CS 3510 — Algorithms, Spring 2022

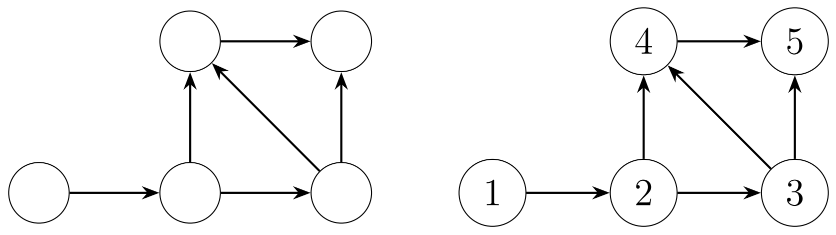
Lecture 19: Hamiltonian Path

April 12, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

1 Hamiltonian Path

A Hamiltonian path is a path that traverses every vertex in a directed graph G exactly once. The HAMPATH problem is a decision problem of whether or not a Hamiltonian path exists. In the following graph, we can make a Hamiltonian path.

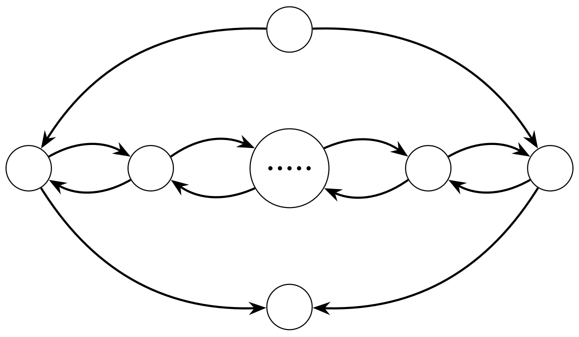


Theorem. *Hamiltonian Path is NP-Complete.*

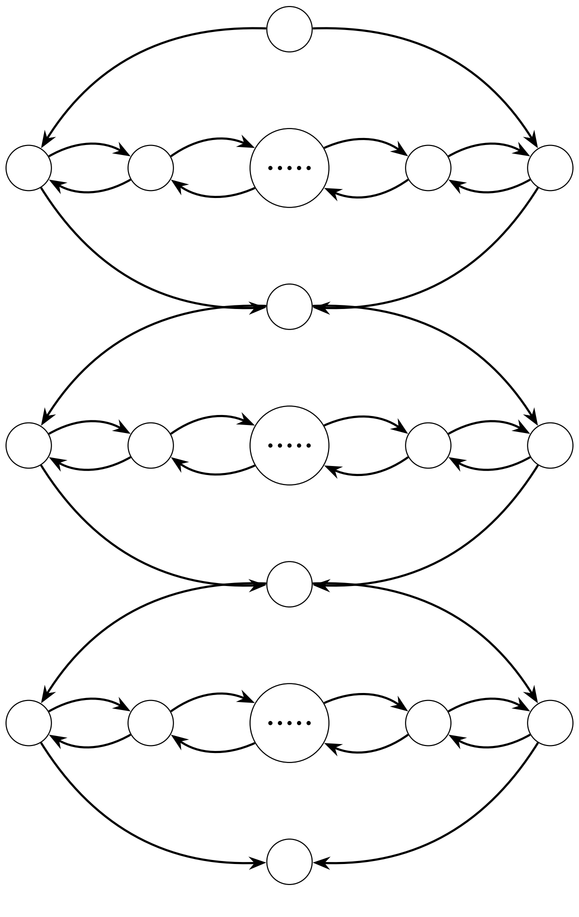
Proof. First, we want to show that HAMPATH $\in$ NP. We know that some candidate solution would be given as an ordered list of vertices, so we can simply traverse in this order and ensure we have traversed each vertex. We can check that it is the correct length in polynomial time and that it does not repeat edges also in polynomial time.

Next, we want to show a reduction from 3SAT to HAMPATH. Given a formula F with n variables x_i and m clauses c_i, create some graph G with property that if F is satisfiable, then G has a Hamiltonian path.

For each variable or literal in the original boolean expression x_i, we construct a *gadget* or subgraph that follows a diamond shape with two vertices at the top of a bottom connected to a row of vertices. Visually, it would like the following:

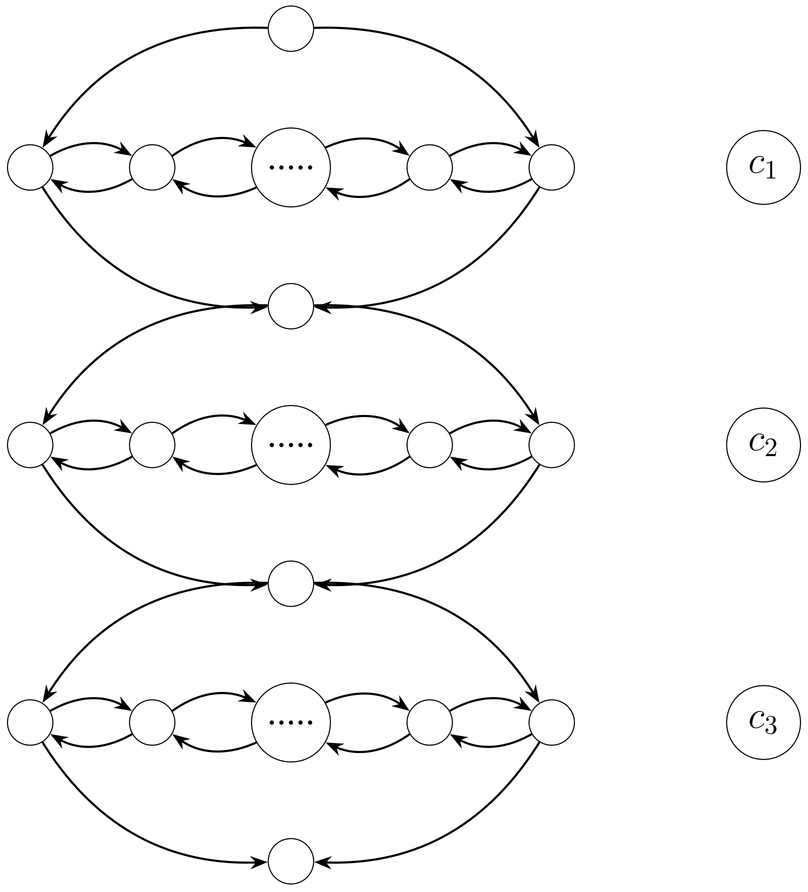


The vertex from the top is the only way we can get to this gadget from other gadgets. The middle row (including the vertex labeled .....) will be a bunch of vertices that correspond to clauses (discussed later). We will stack these segments on top of each other (as each segment corresponds with a literal x_i: when we have n literals, then we will have n segments). Here is an example of 3 segments.



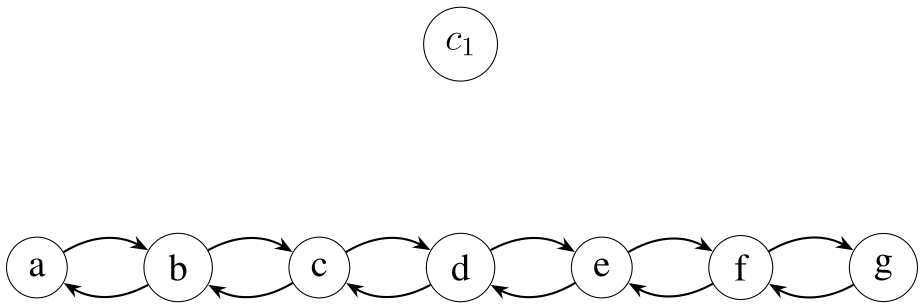
In the above graph, because we are trying to find a Hamiltonian path, we can either traverse each row left or right. If we switch directions midway, then this would cause us to traverse the same vertex twice meaning it would not be a Hamiltonian path. These will correspond to assignments for the original problem.

Now that we have considered each literal in the graph, we need to consider the different clauses. For each clause, we add a single vertex c_i that will be connected to each of the rows in the segments of the corresponding literals in that clause.

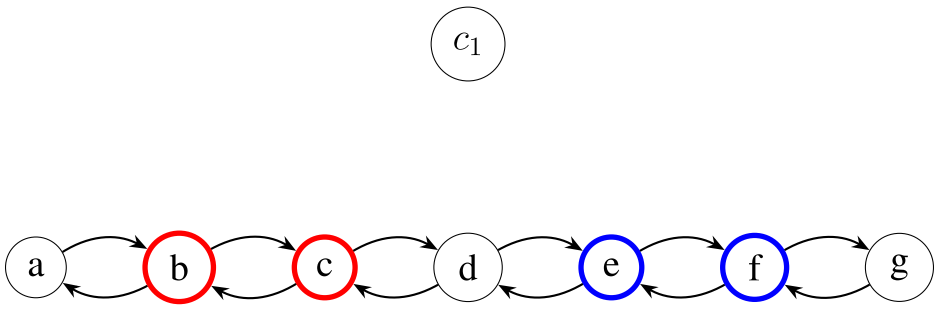


The above graph has the vertices of each clause without the edges. We will now describe how we connect these clause edges to the rest of the graph. In order to describe this connection, we'll take an example. Let $c_1 = x_3 \vee \neg x_4 \vee x_7$. Let's also say that going left to right on a segment row means that the literal corresponding to that row is true, and otherwise false.

With c_1, we want it to be the case that x_3 and x_7 can only reach the vertex c_1 if they go $\underline{\text{left to right}}$ for the corresponding segment row. This corresponds with the fact that x_3 and x_7 are *not* negated in c_1. Likewise, x_4 should only be reachable if we go right to left as x_4 is negated in c_1. With this in mind, we now need to define the specifics of how we connect segments to these clause vertices.

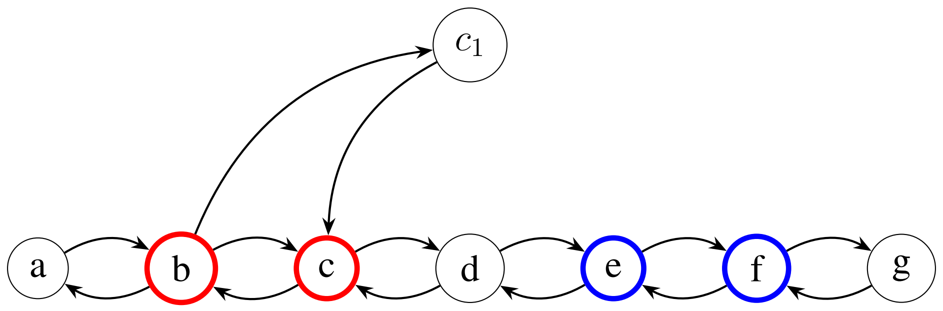


Let the above represent an abridged version of the graph we are constructing that focuses on the row of a specific gadget for the literal x_i. First, we will group vertex pairs in sets of twos representing a specific clause. In the above, b, c will correspond with c_1. d, e will correspond with c_2, etc.



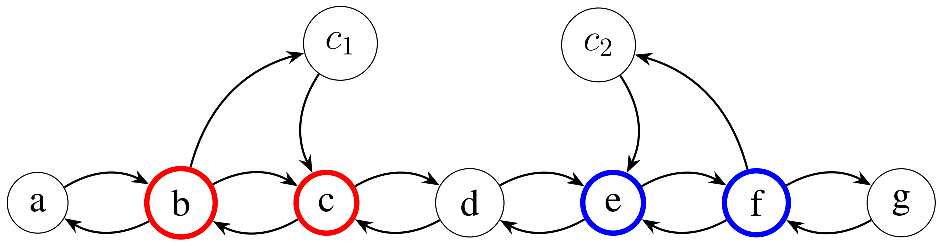
The vertices corresponding to c_1 are now labeled red and the vertices corresponding to c_2 are now labeled blue. If it is the case that x_i (the current gadget we are looking at) is *not in* c_i, we do not do anything with the vertices that correspond to that clause in the current row.

For this specific example, let's say say there is a clause $c_1 = x_i \vee x_j \vee x_k$, then $x_i \in c_1$. Since there is no negation, we want x_i to be true in c_1. Hence, if x_i satisfies c_1, we should be coming from the left, into c_1, then coming back into the row for x_i. This can be done by an edge leaving the left vertex in the clause pair in the row and coming back on the right pair.



In the above, if we were to create a Hamiltonian path on this abridged row, we could go $a \to b \to c_1 \to c \to d \to \cdots$. From here, we may see that if the literal was negated in a clause, we would do a very similar edge addition but swapping the direction. If the literal was negated in the clause, satisfying the clause with this literal would require this literal to be false (moving from the right to left), hence we should leave from the right clause pair and come back to the left clause pair.

Let c_2 be a clause that has x_i negated within it.



In the above, if we were to create a Hamiltonian path, we could go from $g \to f \to c_2 \to e \to d \to \cdots$. In this path, we go from right to left rather than left to right as this literal needs to be false to satisfy the c_2 clause. To construct the graph G, we repeat this process for every literal in every clause and all the clauses.

This concludes the construction of G. Now, we must prove correctness.

Proof. We claim this reduction is correct. In other words, F is satisfiable if and only if G has a Hamiltonian path. Based on the satisfying assignment, we can go left to right or right to left on each row and visit the c_i from one of the variables such that they satisfy that c_i. Hence, F being satisfiable implies G has a Hamiltonian path.

Conversely, we want to show that if G has a Hamiltonian path, then F is satisfiable. If the Hamiltonian path goes row by row through G, then we know there is a satisfying assignment as we can simply determine whether any literal is true or false based on whether we traverse it left or right.

It is also true that we cannot skip around in the graph. In other words, if we are in the subgraph for x_1, we could leave to some clause c_j vertex while traversing the row. From c_j, we could leave to some other literal's subgraph x_i. However, if this happens, then this is not a Hamiltonian path as there would be no way to get back to the vertices you left. We will not prove this (the proof is tedious due to cases).

We must argue that this reduction (constructing G) takes only polynomial time. If the reduction takes more than polynomial time, this is an invalid reduction. In this case, we have m clause vertices and per literal we have $3m + 1$ vertices, hence we have a linear number of vertices. Hence, this reduction takes polynomial time.

2 Challenges

For the remaining questions (the challenges given in class), there will not be proofs of reductions. That is an exercise to you, and it is critical that you understand how to prove them (use the previous reductions we've proved as an outline and check your proofs on Piazza).

2.1 ALMOST-SAT

ALMOST-SAT is a satisfiability process similar to SAT that returns an assignment such that exactly $m - 1$ clauses makes the expression true.

Theorem 2.1. *ALMOST-SAT is NP-Complete.*

Proof. A full proof will not be given. We reduce SAT to ALMOST-SAT. In other words, if we have some blackbox algorithm to solve ALMOST-SAT, in order to show ALMOST-SAT is NP-Hard, we must show that this blackbox algorithm could be used to solve SAT as well.

Say F is an input to SAT, we want to solve F by using ALMOST-SAT. Since ALMOST-SAT gives an expression for $m-1$ clauses, we add k clauses onto F such that $m+k-1$ of those clauses are guaranteed to include the original m clauses. Let $k=2$ where we add $(f) \wedge (\neg f)$. Hence, we are trying to solve a boolean expression with $m+2$ clauses. ALMOST-SAT will return an assignment using $m+1$. Since two of the clauses contradict, the clause it removes will either be (f) or $(\neg f)$. Hence, the assignment for the $m+1$ clauses contains an assignment for the original m clauses as well as an additional clause (either f or $\neg f$).

This shows ALMOST-SAT is NP-Hard (as we have shown a reduction). What else do we need to prove for ALMOST-SAT to be NP-Complete?

2.2 Subgraph Isomorphism

Subgraph Isomorphism (SI) is a problem where given a graph G and H, we need to determine whether or not H is isomorphic to a subgraph of G.

Theorem 2.2. *Subgraph Isomorphism is NP-Complete.*

Proof. We will prove Subgraph Isomorphism is NP-Hard. It is on you to finish the proof to NP-Complete. We choose to reduce from the Clique problem. In order to show that Subgraph Isomorphism is NP-Hard, we need to reduce from some problem. We reduce by using a blackbox algorithm for Subgraph Isomorphism in order to solve the Clique problem.

The input to the Clique problem is some graph G and a value k and returns whether or not G has a clique of size k. We can solve the Clique problem by determining a subgraph of G is isomorphic to a clique. In other words, if we can determine if G has subgraph H where H is a k-clique, we have solved the Clique problem.

The Subgraph Isomorphism is exactly that. The Clique input G, k can be used to make inputs G, H for the Subgraph Isomorphism where H is some k-clique (k vertices that are fully connected). If H is subgraph isomorphic within G, then G has a Clique of size k.

Hence, we have shown Subgraph Isomorphism is NP-Hard.