Analysis of Algorithms, I
CSOR W4231

Eleni Drinea
Computer Science Department

Columbia University

Reductions; independent set and vertex cover; *decision* problems

Outline

1. Reductions
   - A means to design efficient algorithms
   - Arguing about the relative hardness of problems
2. Two *hard* graph problems
3. Complexity classes
   - The class $\mathcal{P}$
   - The class $\mathcal{NP}$
   - The class of $\mathcal{NP}$-complete problems

Today

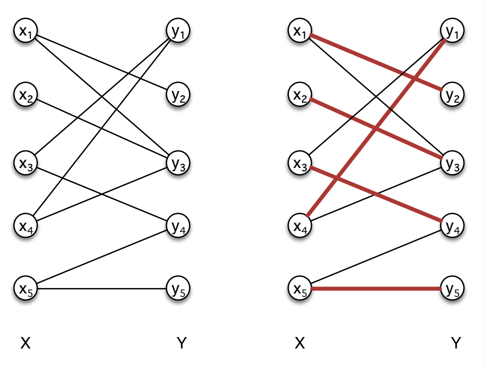
1. Reductions
   - A means to design efficient algorithms
   - Arguing about the relative hardness of problems
2. Two *hard* graph problems
3. Complexity classes
   - The class $\mathcal{P}$
   - The class $\mathcal{NP}$
   - The class of $\mathcal{NP}$-complete problems

Beyond problems that admit efficient algorithms

- Efficient algorithms: worst-case polynomial running time
- So far we solved a variety of problems efficiently
- Today we will look at problems for which we do *not* know of any *efficient* algorithms
- To argue about the relative *hardness* (difficulty) of such problems, we will use the notion of reductions

BPM: does a Bipartite graph have a Perfect Matching?

Input: a bipartite graph $G = (X \cup Y, E)$
Output: **yes**, if and only if G has a matching of size $|X| = |Y|$

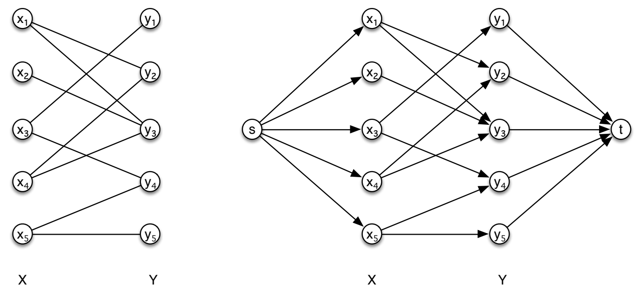


Some terminology

- An instance of BPM is a specific input graph G.
- We may think of an input instance G as a binary string x encoding the graph G, with length $|x|$ bits.
  - we assume reasonable encodings: e.g., a binary number b requires a logarithmic number of bits to be encoded
  - reasonable encodings are related polynomially
- An algorithm that solves BPM admits
  - **Input**: a binary string x encoding a bipartite graph
  - **Output**: **yes**, if and only if x has a perfect matching

We’ve already designed the algorithm that solves BPM

1. Given the bipartite graph G (input to BPM), we constructed a flow network G' (input to max flow).



That is, we transformed the input instance x of BPM into an input instance y of Max Flow.

We've already designed the algorithm that solves BPM

2. We proved that the original bipartite graph G has a max matching of size k if and only if the derived flow network G' has a max flow of value k.

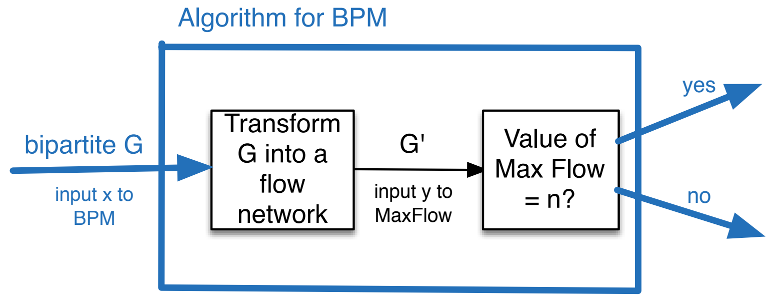
3. We computed max flow in G';

   we answer **yes** to BPM on input x

   if and only if

   we answer **yes** to *Is max flow* $=n$*?* on input y.

A diagram of the algorithm for BPM



Algorithm for BPM

bipartite G

input x to BPM

Transform G into a flow network

G'

input y to MaxFlow

Value of Max Flow = n?

yes

no

Remark 1.

Let x be a binary encoding of G. Transforming G into G' requires only polynomial time in $|x|$.

Reduction

Let X, Y be two computational problems.

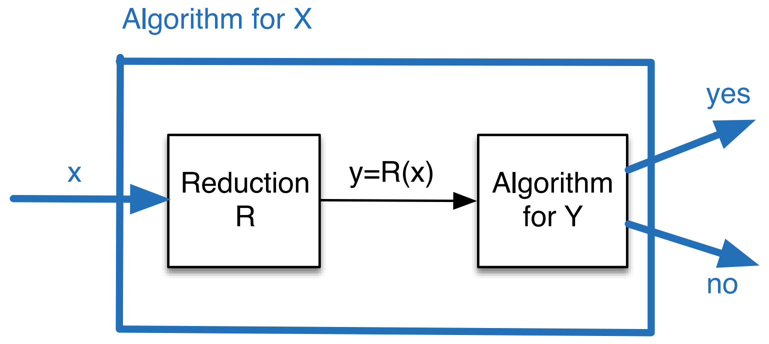
Definition 1.

A reduction is a transformation that for every input x of X produces an equivalent input $y = R(x)$ of Y.

- By equivalent we mean that the answer to $y = R(x)$ considered as an input for Y is a correct **yes/no** answer to x, considered as an input for X.
- We will require that the reduction is completed in a polynomial in $|x|$ number of computational steps.

Diagram of a reduction

X, Y are computational problems.
x, y are inputs to X, Y respectively.



Reductions as a means to design efficient algorithms

- Suppose we had a **black box** for solving Y.
- If arbitrary instances of X can be solved using
  - a polynomial number of standard computational steps; plus
  - a polynomial number of calls to the black box for Y,

  we say that X reduces polynomially to Y; in symbols

$$X \leq_P Y.$$

Fact 2.

Suppose $X \leq_P Y$. If Y is solvable in polynomial time, then X is solvable in polynomial time.

Few observations about reductions

Remark 2.

To solve BPM we made exactly one call to the black box for Max Flow. This will be typical of all our reductions today.

Remark 3.

Reductions between problems provide a powerful technique for designing efficient algorithms.

Reductions as a means to argue about *hard* problems

- Suppose that $X \leq_P Y$.
- Then Y is at least as hard (difficult) as X: given an algorithm to solve Y, we can solve X.

Fact 3 (the contrapositive of Fact 2).

Suppose $X \leq_P Y$. If X cannot be solved in polynomial time, then Y cannot be solved in polynomial time.

- Fact 3 provides a way to conclude that a problem Y does not have an efficient algorithm.
- For the problems we will be discussing today, we have *no proof* that they do not have efficient algorithms.

⇒ So we will be using Fact 3 to establish relative levels of difficulty among these problems.

Today

1. Reductions
   - A means to design efficient algorithms
   - Arguing about the relative hardness of problems
2. Two *hard* graph problems
3. Complexity classes
   - The class $\mathcal{P}$
   - The class $\mathcal{NP}$
   - The class of $\mathcal{NP}$-complete problems

Independent Set

Definition 4 (Independent Set).

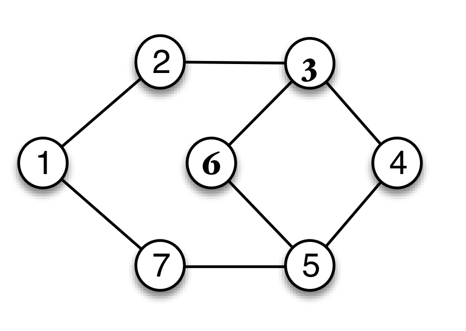
An independent set in $G = (V, E)$ is a subset $S \subseteq V$ of nodes such that there is no edge between any pair of nodes in the set. That is, for all $u, v \in S$, $(u, v) \notin E$.

Independent Set

Definition 4 (Independent Set).

An independent set in $G = (V, E)$ is a subset $S \subseteq V$ of nodes such that there is no edge between any pair of nodes in the set. That is, for all $u, v \in S$, $(u, v) \notin E$.

It is easy to find a small independent set.
What about a large one?

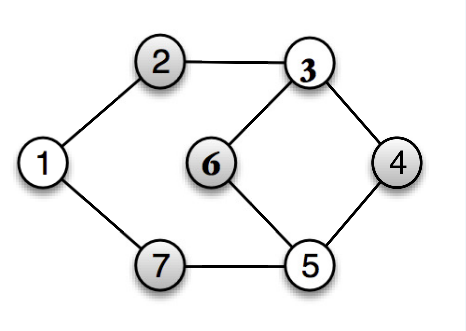


Independent Set

Definition 4 (Independent Set).

An independent set in $G = (V, E)$ is a subset $S \subseteq V$ of nodes such that there is no edge between any pair of nodes in the set. That is, for all $u, v \in S$, $(u, v) \notin E$.

It is easy to find a small independent set.
What about a large one?



Vertex Cover

Definition 5 (Vertex Cover).

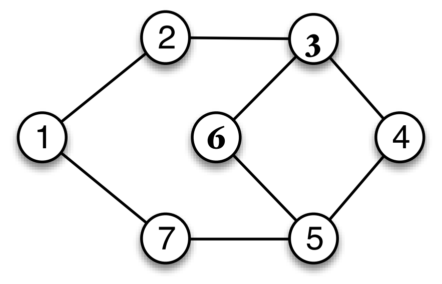
A vertex cover in $G = (V, E)$ is a subset $S \subseteq V$ of nodes such that every edge $e \in E$ has at least one endpoint in S.

Vertex Cover

Definition 5 (Vertex Cover).

A vertex cover in $G = (V, E)$ is a subset $S \subseteq V$ of nodes such that every edge $e \in E$ has at least one endpoint in S.

It is easy to find a large vertex cover.
What about a small one?

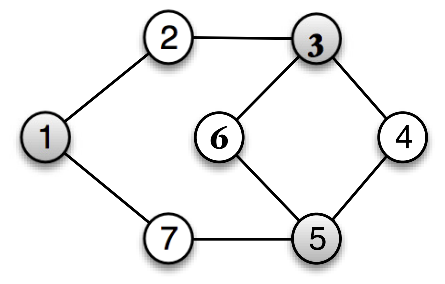


Vertex Cover

Definition 5 (Vertex Cover).

A vertex cover in $G = (V, E)$ is a subset $S \subseteq V$ of nodes such that every edge $e \in E$ has at least one endpoint in S.

It is easy to find a large vertex cover.
What about a small one?



Optimization versions for IS and VC

Definition 6 (`Maximum Independent Set problem`).

Given G, find an independent set of maximum size.

Definition 7 (`Minimum Vertex Cover problem`).

Given G, find a vertex cover of minimum size.

Optimization versions for IS and VC

Definition 6 (`Maximum Independent Set problem`).

Given G, find an independent set of maximum size.

Definition 7 (`Minimum Vertex Cover problem`).

Given G, find a vertex cover of minimum size.

Brute force approach requires exponential time: there are 2^n candidate subsets since every vertex may or may not belong to such a subset.

Decision versions of optimization problems

An optimization problem may be transformed into a roughly equivalent problem with a **yes/no** answer, called the decision version of the optimization problem, by

1. supplying a target value for the quantity to be optimized;
2. asking whether this value can be attained.

We will denote the decision version of a problem `X` by `X(D)`.

Examples of decision versions of optimization problems

- `Max Flow(D)`: Given a flow network G and an integer k (the **target** flow), does G have a flow of value at least k?
- `Max Bipartite Matching(D)`: Given a bipartite graph G and an integer k, does G have a matching of size at least k?
- `IS(D)`: Given a graph G and an integer k, does G have an independent set of size at least k?
- `VC(D)`: Given a graph G and an integer k, does G have a vertex cover of size at most k?

So in a maximization problem the target value is a **lower** bound, while in a minimization problem it is an **upper** bound.

Rough equivalence of decision & optimization problems

1. Suppose we have an algorithm that solves `Maximum Independent Set`. *Algorithm for* `IS(D)`*?*

2. Now suppose we have an algorithm that solves the decision version `IS(D)`, that is, on input $(G = (V, E), k)$, it answers **yes** if G has an independent set of size at least k, and **no** otherwise. *Algorithm for* `Maximum Independent Set`*?*

Rough equivalence of decision & optimization problems

1. Suppose we have an algorithm that solves `Maximum Independent Set`.
   - To solve `IS(D)` on input (G, k), compute the size m of the max independent set; answer **yes** if $m \geq k$, **no** otherwise.
2. Now suppose we have an algorithm that solves `IS(D)`, that is, on input $(G = (V, E), k)$, it answers **yes** if G has an independent set of size at least k, and **no** otherwise.

   Algorithm for `Maximum Independent Set`*?*

Rough equivalence of decision & optimization problems

1. Suppose we have an algorithm that solves `Maximum Independent Set`.
    - To solve `IS(D)` on input (G, k), compute the size m of the max independent set; answer **yes** if $k \leq m$, **no** otherwise.
2. Now suppose we have an algorithm that solves `IS(D)`, that is, on input $(G = (V, E), k)$, it answers **yes** if G has an independent set of size at least k, and **no** otherwise.
    - To find the size m of the maximum independent set, use binary search and at most $\log n$ calls to `IS(D)`.
    - Note that this algorithm indeed runs in time polynomial in the size of the description of the input (G, k).

This rough equivalence between optimization problems and decision problems holds for all the problems we will discuss.

Reduction from Independent Set to Vertex Cover

Fact 8.

Let $G = (V, E)$ *be a graph. Then* S *is an independent set of* G *if and only if* $V - S$ *is a vertex cover of* G*.*

Claim 1.

`IS(D)` $\leq_P$ `VC(D)` *and* `VC(D)` $\leq_P$ `IS(D)`.

Reduction from Independent Set to Vertex Cover

Fact 8.

Let $G = (V, E)$ be a graph. Then S is an independent set of G if and only if $V - S$ is a vertex cover of G.

Claim 1.

`IS(D)` $\leq_P$ `VC(D)` *and* `VC(D)` $\leq_P$ `IS(D)`.

Proof.

- Given an instance $x = (G, k)$ of `IS(D)`, transform it to an instance $y = (G, n - k)$ of `VC(D)`. This completes the reduction.
- Equivalence of x and y follows from Fact 8: `IS(D)` answers **yes** on x if and only if `VC(D)` answers **yes** on y.
- Hence `IS(D)` $\leq_P$ `VC(D)`.

The second part of the claim is entirely similar. □

Proof of Fact 8

Proof.

- **Forward direction:** we will show that $V - S$ is a vertex cover of G.
  If S is an independent set of G, then for all $u, v \in S$, $(u, v) \notin E$. So consider any edge $(u, v) \in E$.
  - Either both $u, v \in V - S$, hence (u, v) is covered by $V - S$.
  - Or, $u \in S$ and $v \in V - S$ (w.l.o.g.); so $V - S$ covers (u, v).
- **Reverse direction:** we will show that $V - S$ is an independent set of G.
  If S is a vertex cover of G, then for every edge $(u, v) \in E$, at least one of u, v is in S. So consider any two nodes $x, y \in V - S$: no edge (x, y) can exist since S would not cover this edge. Hence $V - S$ is an independent set.

☐

Today

1. Reductions
    - A means to design efficient algorithms
    - Arguing about the relative hardness of problems
2. Two *hard* graph problems
3. Complexity classes
    - The class $\mathcal{P}$
    - The class $\mathcal{NP}$
    - The class of $\mathcal{NP}$-complete problems

The class $\mathcal{P}$

Notation: $x \in \texttt{X(D)} \iff x$ is a **yes** instance of `X(D)`.

Example: (triangle, 1) is a **yes** instance of `IS(D)` but (triangle, 2) is a **no** instance of `IS(D)`.

- An algorithm A **solves** (or **decides**) `X(D)` if, for all input x, $A(x) =$ **yes** if and only if $x \in \texttt{X(D)}$.
- A has a polynomial running time, if there is a polynomial $p(\cdot)$ such that for all input strings x of length $|x|$, the worst-case running time of A on input x is $O(p(|x|))$.

Definition 9.

We define $\mathcal{P}$ to be the set of decision problems that can be solved by polynomial-time algorithms.

Problems like `IS(D)` and `VC(D)`

- No polynomial time algorithm has been found despite significant effort
- So we don't believe they are in $\mathcal{P}$

Is there anything positive we can say about such problems?

Problems like `IS(D)` and `VC(D)`

- No polynomial time algorithm has been found despite significant effort
- So we don't believe they are in $\mathcal{P}$

Is there anything positive we can say about such problems?

If we were given a solution for such a problem, we can certify if the instance is a **yes** instance quickly.

Problems like `IS(D)` and `VC(D)`

- No polynomial time algorithm has been found despite significant effort
- So we don't believe they are in $\mathcal{P}$

Is there anything positive we can say about such problems?

For example, given

1. an instance $x = (G, k)$ for `IS(D)`; and
2. a candidate solution S

we can verify quickly that `IS(D)` indeed answers **yes** on x by checking

1. $|S| = k$; and
2. there is no edge between any pair of nodes in S.

Problems like `IS(D)` and `VC(D)`

- No polynomial time algorithm has been found despite significant effort
- So we don't believe they are in $\mathcal{P}$

Is there anything positive we can say about such problems?

- If we were given a solution S for such a problem `X(D)`, we could check if it is correct quickly.

⇒ Such an S is a succinct certificate that $x \in$ `X(D)`.

Note that `VC(D)` also possesses a similar succinct (short) certificate *(why?)*.

The class $\mathcal{NP}$

Definition 10.

An efficient certifier (or *verification algorithm*) B for a problem `X(D)` is a **polynomial-time** algorithm that

1. takes **two** input arguments, the instance x and the *short* certificate t (both encoded as binary strings)
2. there is a polynomial $p(\cdot)$ so that for every string x, we have $x \in$ `X(D)` if and only if there is a string t such that $|t| \leq p(|x|)$ and $B(x, t) =$ **yes**.

Note that **existence** of the certifier B **does not** provide us with any efficient way to **solve** `X(D)`! (*why?*)

Definition 11.

We define $\mathcal{NP}$ to be the set of decision problems that have an efficient certifier.

$\mathcal{P}$ vs $\mathcal{NP}$

Fact 12.

$$\mathcal{P} \subseteq \mathcal{NP}$$

Proof.

Let `X(D)` be a problem in $\mathcal{P}$.

- There is an efficient algorithm $A(x)$ that solves `X(D)`, that is, $A(x)$= **yes** if and only if $x \in$`X(D)`.
- To show that `X(D)`$\in \mathcal{NP}$, we need exhibit an efficient certifier B that takes two inputs x and t and answers **yes** if and only if $x \in$`X(D)`.
- The algorithm B that on inputs x, t, simply discards t and simulates $A(x)$ is such an efficient certifier.

☐

$\mathcal{P}$ vs $\mathcal{NP}$

$$\mathcal{P} \quad = \quad \mathcal{NP} \quad ?$$

$\mathcal{P}$ vs $\mathcal{NP}$

$$\mathcal{P} \quad = \quad \mathcal{NP} \quad ?$$

- *Arguably the biggest question in theoretical CS*
- *We do not think so: finding a solution should be harder than checking one, especially for hard problems...*

Why would $\mathcal{NP}$ contain more problems than $\mathcal{P}$?

- Intuitively, the hardest problems in $\mathcal{NP}$ are the least likely to belong to $\mathcal{P}$.
- How do we identify the hardest problems?

Why would $\mathcal{NP}$ contain more problems than $\mathcal{P}$?

- Intuitively, the hardest problems in $\mathcal{NP}$ are the least likely to belong to $\mathcal{P}$.
- How do we identify the hardest problems?

The notion of reduction is useful again.

Definition 13 ($\mathcal{NP}$-complete problems:).

A problem `X(D)` is $\mathcal{NP}$-complete if

1. `X(D)`$\in\mathcal{NP}$, and
2. for all `Y`$\in\mathcal{NP}$, `Y`$\leq_P$`X(D)`.

Why would $\mathcal{NP}$ contain more problems than $\mathcal{P}$?

- Intuitively, the hardest problems in $\mathcal{NP}$ are the least likely to belong to $\mathcal{P}$.
- How do we identify the hardest problems?

The notion of reduction will be useful again.

Definition 13 ($\mathcal{NP}$-complete problems).

A problem `X(D)` is $\mathcal{NP}$-complete if

1. `X(D)`$\in \mathcal{NP}$ and
2. for all `Y`$\in \mathcal{NP}$, `Y`$\leq_P$`X(D)`.

Fact 14.

Suppose `X` *is $\mathcal{NP}$-complete. Then* `X` *is solvable in polynomial time (i.e.,* `X` $\in \mathcal{P}$*) if and only if $\mathcal{P} = \mathcal{NP}$.*

Why we should care whether a problem is $\mathcal{NP}$-complete

- If a problem is $\mathcal{NP}$-complete it is among the least likely to be in $\mathcal{P}$: it is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.

Why we should care whether a problem is $\mathcal{NP}$-complete

- If a problem is $\mathcal{NP}$-complete it is among the least likely to be in $\mathcal{P}$: it is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.
- Therefore, from an algorithmic perspective, we need to stop looking for efficient algorithms for the problem.

Why we should care whether a problem is $\mathcal{NP}$-complete

- If a problem is $\mathcal{NP}$-complete it is among the least likely to be in $\mathcal{P}$: it is in $\mathcal{P}$ if and only if $\mathcal{P} = \mathcal{NP}$.
- Therefore, from an algorithmic perspective, we need to stop looking for efficient algorithms for the problem.

Instead we have a number of options

1. approximation algorithms, that is, algorithms that return a solution within a provable guarantee from the optimal
2. exponential algorithms practical for small instances
3. work on interesting special cases
4. study the average performance of the algorithm
5. examine heuristics (algorithms that work well in practice, yet provide no theoretical guarantees regarding how close the solution they find is to the optimal one)

How do we show that a problem is $\mathcal{NP}$-complete?

Suppose we had an $\mathcal{NP}$-complete problem `X`.

To show that another problem `Y` is $\mathcal{NP}$-complete, we only need show that

1. `Y` $\in \mathcal{NP}$ and
2. `X` $\leq_P$ `Y`

How do we show that a problem is $\mathcal{NP}$-complete?

Suppose we had an $\mathcal{NP}$-complete problem `X`.

To show that another problem `Y` is $\mathcal{NP}$-complete, we only need show that

1. $\texttt{Y} \in \mathcal{NP}$ and
2. $\texttt{X} \leq_P \texttt{Y}$

Why?

Fact 15 (Transitivity of reductions).

If $\texttt{X} \leq_P \texttt{Y}$ *and* $\texttt{Y} \leq_P \texttt{Z}$, *then* $\texttt{X} \leq_P \texttt{Z}$.

We know that for all $\pi \in \mathcal{NP}$, $\pi \leq_P \texttt{X}$. By Fact 15, $\pi \leq_P \texttt{Y}$. Hence `Y` is $\mathcal{NP}$-complete.

How do we show that a problem is $\mathcal{NP}$-complete?

Suppose we had an $\mathcal{NP}$-complete problem `X`.

To show that another problem `Y` is $\mathcal{NP}$-complete, we only need show that

1. `Y` $\in \mathcal{NP}$ and
2. `X` $\leq_P$ `Y`

So, *if* we had a first $\mathcal{NP}$-complete problem `X`, discovering a new problem `Y` in this class would require an *easier* kind of reduction: just reduce `X` to `Y` (instead of reducing **every** problem in $\mathcal{NP}$ to `Y`!).

How do we show that a problem is $\mathcal{NP}$-complete?

Suppose we had an $\mathcal{NP}$-complete problem `X`.

To show that another problem `Y` is $\mathcal{NP}$-complete, we only need show that

1. `Y` $\in \mathcal{NP}$ and
2. `X` $\leq_P$ `Y`

The first $\mathcal{NP}$-complete problem

Theorem 15 (Cook-Levin).

`Circuit SAT` *is* $\mathcal{NP}$*-complete.*