CSC373: Algorithm Design, Analysis and Complexity
Winter/Spring 2020

Allan Borodin and Sara Rahmati

Week of February 10-14, 2020

Announcements

- Accessability services is looking for someone to share notes taken during the lecture. Please see Feb 2 announcement on web page.
- A student has given me an instance of interval coloring where first fit coloring using the EFT ordering takes a little more than twice the number of colors in an optimal coloring.
  This raises the question as to whether or not there is any constant c such that $\frac{EFT}{OPT} \leq c$. The answer is ...

Announcements

- Accessability services is looking for someone to share notes taken during the lecture. Please see Feb 2 announcement on web page.
- A student has given me an instance of interval coloring where first fit coloring using the EFT ordering takes a little more than twice the number of colors in an optimal coloring.
  This raises the question as to whether or not there is any constant c such that $\frac{EFT}{OPT} \leq c$. The answer is ...
  Yes. In fact, there is a constant c ($c = 26$ will work) such that no matter in what order the intervals arrive, first fit coloring will use at most $c \cdot OPT$. Not sure what is the best known constant c and what is the best constant for the EFT ordering. Algorithms where we have no control over the order of input arrivals (and we are required to make decision for each input item before the next arrival) are called *online algorithms*.

More announcements and this weeks agenda

- Skecthes of Solutions for Assignment 1 have been posted.
- The next assignment will be posted soon.
- This week will be mainly devoted to the start of complexity theory.
- We will first start with a quick review of applications of flow networks.

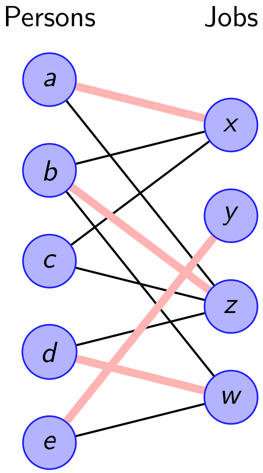
Some ways to achieve polynomial time

- Choose an augmenting path having shortest distance: This is the Edmonds-Karp method and can be found in CLRS. It has running time $O(nm^2)$, where $n = |V|$ and $m = |E|$.
- There is a "weakly polynomial time" algorithm in KT
  - Here the number of arithmetic operations depends on the length of the integral capacities.
  - It follows that always choosing the largest capacity augmenting path is at least weakly polynomial time.
- There is a history of max flow algorithms leading to a recent $O(mn)$ time algorithm (see http://tinyurl.com/bczkdfz).
- Although not the fastest, Dinitz's algorithm has runtime $O(n^2m)$.
  - A shortest augmenting-path method based on the concept of a blocking flow in the leveled graph.
  - Has an additional advantage (i.e. an improved $O(m\sqrt{n})$ bipartite matching time bound) beyond what follows from the better time of Edmonds-Karp max flow.

An application of max-flow: the maximum bipartite matching problem

The maximum bipartite matching problem

- Given a bipartite graph $G = (V, E)$ where
  - $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \varnothing$
  - $E \subseteq V_1 \times V_2$
- **Goal:** Find a maximum size matching.



- We do not know any efficient DP or greedy optimal algorithm for solving this problem.
- But we can efficiently reduce this problem to the max-flow problem.

The reduction

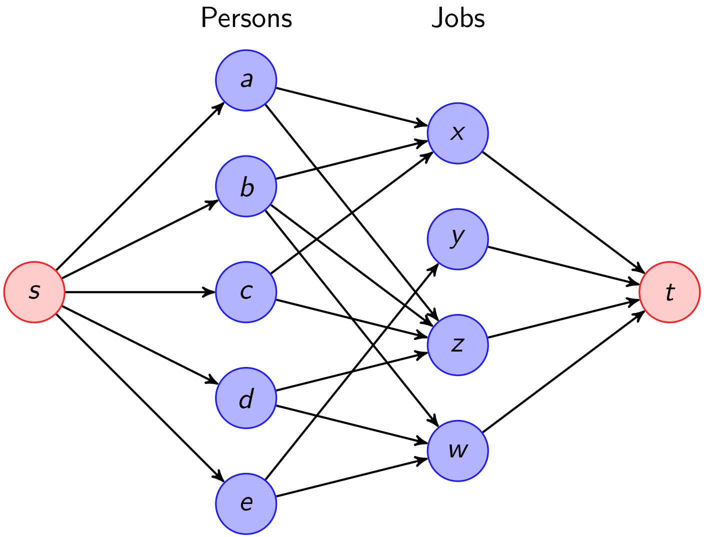


Figure: Assign every edge of the network flow a capacity 1.

The reduction preserves solutions

Claims

1. Every matching M in G gives rise to an integral flow f_M in the newly constructed flow network F_G with $val(f_M) = |M|$
2. Conversely every integral flow f in F_G gives rise to a matching M_f in G with $|M_f| = val(f)$.

Let $m = |E|, n = |V|$

- Time complexity: $O(mn)$ using any Ford Fulkerson algorithm since the max flow is at most n and $C = n$ since all edge capacities are integral and set to 1.
- Dinitz's algorithm can be used to obtain a runtime $O(m\sqrt{n})$.

A few more comments on this reduction

- When we get to our next big topic (NP completeness), we will be focusing on decision problems and as a decision problem we have $|M| \geq k$ iff $val(f_M) \geq k$.
- The reduction we are using is very efficient (linear time in the representation of the graph) and it is a special type of polynomial time reduction which we will call a polynomial time transformation.

Alternating and augmenting paths in graphs

There is a graph theoretic concept of an augmenting path relative to a matching (in an arbitrary graph).

- An alternating path π relative to a matching M is one whose edges alternate between edges in M and edges not in M.
- An augmenting path is an alternating path that starts and ends with an edge not in M.

- The reduction provides a 1-1 correspondence between augmenting paths in the bipartite G wrt. M_f and augmenting paths in G_{f_M}.

Can this reduction be extended to a maximum edge weighted matching

- In the weighted bipartite matching bipartite matching problem, we are given an edge weighted bipartite graph $G = (V, E)$ with $V = V_1 \cup V_2$ (a disjoint partition) and with say integral weights $w : E \rightarrow \mathbb{N}$
- Goal: Compute a matching M so as to maximize $\sum_{e \in M} w(e)$.
- A "reaasonable" idea is to extend the unweighted reduction by again forming a flow network with distinguished source s and terminal t nodes.
- We could then set the capacities of the edges as follows:
  - $c(x, y) = w(x, y)$ for all $(x, y) \in E$
  - $c(s, x) = \max_y \{w(x, y) : x \in V_1\}$
  - $c(y, t) = \max_x \{w(x, y) : y \in V_2\}$
- Why doesn't this work?

Edge disjoint paths: another similar max flow application

- A problem of interest in fault tolerant networks is to ensure that there are sufficiently many edge disjoint paths between any two given nodes.
- Given a directed graph $G = (V, E)$ with distinguished source s and terminal t nodes, the **goal** is to compute the the maximum number of edge disjoint paths from s to t.
- Similar to the bipartite matching transformation, we view G as a flow network $\mathcal{F}_G$ by setting the capacity of all edges equal to 1.
- Once again, because of integrality and unit capacities, we can argue that there are k edge-disjoint paths in G iff $\mathcal{F}_G$ has max (integral) flow k.
- The max flow-min cut theorem implies Menger's theorem which states that the maximum number of edge-disjoint $s - t$ paths in a directed graph is equal to the minimum number of edges in an $s - t$ cut.
- The same theorem holds for undirected graphs.

Another application of max flow-min cut

We consider an application of min cuts. The problem we wish to consider is the following binary classification problem where we want to claasify (label) vertices by one of two labels, say a and b.

We are given an edge and vertex labelled graph $G = (V, E, p, \lambda)$ where $p : E \to \mathbb{R}^{\geq 0}$, and $\lambda : V \times \{a, b\} \to \mathbb{R}^{\geq 0}$.

Our goal is to define a labeling $\ell : V \to \{a, b\}$ so that $\ell(v_i)$ is the label given to vertex v_i.

We interpret $a_i = \lambda(v_i, a)$ (resp. $b_i = \lambda(v_i, b)$) as the benefit we gain from labeling node v_i as a (resp. the benefit by labelling v_i as b). We interpret p_{ij} as the cost of labeling v_i and v_j by different labels.

In fact, I am just now discussing section 7.10 of the text which is titled image segmentation which is the problem of classifying pixels (i.e., nodes) in a grid graph where the labels correspond to foregound and background in an image. However, when reading this section, there is nothing special about grid graphs or the interpretation of the edge and vertex labels.

The objective of the binary classification problem

We want to label the vertices and lets call A, the set of nodes labelled as a and B, the set of nodes labeled b.
The goal is to maximize the following objective:

$$q(A,B) = \sum_{v_i \in A} a_i + \sum_{v_i \in B} b_i - \sum_{(i.j):(\ell(v_i) \neq \ell(v_j))} p_{ij}$$

We want to eventually restate this problem as a minimization problem.
We first restate the problem in the following way:
Let $Q = \sum_i (a_i + b_i)$. Then the equivalent objective is to maximize

$$Q(A,B) = Q - \sum_{v_i \in A} b_i - \sum_{v_i \in B} a_i - \sum_{(i.j):\ell(v_i) \neq \ell(v_j)} p_{ij}$$

The is equivalent then to minimizing:

$$q'(A,B) = \sum_{v_i \in A} b_i + \sum_{v_i \in B} a_i + \sum_{(i.j):\ell(v_i) \neq \ell(v_j)} p_{ij}$$

Minimizing q' as a min cut problem

This is similar to the way we reduced (or transformed) maximizing the size of a matching in a bipartite graph to computing a maximum flow in a related flow network.

Namely, we want change the graph into a network flow graph. Each edge will become two directed edges and we will have new source s and target t nodes where s (resp t) will now be thought of as super node representing nodes labeled as a (resp. as a super node representing nodes labeled b).

We will place capacities between the source s and other nodes to reflect the cost of a "mislabel" and similarly for the terminal t.

The min cut will then correspond to a min cost labelling. We construct the flow network $\mathcal{F} = (G', s, t, c)$ such that

- $G' = (V', E')$
- $V' = V \cup \{s, t\}$
- $E' = \{(u, v) | u \neq v \in V\} \cup \{(s, u) | u \in V\} \cup \{(u, t) | u \in V\}$
- $c(i, j) = c(j, i) = p_{i,j}; c(s, i) = a_i; c(i, t) = b_i$

Depicting the *transformation* of the binary labeling problem to the min cut problem

Consider min cut (A, B) in G'.

- $A =$ foreground.

$$cap(A,B) \;=\; \sum_{j \in B} a_j \;+\; \sum_{i \in A} b_i \;+\; \sum_{\substack{(i,j) \in E \\ i \in A,\, j \in B}} p_{ij}$$

if i and j on different sides, p_{ij} counted exactly once

- Precisely the quantity we want to minimize.

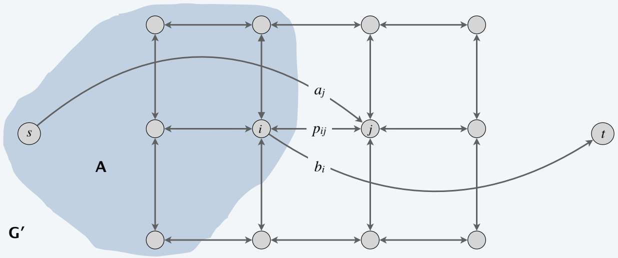


Figure: Figure taken from Kevin Wayne's slides

The more general metric labeling problem

The binary classification problem is a special case of the following more general problem called metric labeling, later considered in the section 12.6 of the text when discusasing approximation algorithms.

- The input is an edge weighted graph $G = (V, E)$, a set of labels $L = \{a_1, \ldots, a_r\}$ in a metric space with distance metric d, and functions $w : E \rightarrow \mathbb{R}^{\geq 0}$ and $\lambda : V \times L \rightarrow \mathbb{R}^{\geq 0}$.
- We want to construct a labeling $\ell : V \rightarrow L$
- $\beta(u, a_j)$ is the benefit of giving label a_j to node u, and d represents the distance between labels.
- Goal: Find a labelling $\lambda : V \rightarrow L$ of the nodes so as to maximize $\sum_u \beta(u, \lambda(u)) - \sum_{(u,v) \in E} w(u, v) \cdot d((\ell(u), \ell(v))$
- For example, the nodes might represent documents, the labels are topics, and the edges are links between documents weighted by the importance of the link.
- When there are 3 or more labels, the problem is NP-hard even for the case of the $\{0,1\}$ metric d where $d(a_i, a_j) = 1$ for $a_i \neq a_j$.

Basic aspects of complexity theory

We consider the following:

- Encoding of inputs/outputs
- Functions, search problems, and optimization problems
- Definitions of decision problems (equivalently) languages in the classes *P* and *NP*
- Polynomial time reductions and the concept of *NP* completeness.
- Polynomial time reductions vs polynomial time transformations.
- A tree of polynomial time transformations starting from 3SAT.
- Examples of polynomial time transformations.

We will also briefly discuss problems where the seqential time complexity *provably* requires more than exponential time.

Computational complexity theory

Computational complexity aims to measure the amount of resources *required* for various computational problems.

In order to makes this into a theory, we need to have precise models of computation and measures of complexity within those models.

We will later introduce the Turing machine model, a precise mathematical model, and claim that all "classical" computational models can be "efficiently" simulated by Turing machines. For our purposes, "efficiently" will mean within a polynomial factor in the context of sequential time.

For what is called "fine-grained complexity", where we care say about the difference between n^2 and n^3, we use other formalizations and appropriate definitions of "efficiently". This may be one of the topics we will consider in the last week of the course.

What are the main measures of complexity that we study?

- sequential time. For the P vs NP issue, this is the relevant measure of complexity.
- memory referred to as space in complexity theory.
- parallel time
- In randomized algorithms, we also consider the amount of randomness used.
- And, in addition, we study tradeoffs bewteen these resources. For example, time vs space.

Encoding of inputs and outputs

- We are always assuming that inputs and outputs are encoded as strings over some finite alphabet S with at least 2 symbols.
  - S^n denotes the set of all finite strings of length n over the alphabet S.
  - The empty string is the only string of length 0.
  - $S^* = \bigcup_{n \geq 0} S^n$ denotes the set of all finite strings over the alphabet S
- We can use as many symbols as we want but 2 suffices for our purpose. (**Note:** finitely many symbols on a keyboard.)
- We can also encode in unary, but that causes an exponential blowup in representation.
- We can encode a set of inputs or outputs $w_1, \ldots, w_n$ by having a special symbol (say $\#$) to separate the inputs but again this can all be encoded back into 2 symbols.
- If we need to distinguish components of an input, we can denote such an encoding of many inputs (or outputs) as $\langle w_1, \ldots, w_n \rangle$.
  - We will usually not need to be so careful about the distinction between an object G and its encoding $\langle G \rangle$.

What does it mean to be efficiently computable?

- Let n denote "size" (i.e. the encoded length) of the inputs and outputs of the problem.
- Using diagonalization, it is not difficult to show that

For any computable time bound $T(n)$, there are computable functions (in particular, decision problems) not computable within time T(n) for inputs/outputs of size n.

- Following Cobham and Edmonds (circa 1965), we will equate the intuitive concept of "efficiently computable" with computable in polynomial time (i.e. time bounded by a polynomial function of the encoded length of the inputs and outputs).
  - This has sometimes been called the Extended Church-Turing Thesis.
  - This hypothesis is not literally believed in contrast to the Church-Turing Thesis.
  - But it is an abstraction that has led to great progress in computing.
- **Informal claim restated:** Any function (polynomial time) computable is (polynomial time) computable by a Turing machine.

What are the objects of study?

- Since it suffices to encode all inputs as finite strings over the alphabet $S = \{0, 1\} = \{\mathrm{false}, \mathrm{true}\}$, we often refer to the complexity issues of such computations as Boolean complexity theory.
  - This is in contrast to, say, arithmetic complexity theory, or the theory of real valued computation.
- Our general objects of study (for Boolean complexity theory) are the computation of:
  1. Functions $f : S^* \to S^*$
  2. Decision problems
     - Let $R(x, y) \subseteq S^* \times S^*$ be a relation.
     - Given x, determine (i.e. output YES or NO) if there exists a y satisfying $R(x, y)$.
  3. Search problems
     - Let $R(x, y) \subseteq S^* \times S^*$ be a relation.
     - Given x, output a y satisfying $R(x, y)$ or say that no such y exists.
  4. Optimization problems
     - Let $R(x, y) \subseteq S^* \times S^*$ be a relation, and $c : S^* \times S^* \to \Re$ be an objective function.
     - Given x, output a y satsifying $R(x, y)$ so as to minimize (or maximize) $c(x, y)$ or say that no such y exists.

Polynomial time computable functions

- A function $f : S^* \rightarrow S^*$ is computable in polynomial time $T(\cdot)$ if:
  1. $T(n)$ is a polynomial.
  2. There is an algorithm (to be precise a Turing machine or an idealized RAM program with an appropriate instruction set) such that: For all inputs $w \in S^*$, the algorithm halts using at most $T(n)$ "time steps" where $n = |w| + |f(w)|$.
- We can define polynomial time search problems, optimization problems and decision problems similarly.
- We will generally not be dealing with functions where $|f(w)| > |w|$. So it will suffice to let n be the encoded length of the input.
- In particular, for decision problems, n will always refer to the length of the encoded input.
- We let P denote the class of decision problems that are solvable (decideable) in polynomial time. We sometimes abuse notation and also use P to denote any problem computable in polynomial time.

Computational Complexity Theory

- Classifies problems according to their computational difficulty.
- The class P (Polynomial Time) [Cobham, Edmonds, 1965]
- P consists of all problems that have an efficient (e.g. $n, n^2 \ldots$) algorithm. (n is the input length)

Examples in P

- Addition, Multiplication, Square Roots
- Shortest Path (Google Maps)
- Network flows (Internet Routing)
- Pattern matching (Spell Checking, Text Processing)
- Fast Fourier Transform (Audio and Image Processing, Oil Exploration)
- Recognizing Prime Numbers [Agrawal-Kayal-Saxena 2002]
- . . .

Polynomial time continued

- One of the nice properties of the Turing machine model is that the concept of a computational step and hence time is well defined.
- **Warning:** If say a RAM has a multiplication operation, then by repeatedly squaring we can compute 2^{2^n} in n operations.
  - We argue that we cannot assume such an operation only costs 1 time unit since the operands will have 2^n bits and hence we might be encoding an exponential time computation within such operations.
- In particular, it can be shown that if we do not account properly for the cost of RAM operations, then we can factor integers (and thus be able to break some encryption schemes such as RSA) in polynomial time but such a computational algorithm would not be considered realistic.
- One way to deal with such RAM models is to say that any operation on operands of length m requires time m.

Classical vs quantum computation

- Quantum computation takes advantage of principles of quantum mechanics (such as “entangled states”) which allow a quantum state involving n ‘qubits’ to be a “superposition” of up to 2^n possible bit strings.
- In 1994 Peter Shor proved that a quantum computer can factor numbers into primes in polynomial time.
- So if large quantum computers can be built, they could crack the RSA encryption scheme. (There are other ways that RSA could be broken (side channel attacks.)
- **The Catch:** Despite substantial effort, physicists have so far been unable to build an actual quantum computer large enough to process more than a dozen or so bits of information.
- **Caveat:** Quantum cryptography provides a promising approach to secure communication in which security (rather than complexity) depends on principles of quantum mechanics.

Note

Quantum computation does NOT change the Church-Turing thesis, that is, what is computable. But it does seem to change what is computable in polynomial time.

What have we been doing so far in this course

- So far, we have almost entirely been presenting polynomial time algorithms.

- As one exception, we did consider the pseudo polynomial time DP for the knapsack problem. Additiionally, we presented the $O(n^2 2^n)$ time DP for the travelling salesman problem improving upon the brute-force $n!$ time algorithm.

- Many times we didn't care if the operands (say in interval scheduling) were real numbers or integers.

- We just assumed that we could do basic arithmetic operations and comparisons in one step.

- This was not a problem because we didn't use algorithms that would build up large integers. (Note that using only addition, repeated doubling can only produce a number as large as 2^n in n steps).

NP-hardness

- We have often been refering to *NP*-hardness when we considered computing optimal solutions to a number of optimization problems.
- We alluded to the widely held belief that such problems cannot be computed efficiently (for all inputs).
- This will be expressed as the conjecture (sometimes called Cook's Hypothesis) that $P \neq NP$. When we say that an optimization problem is *NP*-hard, it implies that we cannot optimally solve such a problem if we assume $P \neq NP$.
- We will later consider approximate optimization algorithms.
- What is the evidence we have for believing in this conjecture?
  - Briefly stated, the main evidence is the extensive number of problems which can be shown to be "equivalent" in the sense that if any one of them can be computed efficiently (i.e. in P) then they all are.
  - These are problems that have been studied for many years (decades and in certain cases centuries) without anyone being able to find polynomial time algorithms.

What is *NP*?

- The class *NP* (Nondeterministic Polynomial Time)
- *NP* consists of all decision problems for which a solution y (to the associated search problem) can be verified in in polynomial time.
- More specifically, a set (or language) L is in the class *NP* if there is a polynomial time computable relation $R(x,y)$ and a polynomial time computable function f such that for all $x \in L$, there is a certificate y such that $|y| \leq f(|x|)$ and $R(x,y)$.

Examples in *NP* (besides everything in *P*)

- Given an integer x (in say binary of decimal representation), is it a composite number? (This is in fact a polynomial time computable decision problem.)
- Given a graph G, can it be vertex colored in 3 colors?
- Given a set $S = \{a_i\}$ of integers can it partitioned into two subsets S_1 and S_2 such that $\sum_{a_i \in S_1} a_i = \sum_{a_i \in S_2} a_i$?

P versus NP

- P: Problems for which solutions/certificates can be efficiently found
- NP: Problems for which solutions/certificateds can be efficiently verified

Conjecture

$$P \neq NP$$

- Most computer scientists believe this conjecture.
- But is seems to be incredibly hard to prove.

Why is proving $P \neq NP$ difficult?

- One reason is that some search problems in NP turn out to be relatively easy. An example is the maximum bipartite matching problem introduced in Week 4). More generally, matching in any graph is polynomial time solvable but this is not an easy result.

The matching problem for undirected graphs

Given a large group of people, we want to pair them up to work on projects. We know which pairs of people are compatible, and (if possible) we want to put them all in compatible pairs.

- If there are 50 or more people, a brute force approach of trying all possible pairings would take billions of years.
- However in 1965 Jack Edmonds found an ingenious efficient algorithm. So this problem is in P.
- There is often a “fine line” between what is and what is not known to be efficiently solvable (e.g. polynomial time 2SAT vs NP-hard 3SAT).

NP-Complete Problems

- These are (in a certain sense) the hardest *NP* problems.
- A problem A is p-reducible to a problem B if an “oracle” (i.e. a subroutine) for B can be used to efficiently solve A.
- If A is p-reducible to B, then any efficient procedure for solving B can be turned into an efficient procedure for A.
- If A is p-reducible to B and B is in P, then A is in P.

Definition

A problem B is NP-complete if B is in NP and every problem A in NP is p-reducible to B.

Theorem

If A is NP-complete and A is in P, then $P = NP$.

To show $P = NP$ you just need to find a fast (polynomial-time) algorithm for any one NP-complete problem!!!

Conjunctive normal form propositional formulas

The SAT problem is defined in terms of inputs given as conjunctive normal form (CNF) formulas.
Here are the standard definitions regarding CNF propositional formulas:

- A literal is a propositional variable x or its negation $\bar{x}$.
- A clause $C = \ell^1 \vee \ell^2 \ldots \vee \ell^r$ is a disjunction of literals.
- A CNF formula $F = C_1 \wedge C_2 \ldots \wedge C_m$ is a conjunction of clauses.
- A CNF formula is a kCNF formula if every clause has at most k literals. For our purposes we will abuse terminology and say that every clause has exactly k literals. It is always assumed that no variable appears twice in any clause.

$$(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee x_4)$$

Figure: An example of a 3CNF formula

Sat and 3SAT

A formula is *satisfiable* if there is an assignment of truth values (i.e. TRUE, FALSE) to the propositional variables such that the formula evaluate to TRUE. For CNF formulas, this means that there is an assignment of truth values such that every clause is TRUE.

SAT (resp. 3SAT) is the decision problem that determines if a CNF (resp. 3CNF) formula is satisfiable.

Following the initial results of Cook and Karp, our development of *NP* complete problems rests on showing that SAT is *NP* complete. We have to show how to reduce *any NP* decision problem to SAT.

We delay the proof of that Theorem until after some examples of reductions. Some reductions will be relatively easy and some not.

Many research papers were written in the early 1970's establishiing NP-completeness for specific problems. The monograph by Garey and Johnson was a valued reference keeping track of many examples. There are web sites devoted to particular domains listing NP complete problems.

- A great many (literally, thousands) of problems have been shown to be *NP*-complete.
- Most scheduling related problems (delivery trucks, exams etc) are *NP*-complete.
- The following simple exam scheduling problem is *NP*-complete:

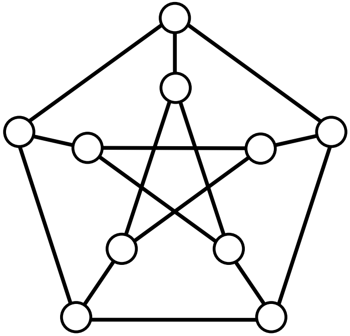
Example

- We need to schedule N examinations, and only three time slots are available.
- We are given a list of exam conflicts: A conflict is a pair of exams that cannot be offered at the same time, because some student needs to take both of them.
- **Problem:** Determine if there is a way of assigning each exam to one of the time slots T_1, T_2, T_3, so that no two conflicting exams are assigned to the same time slot.
- This problem is known as the graph 3-colourability problem.

Graph 3-Colourability

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.

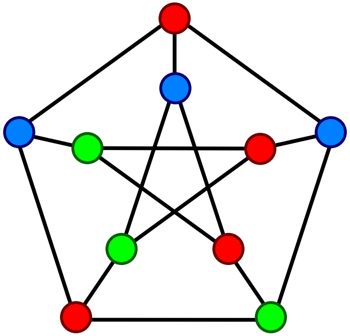


Imagine trying to decide this when there are say hundreds or thousands of nodes.

Graph 3-Colourability

Problem

Given a graph, determine whether each node can be coloured red, blue, or green, so that the endpoints of each edge have different colours.



Imagine trying to decide this when there are say hundreds or thousands of nodes.

Some more remarks on graph coloring

- The natural graph coloring optimization problem is to color a graph with the fewest number of colors.
- We can phrase it as a search or decision problem by saying that the input is a pair (G, k) and then
  1. The search problem is to find a k-coloring of the graph G if one exists.
  2. The decision problem is to determine whether or not G has a k coloring.
  3. Clearly, solving the optimization problem solves the search problem which in turn solves the decision problem.
  4. Conversely, if we can efficiently solve an NP complete decision problem then we can efficiently solve the search and optimization problems. This can be shown in general but we can also show it for specific problems (e.g., for graph coloring as you should try to show).
- Formally it is the graph coloring decision problem which is NP-complete. More precisely, the graph coloring decision problem for any fixed $k \geq 3$ is NP-complete. However, 2-Colorability is in P.
- Search or optimization problems to which any NP-complete decision problem can be reduced are then called NP-hard.

Reducing Graph 3-Colourability to 3SAT

We begin our examples of reductions between *NP* decision problems with a reduction that is implied by the fact that graph 3-colouring is in *NP* and hence must reduce to 3SAT which is *NP*-complete. This reduction would now be considered a relatively easy reduction (certainly in hindsight) but it illustrates how reductions can be between problems coming from what are traditionally thought of as different research areas.

- We are given a graph G with nodes, say $V = \{v_1, v_2, \ldots, v_n\}$
- We are given a list of edges, say $(v_3, v_5), (v_2, v_6), (v_3, v_6), \ldots$
- We need to find a 3CNF formula F which is satisfiable if and only if G can be colored with 3 colors (say red, blue, green). **Note:** Any permutation or renaming of the colors does the change what follows.
- We use three different types of Boolean VARIABLES
  $r_1, r_2, ..., r_n$ (r_i means node i is colored red)
  $b_1, b_2, ..., b_n$ (b_i means node i is colored blue)
  $g_1, g_2, ..., g_n$ (g_i means node i is colored green)

Here we are abusing terminology as “means” is really “intended meaning”

- Here are the CLAUSES for the formula F:
  - We need one clause for each node:
    $(r_1 \vee b_1 \vee g_1)$ (node 1 gets at least one color)
    $(r_2 \vee b_2 \vee g_2)$ (node 2 gets at least one color)
    ...
    $(r_n \vee b_n \vee g_n)$ (node n gets at least one color)
  - We could put in clauses saying that no node gets colored with more than one color but coloring a node with more than one color can only make it more difficult to color so we really don't need these clauses.
  - We need 3 clauses for each edge: For the edge (v_3, v_5) we need
    $(\overline{r_3} \vee \overline{r_5})$ (v_3 and v_5 not both red)
    $(\overline{b_3} \vee \overline{b_5})$ (v_3 and v_5 not both blue)
    $(\overline{g_3} \vee \overline{g_5})$ (v_3 and v_5 not both green)
- The size of the formula F is comparable to the size of the graph G.
- **Check:** G is 3-colorable if and only if F is satisfiable.

On the nature of this polynomial time reduction

- If we consider the previous reduction of 3-coloring to 3-SAT, it can be seen as a very simple type of reduction.
- Namely, given an input w to the 3-coloring problem, it is transformed (in polynomial time) to say $h(w)$ such that

$$w \in \{G \mid G \text{ can be 3-colored}\} \text{ iff}$$
$$h(w) \in \{F \mid F \text{ is a satisfiable 3CNF formula}\}.$$

- If we express the problems as decision probems, the reduction of bipartite matching to maximum flows is also a transformation.

Polynomial time transformations

- We say that a language L_1 is polynomial time transformable to L_2 if there exists a polynomial time function h such that

$$w \in L_1 \text{ iff } h(w) \in L_2.$$

- The function h is called a polynomial time transformation.