Any Traveling Salesman Problem that uses these expensive-costing edges will be more expensive than if we decided to use the edges with cost 1.
- Therefore, we can run $A(H)$. If $A(H) \leq n$, then G has a Hamiltonian cycle. Otherwise, it does not.

From this, we get a new algorithm that can solve the Hamiltonian Cycle. This gives us a relation: the Hamiltonian Cycle problem is *no harder* than the Traveling Salesman Problem. Now, since Hamiltonian Cycle is in NP, then it follows that the Traveling Salesman Problem must also be in NP.

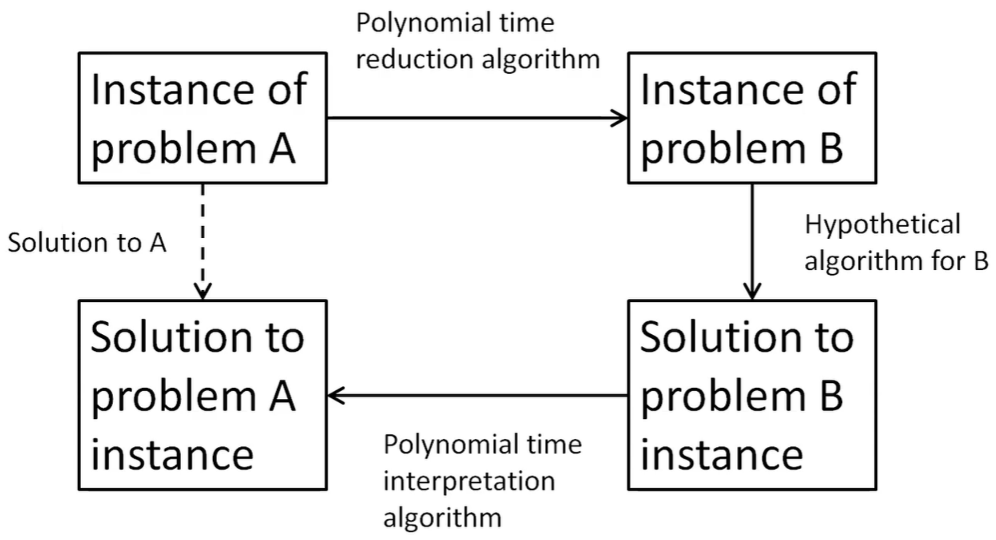
So, what we've shown is that if we have an algorithm that can solve the Traveling Salesman problem, then we can turn that into an algorithm that solves the Hamiltonian Cycle path.

1.5.3 Generalization

If we want to find a reduction from problem A to problem B, then what we want to show is:

- If we are given an algorithm to solve that solves problem B, we can turn that into an algorithm that solves problem A.
- Generally, we start with an instance of problem A. Then, with some polynomial time reduction algorithm, we can turn this instance of problem A into an instance of problem B. Then, with a *hypothetical* algorithm for problem B, we can find a solution to the problem B instance. Then, we need to turn this solution of the problem B instance into a solution for the problem A instance; this is done by using a polynomial time interpretation algorithm.

This process looks something like:


Polynomial time reduction algorithm
Instance of problem A
Instance of problem B
Solution to A
Hypothetical algorithm for B
Solution to problem A instance
Solution to problem B instance
Polynomial time interpretation algorithm

It's important that the reduction and interpretation algorithms run in *polynomial time* by construction. If you have a polynomial time algorithm for problem B, then this entire process runs in polynomial time (i.e. you have a polynomial time algorithm for problem A). If you don't have a polynomial time algorithm for problem B, then there might be some other way to solve problem A, but you can't use this particular method.

So, effectively, if we have algorithms for reduction and interpretation, then:

- Given an algorithm to solve problem B, we can turn it into an algorithm to solve problem A.
- This means that problem A might be *easier* to solve than problem B, but it cannot be *harder*.