

## Decision versus Search

When we want to solve a problem in “real life”, we are interested in finding a solution, not just knowing if one exists. Yet we have always formulated problems in terms of decision, meaning we pose a question and want to know whether the answer is “yes” or “no”. In fact, for every problem we have considered, there is a corresponding search problem. Today we will look at the relation between these.

### 1 Search and decision problems

Every **NP** problem comes in two versions, a decision version and a search version. For example:

**Problem:** SAT

**Input:**  $\langle \varphi \rangle$  where  $\varphi$  is a CNF formula

**Decision Problem:** Is  $\varphi$  satisfiable?

**Search Problem:** Find a satisfying assignment to  $\varphi$  if one exists, else output  $\perp$ .

Here  $\perp$  is a special symbol used to indicate that there is no solution.

The SAT problem we have looked at, and shown to be **NP**-complete, is the decision problem. In the search version of the problem, you can’t stop merely at saying “yes” or “no” to the question of whether the given formula is satisfiable; if the answer is “yes” you must actually find and output a satisfying assignment.

Recall a clique is a subset  $C$  of the vertex set  $V$  of a graph  $G = (V, E)$  such that for all distinct vertices  $u, v \in C$  it is the case that  $\{u, v\} \in E$ . The two versions of the CLIQUE problem are:

**Problem:** CLIQUE

**Input:**  $\langle G, K \rangle$  where  $G = (V, E)$  is a graph and  $K \in \mathbb{N}$

**Decision Problem:** Does  $G$  have a clique of size  $K$ ?

**Search Problem:** Find a clique  $C$  in  $G$  with  $|C| = K$ , if one exists, or output  $\perp$  if none exists.

The CLIQUE problem we have looked at, and shown to be **NP**-complete, is the decision problem. In the search problem, you actually have to find the clique, meaning identify a set  $C$  which forms a clique.

Notice that if you can solve the search problem you can certainly solve the decision problem. Why? Take SAT. Given a formula  $\varphi$ , if I can find a satisfying assignment or tell that one does not exist, I can certainly say whether or not there exists a satisfying assignment. So search is harder; if you can solve it you can certainly solve decision.

So then the question is why do we focus on decision problems? After all the real object of interest is the search problem. What use is it to know a solution exists if you can’t find it?

## 2 Self-reducibility of SAT

One of the reasons we consider decision problems is that if we can solve them, we can often solve the corresponding search problems. That is, the two are “usually equivalent.” Let’s illustrate this for SAT.

As usual, it is a reductionist game. What we will show is: suppose someone gives you a box, or genie, to solve the SAT decision problem. Then we can build a procedure to solve the search problem in polynomial time. That is, search reduces to decision for SAT.

This implies that if SAT is in  $\mathbf{P}$  then its search version is solvable in polynomial time. (Don’t say that the search problem is “in  $\mathbf{P}$ ”. That is not meaningful since  $\mathbf{P}$  only contains languages, meaning decision problems.) So we can concentrate on the decision version, which is simpler. The technique is called self-reducibility.

The box, which we call  $\text{MB}_{\text{Sat}}$  for magic box for SAT, or *oracle*, takes any formula and solves SAT on it. In other words, for any formula  $\phi$  it is the case that  $\text{MB}_{\text{Sat}}(\langle\phi\rangle) = 1$  if  $\phi$  is satisfiable and 0 otherwise. We imagine we have this box as a subroutine, and can invoke it on any input  $\phi$  that we like. Moreover, it returns a response in one step: we are not charged for its running time.

The box always works, for any formula. But we can invoke it only polynomially often, since we have to run in polynomial time.

This illustrates a new setup / concept we will see more of later, namely solving one problem given an oracle to solve another. Think of it as a subroutine, or a piece of library code that someone gives you. They don’t tell you how it works, just what it does. Now you have to write a program to do something, and you are allowed to invoke this subroutine. Every call to the oracle has cost one unit.

Now, our problem is that we are given a formula  $\varphi$  and want to find a satisfying assignment to it if one exists. To help us, we have  $\text{MB}_{\text{Sat}}$ . We can certainly use  $\text{MB}_{\text{Sat}}$  to test whether or not  $\varphi$  is satisfiable. Suppose it says yes. But then what? How do we actually find a satisfying assignment?

The key point is that we can invoke  $\text{MB}_{\text{Sat}}$  on any formula, not just  $\varphi$ . We will invoke it on a bunch of formulas constructed out of  $\varphi$ , and “peel off” a truth assignment one bit at a time.

**Example 2.1** Let’s look at an example. Consider

$$\varphi(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee \neg x_1) \wedge (x_4 \vee \neg x_3)$$

First ask  $\text{MB}_{\text{Sat}}$  whether  $\varphi$  is satisfiable. If it says no, we output no and are done. What if it says yes? The idea is to find the values of the variables one by one. Consider two formulas, which we denote  $\varphi_0$  and  $\varphi_1$ . Each is a formula on one less variable, formed by setting  $x_1$  to the value in question:

- $\varphi_0(x_2, x_3, x_4) = \varphi(0, x_2, x_3, x_4)$ . Simplifying this is  $(0 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee 1) \wedge (x_4 \vee \neg x_3)$ , which is  $\neg x_2 \wedge (x_4 \vee \neg x_3)$ .
- $\varphi_1(x_2, x_3, x_4) = \varphi(1, x_2, x_3, x_4)$ . Simplifying this is  $(1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee 0) \wedge (x_4 \vee \neg x_3)$ , which is  $(x_2 \vee x_3) \wedge (x_4 \vee \neg x_3)$ .

The key point is that *one of these must be satisfiable*. Because  $x_1$  must be either 0 or 1. So let’s ask the  $\text{MB}_{\text{Sat}}$  which! That is run  $\text{MB}_{\text{Sat}}(\langle\varphi_0\rangle)$ . Suppose it says yes. Then we can set  $x_1 = 0$  because we know there exists some satisfying assignment to  $\varphi$  in which  $x_1 = 0$ . Now we go on with  $\varphi_0$ . On

the other hand if it says no, it must be that  $\varphi_1$  is satisfiable, so we can set  $x_1 = 1$ , and go on.

We make a total of  $n + 1$  calls to the oracle, and end up with a satisfying assignment. Now let us say this more properly.

**Theorem 2.2** The SAT search problem is solvable in polynomial time given an oracle for the SAT decision problem.

**Proof:** We must specify a polynomial time algorithm  $F$  which, given an oracle  $\text{MB}_{\text{Sat}}$  for SAT, and given input  $\varphi$ , outputs either a satisfying assignment for  $\varphi$ , or indicates none exists.

First, some notation. Let  $n$  be the number of variables in  $\varphi$ . Let  $b_1, \dots, b_n$  be bits. Let  $\varphi_{b_1}$  denote the formula obtained by plugging in  $x_1 = b_1$  in  $\varphi$ . (This is a formula over the  $n - 1$  variables  $x_2, \dots, x_n$ .) Let  $\varphi_{b_1, b_2}$  be the formula obtained by plugging in  $x_1 = b_1$  and  $x_2 = b_2$  in  $\varphi$ . (This is a formula over the  $n - 2$  variables  $x_3, \dots, x_n$ .) And so on, until  $\varphi_{b_1, \dots, b_n}$ , which is a formula in 0 variables, meaning it is either 0 or 1.

Now for the algorithm. Do the following:

```

If  $\text{MB}_{\text{Sat}}(\langle\varphi\rangle) = 0$  then output  $\perp$  (the formula is unsatisfiable)
Else for  $i = 1, \dots, n$  :
    If  $\text{MB}_{\text{Sat}}(\langle\varphi_{b_1, \dots, b_{i-1}, 0}\rangle) = 1$  then let  $b_i = 0$  else let  $b_i = 1$ 
Output  $b_1, \dots, b_n$ 

```

The first time through the loop, when  $i = 1$ , we are looking at  $\varphi_0$ , meaning  $x_1$  is set to 0. If this is satisfiable, it means there exists *some* satisfying assignment to  $\varphi$  that has  $x_1 = 0$ . So we can set  $x_1 = 0$  and look for a satisfying assignment to  $\varphi_0$ . (Note it could be that there was also some satisfying assignment having  $x_1 = 1$ , but we don't care: it suffices to go with one of them.) Continuing this reasoning, we can see that at the end we do get a satisfying assignment.

You should convince yourself this works. Why can't the algorithm get "stuck" at any point? Why is it OK to set  $b_i = 1$  if  $\text{MB}_{\text{Sat}}$  outputs 0 in the above?

Also check the procedure is polynomial time under the convention that each call to  $\text{MB}_{\text{Sat}}$  counts as one step. ■

Notice that in this proof we invoked  $\text{MB}_{\text{Sat}}$  not once, but  $n + 1$  times, and on *different* formulas, depending on  $\varphi$  and our current partial truth assignment. We made good use of  $\text{MB}_{\text{Sat}}$ : we used the fact that it worked for any formula.

### 3 Self reducibility of Clique

The same holds for the Clique problem. Given an oracle  $\text{MB}_{\text{Clique}}$  to solve the decision problem, it is actually possible to *find* a clique.

**Theorem 3.1** The CLIQUE search problem is solvable in polynomial time given an oracle for the CLIQUE decision problem.

**Proof:** For any graph  $H = (V, E)$  and any  $v \in V$  let  $H \setminus v$  denote the graph formed from  $H$  by removing  $v$  and all edges adjacent to it. Let  $G$  be the given graph, and let  $\{v_1, \dots, v_n\}$  be its vertex set. Now we are given  $\text{MB}_{\text{Clique}}$  which takes  $\langle G, K \rangle$  and outputs 1 if  $G$  has a clique of size  $K$ , and 0 otherwise. The procedure to solve the search problem is as follows:

```

If  $\text{MB}_{\text{Clique}}(\langle G, K \rangle) = 0$  then return  $\perp$  and halt
For  $i = 1, \dots, n$  do
    If  $\text{MB}_{\text{Clique}}(\langle G \setminus v_i, K \rangle) = 1$  then  $G \leftarrow G \setminus v_i$ 
EndFor
Repeat
    Arbitrarily pick a vertex  $v$  in  $G$  and let  $G \leftarrow G \setminus v$ 
Until  $G$  has exactly  $K$  nodes
Output  $G$ 

```

The idea is to remove nodes from  $G$ , one by one, and see what happens to the clique size.

For the analysis, let  $G_i$  be the graph after the  $i$ -th execution of the loop. Let  $G_0 = G$ . We know from the first step that  $G_0$  has a clique of size  $K$ . Observe the following is true for all  $i \geq 1$ :

- (1) If  $v_i$  is not removed then all cliques of size  $K$  in  $G_{i-1} = G_i$  contain  $v_i$ .
- (2)  $G_i$  contains a clique of size  $K$ .

Observe that a clique in  $G_j$  is also a clique in  $G_i$  for  $j \geq i$ . Inductively this means that for any  $j$  the graph  $G_j$  has a clique of size  $K$ , and for any  $i \leq j$ , if  $v_i$  is in  $G_j$  then it is a member of all cliques of size  $K$  in  $G_j$ . This means that in  $G_n$ , there is a clique of size  $\geq K$ , and all vertices are members of all cliques of size  $\geq K$ . So  $G_n$  must itself be a clique of size  $\geq K$ . ■

## 4 Decision vs Search for NP-complete languages

What about other languages? We saw above that we really exploited the structure of the problem. So it is not clear we can always do something like this. What we claim is that can for **NP**-complete languages. To present this, we need to set things up.

What we are really looking at above is not SAT, but the underlying verification process. That is, let  $\text{EVAL}(\varphi, b_1 \dots b_n) = 1$  if  $\varphi(b_1, \dots, b_n) = 1$  and 0 otherwise. Given  $\varphi$  we are finding  $b_1, \dots, b_n$  so that this relation is true. Thus the form of the underlying search problem is given some verifier-defined relation  $\rho(x, w)$  and some input  $x$ , find  $w$  such that  $\rho(x, w) = 1$ , if one exists.

Recall an **NP** language is one for which it is possible to provide a succinct proof, or certificate, that a verifier can check. Let us view the verifier as returning 1 if it accepts and 0 if it rejects. In this context we will denote the verifier by  $\rho$ .

**Definition 4.1** A *verifier* is a function  $\rho$  that takes two arguments  $x, y$ , both strings, and returns either 1 (true) or 0 (false). We say that  $\rho$  is an **NP-verifier** if there exists a polynomial  $p$  such that for all  $x, y$  the computation  $\rho(x, y)$  halts and returns its answer in at most  $p(|x|)$  steps. For

any  $x \in \{0, 1\}^*$  we let  $\rho(x) = \{ w \in \{0, 1\}^* : \rho(x, w) = 1 \}$  denote the witness set of  $x$ . We let  $L_\rho = \{ x \in \{0, 1\}^* : \rho(x) \neq \emptyset \}$  denote the language defined by  $\rho$ . We say that  $\rho$  is **NP**-complete if  $L_\rho$  is **NP**-complete. ■

Notice that we measure the time-complexity of computing  $\rho$  as a function of the length of its first input only, not both inputs. A consequence of this is that if  $|y|$  is more than  $p(|x|)$  then  $\rho$  cannot even read in all of  $y$ , and its decision must be taken based only on some prefix of  $y$ . This means that for all practical purposes we may assume that we only consider strings  $y$  whose length is at most  $p(|x|)$ . The candidate set of  $y$  values is thus finite.

You can easily check out the following. It is just terminology juggling:

**Proposition 4.2** A language  $A$  is in **NP** iff there exists an **NP**-verifier  $\rho$  such that  $A = L_\rho$ . ■

For example  $\rho = \text{EVAL}$  is an **NP**-verifier, and moreover,  $L_\rho = \text{SAT}$ .

However, it is important to note that if  $A \in \text{NP}$ , there may be *many different* **NP**-verifiers  $\rho$  for which  $A = L_\rho$ .

Now associated to an **NP**-verifier  $\rho$  there are two problems. For both, the input is a string  $x$ . The problems are:

- **Decision:** Does there exist  $w$  such that  $\rho(x, w) = 1$ ? That is, is  $x \in L_\rho$ ?
- **Search:** Find  $w$  such that  $\rho(x, w) = 1$  if one exists, else output  $\perp$ .

**Definition 4.3** We say search reduces to decision for an **NP**-verifier  $\rho$  if there is a polynomial-time algorithm which given any input  $x$  and an oracle for  $L_\rho$ , outputs  $w$  such that  $\rho(x, w) = 1$  if one exists, and otherwise outputs  $\perp$ . ■

Thus what we saw above is that search reduces to decision for **EVAL**. Similarly, search reduces to decision for the relation  $\rho$  underlying the **CLIQUE** problem. (Can you define this relation?) Now, our question is: how general is this? Is it true for any **NP**-verifier?

**Question:** Let  $\rho$  be an **NP**-verifier. Does search reduce to decision for  $\rho$ ? ■

The first thought is that the answer is “yes” due to **NP**-completeness. Since the language  $A = L_\rho$  is in **NP** we know that  $A \leq_p \text{SAT}$ . So if you can decide **SAT** you can decide  $A$ . But we know search reduces to decision for (the **NP**-verifier underlying) **SAT**. So does that mean search reduces to decision for  $\rho$ ? Not necessarily. The problem is that we have an oracle for  $A$ , not an oracle for **SAT**, so can’t run the **SAT** self-reducibility algorithm. But if  $A = L_\rho$  is itself also **NP**-complete then things work out. To see this, we need first to recall something that came out of the Cook-Levin theorem.

**Lemma 4.4** Let  $\rho$  be an **NP**-verifier. Then there exist polynomial-time computable functions  $f, W$  such that for all  $x \in \Sigma^*$

- (1)  $x \in L_\rho$  iff  $f(x) \in \text{SAT}$
- (2) If  $T$  is a satisfying assignment to formula  $f(x)$  then  $\rho(x, W(x, T)) = 1$ . ■

**Proof:** Recall that our proof of the Cook-Levin theorem began by constructing a polynomial-time computable function  $g$  which on input  $x$  returns a circuit  $C_x$  such that

$$\exists w [\rho(x, w) = 1] \quad \text{iff} \quad \exists w, z [C_x(w, z) = 1] .$$

Notice that given an input  $w, z$  satisfying circuit  $C_x$ , it is easy to recover  $w$ , since the latter is just the first part of  $w, z$ . Next we showed that  $\text{CIRC-SAT} \leq_p \text{SAT}$ . Look at the reduction function  $h$  we used, and you will see that given a satisfying assignment to formula  $h(C)$  one can easily construct an input satisfying circuit  $C$ . (Indeed, we did this construction to prove that the reduction worked. We are now observing that it was polynomial-time.) ■

In other words, if you know a satisfying assignment  $T$  to the formula  $f(x)$  then you effectively know a witness  $w$  to the membership of  $x$  in  $L_\rho$ , in the sense that given  $T$  (and  $x$ ) you can construct  $w$  in polynomial time. So to find  $w$  we need only find  $T$ .

**Theorem 4.5** Let  $\rho$  be an NP-verifier. If the language  $A = L_\rho$  is NP-complete then search reduces to decision for  $\rho$ . ■

**Proof:** We are given  $x$  and an oracle  $\text{MB}_A$  for  $A$ . We want to find a witness  $w$  for  $x$  if one exists. We would like to ask the oracle questions, but what? We do not know what is the structure of the problem.

But  $A \leq_p \text{SAT}$ . So there are functions  $f, W$  satisfying the conditions of Lemma 4.4. Now let  $\varphi = f(x)$ . Suppose we find a truth assignment  $T$  to  $\varphi$ . Then we are done, because we can find  $w$  satisfying  $\rho(x, w) = 1$  via  $w = W(x, T)$ .

But how do we find  $T$ ? We want to use the self-reducibility of SAT. But there is a catch. How do we do the self-reduction? We do not have a SAT oracle! But we can build one, out of the  $A$  oracle, because  $A$  is itself assumed NP-complete. Namely let  $g$  be a reduction of SAT to  $A$ . Then we can effectively define a SAT oracle. Let  $\text{MB}_A$  denote the given oracle (magic box) for the language  $A = L_\rho$ . Then define:

Subroutine  $\text{MB}_{\text{Sat}}(\langle\phi\rangle)$

Let  $\alpha = g(\phi)$

If  $\text{MB}_A(\alpha) = 1$  then return 1 else return 0.

You should check out that this really is a SAT oracle, meaning it solves the SAT problem. This will be true because  $g$  is a reduction of SAT to  $A$ . Now use this as a SAT oracle in the SAT self-reduction. Apply that self reduction to the formula  $\varphi$  to get an assignment  $T$  to it. Then apply Lemma 4.4 to get  $w$ . ■

## 5 Decision vs Search for non NP-complete languages

However, what if  $A = L_\rho$  is not NP-complete? Well, in this case the search and decision problems might be of different complexities. For example, it could be that the decision problem is easy (ie. in P) yet the search problem is hard.

**Theorem 5.1** Assume  $\mathbf{P} \neq \mathbf{NP} \cap \text{coNP}$ . Then there is an  $\mathbf{NP}$ -verifier  $\rho$  such that  $L_\rho \in \mathbf{P}$  but the search problem for  $\rho$  cannot be solved in polynomial time. (In particular, search does not reduce to decision for  $\rho$ .) ■

**Proof:** Let  $A \in (\mathbf{NP} \cap \text{coNP}) - \mathbf{P}$ . So

- $A = L_{\rho_1}$  for some  $\mathbf{NP}$ -verifier  $\rho_1$ , because  $A \in \mathbf{NP}$  by assumption.
- $\bar{A} = L_{\rho_2}$  for some  $\mathbf{NP}$ -verifier  $\rho_2$ , because  $\bar{A} \in \mathbf{NP}$  by assumption.

Now note that for any  $x$  either there is a  $w_1$  such that  $\rho_1(x, w_1) = 1$  or there is a  $w_2$  such that  $\rho_2(x, w_2) = 1$ , but *both* can never be true! So let us define  $\rho(x, w) = \rho_1(x, w) \vee \rho_2(x, w)$ . Then for every  $x$  there is  $w$  such that  $\rho(x, w) = 1$ . This means  $L_\rho = \Sigma^*$ . So the decision problem for  $\rho$  is trivial, and in particular  $L_\rho \in \mathbf{P}$ .

On the other hand, we claim that the search problem for  $\rho$  is not solvable in polynomial time, given our assumption that  $A \notin \mathbf{P}$ . Indeed, given  $x$ , suppose we could quickly (in polynomial time) find  $w$  such that  $\rho(x, w)$  holds. Check  $\rho_1(x, w)$  and  $\rho_2(x, w)$ . Exactly one of these can hold. If the first then  $x \in A$ , if the second then  $x \notin A$ . So we would be able to decide  $A$  in polynomial time, contradicting the assumption that  $A$  was not in  $\mathbf{P}$ . ■

Under stronger complexity assumptions it is possible to show that there is a  $\mathbf{NP}$ -verifier  $\rho$  such that  $L_\rho \in \mathbf{NP} - \mathbf{P}$  but still search does not reduce to decision for  $\rho$ . With even stronger assumptions one can show there is an  $A \in \mathbf{NP} - \mathbf{P}$  such that search does not reduce to decision for *any* of the many possible  $\mathbf{NP}$ -verifier  $\rho$  for which  $A = L_\rho$ . See for example reference [1].

## References

- [1] M. BELLARE AND S. GOLDWASSER. The complexity of decision versus search. *SIAM J. on Computing*, Vol. 23, No. 1, February 1994. Available via <http://www-cse.ucsd.edu/users/mihir>.