CS 3510 Algorithms 3/26/2024

NP Hard Problems

Lecturer: Jan van den Brand *Scribe(s): Ankith Thalanki*

In today's lecture we covered several new **NP-Hard** problems, and showed they were **NP-Hard** by making a reduction from SAT or 3-SAT to them. We specifically explored two questions: "Is there a Hamiltonian path in graph **G**?" and "And is the size of the largest independent set in graph **G** greater than or equal to k?" We also explored how every SAT problem can be easily converted into the 3-SAT problem, showing that 3-SAT is as hard as SAT.

SAT

The SAT, or The Propositional Satisfiability Problem asks if an assignment of variables exists such that a boolean formula in CNF (conjuctive normal form) evaluates to true. If a boolean formula is represented in conjunctive normal form, then it is represented as a series of clauses where variables inside the clause are connected by OR's and clauses are connected by AND's. For example, $(x_1 \vee x_2) \wedge (x_3 \vee \neg x_4)$ is an example of the conjunctive normal form, where there are two clauses and four variables. This problem would evaluate to true, as if $X_1 = T$ and $x_3 = T$, regardless of the values of the other variables, this boolean expression would evaluate to true. The SAT problem was the first problem to be proven to be in **NP-Hard** (and **NP-Complete**) via the Cook-Levin theorem, which in essence showed that every algorithm could be represented as a boolean formula (solvable by SAT). We will mostly focus in these notes in showing that other algorithms are **NP-Hard** by reducing SAT to them.

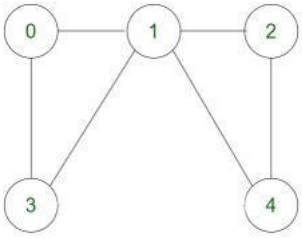


Figure 1: Example of a connected graph where a Hamiltonian path exists. A Hamiltonian path that covers every node can be made, starting at 0, then going to 3,1,4,2.

Hamiltonian Path

The Hamiltonian Path problem asks whether there is a path that exists in a graph such that every node is traversed once, and exactly once. There are other similar variations of the problem, like the Hamiltonian Cycle, where the starting and ending node must be the same, but all of the variants of these problems are **NP-Complete**. It is trivial to show

that the Hamiltonian Path problem is in **NP**, because it is easy to verify that any path follows the restrictions of the problem in polynomial time. It is however much harder to prove that the problem is in **NP-Hard**, as that requires proving it is harder than all other problems in **NP**. Luckily that has already been done via SAT, so we can simply show that the Hamiltonian Path is harder than SAT, by making a polynomial time reduction from SAT to the Hamiltonian Path problem.

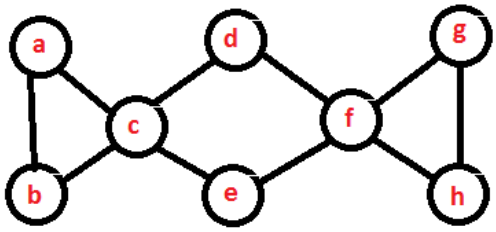
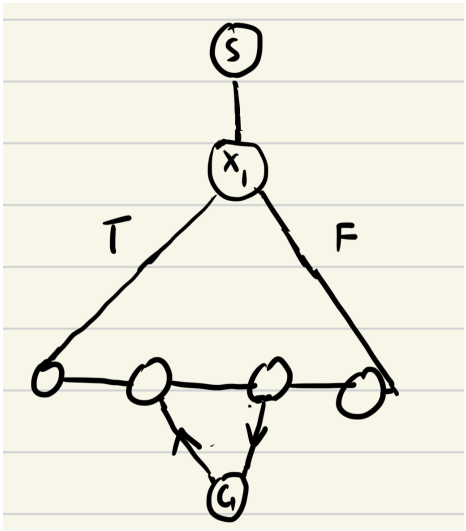


Figure 2: Example of a graph where no Hamiltonian path exists.

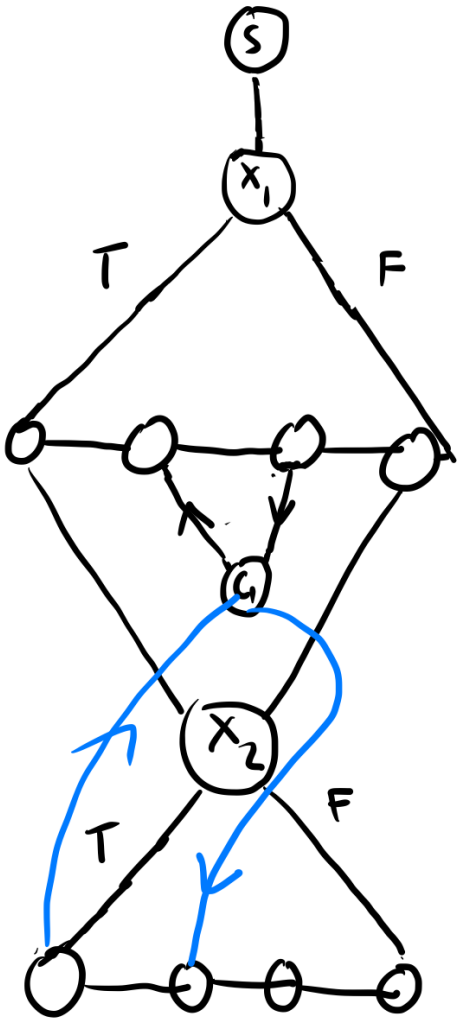
Reducing SAT to Hamiltonian Path

Ultimately, in order to represent SAT as a Hamiltonian Path, we need to convey two ideas as a Hamiltonian Path problem: boolean variables and logical constraints on those boolean variables. How can we represent that a boolean variable is true and false, and add specific constraints toward that? The following image showcases how we might a single variable with a single constraint, like $\neg x_1$.



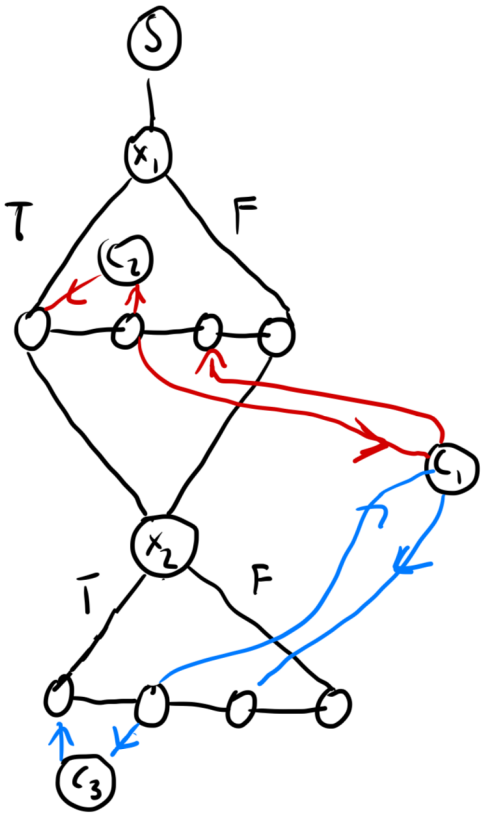
In the above graph, S is the start node, and x_1 represents the choice of whether x_1 is true or false. The only way to make a Hamiltonian path in this graph is if x_1 is false, and

goes down the negative path, because that enables us to go through c_1 condition, which is not possible if you go the other way. It's not to hard to add another condition and variable to the boolean expression, for something like $\neg x_1 \vee x_2$. The corresponding graph is right below:



In the above Hamiltonian graph, we have two variables, whose true and false values are determined by which edge is chosen by the Hamiltonian path. You might wonder if after going past x_1 if you can skip the condition and go straight to x_2, but the problem with that approach is that you would never be able to go back up properly traverse all the nodes there. In this graph, c_1 can be filled by either x_1 being false, or x_2 being true, as we only need one of the variables to fill the condition. If we have another condition/clause, we can simply add a c_2 node with the proper connections to the variables that dictate their corresponding values. With these building blocks we can represent any CNF boolean

formula as a graph. And if the graph does not have a Hamiltonian path, that means that the corresponding CNF boolean formula is over constrained and can never be true. For example, this is what a more complicated CNF that doesn't have a corresponding Hamiltonian Path $((x_1 \vee x_2) \wedge (\neg x_1) \wedge (\neg x_2))$ would look like as a graph:



There is no Hamiltonian Path because c_2 and c_1 can only be traversed if x_1 and x_2 are false, but that makes c_3 impossible to traverse as it needs at least one of them to be true. This matches up with the CNF boolean formula being impossible to be true.

As we have found a way to solve SAT via the Hamiltonian Path problem, via a conversion that only takes polynomial time (making the graph is relatively fast), we have reduced SAT to the Hamiltonian Path problem, indicating that the Hamiltonian Path problem is also **NP-Hard**, and since it is in **NP** as well, it is **NP-Complete** in general.

3-SAT

3-SAT is equivalent to SAT, with the additional requirement however that the clauses in the boolean must have 3 components each. For example, this would be a valid conjunctive normal form for 3-SAT: $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_4 \vee \neg x_1)$, because each clause has 3 and only 3 terms. There is fundamentally no difference between SAT and 3-SAT however as

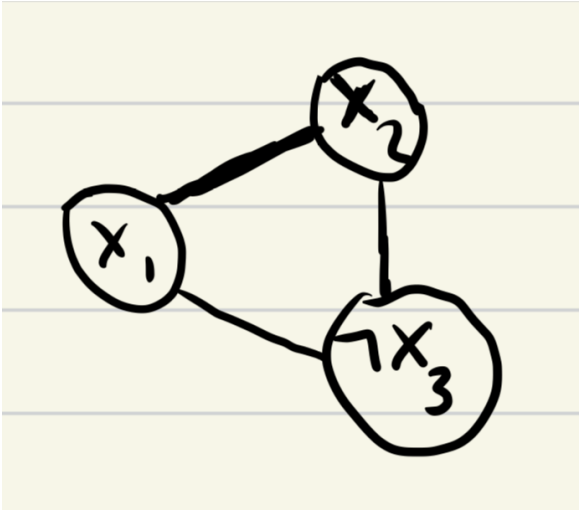
any conjunctive normal form can be converted to have clauses of size 3 without actually changing any of the functionality. This makes 3-SAT also **NP-Hard** and **NP-Complete**, since it is identical to SAT. For example, the CNF SAT formula $(x_1 \vee x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2)$ is equivalent to $(x_1 \vee x_2 \vee y) \wedge (x_3 \vee x_4 \vee \neg y) \wedge (x_1 \vee x_2 \vee z) \wedge (x_1 \vee x_2 \vee \neg z)$ in 3-SAT.

Independent Set

The Independent Set problem asks if there is an independent set in a graph **G** with a size greater than or equal to some number k. We can show that this problem is **NP-Complete** by showing that it is **NP-Hard** by reducing 3-SAT to it. It is easy to show that the problem is **NP**, as given any potential solution we simply need to check that each node in the independent set does not have any of the other nodes as potential neighbours. Once we've proved this problem is in **NP-Complete**, it is rather easy to prove that other problems like Vertex-Cover or Max-Clique is **NP-Complete**.

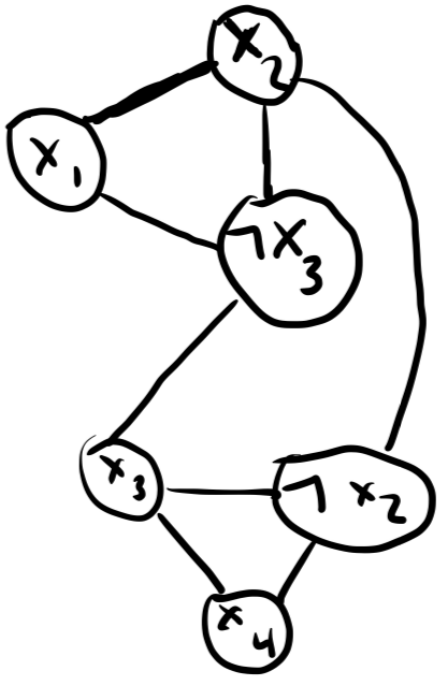
Reducing 3-SAT to Independent Set

In order to solve 3-SAT as Independent Set, we need to represent two ideas in the Independent Set problem: That one item in each clause must be true, and that a variable x and $\neg x$ cannot both be true at the same time. In the Independent Set, we can define a variable being chosen as it being true. Therefore if we can show that the Independent set version of a 3-SAT problem that has k clauses has k independent nodes, we will have shown that the 3-SAT problem evaluates to yes, and vice versa. Let us first define a graph gadget that defines a clause in 3-SAT as simply a triangle containing the three vars in the graph. So for $(x_1 \vee x_2 \vee \neg x_3)$, the gadget would be as below:



In the above gadget, only one of x_1, x_2, or $\neg x_3$ can be chosen as true or be included in the independent set. While it is indeed possible for a clause in 3-SAT to have multiple true boolean components, if we have two nodes with x_1 and x_1, we don't need to include both, only one. This means that even if we have multiple true components in a 3-SAT clause, we

only need to include one in the independent set. This also ensures that we don't double count clauses, as we want the presence of each node/variable in the indepndent set to denote one clause being true. Another situation we do need to worry about is when we have x_1 and $\neg x_1$, as we need to enforce the fact somehow that both cannot be in the independent set. We can do this by adding an edge between any node x_i and its opposite it $\neg x_i$. Therefore, if we had a complicated 3-SAT equation with multiple clauses, like $(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_2 \vee x_4)$, we can represent below as so:



The edges between gadgets represent restrictions on variable values. As long as we can find an independent set of size 2, the number of clauses, that means we can satisfy this version of 3-SAT.