1 Decision vs Search problems

Search problems are described functions $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$ which return an answer to a question, e.g. ”what is the length of the shortest path between vertices $s, t$ in a graph $G$?”. 

Decision problems are described functions $f : \{0, 1\}^* \rightarrow \{0, 1\}$ which return a YES/NO answer to a question, e.g. ”is the length of the shortest path between vertices $s, t$ at most $\ell$”. In nearly all cases, we can reduce search problems to decision problems (e.g. by binary search in this example).

So we will mainly restrict our attention from now on to decision problems. These can be equally given as languages,

$$L = \{x \in \{0, 1\}^* : f(x) = 1\}.$$

Let us recall some definitions in this language. A language $L \subset \{0, 1\}^*$ is decideable or computable if the function $f(x) = 1_{x \in L}$ is computable. For $T : \mathbb{N} \rightarrow \mathbb{N}$,

$$TIME(T(n)) = \{L \subset \{0, 1\}^* : L \text{ is decideable in time } O(T(n))\}.$$

and

$$P = \cup_{c \geq 1} TIME(n^c)$$

Recall that we think of $P$ as the class of problems which can be solved efficiently.

2 The class NP

The class NP captures problems, where solutions can be verified efficiently. More concretely,

Definition 2.1 (NP). A language $L \subset \{0, 1\}^*$ is in NP if there exists a polynomial time $TM M$ and $c \geq 0$ such that

$$L = \{x : \exists y, |y| \leq |x|^c, M(x, y) = TRUE\}$$
That is, $x \in L$ is there exist a "proof" or "witness" for it which is easy (polynomial time) to verify. We think of $M$ as a verifier for the fact that $x \in L$ given the proof. Note that since $M$ runs in time at most $|x|^c$ we can assume that $|y| \leq |x|^c$.

Examples:

- CLIQUE is language of $(G, k)$ where $G$ is a graph with a clique of size at least $k$. CLIQUE is in NP since if $G$ has $n$ vertices, then a proof that $G$ has a clique of size $k$ is a list of the vertices in the clique.

- Traveling Salesman Problem (TSP) is the language of $(G, k)$ where $G$ is a graph with weights, such that there exists a path covering all vertices in $G$ with weights summing to at most $k$. TSP is in NP since a proof is such a path.

- Linear programming is the language of linear constraints which are satisfiable. It is in NP since a proof is a solution.

- 0–1 linear programming is the language of linear constraints which are satisfiable by a 0–1 solution. It is in NP since a proof is a solution.

- COMPOSITE is the language of numbers $n$ which are composite (e.g non-prime). It is in NP since a proof is a decomposition $n = ab$.

Some of these problems are in $P$ (linear programming, composite) while the others seem much harder.

**Claim 2.2.** $P \subset NP \subset EXP$.

**Proof.** $P \subset NP$ is obvious, since a verifier can ignore the proof. $NP \subset EXP$ since we can try all possible witnesses. For every $L \in NP$ and $x \in L$ of length $|x| = n$, the witnesses are of length at most $n^c$ for some $c > 0$. Hence, we can try all of them in time $2^{n^c}$ which is in EXP.

## 3 NP Completeness

It turns out that some languages are harder than others. This is formally defined by a reduction.

**Definition 3.1.** Language $L$ is poly-time reducible to a language $L'$, if there exists a poly-time computable function $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$, such that for all $x \in \{0, 1\}^*$,

$$x \in L \iff f(x) \in L'.$$

We denote this as $L <_p L'$.

**Claim 3.2.** The following holds:

1. If $L <_p L'$ and $L' <_p L''$ then $L <_p L''$. 

2
2. If \( L <_p L' \) and \( L' \in P \) then \( L \in P \).

3. If \( L <_p L' \) and \( L \) is NP-hard then \( L' \) is NP-hard.

Proof. The claim follows from the basic definition.

1. We have \( x \in L \iff f(x) \in L' \) and \( x \in L' \iff f'(x) \in L'' \) where \( f, f' \) are poly-time computable. Hence \( x \in L \iff f(f'(x)) \in L'' \) and \( f(f'(x)) \) is poly-time as well.

2. We have \( x \in L \iff f(x) \in L' \) where \( f(x) \) is poly-time computable. Since \( L' \) is in \( P \) there is a poly-time TM \( M \) such that \( y \in L \iff M(y) = 1 \). So, \( x \in L \iff M(f(x)) = 1 \), and \( M(f(x)) \) is poly-time computable.

3. If \( L \) is NP-hard then for any language \( L'' \in NP \), \( L'' <_p L \). By the previous item \( L'' <_p L' \) as well. So \( L' \) is NP-hard.

\[ \square \]

Definition 3.3 (NP-complete languages). A language \( L \) is called **NP-hard** if for any \( L' \in NP \) we have \( L <_p L' \). If furthermore \( L \in NP \) then we say \( L \) is NP-complete.

Do there exist NP-complete languages? YES.

Theorem 3.4 (Existence of NP complete languages). The following language is NP-complete.

\( L_0 = \{ (\langle M \rangle, x, 1^t) : \exists y \in \{0, 1\}^*, |y| \leq t, M(x, y) = 1, M(x, y) \text{ halts after at most } t \text{ steps} \} \).

Proof. Let's first verify that \( L_0 \in NP \). Let \( M_0 \) be a TM such that

\[ M_0(\langle M \rangle, x, 1^t, y) = 1_{|y|\leq t, M(x, y)=1, M(x, y) \text{ halts after at most } t \text{ steps}}. \]

That is, \( M_0 \) verifies that \( y \) satisfies the requirements in \( L_0 \). Note that \( M_0 \) runs by simulating \( M \) for at most \( t \) steps, and the input has length at least \( t \), hence \( M_0 \in P \). So, \( M \in NP \).

Let now \( L \in NP \). We will show that \( L <_p L_0 \). By definition, there exists a TM \( M \in TIME(n^c) \) such that

\[ x \in L \iff \exists y, |y| \leq |x|^c, M(x, y) = 1. \]

Define the following function \( f_L : \{0, 1\}^* \to \{0, 1\}^* : \)

\[ f_L(x) = (\langle M \rangle, x, 1^{|x|^c}). \]

Then clearly \( x \in L \iff f_L(x) \in L_0 \). Moreover, \( f \) can be computed in poly-time. \[ \square \]
4 Combinatorial NP complete problems

Cook and Levin proved in the early 70s that there are nice combinatorial problems which are NP-complete. Shortly after, Karp gave a list of 20+ well known combinatorial problems, all NP-complete.

Definition 4.1 (SAT). A CNF formula is a conjunction of disjunctions, e.g

\[ \phi(x_1, x_2, x_3, x_4) = (x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor x_3) \land (x_2 \lor \overline{x_4}). \]

Any function \( \phi : \{0,1\}^n \to \{0,1\} \) can be written as a CNF of size at most \( n2^n \). In particular, if \( n \) is constant then the CNF has a constant size. A CNF formula is satisfiable if there exists an assignment to the variables making it true. We define

\[ SAT = \{ \text{Satisfiable CNF formulas} \}. \]

Theorem 4.2 (Cook-Levin). SAT is NP-complete.

Proof. Clearly \( SAT \in NP \) since a proof that \( \phi \in SAT \) is an assignment to the variables of \( \phi \) making it true. The harder claim is that \( SAT \) is NP-hard. Let \( L \in NP \). Then there exists a TM \( M \in \text{TIME}(n^c) \) verifying \( L \). That is,

\[ x \in L \iff \exists y, M(x,y) = 1. \]

We assume w.l.o.g that \( M \) has binary alphabet and three work tapes. Let \( T = n^c \) be a bound on the running time and space used by \( M \). This means that only the first \( T \) cells in the work tapes. We define variables capturing the contents of the work tapes at every step of the computation. For simplicity of notation, say the first work tape is the input, the second is the work tape and the third is the output. We define a number of boolean variables describing the computation process. For \( 1 \leq i, t \leq T \):

- \( TAPE^k_{i,t} = \) the \( i \)-th bit in the \( t \)-th time step in tape \( k = 1, 2, 3 \).
- \( HEAD^k_{i,t} = 1 \) if the head is in the \( i \)-th cell in the \( t \)-th time step for \( k = 1, 2, 3 \).
- \( STATE_{q,t} = 1 \) if the state in the \( t \)-th time step is \( q \), and 0 otherwise.

The theorem is based on the fact that every step of the computation is local.

Initialization. Let \( x \in \{0,1\}^n \) be the input and \( y \in \{0,1\}^{t-n} \) a potential witness.

\[
\begin{align*}
TAPE^1_{i,1} &= x_i \text{ for } 1 \leq i \leq n \\
TAPE^1_{i,1} &= y_{i-n} \text{ for } n < i \leq T \\
TAPE^k_{i,1} &= 0 \text{ for } 1 \leq i \leq T, \ k = 2, 3
\end{align*}
\]
**Update step**  The tape contents, head locations and state in step \( t + 1 \) depend just on those on time \( t \). More explicitly, the symbols under the heads in time \( t \) are

\[
SYMBOL^k_t = [TAPE^k_{i,t} \text{ if } HEAD^k_{i,t} = 1 : i = 1, \ldots, T]
\]

and

\[
TAPE^k_{i,t+1} = \text{Function}(\{STATE_{q,t} : q \in Q\}, \{SYMBOL^r_j : r = 1, 2, 3\}, HEAD^k_{i,t})
\]

\[
HEAD^k_{i,t+1} = \text{Function}(\{STATE_{q,t} : q \in Q\}, \{SYMBOL^r_j : r = 1, 2, 3\}, \{HEAD^k_{i+c,t} : c = -1, 0, 1\})
\]

\[
STATE_{q,t+1} = \text{Function}(\{STATE_{q,t} : q \in Q\}, \{SYMBOL^r_j : r = 1, 2, 3\})
\]

**Output**  The output of the computation is \( WORK^3_t \).

**Writing as a CNF**  We claim that each one of these equalities can be written as a poly-size CNF. Then, requiring them all together to hold is done by conjuncting them, which will still be a poly-size CNF. The initialization steps are clearly like that, since they depend on a constant number of variables. Same for the update for the work tapes, head location and new state. Similarly, we require that the final output will be 1. The only step which requires careful verification is the computation of the symbol under the heads. However, note that

\[
SYMBOL^k_t = [TAPE^k_{i,t} \text{ if } HEAD^k_{i,t} = 1 : i = 1, \ldots, T]
\]

can equivalently be written as

\[
\bigwedge_{i=1}^{t} [(HEAD^k_{i,t} = 1) \Rightarrow (SYMBOL^k_{i,t} = TAPE^k_{i,t})].
\]

Each clause is a function of 3 variables, hence can be written as a constant size CNF. Then, the expression is a CNF of size \( O(t) \).

So, we can write a poly-size CNF \( \phi(x, y) \), such that \( M(x, y) = 1 \iff \phi(x, y) = 1 \). Moreover, \( \phi \) can be computed in time \( O(T^2) = O(n^{2c}) \). Let \( \phi_x(y) = \phi(x, y) \) be the CNF formula obtained by plugging in the value of \( x \). Then, we conclude that

\[
x \in L \iff \exists y, M(x, y) = 1 \iff \exists y, \phi(x, y) = 1 \iff \exists y, \phi_x(y) = 1.
\]

So \( \{L \} \leq_p SAT \) by the reduction \( f(x) = \phi_x \).

Once we showed that SAT is NP-complete, we can use this to show that many other combinatorial problems are also NP-complete.

**Definition 4.3 (kSAT).** A \( k \)-CNF is a CNF where each clause has at most \( k \) variables.

\[
k - SAT = \{ \text{Satisfiable } k \text{-CNF formulas} \}.
\]

**Theorem 4.4.** \( 3SAT \) is NP-complete (and hence also \( 4SAT, 5SAT, \) etc).
Proof. We need to show that every CNF formula can be reduced to 3-CNF formula, with only a polynomial blow up in the size, and where the reduction can be computed by a poly-time TM. Let \( \phi(x) = C_1(x) \land \ldots \land C_m(x) \) be a CNF formula, where \(|x| = n\) and \(m = \text{poly}(n)\). It suffices to change each clause to a 3-CNF. Let \( C(x) \) be a clause, and assume w.l.o.g that

\[
C(x) = v_1 \lor v_2 \lor \ldots \lor v_k,
\]

where \( v_i \) is either a variable or its negation. Define new variables \( y_1, \ldots, y_{k-1} \) given by

\[
y_1 = v_1 \lor v_2, \quad y_2 = y_1 \lor v_3, \ldots, \quad y_{k-1} = y_{k-2} \lor v_k
\]

Then \( C(x) = y_{k-1} \), and we can express each of these formulas by a 3-CNF since they involve just 3 variables. To summarize, we transformed a CNF formula \( \phi(x) \) to a 3-CNF formula \( \psi(x, y) \), where \( \phi \) is satisfiable iff \( \psi \) is. Note that we can create \( \psi \) given a description of \( \phi \) by a poly-time Turing machine, hence \( \text{SAT} \leq_p \text{3SAT} \). This shows 3SAT is NP-hard. Clearly 3SAT is also in NP, hence it is NP-complete. \( \square \)

**Theorem 4.5.** 2SAT is in \( P \).

Proof. Let \( \phi(x) \) be a 2-CNF on \( n \) variables. Any clause in \( \phi \) has at most two literals (variables or their negations). Let’s assume any clause in \( \phi \) has exactly two literals, e.g. by replacing \( x_1 \) with \( x_1 \lor \neg x_1 \). We create a directed graph \( G \) with vertices \( V = \{ x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n \} \). The edges of \( G \) are as follows: for a clause is of the form \( v_i \lor \neg v_j \), it can be equivalently written as

\[
\neg v_i \rightarrow v_j \quad \text{or} \quad \neg v_j \rightarrow v_i
\]

Which means ”if \( \neg v_i = 1 \) then we must have \( v_j = 1 \)”, and similarly for the second term. We will record these by adding the edges \( \neg v_i \rightarrow v_j \) and \( \neg v_j \rightarrow v_i \) to \( G \).

Next, we decompose \( G \) to strongly connected components. A strongly connected components is \( S \subset V \) such that for any two vertices \( u, v \in S \) there is a directed path from \( a \) to \( b \) and from \( b \) to \( a \). Let’s order the set of strongly connected components \( S_1, \ldots, S_m \) so that edges from \( v \in S_i \) only go to \( u \in S_j \) for \( j \geq i \). This can be done via putting the strongly connected components in a DAG.

Now, if for some variable both \( x_i \) and \( \neg x_i \) are in the same strongly connected components then \( \phi \) is unsatisfiable, since if \( x \) is a satisfying assignment and \( x_i = 1 \) then, by following the path from \( x_i \) to \( \neg x_i \) we get that \( \neg x_i = 1 \), e.g. \( x_i = 0 \); and similarly if \( x_i = 0 \).

Otherwise, we will show that a satisfying assignment exist. Perform the following process: starting from \( S_m \) and going backwards, if \( x_i \) is an unassigned variable appearing for the first time then set \( x_i = 1 \); if \( \neg x_i \) appears for the first time assign \( x_i = 0 \). The order inside the strongly connected components doesn’t matter as we assume that \( x_i, \neg x_i \) don’t appear in the same component.

Let’s see why the assignment we get is satisfying. Assume \( v_i \lor v_j \) is a clause in \( \phi \) which is not satisfied by the assignment. This means we set \( v_i = v_j = 0 \). This means we found \( \neg v_i \) before \( v_i \) and \( \neg v_j \) before \( v_j \). Note that \( \neg v_i, \neg v_j \) can’t both be in the same component since otherwise, the edge \( \neg v_i \rightarrow v_j \) implies that there is a path \( \neg v_j \sim \neg v_i \rightarrow v_j \), so \( v_j \) should
have appeared before \( \neg v_j \) in our process. So, we encounter say \( \neg v_i \) in a prior component to \( \neg v_j \). However, as \( \neg v_i \rightarrow v_j \) it means that we would encounter \( v_j \) before \( \neg v_j \), e.g. setting \( v_j = 1 \).

**Theorem 4.6.** CLIQUE is NP-complete.

**Proof.** CLIQUE is clearly in NP. We need to show CLIQUE is NP-hard. We will reduce 3SAT to CLIQUE. Let \( \phi \) be a 3-CNF, where we assume w.l.o.g that each clause has three variables. Assume \( \phi \) has \( n \) variables and \( m \) clauses. We define a graph \( G \) on \( 7m \) vertices. For each clause of \( G \) there will be 7 vertices. For example, for \( C(x) = x_1 \lor \neg x_2 \lor x_3 \) these will correspond to the 7 assignments to \( x_1, x_2, x_3 \) which make \( C \) true. We connect two vertices by an edge if they don’t share a variable and its negation.

We next claim \( \phi \) is satisfiable iff \( G \) has a clique of size \( m \). If \( \phi \) is satisfiable, let \( x \) be satisfying assignment. For each clause choose the vertex corresponding to \( x \). They form a clique of size \( m \). On the other side, any clique in \( G \) can contain at most one vertex for each clause, hence a clique of size \( m \) contains exactly one vertex for each clause; these specify a consistent assignment to the variables which satisfy all the clauses.

**Theorem 4.7.** 0-1 PROGRAMMING is NP-complete.

**Proof.** It is clearly in NP. We can reduce 3SAT to 0-1 PROGRAMMING. Represent each clause \( x_1 \lor \neg x_2 \lor x_3 \) as \( x_1 + (1 - x_2) + x_3 \geq 1 \).

More famous NP-complete problems:

- 3-COLORABILITY
- HAMILTONIAN PATH
- VERTEX COVER
- MAX CUT
- TSP
- SUBSET SUM
- BIN PACKING