CS 374B Review MT3

Turing Complete Spookiness

ACM @ UIUC

November 3, 2024

Disclaimers and Logistics

- **Disclaimer:** Some of us are CAs, but we have not seen the exam. We have no idea what the questions are. However, we've taken the course and reviewed Kani's previous exams, so we have **suspicions** as to what the questions will be like.
- This review session is being recorded. Recordings and slides will be distributed on EdStem after the end.
- **Agenda:** We'll quickly review all topics likely to be covered, then go through a practice exam, then review individual topics by request.
  - Questions are designed to be written in the same style as Kani's previous exams but to be *slightly* harder, so don't worry if you don't get everything right away!
- Please let us know if we're going too fast/slow, not speaking loud enough/speaking too loud, etc.
- If you have a question anytime during the review session, please ask! Someone else almost surely has a similar question.
- We'll provide a feedback form at the end of the session.

P and NP

- A **decision problem** is a problem with a true/false answer. (yes/no, etc.)
- **P** is the set of decision problems with a polynomial-time solver.
- **NP** is the set of decision problems with a polynomial-time *nondeterministic* solver.
- Alternatively, NP is the set of decision problems with a polynomial-time *certifier* for "true" answers, given a polynomial-size *certificate*.
  - Intuitively, with an NP problem, we can verify a "yes" answer quickly if we have the solution in front of us.

P and NP

- A **decision problem** is a problem with a true/false answer. (yes/no, etc.)
- **P** is the set of decision problems with a polynomial-time solver.
- **NP** is the set of decision problems with a polynomial-time *nondeterministic* solver.
- Alternatively, NP is the set of decision problems with a polynomial-time *certifier* for "true" answers, given a polynomial-size *certificate*.
  - Intuitively, with an NP problem, we can verify a "yes" answer quickly if we have the solution in front of us.

For example, consider the yes/no problem of deciding whether a graph $G = (V, E)$ has a path containing all its vertices. (Hamiltonian Path)

- If you were given the path already ($O(V)$ length) as a certificate, you could certify that the answer is "yes" in polynomial time.
- Therefore, this problem is in NP.

P and NP

- A **decision problem** is a problem with a true/false answer. (yes/no, etc.)
- **P** is the set of decision problems with a polynomial-time solver.
- **NP** is the set of decision problems with a polynomial-time *nondeterministic* solver.
- Alternatively, NP is the set of decision problems with a polynomial-time *certifier* for "true" answers, given a polynomial-size *certificate*.
  - Intuitively, with an NP problem, we can verify a "yes" answer quickly if we have the solution in front of us.

For example, consider the yes/no problem of deciding whether a graph $G = (V, E)$ has a path containing all its vertices. (Hamiltonian Path)

- If you were given the path already ($O(V)$ length) as a certificate, you could certify that the answer is "yes" in polynomial time.
- Therefore, this problem is in NP.

Formally, an algorithm C is a certifier for problem X when $s \in X$ if and only if there exists string t such that $C(s, t) =$ true.

- t here is a "certificate."
- We can show X is NP by providing this information, and showing C is polynomial-time and t is polynomial-size (with respect to the size of the input s).

co-NP

- **co-NP** is the set of decision problems X whose complements $\overline{X}$ are in NP.
- Alternatively, NP is the set of decision problems with a polynomial-time certifier for **"false"** answers, given a polynomial-size certificate.
- For example, the problem of deciding whether a graph *doesn't* have a Hamiltonian path is in co-NP.

co-NP isn't on your skillset, but be aware that this is *not* the same thing as NP.

Reductions I: Intuition

- **Intuition:** Problem B is “at least as hard” than problem A if we can use a black-box problem B solver (B oracle) to solve problem A with limited overhead (generally, polynomial-time).

Reductions I: Intuition

- **Intuition:** Problem *B* is "at least as hard" than problem *A* if we can use a black-box problem *B* solver (*B* oracle) to solve problem *A* with limited overhead (generally, polynomial-time).
- We know a variety of "hard" problems, so if we want to show that a problem *B* is hard, we need to show that oracles can be used to quickly solve some hard problem *A* (even if we believe that that oracle doesn't exist!). This is building a **reduction from** problem *A* **to** problem *B*

Reductions I: Intuition

- **Intuition:** Problem *B* is "at least as hard" than problem *A* if we can use a black-box problem *B* solver (*B* oracle) to solve problem *A* with limited overhead (generally, polynomial-time).
- We know a variety of "hard" problems, so if we want to show that a problem *B* is hard, we need to show that oracles can be used to quickly solve some hard problem *A* (even if we believe that that oracle doesn't exist!). This is building a **reduction from** problem *A* **to** problem *B*
- A problem is **NP-hard** if the existence of a polynomial-time algorithm for that problem would imply the existence of a polynomial-time algorithm for any problem in NP. We'll prove that problems are NP-hard by providing **polynomial-time** reductions from a known NP-hard problem to the problem in question.
  - **NP-complete** problems are NP and NP-hard.

Reductions I: Intuition

- **Intuition:** Problem *B* is "at least as hard" than problem *A* if we can use a black-box problem *B* solver (*B* oracle) to solve problem *A* with limited overhead (generally, polynomial-time).
- We know a variety of "hard" problems, so if we want to show that a problem *B* is hard, we need to show that oracles can be used to quickly solve some hard problem *A* (even if we believe that that oracle doesn't exist!). This is building a **reduction from** problem *A* **to** problem *B*
- A problem is **NP-hard** if the existence of a polynomial-time algorithm for that problem would imply the existence of a polynomial-time algorithm for any problem in NP. We'll prove that problems are NP-hard by providing **polynomial-time** reductions from a known NP-hard problem to the problem in question.
  - **NP-complete** problems are NP and NP-hard.
- A problem is **undecidable** if *no* algorithm exists that always completes in the right answer. We'll prove that problems are undecidable by providing reductions from a known undecidable problem to the problem in question.

Reductions I: Intuition

- **Intuition:** Problem B is "at least as hard" than problem A if we can use a black-box problem B solver (B oracle) to solve problem A with limited overhead (generally, polynomial-time).
- We know a variety of "hard" problems, so if we want to show that a problem B is hard, we need to show that oracles can be used to quickly solve some hard problem A (even if we believe that that oracle doesn't exist!). This is building a **reduction from** problem A **to** problem B
- A problem is **NP-hard** if the existence of a polynomial-time algorithm for that problem would imply the existence of a polynomial-time algorithm for any problem in NP. We'll prove that problems are NP-hard by providing **polynomial-time** reductions from a known NP-hard problem to the problem in question.
  - **NP-complete** problems are NP and NP-hard.
- A problem is **undecidable** if *no* algorithm exists that always completes in the right answer. We'll prove that problems are undecidable by providing reductions from a known undecidable problem to the problem in question.

Make sure you're going in the right direction!

If you're trying to prove that a problem is NP-hard or undecidable, you need to reduce **from** an NP-hard/undecidable problem **to** the problem you want to prove is hard (in other words, show that an oracle for your problem can be used to solve an NP-hard/undecidable problem). The most common mistake on exams is reducing in the wrong direction.

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time
  2. Provide an algorithm for the problem that you're reducing to using the problem that you're reducing from.

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
    1. Consider an oracle for the problem that you're reducing to. If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time
    2. Provide an algorithm for the problem that you're reducing to using the problem that you're reducing from.
    3. Analyze the runtime for your algorithm, and show it is within your target.

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time
  2. Provide an algorithm for the problem that you're reducing to using the problem that you're reducing from.
  3. Analyze the runtime for your algorithm, and show it is within your target.
  4. Provide a proof of correctness.

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time
  2. Provide an algorithm for the problem that you're reducing to using the problem that you're reducing from.
  3. Analyze the runtime for your algorithm, and show it is within your target.
  4. Provide a proof of correctness.
- We're mostly going to be talking about **decision variants** of problems (where you only need to return YES/NO) since the main complexity classes are defined with respect to them, and since, usually, decision variants are equally hard as their calculation equivalents

Reductions II: Tutorial

- To show a problem is NP-hard/undecidable you need to do the following:
  1. Consider an oracle for the problem that you're reducing to. If you're showing that something is NP-hard, you should assume that the oracle is polynomial-time
  2. Provide an algorithm for the problem that you're reducing to using the problem that you're reducing from.
  3. Analyze the runtime for your algorithm, and show it is within your target.
  4. Provide a proof of correctness.
- We're mostly going to be talking about **decision variants** of problems (where you only need to return YES/NO) since the main complexity classes are defined with respect to them, and since, usually, decision variants are equally hard as their calculation equivalents

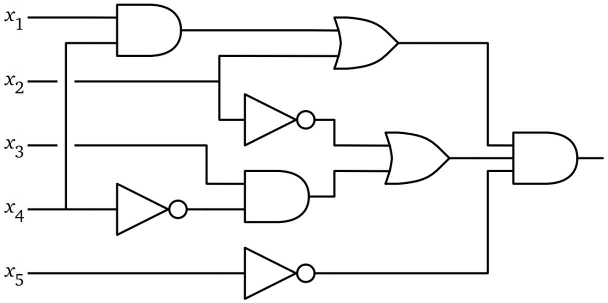
Template- Reduction

Assume that there exists an oracle function *B* which runs in [TIME CONSTRAINT]. Thus, we can solve *A* as follows:

```
procedure A(input):
    Do some preprocessing to create instances of problem B
    outputs ← B(generated inputs)
    Do some postprocessing on outputs to get the correct answer for A
```

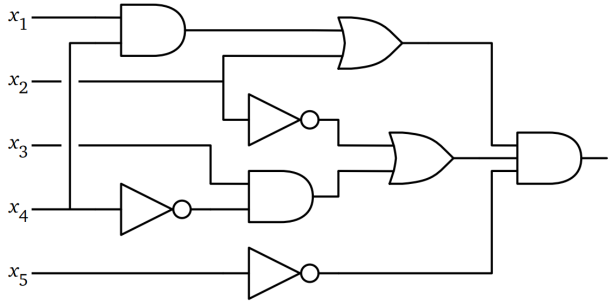
A Tour of NP-Hard Problems: CircuitSAT and 3SAT

- **CircuitSAT**: The “original” NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



A Tour of NP-Hard Problems: CircuitSAT and 3SAT

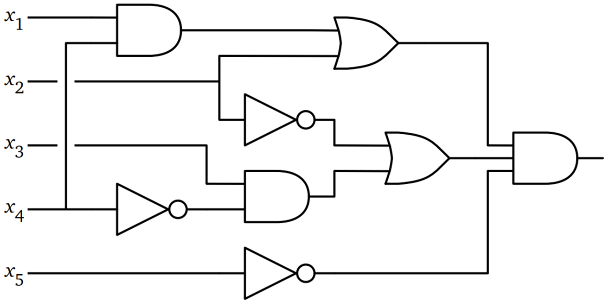
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

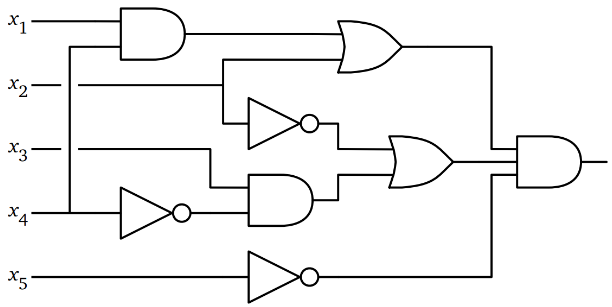
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?
- Consider reducing from 3SAT if. . . :

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

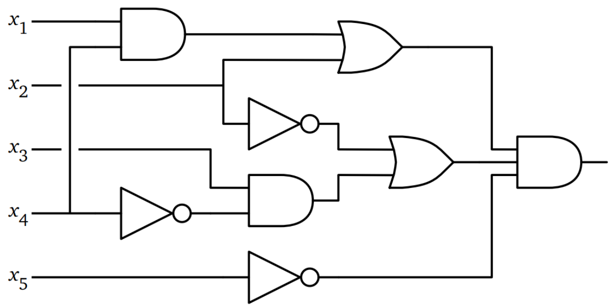
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?
- Consider reducing from 3SAT if. . . :
  - There's some structure of choice within the problem (i.e. the goal is to decide either A or B)

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

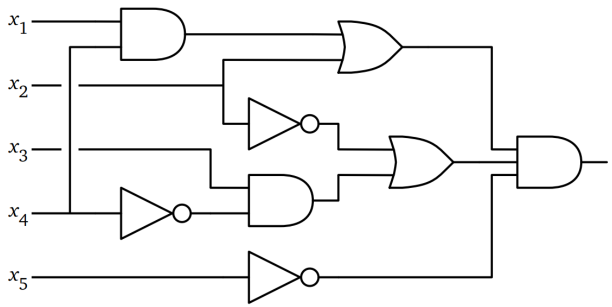
- **CircuitSAT**: The "original" NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \vee b \vee c) \wedge (\overline{a} \vee d \vee e) \wedge \cdots$, is there an assignment to the input variables that makes it return `true`?
- Consider reducing from 3SAT if. . . :
  - There's some structure of choice within the problem (i.e. the goal is to decide either A or B)
  - There's a 3 in the problem, and you don't know why

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

- **CircuitSAT**: The “original” NP-complete problem. Given a boolean circuit, is there a set of inputs that makes it return `true`?



- **3SAT**: Given a boolean formula of the form $(a \lor b \lor c) \land (\overline{a} \lor d \lor e) \land \cdots$, is there an assignment to the input variables that makes it return `true`?
- Consider reducing from 3SAT if. . . :
  - There’s some structure of choice within the problem (i.e. the goal is to decide either A or B)
  - There’s a 3 in the problem, and you don’t know why

Be careful with *k*-SAT variants!

While k-SAT for $k \geq 3$ is NP-complete, there is a polynomial-time algorithm for 2SAT. (Using strongly connected components!)

A Tour of NP-Hard Problems: CircuitSAT and 3SAT

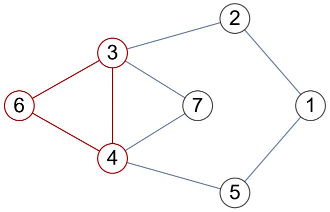
Consider the problem **MajSAT**: Clauses now consist of 5 literals, and you must satisfy at least 3 literals in each clause. Is **MajSAT** in NP, NP-hard, both, or neither? Prove why by either stating an algorithm or providing a reduction.

A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

- **MaxClique**: Given a graph G and positive integer h, can we find a K_h subgraph in G (i.e. a set of h nodes where each one has an edge to every other)?

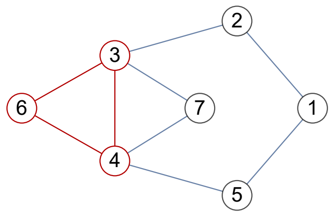
A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

- **MaxClique**: Given a graph G and positive integer h, can we find a K_h subgraph in G (i.e. a set of h nodes where each one has an edge to every other)?



A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

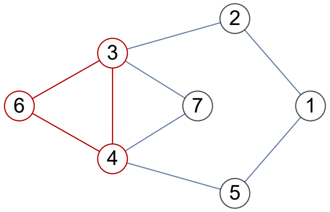
- **MaxClique**: Given a graph G and positive integer h, can we find a K_h subgraph in G (i.e. a set of h nodes where each one has an edge to every other)?



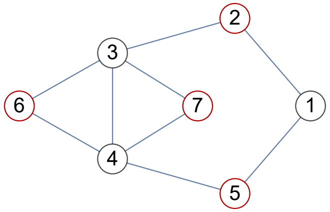
- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?

A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

- **MaxClique**: Given a graph G and positive integer h, can we find a K_h subgraph in G (i.e. a set of h nodes where each one has an edge to every other)?

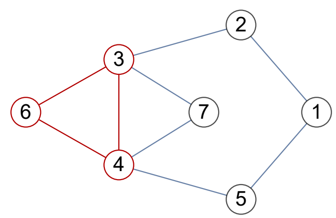


- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?

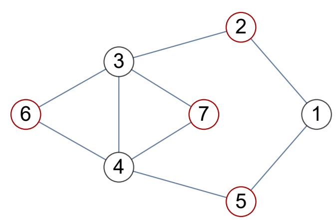


A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

- **MaxClique**: Given a graph G and positive integer h, can we find a K_h subgraph in G (i.e. a set of h nodes where each one has an edge to every other)?



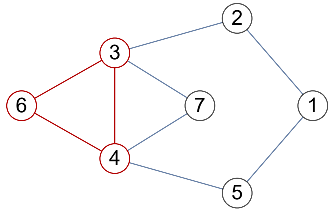
- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?



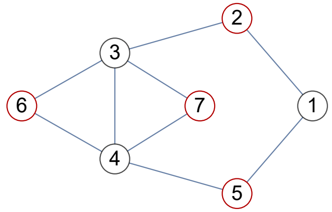
- **MinVertexCover**: Given a graph G and positive integer h, can we find a set of h nodes so that all edges have at least one endpoint chosen?

A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

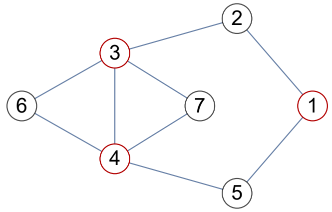
- **MaxClique**: Given a graph G and positive integer h, can we find a K_h subgraph in G (i.e. a set of h nodes where each one has an edge to every other)?



- **MaxIndSet**: Given a graph G and positive integer h, can we find a set of h nodes, none of which share an edge?



- **MinVertexCover**: Given a graph G and positive integer h, can we find a set of h nodes so that all edges have at least one endpoint chosen?

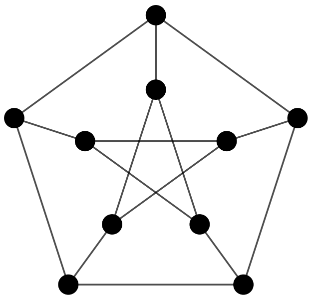


A Tour of NP-Hard Problems: Max{Clique, IndSet}, MinVertexCover

ACM is writing their review session for CS/ECE 374B MT3. While making slides, each CA writes 2 problems, either alone or in collaboration with other CAs. Since all of the CAs all have inflated egos, they won't show up to the review session unless one of the problems that they worked on is in the review session. Show that determining whether we can run a review session with at most k problems is NP-complete.

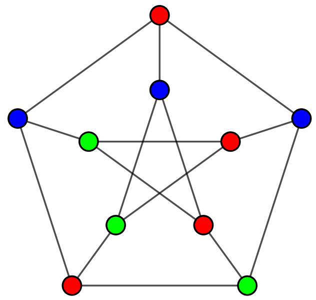
A Tour of NP-Hard Problems: Graph Coloring

- Given an (undirected) graph, can we color the nodes with at most *k* colors so that no two vertices that share an edge are of the same color?



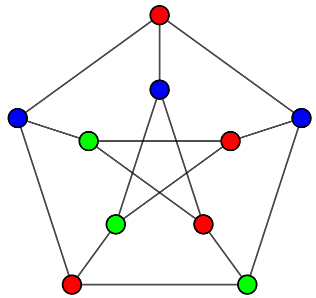
A Tour of NP-Hard Problems: Graph Coloring

- Given an (undirected) graph, can we color the nodes with at most *k* colors so that no two vertices that share an edge are of the same color?



A Tour of NP-Hard Problems: Graph Coloring

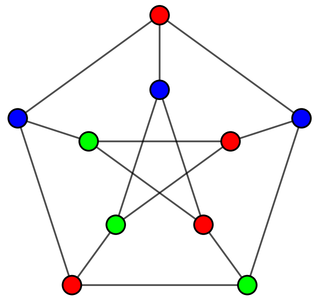
- Given an (undirected) graph, can we color the nodes with at most *k* colors so that no two vertices that share an edge are of the same color?



- Consider reducing from *k*-coloring if. . . :

A Tour of NP-Hard Problems: Graph Coloring

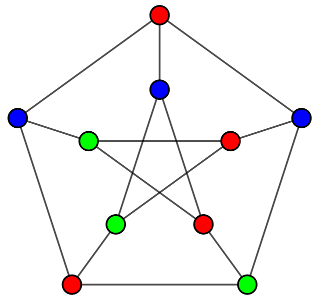
- Given an (undirected) graph, can we color the nodes with at most *k* colors so that no two vertices that share an edge are of the same color?



- Consider reducing from *k*-coloring if. . . :
    - You need to assign objects to groups, and assigning one object to a group limits your choices for some local set of others

A Tour of NP-Hard Problems: Graph Coloring

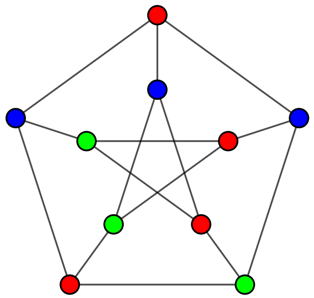
- Given an (undirected) graph, can we color the nodes with at most *k* colors so that no two vertices that share an edge are of the same color?



- Consider reducing from *k*-coloring if. . . :
  - You need to assign objects to groups, and assigning one object to a group limits your choices for some local set of others
  - There's a graph where you need to solve for some *vertex* properties

A Tour of NP-Hard Problems: Graph Coloring

- Given an (undirected) graph, can we color the nodes with at most k colors so that no two vertices that share an edge are of the same color?



- Consider reducing from k-coloring if. . . :
  - You need to assign objects to groups, and assigning one object to a group limits your choices for some local set of others
  - There's a graph where you need to solve for some *vertex* properties

Be careful with *k*-coloring variants!

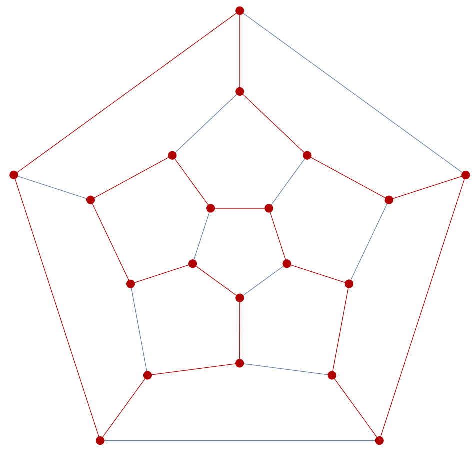
While k-coloring for $k \geq 3$ is NP-complete, you can find whether a graph is bipartite (2-colorable) using DFS.

A Tour of NP-Hard Problems: Graph Coloring

Consider the problem **Safe7Color**, which asks you to color a graph with 7 colors, such that it is a violation if there is an edge $u \leftrightarrow v$ where $c(u)$ and $c(v)$ differ by 0 or 1 (mod 7). Is this problem NP-hard?

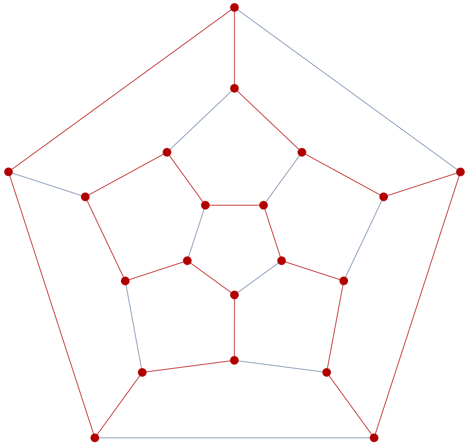
A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian cycle has a Hamiltonian path.



A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

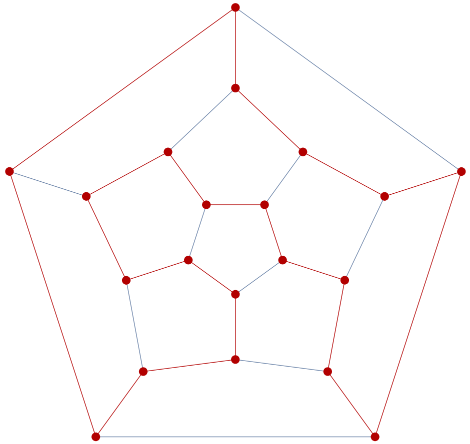
- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian cycle has a Hamiltonian path.



- Consider reducing from **HamPath** or **HamCycle** if. . .

A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

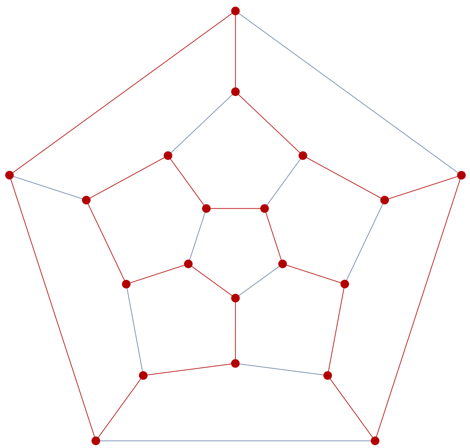
- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
  - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian cycle has a Hamiltonian path.



- Consider reducing from **HamPath** or **HamCycle** if. . .
  - You're given a graph, and you're asked to find a sequence of vertices

A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

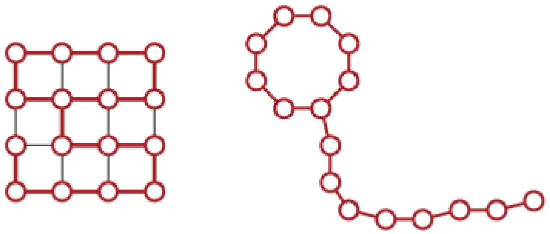
- A **Hamilton Path** is a path that goes through each vertex *exactly* once. Likewise, a **Hamiltonian Cycle** is a cycle that goes through each node *exactly* once.
    - Every graph with a Hamiltonian cycle has a Hamiltonian path, but not every graph with a Hamiltonian cycle has a Hamiltonian path.



- Consider reducing from **HamPath** or **HamCycle** if. . .
    - You're given a graph, and you're asked to find a sequence of vertices
    - You have a resource pool, and you want to use up everuthing

A Tour of NP-Hard Problems: Hamiltonian Paths and Cycles

A **balloon graph** of size ℓ is a cycle of length ℓ attached to a path of length ℓ, where the cycle and the path are disjoint except for the connecting vertex. Show that it is NP-hard to determine whether a graph has a balloon subgraph of size at least k.



A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?
- **IntegerLinearProgramming**: given a linear objective function to optimize, as well as linear constraints, what is the largest objective achievable where (all/some) variables are restricted to integers?

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?
- **IntegerLinearProgramming**: given a linear objective function to optimize, as well as linear constraints, what is the largest objective achievable where (all/some) variables are restricted to integers?
- **TravelingSalesman**: given a weighted graph G, what is there a Hamiltonian path in G of length at most k?

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?
- **IntegerLinearProgramming**: given a linear objective function to optimize, as well as linear constraints, what is the largest objective achievable where (all/some) variables are restricted to integers?
- **TravelingSalesman**: given a weighted graph G, what is there a Hamiltonian path in G of length at most k?
- **SubsetSum**: given a list of integers, is there a subset that sums to exactly k?

A Tour of NP-hard Problems: Others

- These likely won't come up on exams, but they're useful to know.
- **LongestPath**: given a (directed, weighted) graph G, is there a path of length at least k?
- **IntegerLinearProgramming**: given a linear objective function to optimize, as well as linear constraints, what is the largest objective achievable where (all/some) variables are restricted to integers?
- **TravelingSalesman**: given a weighted graph G, what is there a Hamiltonian path in G of length at most k?
- **SubsetSum**: given a list of integers, is there a subset that sums to exactly k?
- **Checkers**: given a $n \times n$ checkerboard, is there a move that captures at least k checkers?

Undecidability

- A language is **decidable** if there exists an algorithm which always returns `true` to all inputs in L and `false` to inputs not in L
  - If we can only return `true` to all inputs in L and either return `false` *or* infinite-loop for all other inputs, the language is merely **acceptable**.

Theorem (Turing, 1936)

The language `Halt`*:* $\{(f, w) :$ *the function f does not infinite loop on input w*$\}$ *is undecidable.*

- 3 main ways to prove that a problem is undecidable:

Undecidability

- A language is **decidable** if there exists an algorithm which always returns `true` to all inputs in L and `false` to inputs not in L
  - If we can only return `true` to all inputs in L and either return `false` *or* infinite-loop for all other inputs, the language is merely **acceptable**.

Theorem (Turing, 1936)

The language `Halt`*:* $\{(f, w) : \textit{ the function } f \textit{ does not infinite loop on input } w\}$ *is undecidable.*

- 3 main ways to prove that a problem is undecidable:
  1. Reduce from Halt: Given an oracle for your problem, design an algorithm to decide Halt. No runtime requirement!

Undecidability

- A language is **decidable** if there exists an algorithm which always returns `true` to all inputs in L and `false` to inputs not in L
  - If we can only return `true` to all inputs in L and either return `false` *or* infinite-loop for all other inputs, the language is merely **acceptable**.

Theorem (Turing, 1936)

The language `Halt`*:* $\{(f, w) :$ *the function f does not infinite loop on input w*$\}$ *is undecidable.*

- 3 main ways to prove that a problem is undecidable:
  1. Reduce from Halt: Given an oracle for your problem, design an algorithm to decide Halt. No runtime requirement!
  2. Rice's Theorem: Very powerful, basically claims that any non-trivial question about functions/Turing machines is undecidable:

Theorem (Rice)

Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:

- *There is a Turing machine Y such that Accept(Y) $\in \mathcal{L}$.*
- *There is a Turing machine N such that Accept(N) $\notin \mathcal{L}$.*

Then, the language $AcceptIn(\mathcal{L}) \leftarrow \{\langle M \rangle \mid Accept(M) \in \mathcal{L}\}$ *is undecidable.*

Undecidability

- A language is **decidable** if there exists an algorithm which always returns `true` to all inputs in L and `false` to inputs not in L
  - If we can only return `true` to all inputs in L and either return `false` *or* infinite-loop for all other inputs, the language is merely **acceptable**.

Theorem (Turing, 1936)

The language `Halt`*:* $\{(f, w) : \text{ the function } f \text{ does not infinite loop on input } w\}$ *is undecidable.*

- 3 main ways to prove that a problem is undecidable:
  1. Reduce from Halt: Given an oracle for your problem, design an algorithm to decide Halt. No runtime requirement!
  2. Rice's Theorem: Very powerful, basically claims that any non-trivial question about functions/Turing machines is undecidable:

Theorem (Rice)

Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:

- *There is a Turing machine Y such that Accept(Y) $\in \mathcal{L}$.*
- *There is a Turing machine N such that Accept(N) $\notin \mathcal{L}$.*

Then, the language $AcceptIn(\mathcal{L}) \leftarrow \{\langle M \rangle \mid Accept(M) \in \mathcal{L}\}$ *is undecidable.*

  3. Abuse the fact that you can put code into a function to derive a contradiction.

Undecidability

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M$'s accept set is regular$\}$

Undecidability

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M$'s accept set is regular$\}$

(b) HaltsQuadratically $= \{\langle M \rangle, r : M$ halts on r in at most $|r|^2$ arithmetic operations$\}$

Undecidability

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M\rangle : M\text{'s accept set is regular}\}$

(b) HaltsQuadratically $= \{\langle M\rangle, r : M \text{ halts on } r \text{ in at most } |r|^2 \text{ arithmetic operations}\}$

(c) AcceptsRejects $= \{\langle M\rangle : M\text{'s accept set} = M\text{'s reject set}\}$

Undecidability

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M$'s accept set is regular$\}$

(b) HaltsQuadratically $= \{\langle M \rangle, r : M$ halts on r in at most $|r|^2$ arithmetic operations$\}$

(c) AcceptsRejects $= \{\langle M \rangle : M$'s accept set $= M$'s reject set$\}$

(d) CFLAccepts374 $= \{c \in CFGs : c$ accepts exactly 374 strings$\}$

Undecidability

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M$'s accept set is regular$\}$
(b) HaltsQuadratically $= \{\langle M \rangle, r : M$ halts on r in at most $|r|^2$ arithmetic operations$\}$
(c) AcceptsRejects $= \{\langle M \rangle : M$'s accept set $= M$'s reject set$\}$
(d) CFLAccepts374 $= \{c \in CFGs : c$ accepts exactly 374 strings$\}$
(e) LeftThrice $= \{\langle M, w \rangle : M$ moves left on input w three times in a row$\}$

Undecidability

For each of the following languages, either show that they are decidable by describing an algorithm that decides them, or show that they are undecidable by reduction and by Rice's theorem when possible.

(a) AcceptsRegular $= \{\langle M \rangle : M\text{'s accept set is regular}\}$

(b) HaltsQuadratically $= \{\langle M \rangle, r : M \text{ halts on } r \text{ in at most } |r|^2 \text{ arithmetic operations}\}$

(c) AcceptsRejects $= \{\langle M \rangle : M\text{'s accept set} = M\text{'s reject set}\}$

(d) CFLAccepts374 $= \{c \in \mathit{CFGs} : c \text{ accepts exactly 374 strings}\}$

(e) LeftThrice $= \{\langle M, w \rangle : M \text{ moves left on input } w \text{ three times in a row}\}$

(f) NeverLeft $= \{\langle M, w \rangle : M \text{ never moves left on input } w\}$

Feedback

- Please fill out the feedback form:
  `go.acm.illinois.edu/cs374b_mt3_feedback`