Lecture 22 - NP Complete Problems

Eric A. Autry

Last time: Amortized Analysis

This time: NP Complete Problems

Next time: Solving NP Complete Problems

Project 3 due tonight!

Homework 7 due this Thursday the 29th.

Project 4 due Thursday, December 6th.

Notes on Bellman-Ford, Greed, and MST have all been updated on Sakai. Notes on Amortized Analysis will be updated with examples by tomorrow's office hours.

NP Hard Problems

So far in this course, we've seen a number of problems that can be solved optimally and efficiently using a number of techniques:

- Divide and Conquer
- Use-it or Lose-it
- Dynamic Programming
- BFS/DFS and variations (Dijkstra's)

However, there exist many problems that cannot be solved optimally in an efficient way.

For these **NP Hard** problems, there are no known polynomial time algorithms.

NP Hard Problems

For these **NP Hard** problems, there are no known polynomial time algorithms.

Very important note: claiming that there cannot exist polynomial time algorithms for these problems is FALSE.

This hasn't been proven one way or the other. Nobody knows if there might exist polynomial time algorithms for these problems.

Overview of NP Completeness

Our goals for today:

- Introduce a number of NP complete problems.
- Discuss the meaning of NP hard, NP complete, and P
- Show several *reductions* that are used to prove that problems are NP complete.

For tomorrow:

- Examine solution techniques for NP hard problems.
  - Brute force vs Dynamic Programming
- Approximation Algorithms
- Introduce the idea of a search algorithm.

SAT

The first NP hard problem we will discuss is the satisfiability problem (SAT):

- We are given n literals, which can be either True or False.
  - Either a Boolean variable (x) .
  - Or the negation of a Boolean variable $(\neg x)$.
- These literals are then used to build clauses which consist of the disjunction (logical *or*) of several literals.
  For example:
  $$(x \vee \neg y \vee z)$$
- The full SAT problems then considers a Boolean formula in *conjunctive normal form*, which is the conjunction (logical *and*) of these clauses. Example:
  $$(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z)$$
- The goal of the SAT problem is to identify a satisfying truth assignment for each variable such that each clause is true, or report that no such truth assignment exists.

SAT Example

Let's take a look at that example again:

$$(x \vee y \vee z) \wedge (x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x) \wedge (\neg x \vee \neg y \vee \neg z)$$

Is there a way to satisfy all of the clauses?

There is not: the middle three clauses ensure that the variables have the same value!

Traveling Salesman Problem (TSP)

For the Traveling Salesman Problem, we are given:

- A set of n vertices.
- All $n(n-1)/2$ distances between them.
- A budget b.

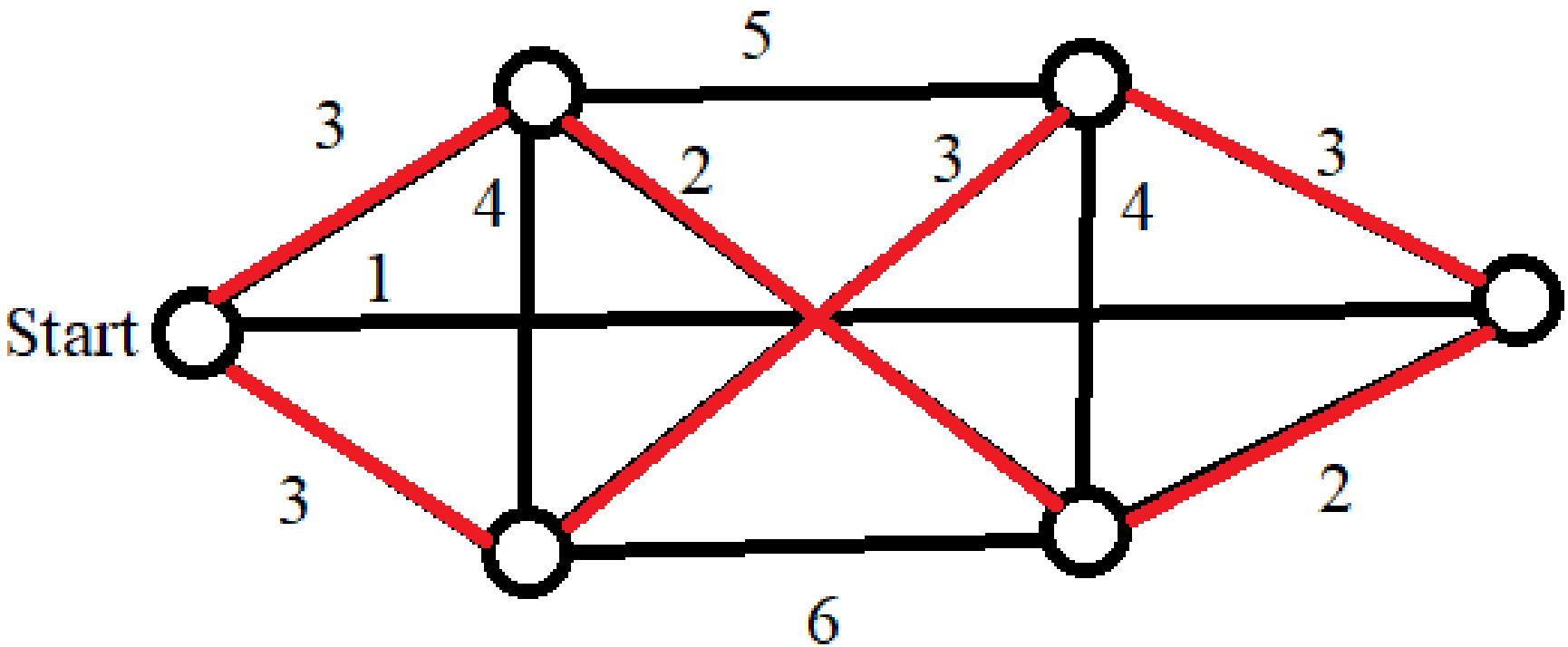
Our goal is to find a **tour** of all of the vertices with a cost less than b, or report that such a tour does not exist.

- A **tour** is a cycle of the vertices that passes through each vertex exactly once.

Quick example on board.

Traveling Salesman Problem (TSP)

The TSP for the following graph is in red:



Decision Problems

But wait: the version of TSP defined in the project is seeking an optimal solution. Where did the budget b come from?

We have taken an **optimization problem** and converted it into what is called a **decision problem**.

Instead of finding the optimal solution, we are framing the question as checking whether a given solution satisfies the problem.

These problems are equivalent:

- Say we have an algorithm for finding the optimal solution.
  - Then when we are given a budget, simply find the optimal solution and compare its value to the budget.
- Say we have an algorithm for solving the decision problem.
  - Use binary search to find the optimum value (i.e., smallest budget for which a tour still exists).

NP Hard

Both SAT and TSP are examples of **NP Hard** problems.

These are problems that can be solved and verified in polynomial time by a nondeterministic Turing Machine.

- i.e., they are 'Nondeterministic Polynomial time' problems.
- These problems are considered to be difficult problems.
- There are no known deterministic polynomial time algorithms to solve them.

There is a special group of NP hard problems that are called **NP complete** problems (includes SAT and TSP).

- We will prove a very important fact about these NP complete problems that relates them all together.

This leads to the million dollar question: does $P = NP$?

- Problems in class P can be solved in polynomial time.
- It is commonly believed that they are not equal since NP seems harder than P.
- But we don't actually know.

Independent Set

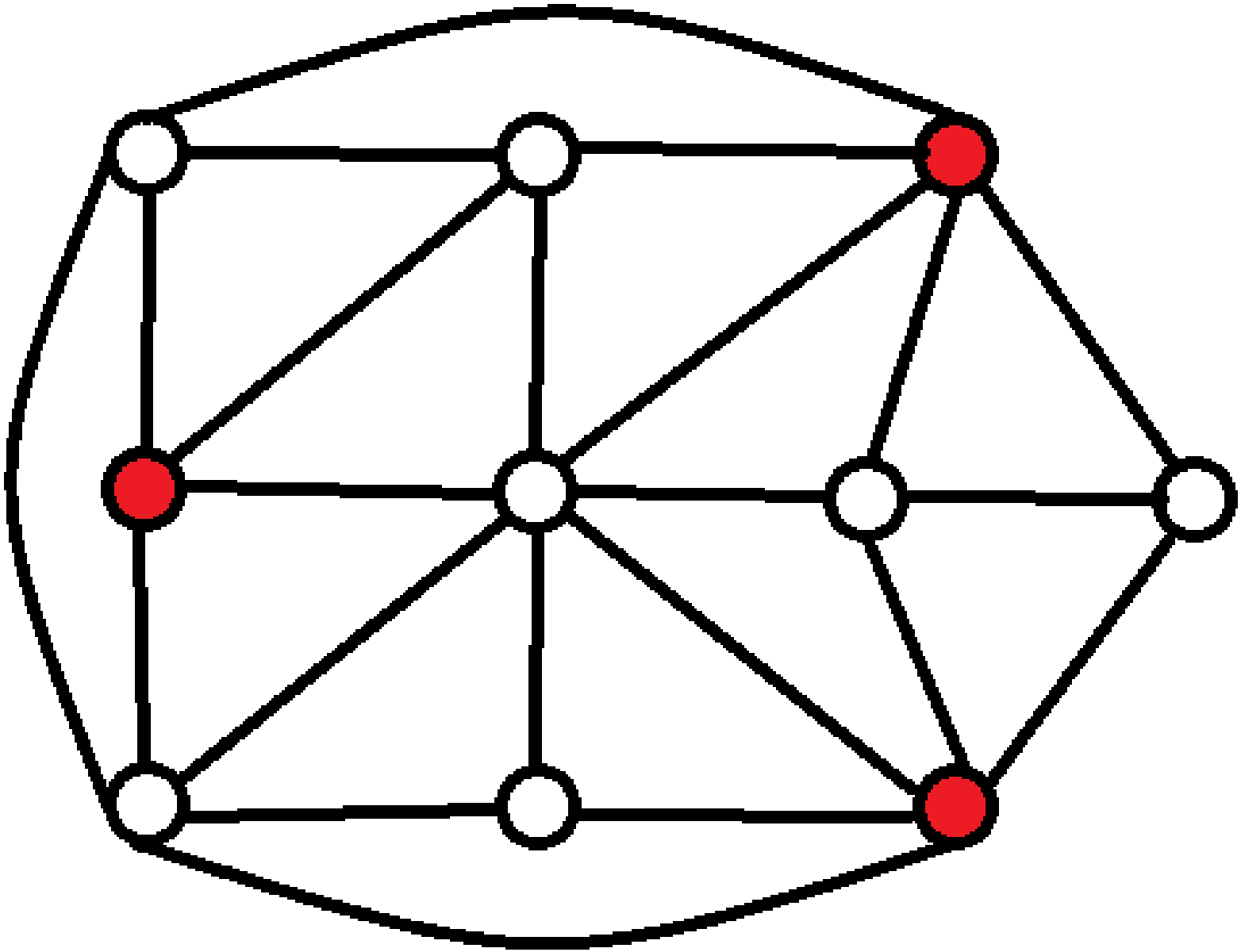
The next NP complete problem we'll introduce is called the Independent Set problem.

- We are given a graph and an integer g.
- The problem is to determine whether there is a set of g vertices that are independent.
- i.e., a set of vertices that have no edges between any two of them.

It turns out that if the graph is a tree, this problem can be solved efficiently using dynamic programming or greed. (This is the party planning problem!)

However, in a general graph, no polynomial time algorithm is known.

Independent Set



Vertex Cover and Clique

For the Vertex Cover problem:

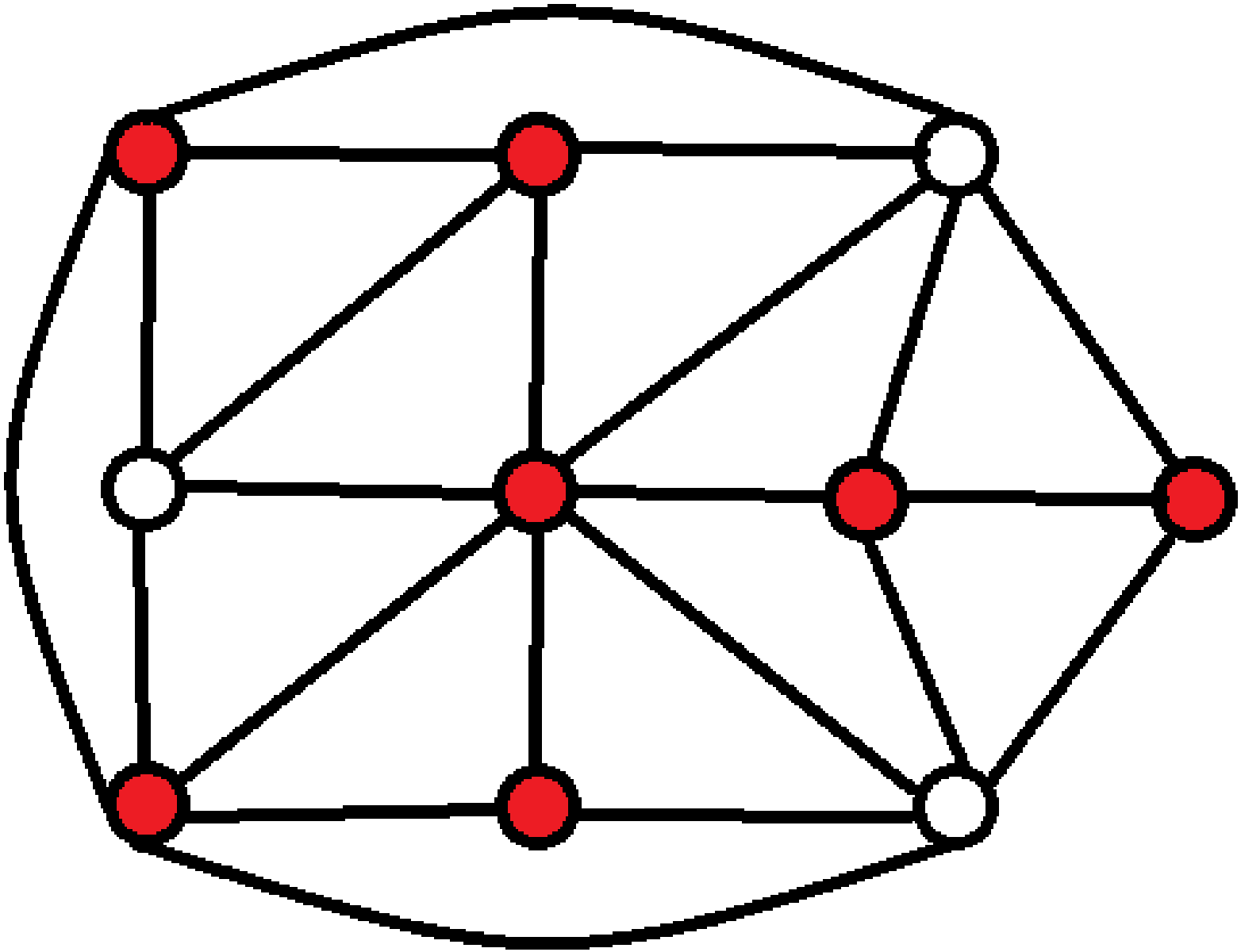
- We are given a graph $G = (V, E)$ and a budget b.
- The goal is to find a set of b vertices such that each edge is *covered*.

For the Clique problem:

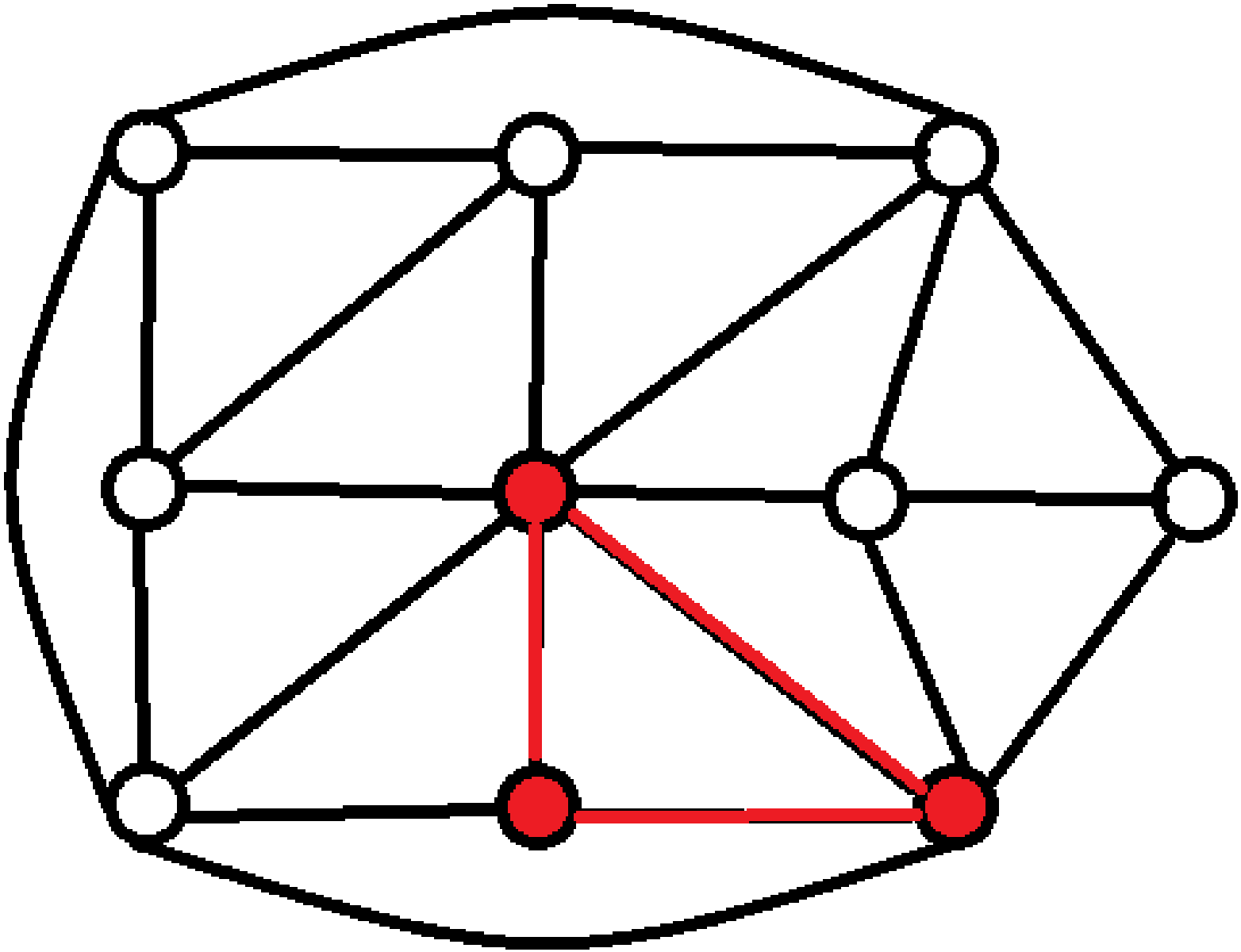
- We are given a graph $G = (V, E)$ and a goal g.
- Is it possible to find a set of g vertices such that all possible edges between them are present?

Note: these last three problems (Independent Set, Vertex Cover, Clique) are closely related to each other.

Vertex Cover



Clique



Hamiltonian/Rudrata Path and Cycle

9th century Kashmiri poet Rudrata asked:

- Is it possible for a knight to visit every square on a chess board without repeating a square?

To approach solving this problem, we will:

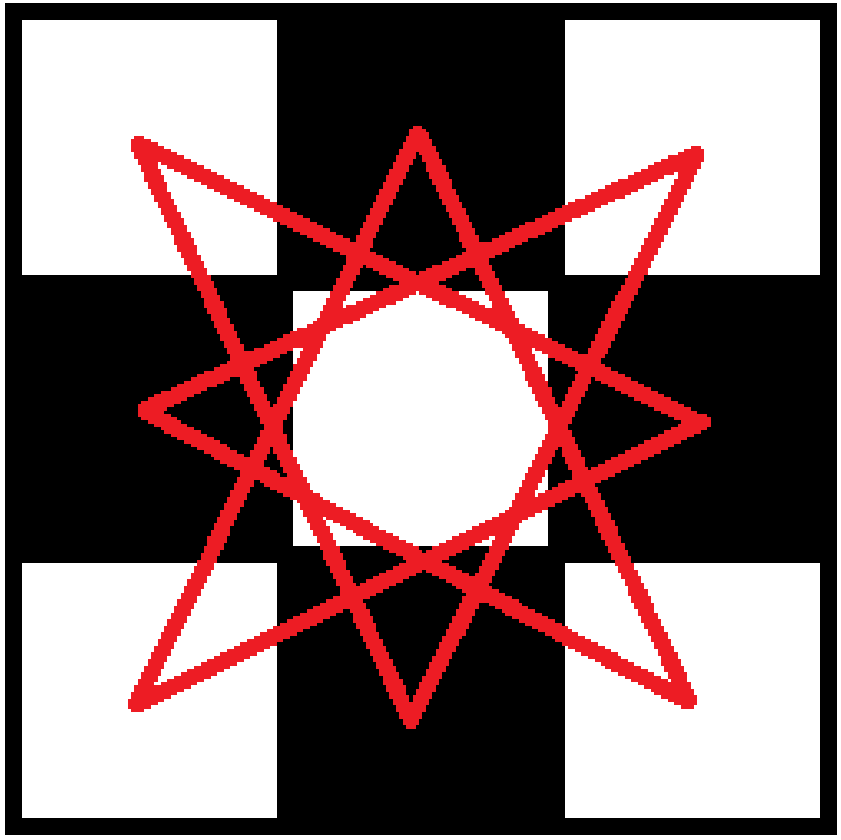
- Represent the squares of the chessboard as vertices.
- Edges will be present between vertices when a legal knight move connects the two squares.

Now we can define what is known as the Hamiltonian Cycle (named after 19th century mathematician who studied the problem):

- Given a graph, find a cycle that visits each vertex only once before returning to the starting vertex, or report that such a cycle is not possible.

This sounds very similar to the Traveling Salesman Problem...

Hamiltonian/Rudrata Path and Cycle



Hamiltonian/Rudrata Path and Cycle

The Hamiltonian Cycle Problem:

- Given a graph, find a cycle that visits each vertex only once before returning to the starting vertex, or report that such a cycle is not possible.

A variation of this problem is the Hamiltonian Path:

- Given a graph, find a path that visits each vertex only once, or report that such a path is not possible.

Again, we see two NP hard problems that seem very closely related.

Longest Path

Another NP hard problem is the Longest Path problem:

- Given a graph with nonnegative edges.
- Given a start vertex s and a destination vertex t.
- Given a goal g.
- Decision problem: is there a simple path from s to t that has a weight of at least g?

This is sometimes called the *taxicab rip-off* problem because the taxi driver wants you to spend as much money as possible during the drive.

Knapsack and Subset Sum

The famous Knapsack problem is defined by:

- We have n items that we want to place into our knapsack.
- Each item has an integer weight w_i and value v_i.
- The knapsack has a *capacity* weight W.
- We are given a goal g.

The Knapsack decision problem is to determine if there is a set of items we can place into our knapsack (without exceeding the weight limit) that have a total value greater than g.

A special case of the Knapsack problem is called the Subset Sum problem, and it is also NP hard:

- Each item's weight is its value, i.e., $w_i = v_i$.
- The goal is to fill the knapsack completely, i.e., $g = W$.
- Note: this corresponds to finding a subset of a given set of integers that adds up to exactly W.

Bin Packing

A similar problem to the Knapsack problem is known as the Bin Packing problem:

- We are given a set of items of varying volumes v_i.
- We are given bins that each have a capacity volume V.
- We are given a goal g.

The Bin Packing problem is to determine if the items can all fit in g bins, or report if such a packing is not possible.

NP Completeness

We've seen that a number of these problems all seem related to each other:

- TSP and Hamiltonian Cycles both required a tour of vertices in a graph.
- Independent Set, Vertex Cover, and Clique all asked for a subset of vertices of a graph that had specified edge properties.
- Knapsack, Subset Sum, and Bin Packing all involved packing items in containers.

It turns out that we are correct and all of these problems are related to each other.

They are all examples of NP Complete problems.

NP Completeness

NP Complete problems have a very special property:

- They can all be **reduced** to each other in polynomial time!

Consider NP Complete problems A and B:

- Suppose we have an algorithm that can solve problem B.
- The we can solve A by reducing it to problem B, i.e., converting it into an equivalent version of problem B.
- Then we use our algorithm to solve this new version of B.
- And finally convert the solution back to problem A.

So, all NP complete problems are equivalent to each other!

Significance of P=NP:

- If we can find a polynomial time algorithm to solve **even one** of the NP complete problems, we can solve **all** of them in polynomial time!

Polynomial Time Reductions

Say we are reducing from problem X to problem Y.

- Step 1: For these reductions, we need to consider the **decision problem** for both X and Y.
- Step 2: We start with the inputs to problem X. Our goal is to use problem Y in order to obtain a solution for problem X. (Imagine we already have code that can solve problem Y.)
- Step 3: We need to describe how to convert the inputs of problem X into the inputs for problem Y in polynomial time.
- Step 4: We then assume that problem Y gives us a solution or reports that no solution is possible. (i.e., we call the function for solving problem Y using our newly defined inputs.)
- Step 5: We then convert, in polynomial time, the solution returned by problem Y into a solution for problem X. Note: we have to prove that if Y gives a solution, then our solution for X is correct, **and** if Y does not give a solution, then there could not have been a solution for X.

Independent Set to Vertex Cover

The setup for this reduction is that we are given an instance of the Independent Set problem: a graph and a goal size g of the independent set.

We want to convert this problem into a Vertex Cover problem so that we can later reconstruct the solution.

We make one important observation:

- The set of vertices S is a vertex cover of a graph if and only if the set $V - S$ is an independent set.

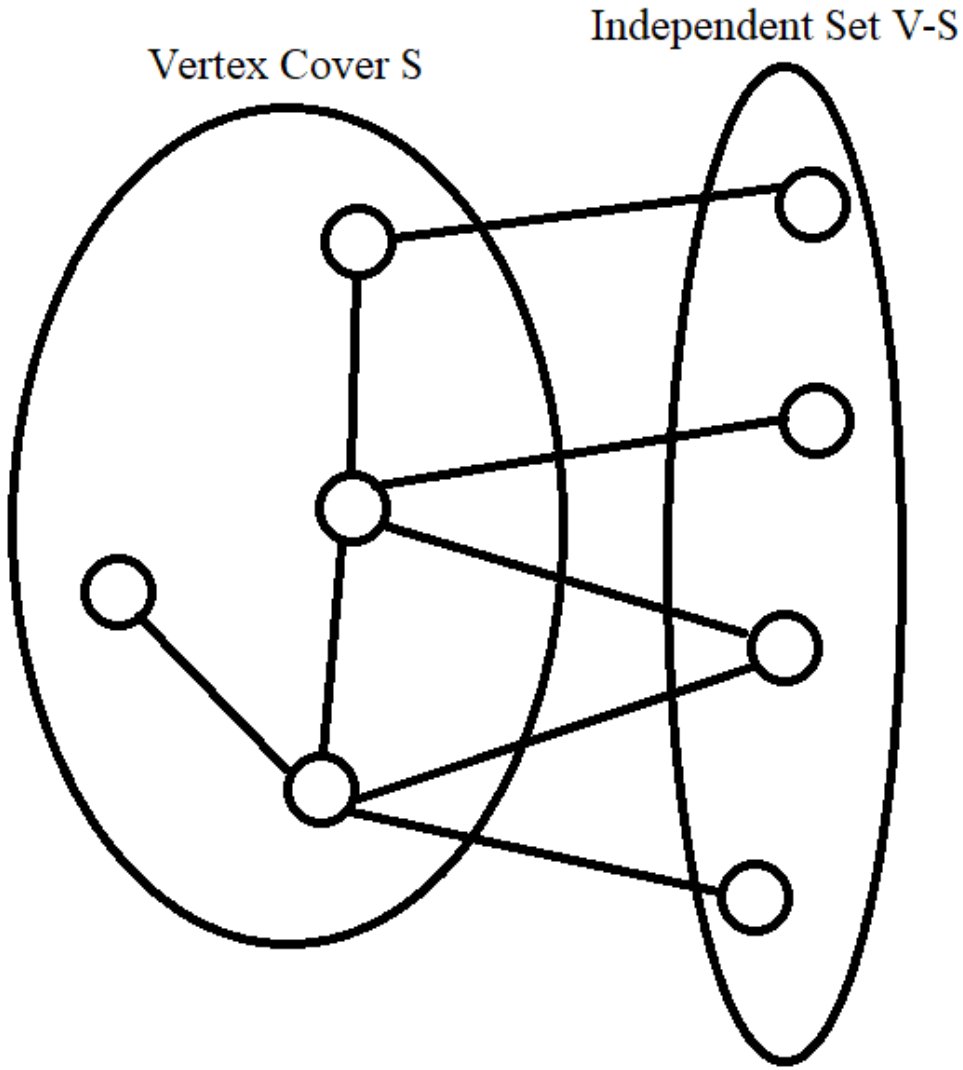
We want an independent set of size g, so simply look for a vertex cover of size $|V| - g$:

- If such a vertex cover exists, then the independent set are all the vertices not in the vertex cover.
- If such a vertex cover does not exist, then the graph cannot have an independent set of size g.

One final check: did the conversion take polynomial time?

Independent Set to Vertex Cover

Note: if there was an edge between vertices in the set $V - S$ on the right, then the set S on the left **could not have been a proper Vertex Cover**!



Independent Set to Vertex Cover

To be more precise, note the statements for the Independent Set and Vertex Cover **decision problems**:

- Independent Set: given a graph G and a goal g, find an independent set of size g, or report that this is not possible.
- Vertex Cover: given a graph H and a size k, find a vertex cover of size k, or report that this is not possible.

So, our reduction is:

- We are initially given the inputs for the Independent Set problem: a graph G and a goal g.
- We define: $H = G$ and $k = |V| - g$.
- These are now the inputs for the Vertex Cover problem.
- Assume that the Vertex Cover problem gives us an answer.
  - If the Vertex Cover problem does return a vertex cover of size $k = |V| - g$ in graph H, then the remaining g vertices are an independent set of size g in graph G. Return these g vertices as the independent set.
  - If the Vertex Cover problem returns that there is no vertex cover of size $k = |V| - g$ in graph H, then there was no independent set of size g in graph G. Return false.

Independent Set to Clique

For this reduction, we will need to consider the complement of a graph G:

- The compliment of a graph G is denoted $\bar{G}$ and is the graph with the same verices, but whose edges are exactly those that are not in graph G.

We make an important observation:

- A set of nodes S is an independent set in G if and only if they are a clique in $\bar{G}$.

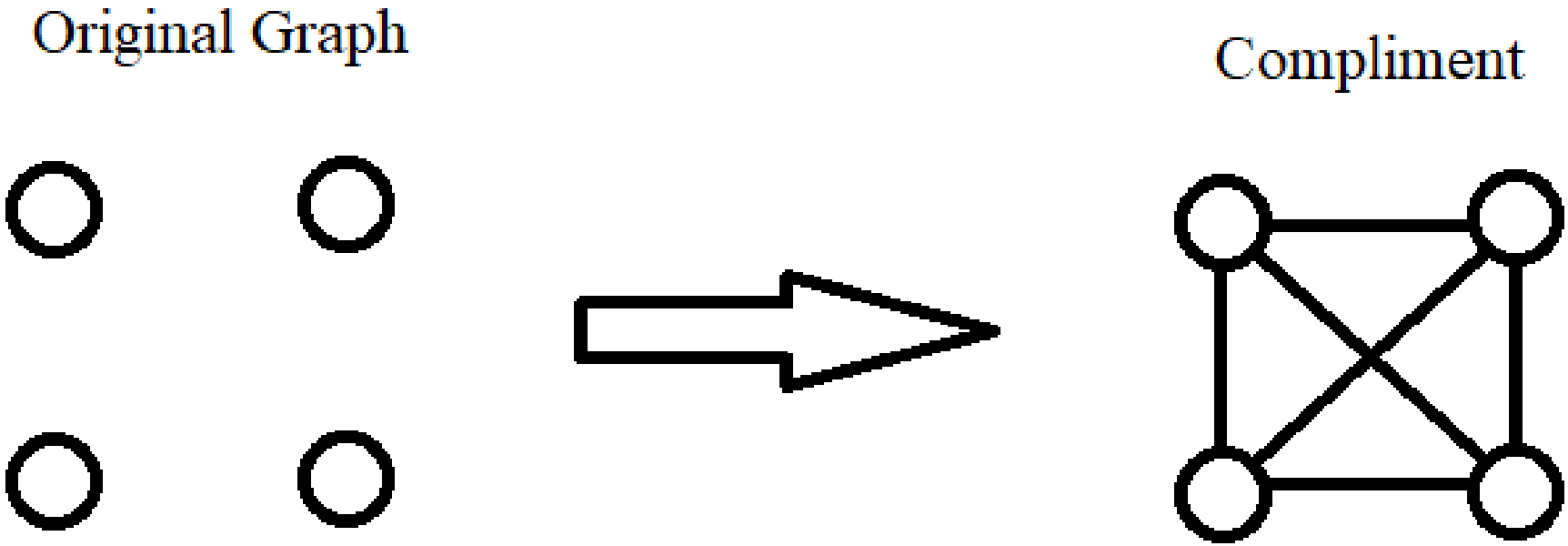
Therefore, given a graph G and a goal g for the size of the independent set:

- Create the compliment graph $\bar{G}$.
- Solve the Clique problem using graph $\bar{G}$ with a goal of g for the clique size.
- The resulting solution will also be a solution for the Independent Set problem.

One final check: did the conversion take polynomial time?

Independent Set to Clique

A set of nodes S is an independent set in G if and only if they are a clique in $\bar{G}$.



Note that all 4 independent set vertices in the original graph have the missing shared edges, that are then all present in the compliment graph, forming a clique of size 4.

Independent Set to Clique

To be more precise, note the statements for the Independent Set and Clique **decision problems**:

- Independent Set: given a graph G and a goal g, find an independent set of size g, or report that this is not possible.
- Clique: given a graph H and a size k, find a clique of size k, or report that this is not possible.

So, our reduction is:

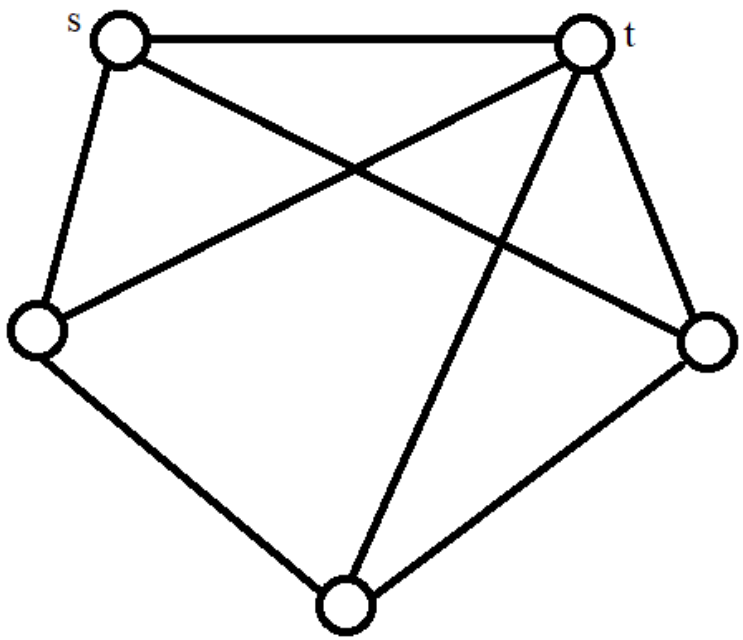
- We are initially given the inputs for the Independent Set problem: a graph G and a goal g.
- We define: $H = \bar{G}$ and $k = g$.
- These are now the inputs for the Clique problem.
- Assume that the Clique problem gives us an answer.
  - If the Clique problem does return a clique of size $k = g$ in graph H, then this is an independent set of size g in graph G. Return these g vertices as the independent set.
  - If the Clique problem returns that there is no clique of size $k = g$ in graph H, then there was no independent set of size g in graph G. Return false.

Hamiltonian Path to Hamiltonian Cycle

Here we start with an instance of the Hamiltonian Path problem:

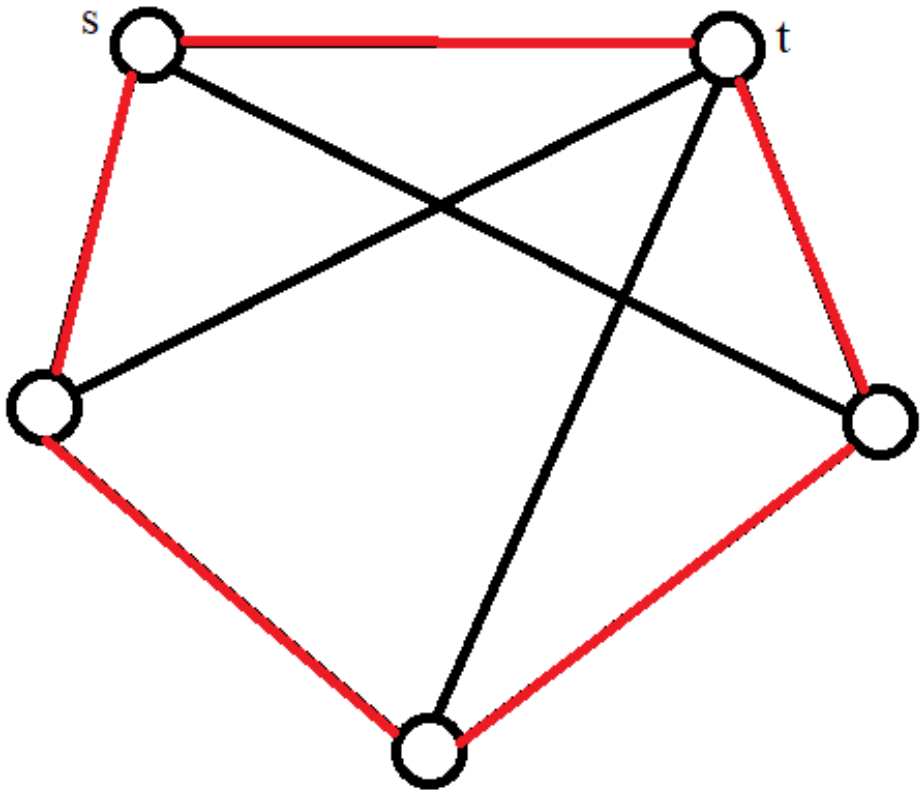
- Given a graph, find a path that visits each vertex only once, or report that such a path is not possible.

Specifically, we will assume that we have some start vertex s and some destination vertex t, and ask whether a Hamiltonian Path connecting s to t is possible.



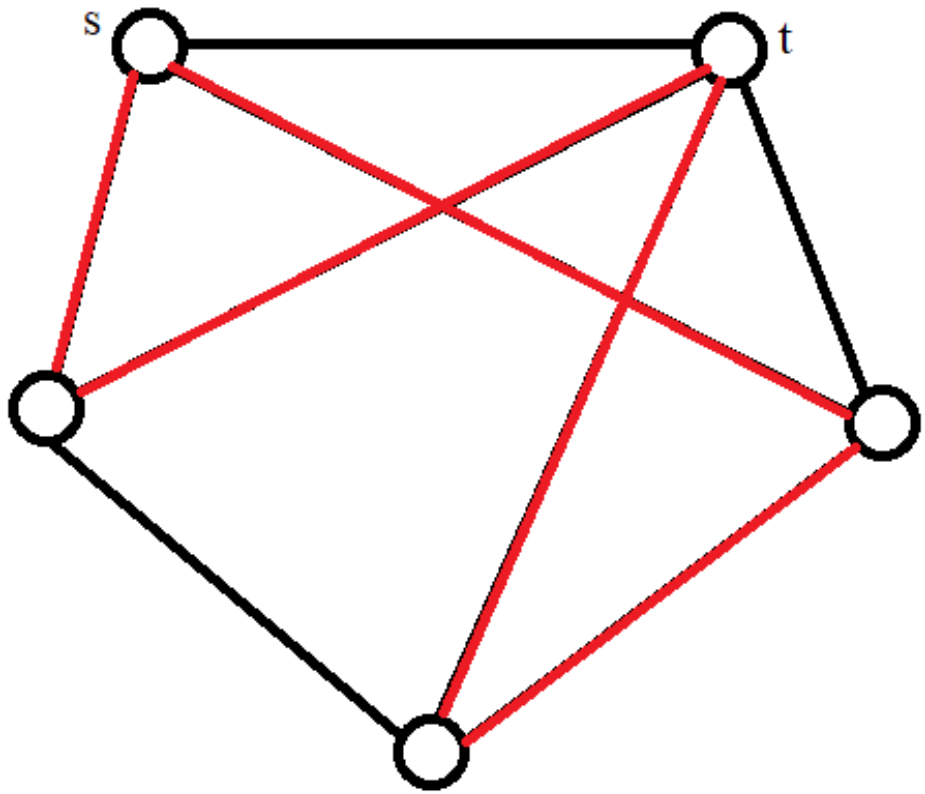
Hamiltonian Path to Hamiltonian Cycle

Note: if we have a Hamiltonian Cycle in our graph that uses the edge between s and t, then removing this edge would give us a Hamiltonian Path!



Hamiltonian Path to Hamiltonian Cycle

However, it is possible to have a cycle that doesn't use the edge between s and t. There is no single edge we can remove to obtain our Hamiltonian Path from this cycle:



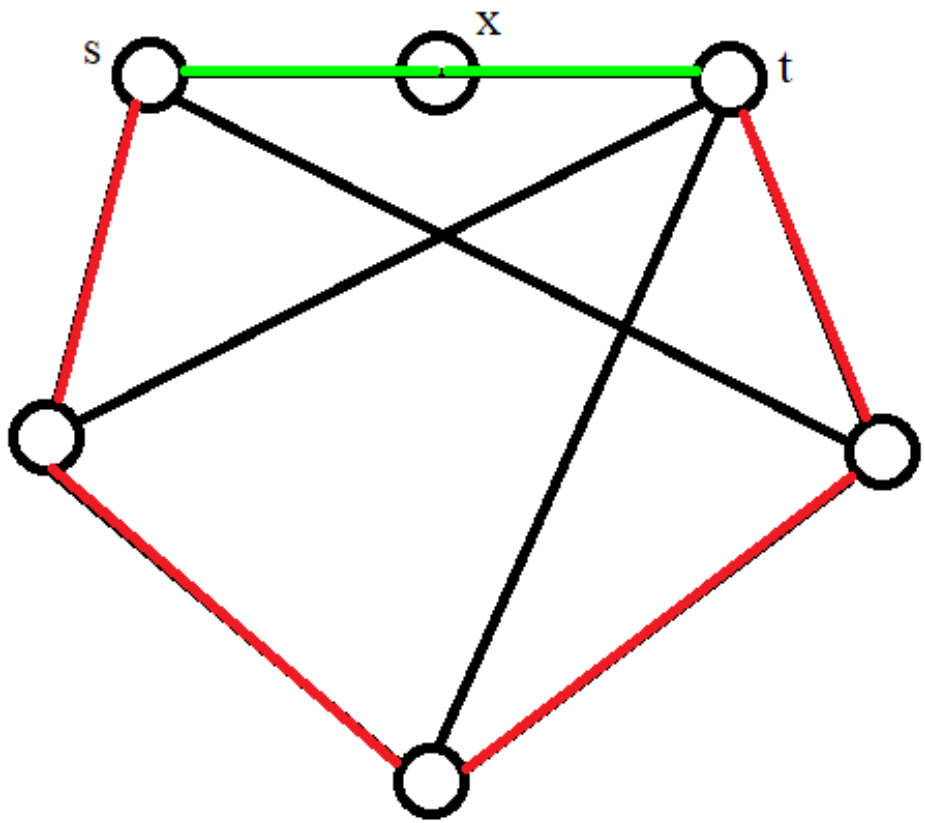
Hamiltonian Path to Hamiltonian Cycle

We can convert this problem into a Hamiltonian Cycle problem by adding a single vertex x to the graph that is connected only to vertices s and t.

- If this new graph has a Hamiltonian Cycle, then clearly the edges from s to x and from x to t must have been used consecutively.
- And all other vertices must have been visited.
- So removing x from the cycle results in a Hamiltonian Path between s and t.

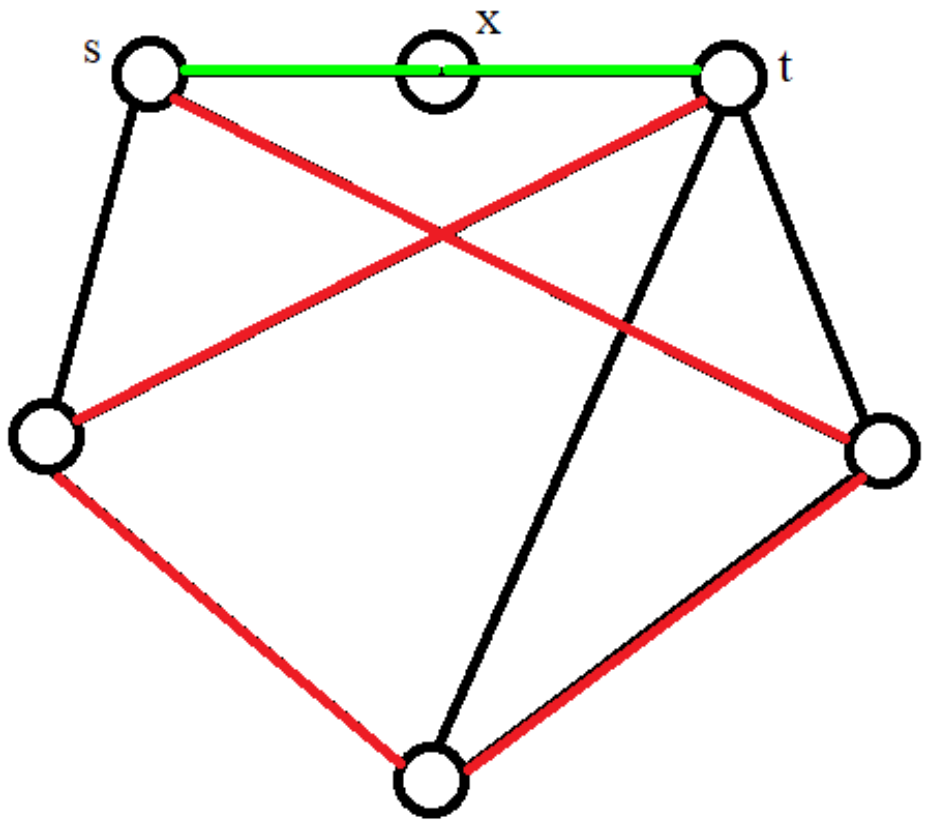
Hamiltonian Path to Hamiltonian Cycle

Note: we are forcing the Hamiltonian Cycle to use an 'edge' between s and t by forcing it to go through the dummy vertex x. This allows us to remove vertex x and obtain a Hamiltonian Path from s to t.



Hamiltonian Path to Hamiltonian Cycle

There may be multiple possible Hamiltonian Cycles, but since they all have to go through vertex x, we can always reconstruct a Hamiltonian Path from s to t by removing vertex x.



Hamiltonian Path to Hamiltonian Cycle

We can convert this problem into a Hamiltonian Cycle problem by adding a single vertex x to the graph that is connected only to vertices s and t.

- What if there is no Hamiltonian Cycle in the new graph?
- This means that there was no Hamiltonian Path in the original graph.

Note that the following statements are equivalent (contrapositives):

- If the new graph has no Hamiltonian Cycle, then the original graph had no Hamiltonian Path.
- If the original graph has a Hamiltonian Path, then the new graph has a Hamiltonian Cycle.

But the second statement is clearly true, because we could just add the edges (s, x) and (x, t) to the path to get the cycle.

Hamiltonian Path to Hamiltonian Cycle

So our reduction is:

- Given a graph G and vertices s and t for the Hamiltonian Path problem.
- Create a new graph G' that has an extra vertex x and edges (s, x) and (s, t).
- Solve the Hamiltonian Cycle problem on this new graph G'.
- If there is a cycle, then the Hamiltonian Path comes from removing vertex x from the cycle.
- If there is no cycle, then there is no path.
- As a final check: it takes only polynomial time to convert from G to G' and to convert the cycle to the path.

Note: we can also reduce Hamiltonian Cycle to Hamiltonian Path, i.e., the two are equivalent.

3SAT to Independent Set

3SAT is a special version of the SAT problem where each clause has only 3 literals in it:

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$

This problem is the canonical NP complete problem, and has been studied for many years.

We would like to reduce this problem to the Independent Set problem.

Note: this would prove that the Independent Set problem is also NP Complete.

- Because if the Independent Set problem can be solved efficiently, we could use this reduction to solve 3SAT efficiently.

3SAT to Independent Set

3SAT is a special version of the SAT problem where each clause has only 3 literals in it:

$$(\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee y \vee z) \wedge (\neg x \vee \neg y)$$

To reduce this problem to an Independent Set problem, we need to create some sort of graph.

Idea: we will create gadgets where each literal becomes a vertex while each clause becomes a triangle.

Why a triangle?

The three vertices are connected, which forces us to choose only one vertex for each clause in the independent set problem.

More next time.