P and NP
CISC5835, Algorithms for Big Data
CIS, Fordham Univ.

Instructor: X. Zhang

Efficient Algorithms

- So far, we have developed algorithms for finding
  - shortest paths in graphs,
  - minimum spanning trees in graphs,
  - matchings in bipartite graphs,
  - maximum increasing subsequences,
  - maximum flows in networks,
  - …
- All these algorithms are efficient, because in each case their time requirement grows as a polynomial function (such as n, n^2, or n^3) of the size of the input (n).
  - These problems are tractable.

Exponential search space

- In all these problems we are searching for a solution (path, tree, matching, etc.) from among an exponential number of possibilities.
  - Brute force solution: checking through all candidate solutions, one by one.
  - Running time is 2^n, or worse, useless in practice
- Quest for efficient algorithms: finding clever ways to bypass exhaustive search, using clues from input in order to dramatically narrow down the search space.
  - for many problems, this quest hasn't been successful: fastest algorithms we know for them are all exponential.

Satisfiability Problem

- A boolean expression in **conjunctive normal form (CNF)**

$$(x \vee y \vee \bar{z})(x \vee \bar{y})(y \vee \bar{z})(z \vee \bar{x})(\bar{x} \vee \bar{y} \vee \bar{z})$$

  - **literals**: a boolean variable or negation of one
  - a collection of clauses (in parentheses), each consisting of **disjunction** (logical or, $\vee$) of several literals
- A satisfying truth assignment: an assignment of false or true to each variable so that every clause contains a literal whose value is true, and whole expression is satisfied (true)
  - is (x=T, y=T, z=F) satisfying truth assignment to above CNF?
- **SAT Problem**
  - Given a Boolean formula in CNF
  - Either find a satisfying truth assignment or report that none exists.

SAT as a search problem

- SAT is a typical **search problem (or decision problem)**
  - Given an instance I (i.e., some input data specifying problem at hand),
  - To find a solution S (an object that meets a particular specification). If no such solution exists, we must say so.
- **In SAT:** input data is a Boolean formula in conjunctive normal form, and solution we are searching for is an assignment that satisfies each clause.

Search Problems

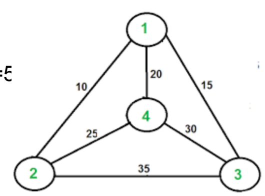
- For each such search problem, consider corresponding **checking/verifying algorithm C, which:**
  - Given inputs: an instance I and a proposed solution S
  - Runs in time polynomial in size of instance, i.e., |I|.
  - Return true if S is a solution to I, and return false if otherwise
- For SAT problem, checking/verifying algorithm C
  - take instance I, such as,

$$(x \vee y \vee \bar{z})(x \vee \bar{y})(y \vee \bar{z})(z \vee \bar{x})(\bar{x} \vee \bar{y} \vee \bar{z})$$

  - solution S, such as (x=T, y=T, z=F)
  - return true if S is a satisfying truth assignment for I.

Traveling Salesman Problem

- Given n vertices 1, . . . , n, and all n(n − 1)/2 distances between them, as well as a budget b.
  - Can we tour 4 nodes with budge b=5



- Output: find a tour (a cycle that passes through every vertex exactly once) of total cost b or less – or to report that no such tour exists.
  - find permutation $\tau(1),\ldots,\tau(n)$ of vertices such that when they are toured in this order, total distance covered is at most b:
  - $d_{\tau(1),\tau(2)} + d_{\tau(2),\tau(3)} + \cdots + d_{\tau(n),\tau(1)} \leq b$.

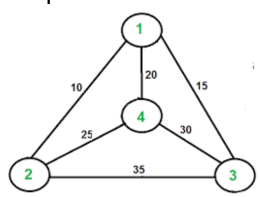
NP Problem

For a search/decision problem, if :

- There is an efficient checking algorithm C that takes as input **the given instance I**, **the proposed solution S**, and outputs true if and only if S really is a solution to instance I; and outputs false o.w.
- Moreover running time of C(I,S) is bounded by a polynomial in |I|, the length of the instance.
- Then the search/decision problem belongs to NP, the set of search problem for which there is a polynomial time checking algorithms
  - Origin: such search problem can be solved in **polynomial** time by **nondeterministic** Turing machine

Traveling Salesman Problem

- Given n vertices 1, . . . , n, and all $n(n - 1)/2$ distances between them, as well as a budget b.
- Output: find a tour (a cycle that passes through every vertex exactly once) of total cost b or less – or to report that no such tour exists.
- Here, TSP is defined as a search/decision problem
  - given an instance, find a tour within the budget (or report that none exists).
- Usually, TSP is posed as optimization problem
  - i.e., find shortest possible tour
  - 1->2->3->4, total cost: 60



Search vs Optimization

- Turning an optimization problem into a search problem does not change its difficulty at all, because the two versions reduce to one another.
- Any algorithm that solves the optimization TSP also readily solves search problem: find the optimum tour and if it is within budget, return it; if not, there is no solution.
- Conversely, an algorithm for search problem can also be used to solve optimization problem:
  - First suppose that we somehow knew cost of optimum tour; then we could find this tour by calling algorithm for search problem, using optimum cost as the budget.
  - We can find optimum cost by binary search.

Why Search (not Optimize)?

- Isn't any optimization problem also a search problem in the sense that we are searching for a solution that has the property of being optimal?
  - The solution to a search problem should be easy to recognize, or as we put it earlier, polynomial-time checkable.
- Given a potential solution to the TSP, it is easy to check the properties "is a tour" (just check that each vertex is visited exactly once) and "has total length ≤ b."
- But how could one check the property "is optimal"?

Next: a collection of problems …

apparently similar problems have different complexities

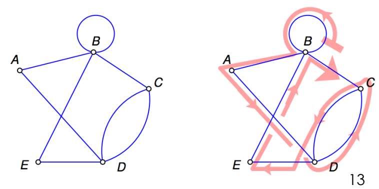
Euler Path:

Given a graph, find a path that contains each edge exactly once.

Possible, if and only if

- (a) the graph is connected and
- (b) every vertex, with the possible exception of two vertices (the start and final vertices of the walk), has even degree.

A polynomial time algorithm for Euler Path?

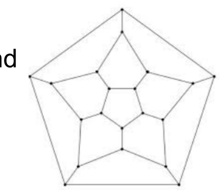


Hamilton/Rudrata Cycle

Rudrata/Hamilton Cycle:

Given a graph, find a cycle that visits each vertex exactly once.

Recall: a cycle is a path that starts and stops at same vertex



Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once.

Minimum Cut

A cut is a set of edges whose removal leaves a graph disconnected.

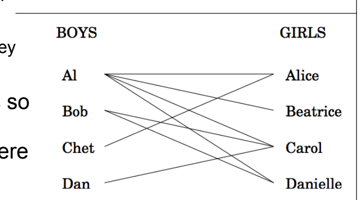
minimum cut: given a graph and a budget b, find a cut with at most b edges.

This problem can be solved in polynomial time by n − 1 max-flow computations: give each edge a capacity of 1, and find the maximum flow between some fixed node and every single other node.

The smallest such flow will correspond (via the max-flow min-cut theorem) to the smallest cut.

Bipartite Matching

- Input: a (bipartite) graph
  - four nodes on left representing boys and four nodes on the right representing girls.
  - there is an edge between a boy and girl if they like each other
- Output: Is it possible to choose couples so that everyone has exactly one partner, and it is someone they like? I (i.e., is there a *perfect matching*?)
- Reduced to maximum-flow problem.
  - Create a super source node, s, with outgoing edges to all boys
  - Add a super sink node, t, with incoming edges from all girls
  - direct all edges from boy to girl, assigned cap. of 1

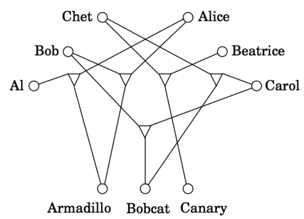


3D matching

3D matching: there are n boys and n girls, but also n pets, and the compatibilities among them are specified by a set of triples, each containing a boy, a girl, and a pet.

Intuitively, a triple (b,g,p) means that boy b, girl g, and pet p get along well together.

We want to find n disjoint triples and thereby create n harmonious households.



Graph Problems

independent set:
Given a graph and an integer g, find g vertices, no two of which have an edge between them.

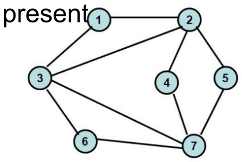
g=3, {3, 4, 5}

vertex cover: Given a graph and an integer b, find b vertices cover (touch) every edge.

b=1, no solution; b=2, {3, 7}

Clique: Given a graph and an integer g, find g vertices such that all possible edges between them are present.

g=3, {1, 2, 3}.



Knapsack

knapsack: We are given integer weights $w_1, \ldots, w_n$ and integer values $v_1,\ldots,v_n$ for n items.

We are also given a weight capacity W and a goal g

We seek a set of items whose total weight is at most W and whose total value is at least g.

The problem is solvable in time O(nW) by dynamic programming.

subset sum:
Find a subset of a given set of integers that adds up to exactly W .

Hard Problems, Easy Problems

Hard problems (**NP**-complete)	Easy problems (in **P**)
3SAT	2SAT, HORN SAT
TRAVELING SALESMAN PROBLEM	MINIMUM SPANNING TREE
LONGEST PATH	SHORTEST PATH
3D MATCHING	BIPARTITE MATCHING
KNAPSACK	UNARY KNAPSACK
INDEPENDENT SET	INDEPENDENT SET on trees
INTEGER LINEAR PROGRAMMING	LINEAR PROGRAMMING
RUDRATA PATH	EULER PATH
BALANCED CUT	MINIMUM CUT

P

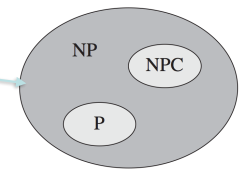
We've seen many examples of NP search problems that are solvable in polynomial time.

In such cases, there is an algorithm that takes as input an instance I and has a running time polynomial in |I|. If I has a solution, the algorithm returns such a solution; and if I has no solution, the algorithm correctly reports so.

The class of all search problems that can be solved in polynomial time is denoted P.

P=NP?

- Most people believe not



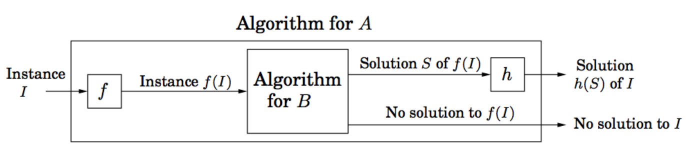
- Many problems have no polynomial time algorithms … yet.
- All problems on left side of table are same problem.
  - If one of them has a polynomial time algorithm, then every problem has a polynomial time algorithm.
  - **NP Complete (NPC)**

Reduce A -> B

A reduction from search problem A to search problem B

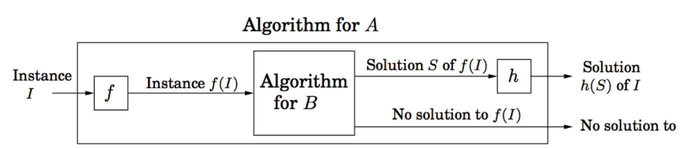
- a polynomial time algorithm **f** that transforms any instance I of A into an instance f(I) of B
- and another polynomial time algorithm **h** that maps any solution S of f(I) back into a solution h(S) of I.
- If f (I) has no solution, then neither does I .

Any algorithm for B can be converted into an algorithm for A by bracketing it between f and h.



Reduction

- Assume there is a reduction from a problem A to a problem B. A → B.
  - If we can solve B efficiently, then we can also solve A efficiently.
  - If we know A is hard, then B must be hard too.
    Reductions also have the convenient property that they compose. If A → B and B → C, then A → C.



NP Complete

Definition:

A search problem C is NP-complete

1) It's NP

2) Every NP problem can be reduced to C.

3SAT -> Independent Set

3SAT (special case of SAT)

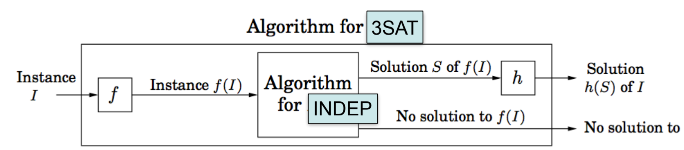
input: a set of clauses, each with three or fewer literals,

Output: a satisfying truth assignment (if exists)

Independent Set

Input: a graph and a number g

Output: a set of g pairwise non-adjacent vertices (if exists)



3SAT -> Independent Set

3SAT: find satisfying truth assignment for a set of clauses

$$(\overline{x} \vee y \vee \overline{z})\ (x \vee \overline{y} \vee z)\ (x \vee y \vee z)\ (\overline{x} \vee \overline{y}),$$

Independent Set input: a graph and a number g

Output: find a set of g pairwise non-adjacent vertices.

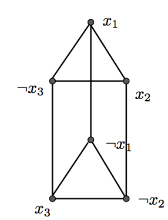
Given an instance I of 3SAT, create an instance (G,g) of Independent Set as follows:

- Graph G has a triangle for each clause (or just an edge, if the clause has two literals), with vertices labeled by the clause's literals, edges between any two vertices that represent opposite literals.
- Goal g is set to the number of clauses.

3SAT -> Independent Set

3SAT: find satisfying truth assignment for a set of clauses

$(x_1 \vee x_2 \vee \neg x_3) \wedge (x_3 \vee \neg x_2 \vee \neg x_1)$



Find independent set of size 2

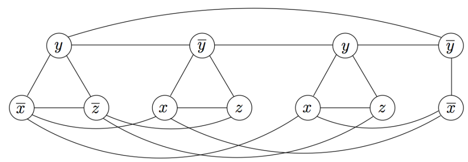
Independent Set input: a graph and a number g

Output: find a set of g pairwise non-adjacent vertices.

3SAT -> Independent Set

3SAT: find satisfying truth assignment for a set of clauses

Figure 8.8 The graph corresponding to $(\bar{x} \vee y \vee \bar{z})\ (x \vee \bar{y} \vee z)\ (x \vee y \vee z)\ (\bar{x} \vee \bar{y})$.



Find independent set of size 4

Independent Set input: a graph and a number g

Output: find a set of g pairwise non-adjacent vertices.

SAT -> 3SAT

Given an **instance I** of SAT where clauses have more than three literals, $(a_1 \vee a_2 \vee \cdots \vee a_k)$ (a_i 's are literals, k > 3), is replaced by a set of clauses.

$$(a_1 \vee a_2 \vee y_1)(\bar{y}_1 \vee a_3 \vee y_2)(\bar{y}_2 \vee a_4 \vee y_3)\cdots(\bar{y}_{k-3} \vee a_{k-1} \vee a_k),$$

where y_i 's are new variables.

The conversion takes polynomial time.

Resulting CNF, I', is equivalent to I in terms of satisfiability, because for any assignment to the a_i 's,

$$\left\{ \begin{array}{c} (a_1 \vee a_2 \vee \cdots \vee a_k) \\ \text{is satisfied} \end{array} \right\} \iff \left\{ \begin{array}{c} \text{there is a setting for the } y_i\text{'s for which} \\ (a_1 \vee a_2 \vee y_1)(\bar{y}_1 \vee a_3 \vee y_2)(\bar{y}_2 \vee a_4 \vee y_3)\cdots(\bar{y}_{k-3} \vee a_{k-1} \vee a_k) \\ \text{are all satisfied} \end{array} \right\}$$

Summary