4.4 More "NP-Complete" Problems

Definition 6 *A problem in NP which has the property that every problem in NP can be transformed into it by a polynomial-time transformation, is called* NP-complete.

Cook's Theorem states that SAT is NP-complete. In 1972, Richard Karp ([Kar72]) showed that other problems were NP-complete as well. VERTEX COVER was one of them.

Theorem 22 VERTEX COVER *is NP-complete.*

Proof It is easy to see that VERTEX COVER can be solved by the exhaustive search approach that costs only "nondeterministic polynomial time" — VERTEX COVER is in NP.

To show that VERTEX COVER is NP-complete we describe a polynomial transformation from SATISFIABILITY into VERTEX COVER. Since there is a polynomial-time transformation from any problem in NP to SATISFIABILITY this then implies that there is (a two-step) polynomial-time transformation from any problem in NP to VERTEX COVER.

This transformation is made easier if we transform only a special form of SATISFIABILITY known as "SATISFIABILITY of boolean expressions in 3-conjunctive normal form," or 3-SAT for short. Expressions of this form look like this:

$$(x \vee \neg y \vee z) \wedge (w \vee z \vee w) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (z \vee \neg w \vee \neg z)$$

They are conjunctions (ANDs) of disjunctions (ORs) of three "literals" — variables or negated variables. (There is no requirement that literals in a clause have to be different.) Each 3-literal disjunction is called a "clause" of the expression.

For such a transformation of 3-SAT to VERTEX COVER to prove that VERTEX COVER is NP-complete we have to show that 3-SAT is NP-complete. We do this by slightly modifying our original proof showing that SATISFIABILITY was NP-complete.

That proof showed that SATISFIABILITY of circuits was NP-complete. Turning circuits into boolean expressions results in one "clause" per gate, with all those clauses ANDed together. If we could turn each clause into 3-conjunctive normal form, we would have shown 3-SAT to be NP-complete.

To simplify matters we could build those circuits out of just one type of gate. Let's choose NAND-gates. (NOR-gates would have been the other possibility.) Each gate then turns into a clause of the form

$$(x \vee \neg y \vee z) \wedge (x \vee z \vee w) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee w \vee \neg z)$$

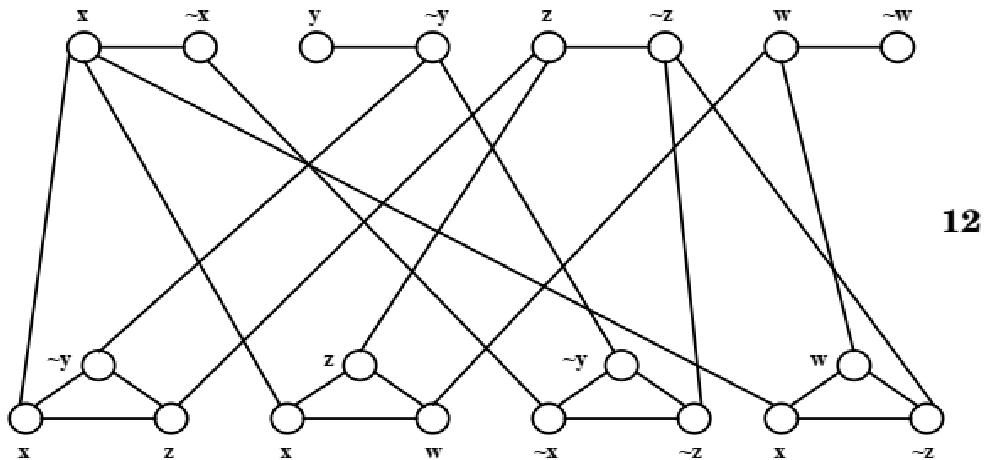

Figure 4.6: Translating 3-SAT into VERTEX COVER

$$(\neg(x_t \wedge y_t) \Leftrightarrow z_{t+1})$$

which we could split into

$$(\neg(x_t \wedge y_t) \Rightarrow z_{t+1}) \wedge (z_{t+1} \Rightarrow \neg(x_t \wedge y_t))$$

and further rewrite into

$$((x_t \wedge y_t) \vee z_{t+1}) \wedge (\neg z_{t+1} \vee \neg(x_t \wedge y_t))$$

One more rewriting get us close to the right form:

$$(x_t \vee z_{t+1}) \wedge (y_t \vee z_{t+1}) \wedge (\neg z_{t+1} \vee \neg x_t \vee \neg y_t)$$

The rewriting into 3-SAT is complete if we repeat a literal in those clauses that are short of having three literals:

$$(x_t \vee z_{t+1} \vee z_{t+1}) \wedge (y_t \vee z_{t+1} \vee z_{t+1}) \wedge (\neg z_{t+1} \vee \neg x_t \vee \neg y_t)$$

This proves

Lemma 5 3-SAT *is NP-complete.*

What's left to do in order to show that VERTEX COVER is NP-complete is to transform 3-SAT into VERTEX COVER. As an example, consider the expression

$$(x \vee \neg y \vee z) \wedge (x \vee z \vee w) \wedge (\neg x \vee \neg y \vee \neg z) \wedge (x \vee w \vee \neg z)$$

The translation algorithm is as follows. (See Figure 4.6 for an illustration.) For each variable in the expression there is an edge in the graph with one of its endpoints labeled by that variable and the other by the negation of the variable. (These "variable edges" are at the top of the graph in Figure 4.6.) For each clause in the expression there is a "triangle" in the graph with its edges labeled by the three literals in the clause. (These "clause triangles" are at the bottom of the graph in Figure 4.6.) Now edges get drawn between the variable edges and the clause triangles, but only between nodes that have the same label. Finally, for this to be a "Yes/No" VERTEX COVER problem, there needs to be a number — the number of nodes about which we ask whether or not there is a vertex cover of that size. This number is equal to the number of different variables in the expression plus twice the number of clauses in the expression. In the example, this number is 12.

To explore the connection between a cover of this size for the graph and a satisfying assignment of truth values for the expression, let's build up some intuition about the matter by trying first to find a vertex cover for the graph. How would we go about that?

One might be tempted to think about some algorithm for this. That's not a good idea, since no sub-exponential algorithms are known — and it is conjectured that none exist. So just "eyeball" the given example. It is probably small enough to do that with success.

Heuristics may help. For example, to cover a triangle, any two of its three corners suffice, but a single corner does not. So we might as well set aside two of our 12 choices of nodes for each of the triangles. And to cover a variable edge, either one of its endpoints suffices. So we set aside one of our 12 choices for each variable edge. No matter how we choose nodes within these set-aside allocations, the variable edges and the clause triangles will all be covered. The challenge is to make the remaining choices such that the edges than run *between* variable edges and clause triangles get covered as well.

Consider the $(w, \neg w)$ variable edge. w covers the only edge that $\neg w$ covers and then some. Thus there is no disadvantage in choosing w. This has further ramifications. The (w, w) edge down to the rightmost triangle is now covered and there is no disadvantage in choosing the other two corners of that triangle for covering it. The same applies to the second triangle from the left.

We are doing well so far. Two triangles are covered (which as said before is automatic) and, importantly, all the edges between those two triangles and variable edges are covered as well. Here is the secret for success: We have to choose endpoints of the variable edges such that for each triangle at least one of three edges from the triangle corners up to the variable edges is covered *from above*. (One can already smell the connection to the 3-SAT problem: To satisfy the expression, we have to choose values for the variables such that for each clause at least one of its literals is true.)

Choosing the $\neg y$ endpoint of the $(y, \neg y)$ edge then lets us take care of the remaining two triangles.

What happened to the $(x, \neg x)$ and the $(z, \neg z)$ edges? They are not covered yet, and they are the only edges that are not covered yet. But we have spent

only 10 of our 12 node choices so far and can choose either endpoint of those two edges.

Back to the question of how such a solution to the vertex cover problem implies that the expression is satisfiable. Choosing w over $\neg w$ initially in the graph corresponds to making the variable w in the expression *true*. (The reason why this was a no-loss choice in the graph translates, on the expression side, into the observation that $\neg w$ does not occur in any clause.) Choosing the $\neg y$ over the y node in the graph corresponds to making the variable y in the expression *false* and satisfies the remaining clauses. Being free to choose either endpoint of the $(x, \neg x)$ and the $(z, \neg z)$ edges to complete the cover of the graph corresponds to the fact that the expression is already satisfied no matter what we assign to x and z.

A similar discussion illustrates how a satisfying assignment to the variables of the expression translates into a cover of the right size for the graph. Making a variable x true (false) in the expression translates into choosing the x ($\neg x$) endpoint of the $(x, \neg x)$ variable edge of the graph. The rest of the cover then falls into place without any difficult choices. □

4.5 The "Holy Grail": P = NP?

In order not to lose sight of the big picture over all these technical details, let's summarize the results of this chapter and contrast them with the results of Chapter 1. In both chapters we deal with proving problems to be difficult. In Chapter 1, we ignored the importance of time and considered as "difficult" problems that could not be solved by any program running in a finite amount of time. In the current chapter, the notion of difficulty of a problem is that there is no program that solves the problem *in polynomial time*. Figures 4.7 and 4.8 illustrate the results in both of these two settings.

In both settings, there are problems that are known to be "easy" — solvable (ODDLENGTH, BINARY ADDITION, SATISFIABILITY) or solvable in polynomial time (ODDLENGTH, BINARY ADDITION).

In both settings there are transformations that turn one problem into another. In both settings these transformations "propagate difficulty" — if A can easily be transformed into B and A is difficult, then B is difficult as well. In Chapter 1, for "difficulty," i.e., unsolvability, to propagate, the transformations merely had to be computable[7]. In the current chapter, for "difficulty," i.e., the lack of a polynomial-time solution, to propagate, the transformations had to be computable in polynomial time.

So far, so similar. But there is one huge difference. In Chapter 1, we did have a method to prove that one of the problems (SELFLOOPING) was indeed difficult. That method was diagonalization. With this starting point of one problem provably difficult, the transformations take over and prove all sorts of other problems to be difficult as well.

[7] All the examples were actually computable in polynomial time.

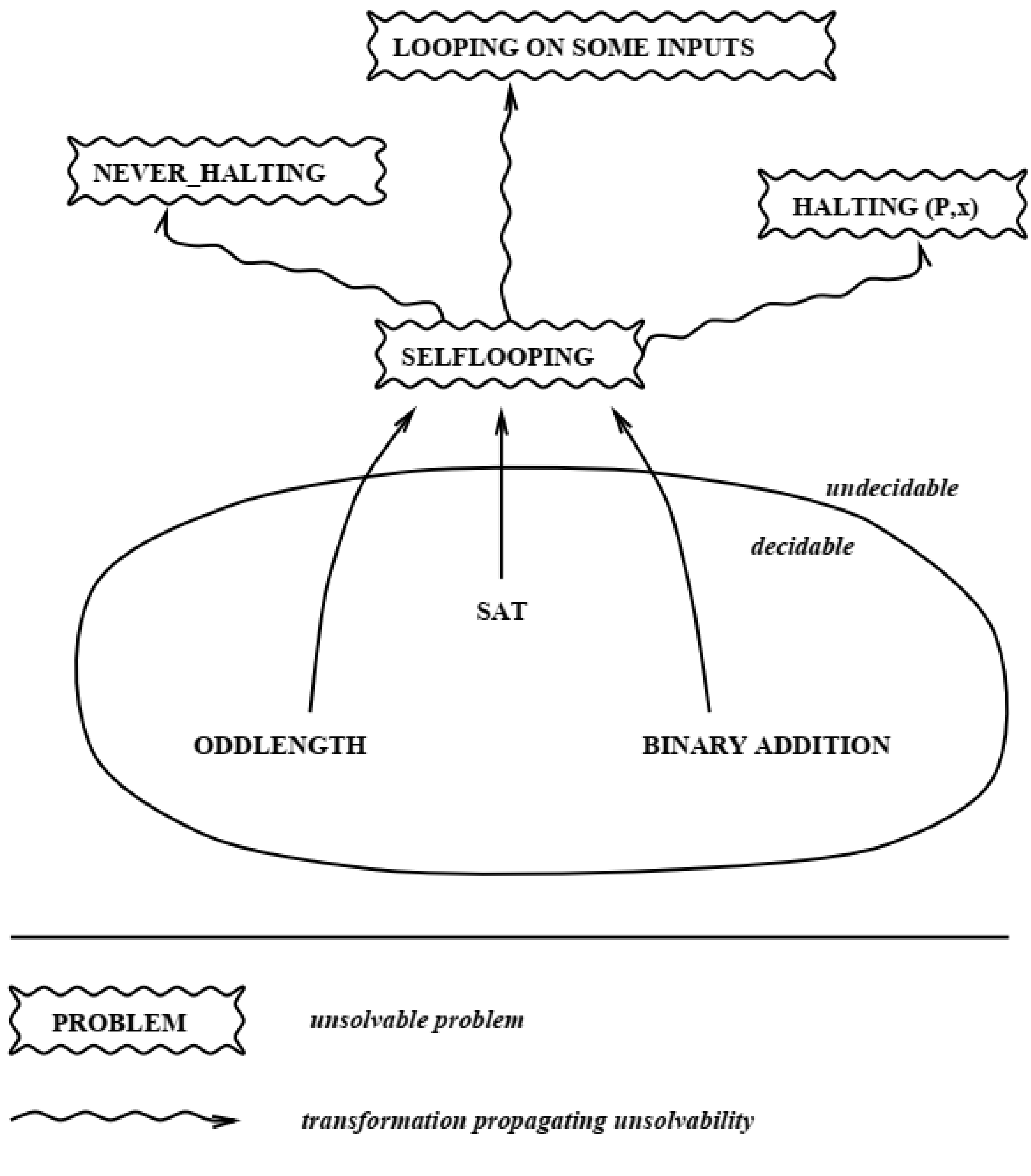

Figure 4.7

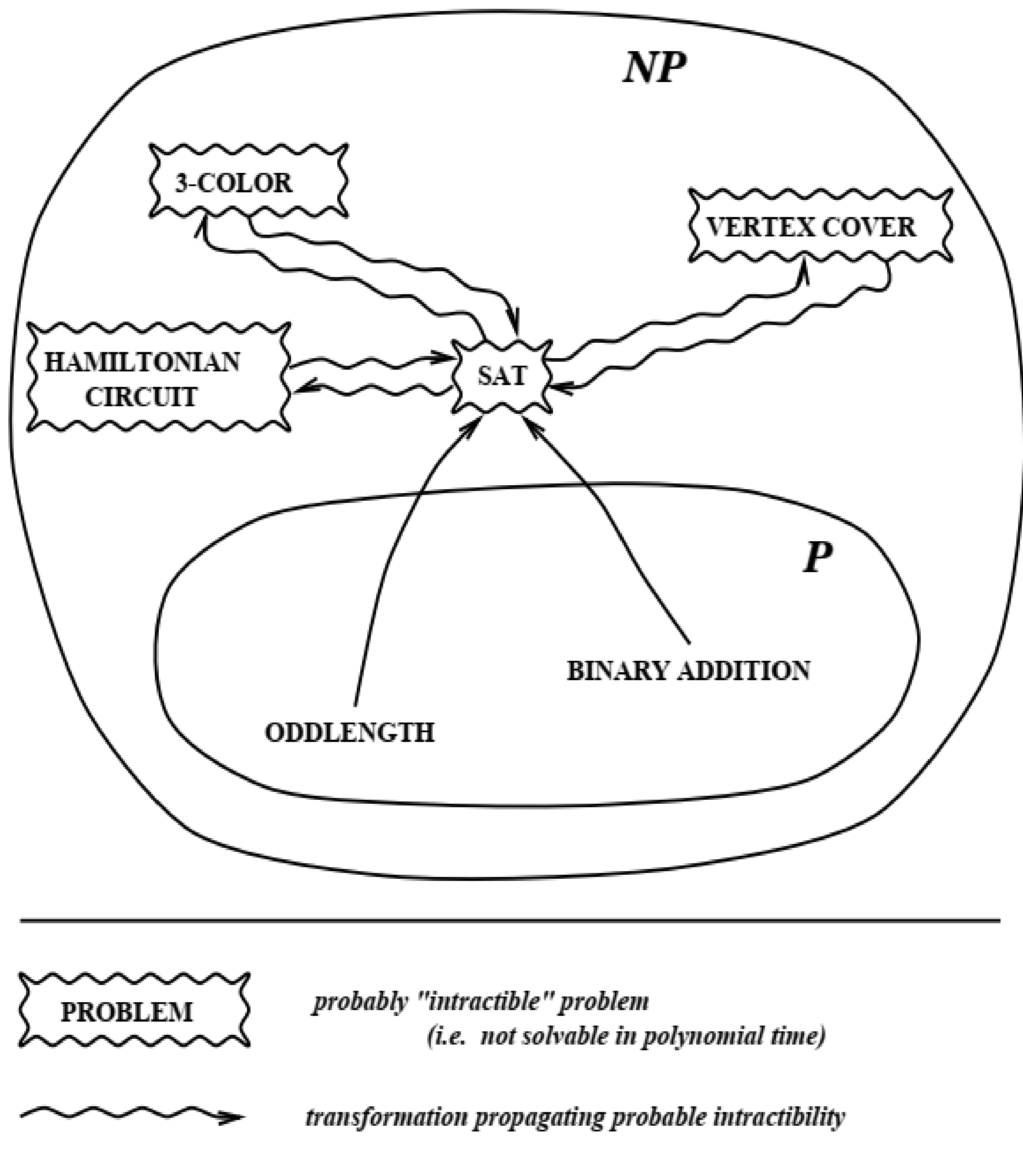

Figure 4.8: Results from the current chapter

In the current chapter, there is no such starting point. We do not have a proof that SAT*, or* VERTEX COVER*, or* 3-COLOR*, or* HAMILTONIAN CIRCUIT*, do indeed require more than polynomial running time to be solved.*

So what's missing in the current chapter is a proof technique — some way to show that, say, SAT cannot be solved in polynomial time. Such a result would show that $P \neq NP$. Conversely, finding a polynomial-time algorithm for SAT because of the NP-completeness of SAT would imply that $P = NP$.[8]

Why is $P \stackrel{?}{=} NP$ So Hard to Resolve?

The question of whether or not there are polynomial-time algorithms for SAT, VERTEX COVER, or — equivalently — any other NP-complete problem has been the main unresolved problem in theory of computation for twenty years. While most people conjecture that there are no such polynomial-time algorithms, in other words $P \neq NP$, there is no proof. What then makes this question so hard?

The sensitivity of the question

One reason for the difficulty is that there are models of computation that are very close to the standard model — very close to the kind of programs you write all the time — in which $P = NP$ and, in a different model, $P \neq NP$. In other words, a slight change in our concept of what a computer is (or, because the distinction between hardware and software is of no consequence here, in the definition of what a program is) changes the answer to the $P \stackrel{?}{=} NP$ question.

Why is this a reason that resolving the $P \stackrel{?}{=} NP$ question might be hard? Because it implies that any proof technique that can resolve the $P \stackrel{?}{=} NP$ question must be sensitive to those slight changes. It must work in the standard model of computation but must not work in a closely related model. Worse yet, of the two obvious approaches for trying to resolve $P \stackrel{?}{=} NP$ question — one approach for showing equality, the other for showing inequality — neither is sensitive to those slight changes in the model of computation.

Co-processors

What are these "slight changes" in the model? In terms of hardware, the change is to plug in a "co-processor" — a circuit, or chip, which takes some input and produces some output. The most common co-processor in practice may be a "numerical co-processor" — a chip that takes as input two floating-point numbers and produces, say, their product. The co-processors we are interested in here take as input one bitstring and produce one bit as output. Each such co-processor can be thought of as implementing a set of strings. For each string it responds with a "Yes" or a "No." Figure 4.9 illustrates this. In terms of software, this amounts to stipulating that there is a new boolean function built into our programming language which takes a bitstring as argument and that we do not count the execution time of that function as part of our programs' execution time.

What happens to our computer (or programming language) once we add this new feature? Programs that do not access that co-processor (do not call that new function) run like they always did. What about programs that do

[8] Current betting is overwhelmingly against this resolution of the question.

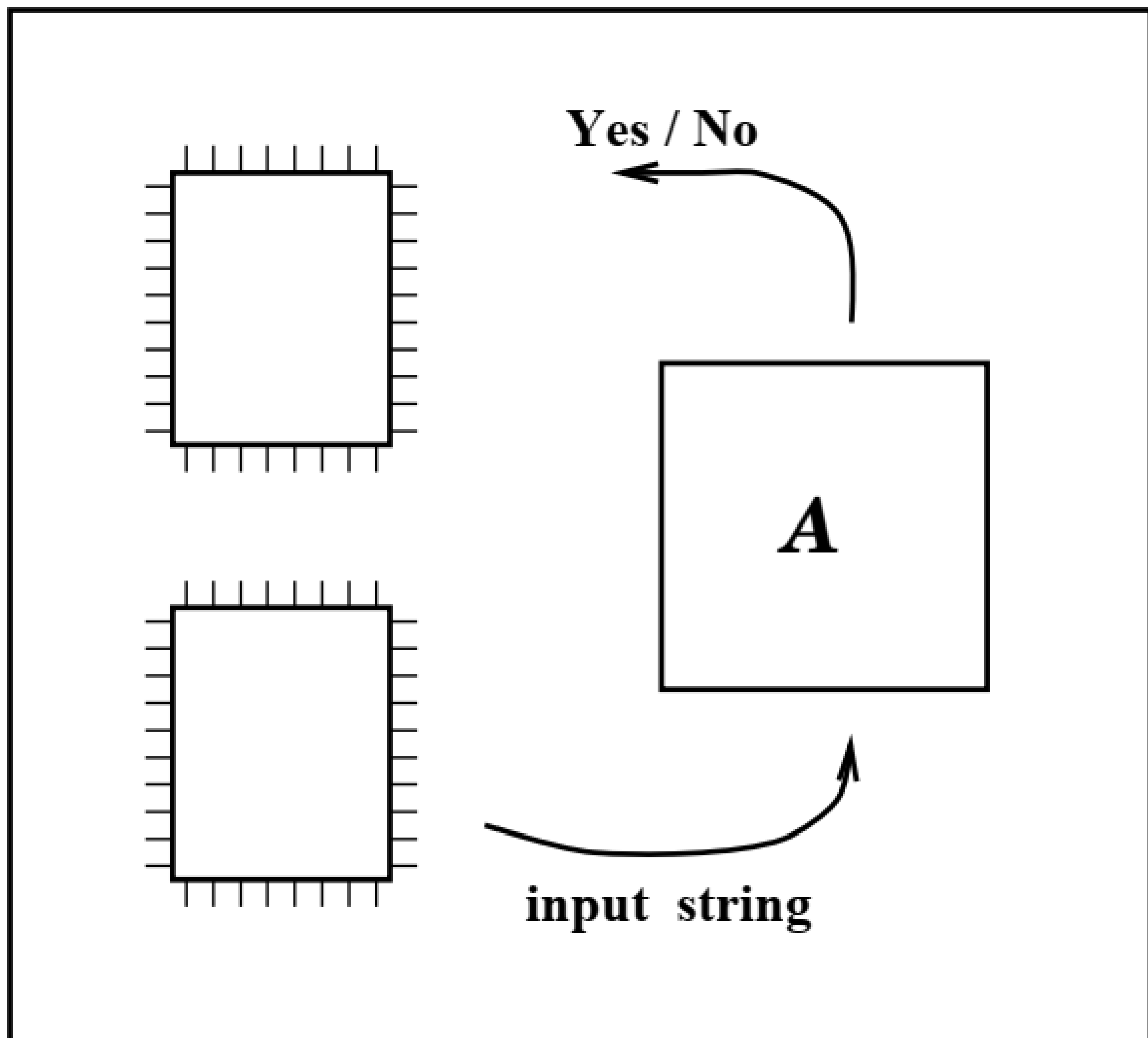

Figure 4.9: A co-processor

access the new feature? What they do depends on exactly which model of a co-processor we plugged in, i.e., which set it implements.

As long as that new co-processor only does things we could have done by programming our own boolean function it is not very interesting. But what if somehow, mysteriously, we had come across a co-processor that does in one single step something that we might otherwise not be able to do at all with a program or at least not as fast?

access the new feature? What they do depends on exactly which model of a co-processor we plugged in, i.e., which set it implements.

As long as that new co-processor only does things we could have done by programming our own boolean function it is not very interesting. But what if somehow, mysteriously, we had come across a co-processor that does in one single step something that we might otherwise not be able to do at all with a program or at least not as fast?

Co-processors that are "out of this world"

We can consider co-processors of any kind of power, e.g. even a co-processor that solves HALTING in a single leaping bound: a co-processor which, once you have fed it a bitstring that describes a program and input, responds with a "Yes," if that program on that input would terminate, and with a "No" otherwise.

Does such a co-processor exist? Definitely not on this planet, but that need not keep us from considering what such a new — very powerful — machine could do.[9]

Another co-processor of interest might be one that solves SATISFIABILITY in a single step.

$\mathbf{P}_A$ and $\mathbf{NP}_A$

For any co-processor A, let P_A be the class of problems that a machine with co-processor A can solve in polynomial time. For any co-processor A that is worth talking about, P_A is bigger than P. This is certainly true if $A =$ HALTING and probably true if $A =$ SAT.

Similarly, let NP_A be the class of problems that a machine with co-processor A can solve in nondeterministic polynomial time. NP_A can be bigger than NP. This is certainly true if $A =$ HALTING and probably true if $A =$ SAT.

Theorem 23 (Baker, Gill, Solovay 1976) *1. There is a co-processor*[10] *A such that*

$$P_A = NP_A.$$

2. There is a co-processor B such that

$$P_B \neq NP_B.$$

(A proof can be found in *Hopcroft and Ullman.* A co-processor A for which $\mathrm{P}_A = \mathrm{NP}_A$ is the one that answers questions of the form, "Is it true that

there is a u such that for all v ... there is a z such that $E(u, v, \ldots y, z)$?"

where E is a boolean expression. This kind of problem — "quantified Boolean expressions" — is a generalization of SAT, which has only one "quantifier." ("There is a set of values that makes the expression true."))

Proof techniques can be "too powerful"

Think of the result of Baker et al as saying that in one set of circumstances

[9] This is a form of transformation. We are asking what problems we could solve if we had a way to solve HALTING.

[10] A set of bitstrings, really. The commonly used term is actually "oracle."

("on planet A") P = NP and in a different set of circumstances ("on planet B") P $\neq$ NP. Why do we on planet Earth care? Because if we want to prove that P $\neq$ NP, we have no chance unless we come up with a proof technique that does not work on planet A. Or, if space travel does not seem the right subject to drag into this discussion, we have no chance unless we come up with a proof technique that will collapse – will not be applicable, will not work — once we plug in co-processor A. Any technique that is so robust that it even works in the presence of any co-processor is "too powerful" to prove P $\neq$ NP. Similarly, any technique that is so robust that it even works in the presence of any co-processor, especially co-processor B from the theorem, is "too powerful" to prove P = NP.

Are we talking about some far-out or nonexisting proof techniques? Not at all. If we want to prove that SAT cannot be solved in polynomial time, what would be a natural first technique to try? What technique do we have to show that something cannot be done? The one and only method we have is diagonalization. How could we try to use it for proving that something takes, say, exponential time to solve?

Time-limited diagonalization

Consider the following "exponential-time" version of HALTING. Given a program P and an input x, decide whether or not P halts within $2^{|x|}$ steps. This is clearly a decidable problem — just run the program for that long and see. Is there a faster way to decide it? No, by a diagonalization proof that is practically identical to the one we used to prove that the (unlimited) halting problem was undecidable. The central argument in such a proof is just like our argument in Chapter 1 that SELFLOOPING was not decidable. There the argument was that if SELFLOOPING was decidable then we could write a program that took itself as input, decided what its input (that is, itself) will do (loop or not) and based on that decision do the opposite. Now, with time playing a role, the argument is that if it was possible to decide in substantially less than 2^n steps that a program was going to halt within that many steps then we could write a program that took itself as input, decided what its input (that is, itself) will do (run for more than 2^n steps or not) and based on that decision do the opposite.

This is of course very exciting. We have a way to prove that something takes exponential time! All that's left to do in order to prove that P $\neq$ NP is to transform this EXPONENTIAL-TIME HALTING problem to SAT.[11] Are we likely to succeed? Not if this approach would also work "on other planets," i.e., in the presence of co-processors. Well, does it? Does an argument like

> "if it was possible to decide in substantially less than 2^n steps that a program was going to halt within that many steps then we could write a program that took itself as input, decided what its input (that is, itself) will do (run for more than 2^n steps or not) and based on that decision do the opposite"

[11] with a polynomial-time transformation, but that should not worry us as we have yet to come up with any transformation that was not polynomial-time

still work when there is a co-processor present? Unfortunately, the answer is yes. There is nothing wrong with the same argument on machines that have a co-processor:

> "if it was possible *on a machine with co-processor A* to decide in substantially less than 2^n steps that a program was going to halt within that many steps then we could write a program *to run on a machine with co-processor A* that took itself as input, decided what its input (that is, itself) will do (run for more than 2^n steps or not) and based on that decision do the opposite"

Thus this diagonalization technique is doomed. It is "too good."

Well, then maybe, just maybe, P = NP ? One reason why a conjecture can be hard, actually impossible, to prove is that the conjecture might not be true. We have used this insight before. If we had a hard time coming up with a context-free grammar for some language, a reasonable alternative was to consider the possibility that the language was not context-free and to try to prove that instead.

Whatever the reason for entertaining the conjecture that P = NP, we should be aware that "extra-terrestrial" arguments similar to the ones presented above show that P = NP is not likely to be provable with "the obvious" kind of approach.

What is the "obvious" kind of approach? What is the problem?

The problem here would be to find a way to solve SAT and its equivalent problems with a deterministic polynomial-time algorithm. How could we even approach this problem? By trying to devise some very clever way of searching nondeterministic execution trees of polynomial height but exponential size in (deterministic) polynomial time.

But if we succeeded in such an efficient deterministic "simulation" of nondeterminism, why should such a simulation not work just because a machine might have a co-processor? Why should this not work "on planet B." If it does, of course, the technique is again "too good."

Now what? That's where the matter stands. We are not able to settle the question of whether or not P = NP and variations of existing proof techniques are not likely to help. We are able to build and program computers but we cannot claim to understand the nature of computation.

Exercises

Ex. 4.1. Is it true that if a circuit is satisfiable then there is more than one way to satisfy it? Explain briefly.

Ex. 4.2. Is it true that if a graph is 3-Colorable then there is more than one way to 3-Color it? Explain briefly.