CSE 431/531: Analysis of Algorithms (Summer 2023)
Theory of complexity

Chen Xu

August 1-8, 2023

Outline

- Problems
- Classes of Problems
- Reductions
- NP-Complete Problem
- FPT and approximation algorithms

What are hard problems?

- There are some problems that people have not found algorithms that solve them in $O(n^k)$ for a small fixed constant k.
- For example, the 0/1-Knapsack problem and traveling salesman problem we have learned in the lecture.
- People found a fact that these problems all stem from one single problem that solving it would lead to the solution for all.

Introduction to Boolean Formula

Boolean Formula:
Formulas in which the variables can take on only two possible values: TRUE (1) or FALSE (0).

Basic Operations:

- AND ($\wedge$): The result is TRUE if both variables are TRUE.
- OR ($\vee$): The result is TRUE if either or both variables are TRUE.
- NOT ($\neg$): The result is the inverse of the input variable.

Example:

$$(a \wedge b) \vee (\neg a \wedge c)$$

Setting a=1, b=1, and c=0, the evaluation will be:

$$(1 \wedge 1) \vee (\neg 1 \wedge 0) = 1 \vee (0 \wedge 0) = 1 \vee 0 = 1$$

Thus, the formula is TRUE with this assignment.

Definition of the SAT Problem

SAT Problem

Input: A Boolean formula ϕ of n variables
Output: **Yes** if there exists an assignment that makes this formula outputs True, otherwise **No**.

Example (Yes instance):

$$(a \wedge b) \vee (\neg a \wedge c) \vee (b \wedge \neg c)$$

a	b	c	$(a \wedge b) \vee (\neg a \wedge c) \vee (b \wedge \neg c)$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

No Instance of the SAT Problem

Example (No instance):
Consider the formula:

$$a \wedge \neg b \wedge b$$

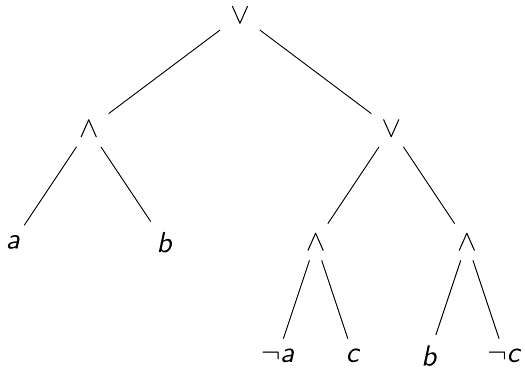
We analyze all possible assignments:

a	b	$a \wedge \neg b \wedge b$
0	0	0
0	1	0
1	0	0
1	1	0

No satisfying assignment exists, so the formula is "unsatisfiable".

Boolean formula parse tree

Parse Tree:



CNF Form:
A common way to encode Boolean formula is in Conjunctive Normal Form (CNF).
A CNF formula is a conjunction of one or more **clauses**, where a clause is a disjunction of **literals** (variables or their negation).
Example:
The formula $(a \wedge b) \vee (\neg a \wedge c) \vee (b \wedge \neg c)$ can be encoded in CNF as:
$(a \vee \neg a) \wedge (b \vee c \vee \neg c)$

3-SAT

- 3-SAT is the SAT problem when we restrict the input boolean formula to have at most three literals per clauses.

3-SAT Problem

Input: A Boolean formula ϕ of n variables, m clauses of at most 3 literals
Output: **Yes** if there exists an assignment that makes this formula outputs True, otherwise **No**.

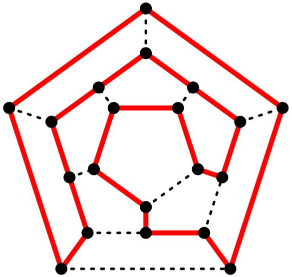
Hamiltonian Cycle Problem

A Hamiltonian cycle in a graph is a closed loop on the graph such that every node is visited exactly once.

HC problem

Input: A graph G.
Output: **Yes** if a Hamiltonian cycle exists, otherwise **No**.

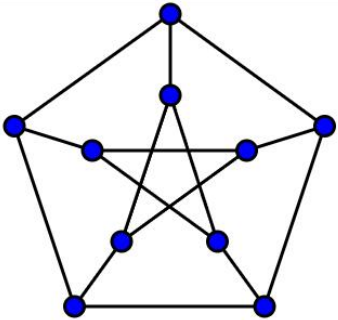
Yes instance:



Hamiltonian Cycle Problem – No instance

Can you find a hamiltonian cycle in the graph below?

No instance:



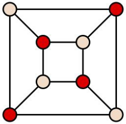
Decision Problem vs Optimization Problem

- Some problems do not answer yes/no. For example, a lot of optimization problems answer how many.
- A problem is called **decision problem** if the answer is 0/1.

Maximum Independent Set Problem

Definition

An **independent set** of $G = (V, E)$ is a subset $I \subseteq V$ such that no two vertices in I are adjacent in G.



Maximum Independent Set Problem

Input: A graph G

Output: What is the maximum size of the independent set?

Note that this is an optimization problem.

Decision version of Maximum Independent Set problem

k-Independent Set problem

Input: A graph G and a positive number k
Output: Does there exist an independent size **at least** k?

- If we can solve the decision version, we can do multiple calls using different k values to get the answer to the maximum independent set. (Think about binary search)

Encoding

- Notice that the graphs and the Boolean formulae, the input of the above problems, can all be encoded as binary string.
- The size of the string is denoted as $|s|$.
- All decision problems can be abstracted as computing a function $X : \{0,1\}^* \to \{0,1\}$. The problem is solved by an algorithm in polynomial time if the algorithm gives the correct output $X(s)$ and terminates in at most $p(|s|)$ steps, $p(*)$ is a polynomial function.

Class of P

- The **complexity class P** is a collection of problems that can be solved in $O(n^c)$ polynomial time.
- We already learned a lot of **P** class problems.

Certificate

- Usually, our decision problem looks like this:

Some problem

Input: Some x

Output: Does there exist some y s.t. something about x, y holds?

- We call the y part a **certificate**.
- The **certificate** is the proof that the instance is a **yes-instance**.

Certificate for SAT

SAT Problem

Input: A Boolean formula ϕ of n variables
Output: Does there exist an assignment that makes this formula output True?

- The assignment of variables $x_1 = 0, x_2 = 1, x_3 = 0, ..., x_n = 1$ **that makes the formula output True** is a certificate.
- Each certificate can be verified whether it really makes the formula output True or not.
- For SAT problem, the verification can be done very efficiently in polynomial time.

Certificate for Hamiltonian Cycle

HC problem

Input: A graph G.

Output: Does there exist a Hamiltonian cycle?

- A cycle that is Hamiltonian is a certificate.
- The verification can also be done very efficiently in polynomial time.

Polynomial Verifier

- For a decision problem, the algorithm that verifies the certificate is called a **verifier**.
- If the verifier, taking the certificate and the input from the original problem, runs in polynomial time, then it is a **polynomial verifier**.

Class of **NP**

- The complexity class **NP** is the set of all problems for which there exists a polynomial verifier.
- Do all decision problems in **P** also have a polynomial verifier? Yes, class of **P** decision problems have polynomial verifiers and they themselves can be solved in polynomial time. So **P** is a subset of **NP**. The tree-joint-matched problem in HW2 is one of them. (Exercise: What is the certificate and the verifier?)

How to show a decision problem is in **NP**

- To show a problem in **NP**, it must first be a decision problem.
- The next step is to construct the polynomial verifier algorithm for it. The input of the verifier algorithm should also be polynomial of the size of the problem.

Example: Show $HC \in \mathbf{NP}$

- Input for the original problem is the graph G.
- Certificate is the Hamiltonian Cycle in the form of vertex sequence C.
- Build a polynomial verifier:

IsHam(G,C)

```
if |C| ≠ n or C[1] ≠ C[n] then
    return False
for i = 2 to n do
    if C[i] is not adjacent to C[i − 1] in G then
        return False
    for j = i + 1 to n do
        if C[i] = C[j] then
            return False
return True
```

- By definition,

$$\exists C \; IsHam(G, C) = 1 \;\Leftrightarrow\; HC(G) = 1$$

What do $\exists, \forall$ mean?

- $\exists x$ – Does there exist x such that it satisfies (some logic)?
  If there is at least one then it evaluates True, otherwise False.
- $\forall x$ – Do all x satisfy (some logic)?
  If all satisfy then it evaluates True, otherwise False.
- $\neg\exists x$ – Does there not exist x such that it satisfies (some logic)?
  We do not evaluate $\neg\exists$, we evaluate $\exists$ then negate the result.
- $\forall x \neg \equiv \neg\exists x$ – Do all x satisfy (negation of some logic) is equivalent to the above statement.
  This $\forall$ can be evaluated on the negation of the logic. Many logics while being polynomial size, their negations might not be polynomial size.

Complement of an **NP** problem

- Let us reiterate. Let $R(x, y)$ be the polynomial verifier where x is the input of original problem and y is the input of the certificate. The characteristic of the **NP** class is that $R(x, y)$ is easy but $\exists y R(x, y)$ is hard.
- The complement of a problem is putting a big negation before $\exists y R(x, y)$. It turns it into $\neg \exists y R(x, y)$ and finally $\forall y \neg R(x, y)$
- Let another $R'(x, y) = \neg R(x, y)$ we have $\forall y R'(x, y)$
- The complement of an **NP** problem goes like this:

Some complement problem

Input:	Some x
Output:	Do all y satisfy something about x, y?
Or:	Does there not exist y s.t. something about x, y holds?

Class of **co-NP**

- If a problem is in **NP**, its complement problem is in **co-NP**.
- If a problem is in **co-NP**, its complement problem is in **NP**.
- Yes-instances of an **NP** problem is exactly No-instances of its complement and vice-versa.

Example of **co-NP** problem

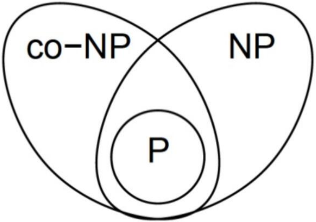
co-HC problem

Input: A graph G.
Output: Does G satisfy that there exists no Hamiltonian cycle in G?

- The yes-instance for this problem is the no-instance for HC problem.
- We are unlikely to find a certificate for the yes-instances of co-HC problem.
- It is only possible to find certificates for the no-instances although not easy.

P,NP,co-NP relations

- Whether **NP** = **co-NP**? is another open problem. Most believe that **NP** $\neq$ **co-NP**
- Exercise: How to show **P** is a subset of **co-NP**.
- Here is the most believed relation between **P**, **NP** and **co-NP**. Note that we also believe **P** $\neq$ **NP**$\cap$**co-NP** too.



Self Reduction

- Recall that we had learned about concept of subproblems in previous lectures. For instance:

Some greedy algorithm

```
Greedy(V,n)
...
(Perform actions on the input)
...
temp = Greedy(V \ {v}, n-1)
(Make some decisions, compile some information)
return result
```

- This is viewed as a **reduction** from a problem on n vertices to a smaller problem on $n-1$ vertices. The problem structure is the same. It is calling itself. To solve problem P_n we need to solve P_{n-1}. P_{n-1} is not harder than P_n.

Reduction from one problem to another problem

- The reduction from problem A to problem B looks like (assume A and B are decision problems):

Algorithm that solves A

$A(x)$:
...
(Perform actions on the input x, get a converted input $f(x)$)
...
temp = B(f(x))
(Process temp and get result)
return result

- We use B as a black box.
- If this algorithm A works, then we say A reduces to B. Denote as $A \preceq B$

Further simplification

- We can simplify it:

Algorithm that solves A

$A(x)$:
...
(Perform actions on the input x, get a converted input $f(x)$)
...
return B(f(x))

- Computing $f(x)$ should not take longer than computing $B(f(x))$. In fact, the polynomial time reduction is putting more restrictions on $f(*)$.

Polynomial time reduction

- The **polynomial time reduction** restricted that $f(*)$ needs to be computed in polynomial time. We denote polynomial time reduction from A to B as $A \preceq_p B$.
- If $A \preceq_p B$ and B can be solved in polynomial time, then A can be solved in polynomial time.
- If $A \preceq_p B$ and A cannot be solved in polynomial time, then B cannot be solve in polynomial time.

NP-complete problem

Definition for **NP-complete** problem

Problem A is **NP-complete**, if

1. $A \in NP$, and
2. For every problem $B \in NP$, $B \preceq_p A$.

- In other words, A is one of the hardest problems in **NP**.

How do we show that a problem is hard in general?

Although you may have a trivial algorithm that runs in exponential time.

Trivial algorithm for HC(G)

1: **for** all possible cycles c in G **do**
2:     **if** c is Hamiltonian **then**
3:         **return** True
4: **return** False

Showing a trivial algorithm does not suffice to claim hardness. Because there is no guarantee your trivial algorithm is the best possible one.

However, you can show the hardness of a problem P by reducing an NPC problem $A(x)$ to P in polynomial time.

Some NPC problem $A(x)$

1: $x' = f(x)$, $f(x)$ runs in poly-time.
2: **return** $P(x')$

If your algorithm for A is **correct**, then you have shown problem P is at least as hard as A. Solving P faster will result in solving A faster.

How to show a problem is NP-complete

- First, show the problem is in **NP**.
- Next, there are two ways of showing NP-completeness.
  1. By proving that the problem matches the definition of NP-complete problem. This is hard.
  2. By reducing from another NP-complete problem using polynomial reductions. This is easier.

Circuit-SAT is **NP-complete**

- In practice, we usually choose the polynomial reduction. This requires a unique **first** problem that is proved NP-complete by definition.
- People have already done the work decades ago. They found the first NP-complete problem called Circuit Satisfiability Problem (Circuit-SAT). It is a very close variant to the SAT problem.

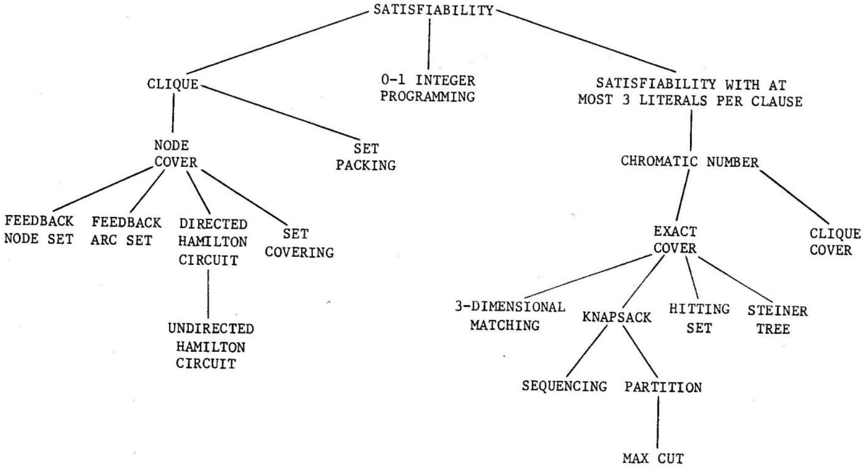
Theorem 1

Circuit-SAT is NP-complete.

- The basic idea is to translate every step of a verifier into boolean circuit implementation so that every **NP** class problem with a polynomial verifier reduces to a boolean circuit SAT of polynomial size. We will not cover this in the lecture. Further readings: [Coo71]

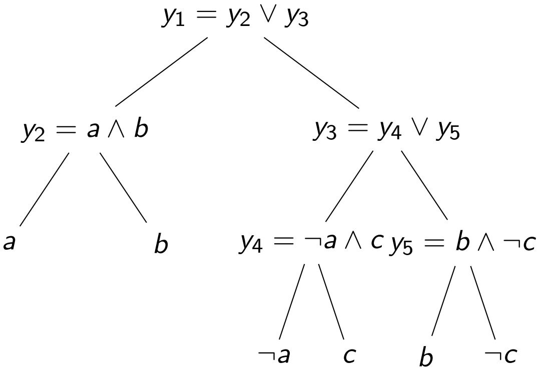
Classical NP-complete problems

- People have also done reductions to a lot of classical problems.



SAT $\preceq_p$ 3-SAT

- Recall that we have a binary parse tree for every boolean formula. We can put additional variables on the inner nodes. Let ϕ be any boolean formula. As an example, we have $\phi = (a \wedge b) \vee (\neg a \wedge c) \vee (b \wedge \neg c)$.



The original ϕ is equivalent to the following boolean formula:

$$
\begin{aligned}
&(y_1 = y_2 \vee y_3) \wedge \\
&(y_2 = a \wedge b) \wedge \\
&(y_3 = y_4 \vee y_5) \wedge \\
&(y_4 = \neg a \wedge c) \wedge \\
&(y_5 = b \wedge \neg c)
\end{aligned}
$$

SAT $\preceq_p$ 3-SAT

- For each of the small clauses we can convert it into a 3-CNF. For example, $y_1 = y_2 \vee y_3$

y_2	y_3	y_1	$y_1 = y_2 \vee y_3$
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$$
\begin{array}{ll}
(y_1 = y_2 \vee y_3) & \Leftrightarrow \\
(y_2 \vee y_3 \vee \neg y_1) & \wedge \\
(y_2 \vee \neg y_3 \vee y_1) & \wedge \\
(\neg y_2 \vee y_3 \vee y_1) & \wedge \\
(\neg y_2 \vee \neg y_3 \vee y_1) &
\end{array}
$$

SAT $\preceq_p$ 3-SAT

- Let the instance from SAT be ϕ. We perform the boolean table conversion for every inner node and conjunct them altogether. This gives us a 3-SAT instance ψ such that

$$\phi \text{ is satisfiable} \Leftrightarrow \psi \text{ is satisfiable}$$

- Therefore we can just feed the ψ to the 3-SAT solver.

SAT solver

1: From ϕ, construct ψ using the procedure described above.
2: **return** 3-SAT(ψ)

Theorem 2

3-SAT is NP-complete.

3-SAT $\preceq_p$ k-Independent Set

Recall that we have the k-Independent Set problem. We would like to reduce from 3-SAT to it.

k-Independent Set problem

Input: A graph G and a positive number k
Output: Does there exist an independent size **at least** k?

- Our target is to construct a (G, k) instance from any 3-SAT instance ϕ such that

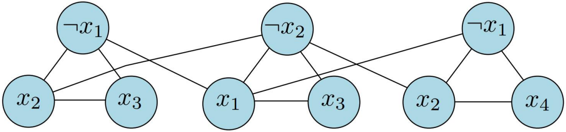
$$\phi \text{ is satisfiable} \Leftrightarrow G \text{ has an independent set of size at least } k$$

3-SAT $\preceq_p$ k-Independent Set

- Given any 3-SAT instance ϕ, create a vertex for every literal in a clause. $3m$ vertices are created.
- Connect the vertices corresponding to the literals within a clause. They form triangles. $3m$ edges are created.
- Across the different clauses, connect the vertices with conflicting literals. i.e. connect x_1 to $\neg x_1$.
- Set k be the number of clauses.

Example

- This is a yes-instance of 3-SAT:
  $\phi = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4)$
- The reduced instance of k-Independent Set is:

$G =$ 

$k = 3$

Correctness

1. ϕ is satisfiable $\Rightarrow$ G has an independent set of size k.
   - Since ϕ is satisfiable at least one of literals in a clause is picked. We can pick one of corresponding vertex in the graph. We cannot pick more than one because the triangle restricts us to pick only one.
   - ϕ is satisfiable implies that there is no conflict, which means no edges whose both ends across the different triangles are picked. There are exactly k vertices in k different triangles in the independent set.
2. G has an independent set of size k $\Rightarrow$ ϕ is satisfiable.
   - Suppose I is the independent set of G, then each triangle has exactly one vertex being picked.
   - The vertices are not conflicting each other, meaning if the vertex named x_1 is picked then $x_1 = 1$ otherwise 0.
   - This is the truth assignment that makes ϕ return true.
3. Therefore, ϕ is satisfiable $\Leftrightarrow$ G has an independent set of size k. The reduction is correct.

Theorem 3

k-Independent Set is NP-complete.

References

[Coo71] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1971), pp. 151–158.