NP-Completeness

Ross M. McConnell *

For this reading, we will use the following graph as a running example, which is coded into the `main` and some of the methods:

```
0
/
/ |
/  |
/
1   2
/
/  |
/   |
/
3
/
/  |
/   |
---
4
/
/  |
/   |
/
5
/
/  |
/   |
6
```

Then run `ipython -i NPC.py` to see how the graph is printed out.

Three Optimization Problems

Our usual measure of the running time is as a function of some natural measure that is specific to the problem at hand. For sorting, it is the number $n$ of elements; for graphs is it $n + m$, where $n$ is the number of vertices and $m$ is the number of edges. A universal measure, which we must use when talking about a broad class of algorithms, is the running time as a function of the number of bits required to specify the instance of the problem. This is the measure that we will adopt here, with a small exception in one case.

We are only interested in the distinction between polynomial and exponential time bounds, so the way you choose to represent the inputs with bits makes little difference to this question, as long as you don’t go out of your way to represent the inputs inefficiently, such as representing integers in base-1 (the stone-age number system).

An independent set in a graph is a set of vertices, no two of which are adjacent. The problem of finding a maximum independent set is the problem of finding an independent set of maximum size. Conceptually, imagine that you are putting together a list of people for a party, the edges are pairs of people that don’t get along, and you want to assemble as big an invitation list as you can without inviting any two people who don’t get along. (There are serious variants on this idea that come up in industrial problems. For example, the “people” can be jobs to be scheduled, and the pairs that “don’t get along” are pairs of jobs that conflict over use of a resource.)

- In our running example, $\{1, 3, 6\}$ is an independent set, and $\{1, 3, 4, 6\}$ is a maximum independent set.

In NPC.py is a method called `maxIndSet` that finds a maximum independent set in a graph. Try it by running `ipython -i NPC.py` and then running `G.maxIndSet()` from the Python prompt. Make sure it returns $\{1, 3, 4, 6\}$.

*Dept. of Computer Science, Colorado State University, Fort Collins, CO 80523-1873, USA. rmm@cs.colostate.edu
Study the code to see how it works. It is a variant of our first cut on dynamic programming problems: we have two recursive calls, one where we include a certain element and one where we exclude it. The problem is that the recursion tree has so many distinct subproblems that a table does us no good. We would need a table of exponential size, and it wouldn’t give us a polynomial time bound.

A weighted version of the problem is the problem of finding a maximum-weight independent set. In this one, the vertices can have different weights, and you want an independent set of maximum total weight.

This is the variant of the job scheduling problem where the jobs have different profits.

```
(1)
(1)
(2)
(2)
(1)
(3) (6)
```

- In this example, a maximum-weight independent set is \{3, 5\}, which has weight 8.

Run ipython -i NPC.py, and from the Python prompt, run `G.maxWeightIndSet(Weights)`. Make sure you get this answer.

A vertex cover is a set of vertices that “cover” all edges. That is, every edge has at least one endpoint in the set. The problem of finding a minimum vertex cover is the problem of finding a vertex cover of minimum size. Think of a bad neighborhood where the police want to station officers at intersections so that every street has a policeman on at least one end. The police want to “cover” all the streets by being able to look down them from at least one end. They want to minimize the number of cops assigned to these jobs.

- In the running example, \{0, 2, 5\} is a minimum vertex cover.

In NPC.py is a method called `minVertCover` that finds this. Run NPC.py, and from the Python prompt, run `G.minVertCover()` to make sure you get this answer.

All three of these problems are optimization problems, which means that you want to minimize something or maximize something, subject to a set of constraints. Nobody knows a polynomial time bound for any of them. The solutions we know of, including the one’s I’ve coded, will overwhelm our computing resources on even fairly small examples of the problems.

The goal of the study of NP-completeness is to be able to identify problems that there is little hope of finding a polynomial-time algorithm for.

Why would we want to show this? If we know that a problem is hopeless, we can avoid wasting time trying to find a good algorithm for it, and instead focus our attention on heuristics and approximation algorithms that usually give us something inferior to an optimum solution, but in a reasonable amount of time. Knowledge of NP-completeness saves billions of dollars a year in resources that would have otherwise been spent on wild goose chases.
Three Corresponding Decision Problem

Though the goal of NP-completeness is usually to show that an optimization problem is hopeless, we get our traction by studying decision problems, which are problems that have a one-bit answer: yes or no.

Every optimization problem has a corresponding decision problem of the form, Can we get a solution that is at least this good?

- The decision problem for maximum independent set, which we will call INDEPENDENT-SET, takes as inputs a graph and an integer $k$, and asks whether the graph has an independent set of size $k$. The answer is “yes” if one exists somewhere in the graph, and “no” otherwise. In the running example, the answer is “yes” for $k \leq 4$, and “no” for $k \geq 5$, since the largest one has size four.

- The decision problem for maximum-weight independent set, which we will call WEIGHTED-INDEPENDENT-SET takes as inputs a graph, vertex weights, and a value $w$, and asks whether there is an independent set of total weight at least $w$.

- The decision problem for minimum vertex cover, which we will call VERTEX-COVER, takes as inputs a graph and an integer $k$, and asks whether there is a vertex cover in the graph of size at most $k$.

In the posted Python file, find the implementations of algorithms for these decision problems. In the code, they are solved by calling the methods for the optimization problems, but this might not be the only way to solve them. We say that these methods are reductions of the decision problems to the optimization problems. The reductions take polynomial time; what takes exponential time is solving the optimization problems.

We can draw the following conclusions from the reductions you see in the code:

- If there exists a polynomial-time solution to one of the optimization problems, there exists one for the corresponding decision problem;

- The contrapositive of this: If there doesn’t exist a polynomial-time solution to one of the decision problems, then there doesn’t exist one for the corresponding optimization problem.

The latter observation is the most important, because it means that if we prove that there doesn’t exist a polynomial algorithm for the decision problem, it will prove that there doesn’t exist one for the optimization problem. If we fail to prove this, but provide compelling evidence that there isn’t one for the decision problem, it will provide compelling evidence that there isn’t one for the optimization problem.

This allows us to focus on dashing any hope of solving decision problems in polynomial time, in order to show that there is no hope of solving the corresponding optimization problems in polynomial time. By focusing on decision problems, we can dash our hopes for the optimization problems with less work and a cleaner theory.
NP: A Class of Decision Problems

All three of the decision problems have the property that if the answer is yes, there exists a way to verify this in polynomial time, but it might require somebody to give you a hint (“certificate”). If somebody points out the right hint for an instance of the independent set problem, namely, an independent set of the given size $k$ in the given graph, it is easy to check whether they are telling the truth. You will know with certainty that the answer to your instance of the problem is “yes” in polynomial time. Without the hint, we don’t know how to do this. The hint is hiding in plain sight, camouflaged among the other vertices, but we don’t know how to smoke it out in polynomial time.

`INDEPENDENT_SET_HINT` in the posted Python file is an algorithm that checks that they are telling the truth, and it runs in polynomial time. If they give a bad hint, it returns False, which may mean that the answer is “no” or just that they gave a bad hint. It only returns True if the hint is good, such as if it ever returns True, we know that the answer to the decision problem really is True.

Similarly, `WEIGHTED_INDEPENDENT_SET_HINT` and `VERTEX_COVER_HINT` do the same for the other two optimization problem.

NP stands for nondeterministic polynomial, which is a fancy way of saying that if the answer is “yes,” you will know your answer in polynomial time if you get lucky and stumble on the right hint.

The class P is the set of decision problems where, if the answer is yes, you can find this out in polynomial time without hints or luck. An example is this: Does there exist a path of length at most $k$ from a given vertex $x$ to a given vertex $y$ in a given graph? You can solve this in polynomial time using BFS without any hints.

To be in NP, a problem can require a hint, but it doesn’t have to. Therefore, $P \subseteq NP$. That is, every problem in P is also a problem in NP.

`INDEPENDENT-SET`, `WEIGHTED-INDEPENDENT-SET`, and `VERTEX-COVER` are problems that are in NP, but not known to be in P. Nobody has found a polynomial algorithm that solves them without a hint. Here are some other problems that are in NP, but not known to be in P:

- **SAT**: Given a boolean expression, does there exist an assignment of values to its variables so that the expression evaluates to True? If the answer is yes, then the hint is an assignment of values that causes it to return True. Given the hint, it’s easy to verify that the answer is yes by just plugging the hint in for the values of the variables.

- **SUBSET-SUM**: Given a weight limit $t$ and a set of items to choose from, each with a weight, is it possible to reach a total weight of exactly $t$ with some subset of the items?

- **KNAPSACK**: Given a weight limit and set of items to choose from, each with a weight and a value, is it possible to achieve a value of at least a given $r$ without exceeding a given weight $w$? A hint is a set of items that achieves $r$ without exceeding weight $w$. You can obviously verify the hint in polynomial time, so the problem is in NP.

- **HAMILTONIAN-CYCLE**: Given an undirected graph, is there a cycle in the graph that visits all the vertices without doubling back through any of them? If somebody points out such a cycle, it’s easy to verify that the answer is yes.
• HAMILTONIAN-PATH: Given an undirected graph, is there a path in the graph that visits each vertex exactly once.

Here is one that was neither known to be in P or NP-complete until about 12 years ago:

• COMPOSITE: Given an positive integer $n$, is $n$ composite, that is, the product of two other integers? If the answer is yes, then a factor, $m$, serves as the hint. In time polynomial in the number of bits in $n$, you can verify that $m$ divides evenly into $n$. This problem was shown to be in P when some researchers found an algorithm that solves it in polynomial time without a hint. It is still the case that nobody knows a polynomial algorithm to find the obvious hint! We are lucky for this, because all of our encryption depends on nobody getting hold of such an algorithm. To make matters worse, nobody has proven that such an algorithm doesn’t exist. This is an enormous security concern.

Aren’t all decision problems in NP? The answer to this is no; there is not always a hint that will allow someone to verify a yes answer in polynomial time. Examples often come up in analyzing games. Here is the intuition.

In Tic-Tac-Toe, two perfect players will always draw. What about chess? Two perfect players, such as two omniscient deities, will either always draw, or white will always win, or black will always win. The answer exists, but we don’t know which of these is the case.

Consider the question, “Can White always force a win?” Suppose you are an omniscient deity and you have examined the entire game tree and know that the answer is yes. How can you prove this to an skeptical mortal? There seems to be no way. You can tell her, “I’ve seen it all! White will always win!” She will say he doesn’t believe you without proof. Being mortal, she will die before you can ever step her through all the steps of your proof.

Though this example conveys the intuition, it has a flaw. A critic of this argument about chess will say that there are $O(1)$ possible configurations of the chessboard, hence the game tree has $O(1)$ size, hence it takes $O(1)$ time to step someone through it. Even though it has a very large constant hiding inside the big-O, $O(1)$ is a polynomial bound.

To get around this argument, our book, the Kleinberg and Tardos algorithms text, considers a game that doesn’t have $O(1)$ size, the Facility Location Problem, in Chapter 1. The intuition is the same, but but it silences this criticism.

Also, notice that when the answer is “no” to any of the above problems in NP (other than COMPOSITE), there doesn’t seem to be any way you could show this in polynomial time.

How a Problem Can Be NP-Complete

A problem is $NP$-complete if a polynomial-time algorithm for it would imply that there is a polynomial-time algorithm for all problems in NP, that is, that $P = NP$. We do not know whether $P = NP$, but most people believe that it is not. There is a one million dollar prize waiting at the Clay Institute for anybody who finds a proof either way.

When the class NP was first described, nobody dreamed that there would turn out to be such a thing as an NP-complete problem. In this section, I will explain the idea behind the proof that there is. This was shown in 1971 by Stephen Cook, who received the Turing Award – the Computer Science equivalent of the Nobel Prize – for it. He showed that a problem called SAT was NP-complete, but I will illustrate it on a related problem that I will call ASAT. This requires me
to take some inconsequential liberties with the definition of its input problem size, but it has the advantage of making it much easier to understand and to play with in a language such as Python.

We say that a method that returns True or False is *satisfiable* if there exists an input that causes it to return True. A boolean method that is not satisfiable is pretty worthless; we might as well replace any call to it with the boolean expression `False`. It would be nice if compilers alerted us when a complicated method we wrote is not satisfiable, since, if this is not what we intended, it would indicate the existence of a bug. There is a reason that they don’t, which is that nobody knows a polynomial time algorithm to figure this out. Moreover, we can show that it is unlikely that anybody will find one, since if they did, it would imply that every problem in NP has a polynomial algorithm, and thousands of problems in NP have resisted every attempt to find a polynomial algorithm for them.

Look at the method called `ASAT` (“algorithm satisfiability”). Its inputs are a reference to a boolean method, `Method`, whose input is a sequence of $b$ bits. It returns a boolean value that is True if there exists an input sequence of $b$ bits that get `Method` to return True. Otherwise it returns False. A precondition is that `Method` halts on every input. We will say that `ASAT` has a polynomial algorithm if the algorithm runs in time that is polynomial in the length of the code for `Method`, plus the number $b$ of bits it takes as input, plus the running time of `Method`. This is a slight variant of our definition of the running time, to avoid a technical loophole, but since we are only passing in methods that implement polynomial algorithms, the distinction is irrelevant for our purposes.

`ASAT` is in NP, since, if `Method` is satisfiable, there exists a bit string that we can input to it and verify in polynomial time in what we have defined to be the input size that it is, in fact, satisfiable.

Other sources use SAT or a closely related problem to show that there exists an NP-complete problem.

To make things easier to program, I am assuming that the method to be tested takes its input bits as a Python list of $k$ integers, where each integer is between 0 and `maxValue`. In other words, its input bits are the last $\lfloor \log_2 \text{maxValue} \rfloor + 1$ bits of each of the $k$ integers, and the number of input bits to it is $b = k \cdot \lfloor \log_2 \text{maxValue} \rfloor$. Any polynomial-time method that takes a sequence of input bits can be rewritten to take its input in this format, possibly by padding the input string with some extra bits that it throws away if $b$ is not a multiple of $\lfloor \log_2 \text{maxValue} \rfloor$.

The algorithm I’ve written for `ASAT` is not polynomial. Let’s now consider what would happen if it could be rewritten to run in polynomial time, using a different algorithm. This would imply that we would get a polynomial-time solution to INDEPENDENT-SET, as follows. Look at `IndSetGadget`. It is written so that you can hardcode an instance of INDEPENDENT-SET in its code, and the only input is the hint. It returns True if the hint is good, and False otherwise.

In this case, `IndSetGadget` asks whether our example graph has an independent set of size 4.

**Here is the punchline:** with the following, we are asking whether `IndSetGadget` can ever be induced to return True.

```
In [1]: ASAT(IndSetGadget, 7, 7)
Out[1]: True
```

The last two parameters explain to `ASAT` that a hint to the gadget must be a list of seven integers, each at least 0 and at most 7. In other words, only the last three bits of each integer are used by the method. (Seven is the largest three-bit number, and we need three bits to express
a vertex number between 0 and 6.) The hint is a way to list up to all seven vertices, some of
them possibly more than once. When the gadget turns the list into a frozenset, it gets rid of any
duplicates in the list.

The fact that this returns True means that there is an independent set of size 4; it is only
possible to get our gadget to return True if such an independent set exists.

Now, change the code of IndSetGadget by changing the line IndSetSize = 4 to IndSetSize
= 5. Now it is satisfiable if our graph has an independent set of size 5. Run the test again:

In [1]: ASAT(IndSetGadget, 7, 7)
Out[1]: False

This says that our gadget always returns False, which means that our graph doesn’t have an
independent set of size 5.

A polynomial algorithm for ASAT would imply that you could get the answer for any instance
of INDEPENDENT-SET by changing the instance in the gadget to be the instance you want to
solve the INDEPENDENT-SET problem on.

The same trick can be used on any problem X in NP, using the fact that these are problems that
can be solved in polynomial time given the right hint. Compile a method that checks a proposed
hint for any given instance of X, and then ask whether there exists an input that causes the method
to return true using a call to ASAT. This would give a polynomial-time algorithm for X if ASAT is
were polynomial. Since X is an arbitrary problem in NP, this would imply a polynomial algorithm
for all problems in NP.

Other NP-Complete Problems

Once one problem was shown to be NP-complete, it was easy to use this to show that other problems
in NP are NP-complete. The fact that ASAT is NP-complete can be used to show that SAT is
NP-complete. The fact that SAT is NP-complete can be used to show that another problem in
NP named 3CNF-SAT is NP-complete, etc. The fact that 3CNF-SAT is NP-complete can be used
to show that INDEPENDENT-SET is NP-complete. The fact that INDEPENDENT-SET is NP-
complete can be used to show that WEIGHTED-INDEPENDENT-SET and VERTEX-COVER
are NP-complete. The fact that VERTEX-COVER is NP-complete can be used to show that
SUBSET-SUM is NP-complete.

The following figure summarizes this chain of proofs:
Each arrow is an *NP-completeness proof*. You start with a problem A that’s known to be NP-complete. This means that if there is a polynomial-time algorithm for A, then \( P = NP \). Then you show that a new problem B must also be NP-complete by giving a polynomial time reduction of A to B. This shows that if there is a polynomial-time algorithm for B, there is one for A, hence \( P = NP \). In other words, B is also NP-complete.

Find the method called **INDEPENDENT_SET2**. This is the NP-completeness proof corresponding to the arrow in the diagram from INDEPENDENT-SET to WEIGHTED-INDEPENDENT-SET. By the time someone got around to doing this proof, they already knew that INDEPENDENT-SET was NP-complete. The method shows that if there is a polynomial-time method for WEIGHTED-INDEPENDENT-SET, then there is one for INDEPENDENT-SET, which means that WEIGHTED-INDEPENDENT-SET is also NP-complete.

The next method, **INDEPENDENT_SET3** shows a reduction to VERTEX-COVER. It takes advantage of the fact that a set is an independent set if and only if its complement is a vertex cover. Therefore, you can find a maximum independent set by finding a minimum vertex cover, and assembling all of the vertices that are not in the vertex cover. In our example above, you may have noticed that our minimum vertex cover was \( \{0, 2, 5\} \), and our maximum independent set was \( \{1, 3, 4, 6\} \), the vertices not in the vertex cover. Every edge is incident to one of the edges of the vertex cover, which means that the vertices not in the vertex cover are an independent set.

A *literal* in a boolean expression is a variable or its negation. A boolean expression is in *3CNF form* if it consists of the AND of a set of clauses, each of which is the OR of a set of literals. An example is \((\neg x_1 \text{ or } \neg x_2 \text{ or } x_3) \text{ and } (x_1 \text{ or } \neg x_2 \text{ or } \neg x_3) \text{ and } (x_1 \text{ or } x_2 \text{ or } x_3) \text{ and } (\neg x_1 \text{ or } x_2 \text{ or } x_4)\). Whether a boolean expression in 3CNF form is satisfiable is called 3CNF-\(SAT\), and it’s NP-complete.

The following illustrates how we can use this to show that INDEPENDENT-SET is NP-complete. We can reduce this to INDEPENDENT-SET in polynomial time. The following illustrates the reduction for the above 3CNF expression:

We create a vertex for each occurrence of a literal and a clique for each of the \( k \) clauses. Two literals in different clauses have an edge if they can’t both be true. Notice that for the expression to be satisfied, we have to select at least one literal in each clause to be true. We can do this if and
only if the constructed graph has an independent set of size $k$.

For example, in the above picture, selecting not $x_1$ from the left clause, not $x_2$ from the top clause, $x_3$ from the right clique, and $x_4$ from the bottom clique gives an independent set. Therefore, setting $x_1 = False$, $x_2 = False$, $x_3 = True$, and $x_4 = True$ will ensure that the expression is satisfied. If there is no independent set of size $k$, there is no way to select one literal in each clause to be true, and the expression is not satisfiable.

The time to create the graph is polynomial in the length of the expression, so a polynomial-time algorithm for INDEPENDENT-SET would imply a polynomial-time algorithm for 3CNF-SET, hence that $P = NP$. This shows that INDEPENDENT-SET is NP-complete, provided it is known that 3CNF-SAT is NP-complete, which is known.

**How to figure out that an optimization problem is NP-hard**

An optimization problem is **NP-hard** if a polynomial-time algorithm for it would imply $P = NP$. In other words, if you can show that an optimization problem is NP-hard, you will have shown that a polynomial-time algorithm for it would imply that An optimization problem is NP-hard if the corresponding decision problem is NP-complete.

Following the above lead, people have shown that there are thousands of NP-complete problems. Some of them are of interest to industry in their own right, but most of them serve to show that the corresponding optimization problems are NP-hard.

Most industrial optimization problems are NP-hard, unfortunately. Optimal traffic-light scheduling and airline scheduling are examples. Think about that the next time you’re waiting at a light on College or stuck on the tarmac.

Knowing about this, however, saves billions of dollars every year in programmer time that would be otherwise wasted on futile algorithm-design efforts.

How can you use it when you graduate? You won’t spend any time figuring out how to get ASAT to give an answer to an NP-complete problem. I am having you study that only so that you can understand how there can be a problem such that a polynomial-time algorithm would imply $P = NP$. Where you could spend time is in trying to figure out whether an optimization problem you are working on is NP-hard. Here is the recipe:

1. Find a decision problem in NP that corresponds to your optimization problem. It should be of the form, *does there exist an answer that’s at least as good as $x$*. Since you can solve the decision problem using an algorithm for your optimization problem, a polynomial-time algorithm for your optimization problem would imply one for your decision problem.

2. Find a reduction of a known NP-complete problem to your decision problem. The reduction should take polynomial time; the only hitch to getting a polynomial-time algorithm to the NP-complete problem should be the running time of an algorithm for your decision problem. This will imply that your decision problem is NP-complete.

If you succeed, you will have shown that a polynomial-time algorithm for your optimization problem would imply $P = NP$, and you should give up on looking for algorithms that find optimum solutions for it.

To do this, you need to get good at figuring out how to reduce a known NP-complete problem to the decision problem corresponding to optimization problems that come up at your job. It’s a skill that takes practice.
More examples of NP-completeness proofs

- According to the drawing above, the NP-completeness of HAMILTONIAN-CYCLE can be used to show that HAMILTONIAN-PATH is NP-complete. Here’s how.

Suppose you have a graph $G$ and you want to know whether it has a Hamiltonian cycle (A). Pick a vertex $a$ and add a new vertex $x$ adjacent only to $a$. Add a vertex $y$ that’s adjacent to all the neighbors of $a$, and add a vertex $z$ that’s adjacent only to $y$ (B). If your original graph $G$ has a Hamiltonian cycle, then this can be extended to a Hamiltonian path in the new graph with endpoints at $x$ and $z$. Conversely, if the new graph has a Hamiltonian path, it must have one endpoints at $x$, proceed through $a$ and then all the other vertices of $G$, through a neighbor $b$ of $a$, and on to $y$ and $z$ (C). Removing $x$, $y$ and $z$ from the ends of this path and adding edge $(b, a)$ gives a Hamiltonian cycle of our original graph (D).

Therefore, the new graph has a Hamiltonian path if and only if $G$ has a Hamiltonian cycle. A polynomial-time algorithm for HAMILTONIAN-PATH would imply that we could find the answer to any instance of HAMILTONIAN-PATH in polynomial time, which would imply $P = NP$. Our conclusion: HAMILTONIAN-PATH is NP-complete also.

- You are the administrator of a network. You found a minimum spanning tree $T$ of the network to route packets on. Some of the customers own nodes that have high degree in $T$, and they have complained that this is unfair, because their node spends all of its time routing packets for others.

You decide to find a spanning tree that minimizes the maximum degree, to make it as fair as possible. You figure that this can’t be too hard, since finding a minimum spanning tree is polynomial. After trying for awhile to come up with an efficient algorithm, you begin to suspect it’s NP-hard.

To resolve the issue, you think up the corresponding decision problem: Does a given graph have a spanning tree in which no node of the tree has degree greater than a given $k$?
This problem is in NP, since if someone points out such a spanning tree, it’s easy to verify that they are telling the truth in polynomial time. Let’s call this decision problem BOUNDED-SPANNING-TREE.

Let’s show it’s NP-complete by reducing a known NP-complete problem to it. Here’s a nice reduction:

```python
def HAMILTONIAN-PATH(self):
    return self.BOUNDED-SPANNING-TREE(2)
```

The reason it works is that a Hamiltonian path is the same thing as a spanning tree where no node of the tree has degree greater than two.

Thus, BOUNDED-SPANNING-TREE is NP-complete, and a polynomial-time algorithm for your optimization problem could be used to solve it in polynomial time. Your network optimization problem is NP-hard.