LEC20: SATISFIABILITY

CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

Lecture slide courtesy of Prof. Steven Skiena
THE MAIN IDEA

Suppose I gave you the following algorithm to solve the *bandersnatch* problem:

- **Bandersnatch**(G)
  - Convert G to an instance of the Bo-billy problem Y.
  - Call the subroutine Bo-billy on Y to solve this instance.
  - Return the answer of Bo-billy(Y) as the answer to G.

Such a translation from instances of one type of problem to instances of another type such that answers are preserved is called a *reduction*. 
Now suppose my reduction translates G to Y in $O(P(n))$:

1. If my Bo-billy subroutine ran in $O(P'(n))$ I can solve the Bandersnatch problem in $O(P(n) + P'(n'))$

2. If I know that $\Omega(P'(n))$ is a lower-bound to compute Bandersnatch, then $\Omega(P'(n) - P(n'))$ must be a lowerbound to compute Bo-billy.

Why? If I could solve Bo-billy any faster, then I could violate my lower bound by solving Bandersnatch using the above reduction. This implies that there can be no way to solve Bo-billy any faster than claimed!
A PORTION OF THE REDUCTION TREE FOR NP-COMPLETE PROBLEMS.
A decision problem $C$ is NP-complete if:

1. $C$ is in NP, and
2. Every problem in NP is reducible to $C$ in polynomial time.

$C$ can be shown to be in NP by demonstrating that a candidate solution to $C$ can be verified in polynomial time.

Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition 1.
SATISFIABILITY

We must start with a single problem that is absolutely, certifiably, undeniably hard: satisfiability problem

- **Problem:** Satisfiability
- **Input:** A set of Boolean variables $V$ and a set of clauses $C$ over $V$.
- **Output:** Does there exist a satisfying truth assignment for $C$—i.e., a way to set the variables $v_1, \ldots, v_n$ true or false so that each clause contains at least one true literal?
Example 1: \( V = v_1, v_2 \) and \( C = \{ \{ v_1, \overline{v_2} \}, \{ \overline{v_1}, v_2 \} \} \)

A clause is satisfied when at least one literal in it is TRUE. \( C \) is satisfied when \( v_1 = v_2 = \text{TRUE} \).

Example 2: \( V = v_1, v_2 \)

\[
C = \{ \{ v_1, v_2 \}, \{ v_1, \overline{v_2} \}, \{ \overline{v_1} \} \}
\]

Although you try, and you try, and you try and you try, you can get no satisfaction.

There is no satisfying assignment since \( v_1 \) must be FALSE (third clause), so \( v_2 \) must be FALSE (second clause), but then the first clause is not satisfiable!
Satisfiability is known/assumed to be a hard problem in the worst case.

Every top-notch algorithm expert in the world has tried and failed to come up with a fast algorithm to test whether a given set of clauses is satisfiable.

Further, many strange and impossible-to-believe things have been shown to be true if someone in fact did find a fast satisfiability algorithm.
3-SATISFIABILITY

- **Problem:** 3-Satisfiability (3-SAT)
- **Input:** A collection of clauses $C$ where each clause contains exactly 3 literals, over a set of Boolean variables $V$.
- **Output:** Is there a truth assignment to $V$ such that each clause is satisfied?

- Note that this is a more restricted problem than SAT.
- If 3-SAT is NP-complete, it implies SAT is NP-complete but not visa-versa, perhaps long clauses are what makes SAT difficult?!
- After all, 1-SAT is trivial!
To prove it is complete, we give a reduction from SAT \( \propto \) 3-SAT.

We will transform each clause independently based on its length.

Suppose the clause \( C_i \) contains \( k \) literals.

- If \( k = 1 \), meaning \( C_i = \{z_1\} \), create two new variables \( v_1 \); \( v_2 \) and four new 3-literal clauses:
  \[
  \{v_1, v_2, z_1\}, \{v_1, \overline{v}_2, z_1\}, \{\overline{v}_1, v_2, z_1\}, \{\overline{v}_1, \overline{v}_2, z_1\}
  \]
- Note that the only way all four of these can be satisfied is if \( z \) is TRUE.
If \( k = 2 \), meaning \( \{z_1; z_2\} \), create one new variable \( v_1 \) and two new clauses: \( \{v_1; z_1; z_2\}, \{\overline{v_1}; z_1; z_2\} \)

If \( k = 3 \), meaning \( \{z_1; z_2; z_3\} \), copy into the 3-SAT instance as it is.

If \( k > 3 \), meaning \( C_i = \{z_1; z_2; \ldots; z_n\} \), create \( n - 3 \) new variables and \( n - 2 \) new clauses in a chain: where for \( 2 \leq j \leq n - 3 \), \( C_{i,j} = \{v_{i,j-1}, z_{j+1}, \overline{v}_{i,j}\} \), \( C_{i,1} = \{z_1, z_2, \overline{v}_{i,1}\} \), and \( C_{i,n-2} = \{v_{i,n-3}, z_{n-1}, z_n\} \).

This transform takes \( O(m+n) \) time if there were \( n \) clauses and \( m \) total literals in the SAT instance.

Since any SAT solution also satisfies the 3-SAT instance and any 3-SAT solution describes how to set the variables giving a SAT solution, the transformed problem is equivalent to the original.
Why Does the Chain Work?

- If none of the original variables in a clause are TRUE, there is no way to satisfy all of them using the additional variable:
  
  \[(F; F; T); (F; F; T); \ldots; (F; F; F)\]

- But if any literal is TRUE, we have \(n - 3\) free variables and \(n-3\) remaining 3-clauses, so we can satisfy each of them.
  
  \[(F; F; T); (F; F; T); \ldots; (F; T; F); \ldots; (T; F; F); (T; F; F)\]

- Any SAT solution will also satisfy the 3-SAT instance and any 3-SAT solution sets variables giving a SAT solution, so the problems are equivalent.
A slight modification to this construction would prove 4-SAT, or 5-SAT,... also NP-complete.

However, it breaks down when we try to use it for 2-SAT, since there is no way to stuff anything into the chain of clauses.

Now that we have shown 3-SAT is NP-complete, we may use it for further reductions. Since the set of 3-SAT instances is smaller and more regular than the SAT instances, it will be easier to use 3-SAT for future reductions.

Remember the direction to reduction!

\[ SAT \propto 3 - SAT \propto X \]
Note carefully the direction of the reduction.

We must transform every instance of a known NP-complete problem to an instance of the problem we are interested in. If we do the reduction the other way, all we get is a slow way to solve x, by using a subroutine which probably will take exponential time.

This always is confusing at first - it seems bass-ackwards.

Make sure you understand the direction of reduction now - and think back to this when you get confused.
**Problem: Vertex Cover**

**Input:** A graph $G = (V,E)$ and integer $k \leq |V|$.

**Output:** Is there a subset $S$ of at most $k$ vertices such that every $e \in E$ has at least one vertex in $S$?

Here, four of the eight vertices suffice to cover.

It is trivial to find a vertex cover of a graph – just take all the vertices. The tricky part is to cover with as small a set as possible.
To prove completeness, we reduce 3-SAT to VC.

From a 3-SAT instance with \( n \) variables and \( C \) clauses, we construct a graph with \( 2N + 3C \) vertices.

For each variable, we create two vertices connected by an edge:

To cover each of these edges, at least \( n \) vertices must be in the cover, one for each pair.
For each clause, we create three new vertices, one for each literal in each clause. Connect these in a triangle.

At least two vertices per triangle must be in the cover to take care of edges in the triangle, for a total of at least $2C$ vertices.

Finally, we will connect each literal in the flat structure to the corresponding vertices in the triangles which share the same literal.
Claim: G has a vertex cover of size N + 2c iff S is satisfiable

- This graph has been designed to have a vertex cover of size \( n + 2c \) if and only if the original expression is satisfiable.
- To show that our reduction is correct, we must show that:
  1. Every satisfying truth assignment gives a cover.
     - Select the N vertices corresponding to the TRUE literals to be in the cover.
     - Since it is a satisfying truth assignment, at least one of the three cross edges associated with each clause must already be covered – pick the other two vertices to complete the cover.
2. *Every vertex cover gives a satisfying truth assignment.*

- Every vertex cover must contain $n$ first stage vertices and $2C$ second stage vertices.
- Let the first stage vertices define the truth assignment.
- To give the cover, at least one cross-edge must be covered, so the truth assignment satisfies.
Every SAT defines a cover and Every Cover Truth values for the SAT!

Example: $V_1 = V_2 = True$, $V_3 = V_4 = False$. 
As you can see, the reductions can be very clever and very complicated.

While theoretically any NP-complete problem can be reduced to any other one, choosing the correct one makes finding a reduction much easier.

$$3 \text{ SAT } \propto \text{ VC}$$

As you can see, the reductions can be very clever and complicated.

While theoretically any NP-complete problem will do, choosing the correct one can make it much easier.
MAXIMUM CLIQUE

- Instance: A graph $G = (V;E)$ and integer $j \leq v$.
- Question: Does the graph contain a clique of $j$ vertices, ie. Is there a subset of $v$ of size $j$ such that every pair of vertices in the subset defines an edge of $G$?
- Example: this graph contains a clique of size 5.
When talking about graph problems, it is most natural to work from a graph problem - the only \( NP \)-complete one we have is vertex cover!

If you take a graph and find its vertex cover, the remaining vertices form an independent set, meaning there are no edges between any two vertices in the independent set, for if there were such an edge the rest of the vertices could not be a vertex cover.
Clearly the **smallest vertex cover gives the biggest independent set**, and so the problems are equivalent.

Delete the subset of vertices in one from the total set of vertices to get the order!

Thus finding the maximum independent set must be **NP-complete**!
In an independent set, there are no edges between two vertices. In a clique, there are always between two vertices.

Thus if we complement a graph (have an edge iff there was no edge in the original graph), a clique becomes an independent set and an independent set becomes a Clique!
Thus finding the largest clique is NP-complete:

- If VC is a vertex cover in G, then V - VC is a clique in G’.
- If C is a clique in G, V - C is a vertex cover in G’.