1 Decision vs Search problems

Search problems are described by functions $f : \{0, 1\}^* \to \{0, 1\}^*$ 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 by functions $f : \{0, 1\}^* \to \{0, 1\}$ 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 $L \subseteq \{0, 1\}^*$ is decideable or computable if the function $f(x) = 1_{x\in L}$ is computable. For $T : \mathbb{N} \to \mathbb{N}$,

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

We have two standard definitions: $P$, polynomial time-solvable problems, which is what we think of as “efficiently solvable problems”,

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

EXP is the class of exponential-time solvable problems,

$$P = \cup_{c > 0} TIME(2^{n^c}).$$

2 The class NP

The class NP captures problems, where solutions can be verified in polynomial time.
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

$$L = \{x : \exists y, |y| \leq |x|^c, M(x, y) = 1\}$$

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:

1. CLIQUE is the 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.

2. 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.

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

4. 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.

5. 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 $L \leq_p L'$. 

2
Definition 3.2 (NP-complete languages). A language $L$ is called **NP-hard** if for any $L' \in NP$ we have $L' \leq_p L$. If furthermore $L \in NP$ then we say $L$ is **NP-complete**.

Claim 3.3. The following holds:

1. If $L \leq_p L'$ and $L' \leq_p L''$ then $L \leq_p L''$.
2. If $L \leq_p L'$ and $L' \in P$ then $L \in P$.
3. If $L \leq_p L'$ 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 $L'' \in NP$, $L'' \leq_p L$. By the previous item $L'' \leq_p L'$ as well. So $L'$ is NP-hard.

Do there exist NP-complete languages? YES.

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

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

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

\[
M_0((M), x, 1^t, y) = \begin{cases} 
\text{accept} & \text{if } |y| \leq t, M(x, y) = 1, M(x, y) \text{ terminates after at most } t \text{ steps} \\
\text{reject} & \text{otherwise}
\end{cases}
\]

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 $L \leq_p L_0$. By definition, there exists $c \geq 1$ and 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 $f_L : \{0, 1\}^* \to \{0, 1\}^*$:

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

Then clearly $x \in L \iff f_L(x) \in L_0$. Moreover, $f$ can be computed in poly-time. \qed
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
\[ \phi(x_1, x_2, x_3, x_4) = (x_1 \lor x_3 \lor \overline{x_4}) \land (x_2 \lor x_3) \land (x_2 \lor \overline{x_1}). \]

Any function \( \phi : \{0, 1\}^n \to \{0, 1\} \) can be written as a CNF of size at most \( n2^n \). A CNF formula is satisfiable if there exists an assignment to the variables making it true. We define
\[ \text{SAT} = \{\text{Satisfiable CNF formulas}\}. \]

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

Proof. Clearly \( \text{SAT} \in \text{NP} \) since a proof that \( \phi \in \text{SAT} \) is an assignment to the variables of \( \phi \) making it true. The harder claim is that \( \text{SAT} \) is NP-hard. Let \( L \in \text{NP} \). 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 without loss of generality that \( M \) has a single work tape with alphabet \( \Gamma \), which initially is initialized by the input, and then is used to perform the work. Recall that \( \{0, 1, b\} \subset \Gamma \).

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 tape is ever used. We define variables capturing the contents of the tape at every step of the computation. For \( 1 \leq i, t \leq T \):
- \( TAPE^t_{i, \gamma} = 1 \) if the \( i \)-th symbol in the \( t \)-th time step in the tape equals \( \gamma \).
- \( HEAD^t_i = 1 \) if the head is in the \( i \)-th cell in the \( t \)-th time step.
- \( STATE^t_q = 1 \) 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 \( y \in \{0, 1\}^t \) a potential witness.

\[
\begin{align*}
TAPE^0_{i, \gamma} &= \begin{cases} 
1 & \text{if } i = 1, \ldots, n \text{ and } \gamma = x_i, \\
1 & \text{if } i = n + 1, \ldots, n + t \text{ and } \gamma = y_{i-n}, \\
0 & \text{otherwise}
\end{cases} \\
HEAD^0_i &= \begin{cases} 
1 & \text{if } i = 0, \\
0 & \text{otherwise}
\end{cases} \\
STATE^0_q &= \begin{cases} 
1 & \text{if } q \text{ is the initial state of } M, \\
0 & \text{otherwise}
\end{cases}
\end{align*}
\]
Consistency. The tape contents, head locations and state in step \( t+1 \) depend just on those on time \( t \). Moreover, these are local computations. For example, the HEAD at time \( t \) is at coordinate \( i \) only if the HEAD at time \( t-1 \), \( t \) or \( t+1 \).

Recall that the transition function of a TM is \( \delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{-1,0,1\} \) where \( \delta(q, \gamma) = (q', \gamma', d) \) where at state \( q \) and input character \( a \), the TM writes \( \gamma' \) at the current location, changes the state to \( q' \) and moves in direction \( d \) in the tape.

So, for every \( q \in Q, \gamma \in \Gamma, d = \{-1,0,1\} \) and every \( t = 1, \ldots, T \) we have the constraint:

\[
\text{If } HEAD^t_i = 1, TAPE^t_i,\gamma = 1, STATE^t_q = 1 \text{ and } \delta(q, \gamma) = (q', \gamma', d) \text{ then } HEAD^{t+1}_{i+d} = 1, TAPE^{t+1}_i,\gamma' = 1, STATE^{t+1}_{q'} = 1.
\]

This is a formula on a constant number of variables, independent of the input size. So we can create a CNF formula expressing it. In addition, we need to add formulas that for every \( t \), exactly one of \( \{HEAD^t_i : i = 1, \ldots, T\} \) is 1; exactly one of \( \{STATE^t_q : q \in Q\} \) is true; and exactly one of \( \{TAPE^t_i,\gamma : \gamma \in \Gamma\} \) is true for every \( i \). All of these can be expressed by CNF formulas in the same way. Then, we just need to AND all the clauses and get the final CNF.

Output. The output of the computation is \( TAPE^T_{0,1} \).

Writing as a CNF. All the identities above can be written as a CNF of size \( \text{poly}(T) \). Then, requiring them all together to hold is done by AND-ing them, which will still be a polynomial size CNF. 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 \( L <_p SAT \) by the reduction \( f(x) = \phi_x \).

5 Proving NP completeness by reductions

Once we showed that SAT is NP-complete, we can use this to show that many other combinatorial problems are also NP-complete. The general recipe is as follows: assume we know that a language \( L \) is NP-Complete (say SAT) and we want to use this to prove that some other language \( L' \) is also NP-Complete. We need to show two things:

1. \( L' \) is in NP. This is usually shown directly.

2. \( L <_p L' \). That is, there is a poly-time computable function \( f \) which maps inputs of \( L \) to inputs of \( L' \), such that \( x \in L \iff f(x) \in L' \).

Definition 5.1 (k-SAT). A k-CNF is a CNF where each clause has at most \( k \) variables.

\[
k - SAT = \{\text{Satisfiable k-CNF formulas}\}.
\]
Theorem 5.2. 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 \( \text{SAT} <_p \text{3SAT} \). This shows 3SAT is NP-hard. Clearly 3SAT is also in NP, hence it is NP-complete. \( \square \)

Theorem 5.3. 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). Lets assume any clause in \( \phi \) has exactly two literals, e.g. by replacing \( x_1 \lor x_1 \). 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 \( v_i \lor v_j \), it can be equivalently written \( \neg v_i \rightarrow v_j \) or \( \neg v_j \rightarrow v_i \) 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 \( a \). Lets 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.

Lets 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 $\neg v_i \to v_j$ implies that there is a path $\neg v_j \leadsto \neg v_i \to v_j$, 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 $\neg v_i \to v_j$ it means that we would encounter $v_j$ before $\neg v_j$, e.g. setting $v_j = 1$.

**Theorem 5.4.** *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 5.5.** 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