# Algorithms (II)

Difficult Problems

Guoqiang Li

School of Software, Shanghai Jiao Tong University

# Efficient Problems, Difficult Problems

# Efficient Algorithms

- 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.

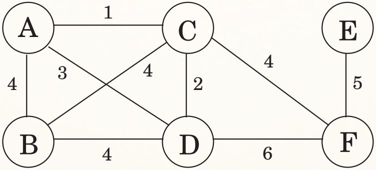
# Exponential Search Space

- In all these problems we are searching for a solution (path, tree, matching,) from among an exponential population of possibilities.
- All these problems could in principle be solved in exponential time by checking through all candidate solutions, one by one.
- An algorithm with running time $2^n$, or worse, is useless in practice.
- The quest for efficient algorithms is about finding clever ways to bypass this process of exhaustive search, using clues from the input in order to dramatically narrow down the search space.
- Other "search problems" in which again we are seeking a solution with particular properties among an exponential chaos of alternatives.
- The fastest algorithms we know for them are all exponential.

# Minimum Spanning Trees

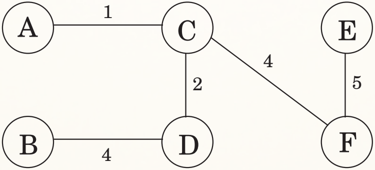
# Build a Network

- Suppose you are asked to network a collection of computers by linking selected pairs of them.
- This translates into a graph problem in which
  - nodes are computers,
  - undirected edges are potential links, each with a maintenance cost.



# Build a Network

- The goal is to
  - pick enough of these edges that the nodes are connected,
  - the total maintenance cost is minimum.
- One immediate observation is that the optimal set of edges cannot contain a cycle.



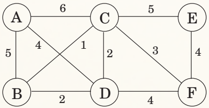
# A Greedy Approach

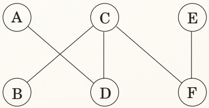
- Kruskal's minimum spanning tree algorithm starts with the empty graph and then selects edges from $E$ according to the following rule.
- Repeatedly add the next lightest edge that doesn't produce a cycle.

Example:

Starting with an empty graph and then attempt to add edges in increasing order of weight

$$B-C; C-D; B-D; C-F; D-F; E-F; A-D; A-B; C-E; A-C$$



# A General Kruskal's Algorithm

$X = \{\ \}$;
repeat until $|X| = |V| - 1$;
    pick a set $S \subset V$ for which $X$ has no edges between $S$ and $V - S$;
    let $e \in E$ be the minimum-weight edge between $S$ and $V - S$;
    $X = X \cup \{e\}$;

# Prim's Algorithm

- A popular alternative to Kruskal's algorithm is Prim's, in which the intermediate set of edges $X$ always forms a subtree, and $S$ is chosen to be the set of this tree's vertices.
- On each iteration, the subtree defined by $X$ grows by one edge, namely, the lightest edge between a vertex in $S$ and a vertex outside $S$. We can equivalently think of $S$ as growing to include the vertex $v \notin S$ of smallest $\texttt{cost}$:

$$\texttt{cost}(v) = \min_{u \in S} w(u, v)$$

# A Little Change of the MST

What if the tree is not allowed to branch?

# Satisfiability Problem

# Satisfiability

The instances of Satisfiability or SAT:

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

That is, a Boolean formula in conjunctive normal form (CNF).

- It is a collection of clauses (the parentheses),
  - each consisting of the disjunction (logical or, denoted $\vee$) of several literals;
  - a literal is either a Boolean variable (such as $x$) or the negation of one (such as $\overline{x}$).
- A satisfying truth assignment is an assignment of `false` or `true` to each variable so that every clause contains a literal whose value is `true`.
- Given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

# 2-SAT

Given a set of clauses, where each clause is the disjunction of two literals. You are looking for a way to assign a value true or false to each of the variables so that all clauses are satisfied. that is, there is at least one true literal in each clause.

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_3) \wedge (x_1 \vee \bar{x}_2) \wedge (x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4)$$

Given an instance $I$ of 2-Sat with $n$ variables and $m$ clauses, construct a directed graph $G_I = (V, E)$ as follows.

- $G_I$ has $2n$ nodes, one for each variable and its negation.
- $G_I$ has $2m$ edges: for each clause $(\alpha \vee \beta)$ of $I$, $G_I$ has an edge from the negation of $\alpha$ to $\beta$, and one from the negation of $\beta$ to $\alpha$.

# 2-SAT

- Show that if $G_I$ has a strongly connected component containing both $x$ and $\overline{x}$ for some variable $x$, then $I$ has no satisfying assignment.
- If none of $G_I$'s strongly connected components contain both a literal and its negation, then the instance $I$ must be satisfiable.
- Conclude that there is a linear-time algorithm for solving 2-SAT.

# Search Problems (Decision Problems)

# Satisfiability

The instances of Satisfiability or SAT:

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

That is, a Boolean formula in conjunctive normal form (CNF).

- It is a collection of clauses (the parentheses),
  - each consisting of the disjunction (logical or, denoted $\vee$) of several literals;
  - a literal is either a Boolean variable (such as $x$) or the negation of one (such as $\overline{x}$).
- A satisfying truth assignment is an assignment of `false` or `true` to each variable so that every clause contains a literal whose value is `true`.
- Given a Boolean formula in conjunctive normal form, either find a satisfying truth assignment or else report that none exists.

# Search Problems

- SAT is a typical search problem.
- We are given an `instance` $I$
  - that is, some input data specifying the problem at hand,
  - in this case a Boolean formula in conjunctive normal form.
- we are asked to find a `solution` $S$
  - an object that meets a particular specification,
  - in this case an assignment that satisfies each clause.
- If no such solution exists, we must say so.

# Search Problems

- A search problem must have the property that any proposed solution $S$ to an instance $I$ can be quickly checked for correctness.
- $S$ must at least be concise, with length polynomially bounded by that of $I$.
    - This is clearly true in the case of SAT, for which $S$ is an assignment to the variables.
- There is a polynomial-time algorithm that takes as input $I$ and $S$ and decides whether or not $S$ is a solution of $I$.
    - For SAT, this is easy as it just involves checking whether the assignment specified by $S$ indeed satisfies every clause in $I$.

# Search Problems

A search problem is specified by an algorithm $C$ that takes two inputs, an instance $I$ and a proposed solution $S$, and runs in time polynomial in $|I|$.

We say $S$ is a solution to $I$ if and only if $C(I, S) = \texttt{true}$.

# Satisfiability Revisit

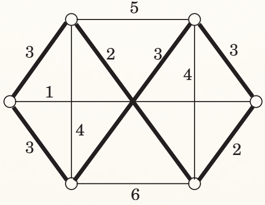
- Researchers over the past 50 years have tried hard to find efficient ways to solve the SAT, but without success.
- The fastest algorithms we have are still exponential on their worst-case inputs.
- There are two natural variants of SAT for which we do have good algorithms.
  - 2-SAT, however, can be solved in linear time.
  - If all clauses contain at most one positive literal, then the Boolean formula is called a Horn formula, and a satisfying truth assignment, if one exists, can be found by the greedy algorithm.

# Traveling Salesman Problem

# The Traveling Salesman Problem

- In the traveling salesman problem(TSP) we are given $n$ vertices and all $n(n-1)/2$ distances between them, and a budget $b$.
- To find a cycle that passes through every vertex exactly once, of total cost $b$ or less - or to report that no such cycle.
- A permutation $\tau(1), \ldots, \tau(n)$ of the vertices such that when they are toured in this order, the total distance covered is at most $b$:

$$d_{\tau(1),\tau(2)} + d_{\tau(2),\tau(3)} + \ldots + d_{\tau(n),\tau(1)} \leq b$$



# The Traveling Salesman Problem

- We have defined the TSP as a search problem: given an instance, find a tour within the budget (or report that none exists).
- But why are we expressing the TSP in this way, when in reality it is an optimization problem, in which the shortest possible tour is sought?

# 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 also readily solves the search problem:
  - find the optimum tour and if it is within budget, return it; if not, there is no solution.
- Conversely, an algorithm for the search problem can also be used to solve the optimization problem:
  - First suppose that we somehow knew the cost of the optimum tour; then we could find this tour by calling the algorithm for the search problem, using the optimum cost as the budget.
  - We can find the optimum cost by binary search.

# Search Instead of Optimization

- 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 $\leq b$."
- But how could one check the property ”is optimal”?

# TSP Revisit

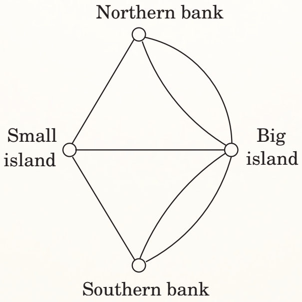
- There are no known polynomial-time algorithms for the TSP, despite much effort by researchers over nearly a century.
- There exists a faster, yet still exponential, dynamic programming algorithm.
- The Minimum spanning tree (MST) problem, for which we do have efficient algorithms, provides a stark contrast here.
- The TSP can be thought of as a tough cousin of the MST problem, in which the tree is not allowed to branch and is therefore a path.
- This extra restriction on the structure of the tree results in a much harder problem.

# Euler and Rudrata

# Euler Path

Euler path:

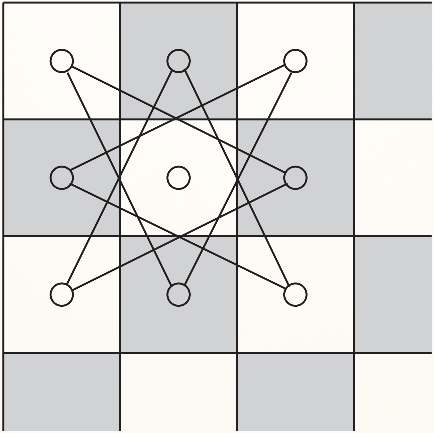
Given a graph, find a path that contains each edge exactly once.



# Euler Path

- The answer is yes if and only if
  1. the graph is connected and
  2. every vertex, with the possible exception of two vertices (the start and final vertices of the walk), has even degree.
- Using above, it is easy to see that there is a polynomial time algorithm for Euler path.

# Rudrata Cycle

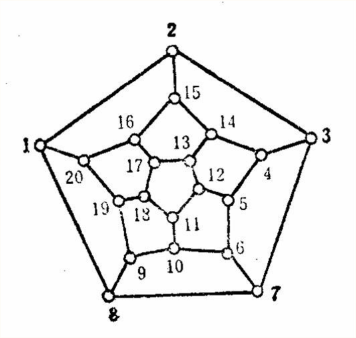


# Rudrata Cycle

Rudrata Cycle:

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

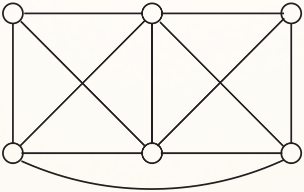
In the literature this problem is known as the Hamilton cycle problem.



# Cuts and Bisections

# 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.



# Minimum Cut

- 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.

# Balanced Cut

- In many graphs, the smallest cut leaves just a singleton vertex on one side - it consists of all edges adjacent to this vertex.
- Far more interesting are small cuts that partition the vertices of the graph into nearly equal-sized sets.
- Balanced cut: Given a graph with $n$ vertices and a budget $b$, partition the vertices into two sets $S$ and $T$ such that $|S|, |T| \geq n/3$ and such that there are at most $b$ edges between $S$ and $T$.

# Integer Linear Programming

# Linear Programming

- In a linear programming problem we are given a set of variables, and we want to assign real values to them so as to
  1. satisfy a set of linear equations and/or linear inequalities involving these variables, and
  2. maximize or minimize a given linear objective function.

# Linear Programming

$$
\begin{gathered}
\max x_1 + 6x_2 + 13x_3 \\
x_1 \leq 200 \\
x_2 \leq 300 \\
x_1 + x_2 + x_3 \leq 400 \\
x_2 + 3x_3 \leq 600 \\
x_1, x_2, x_3 \geq 0
\end{gathered}
$$

# Integer Linear Programming

- Integer linear programming (ILP): We are given a set of linear inequalities $A\mathbf{x} \leq b$, where
  - $A$ is an $m \times n$ matrix and
  - $b$ is an $m$-vector;
  - an objective function specified by an $n$-vector $c$;
  - and finally, a goal $g$ (the counterpart of a budget in maximization problems).
- We want to find a nonnegative integer $n$-vector $x$ such that $A\mathbf{x} \leq b$ and $c \cdot \mathbf{x} \geq g$.

$$\begin{gathered} \max 2x_1 + 5x_2 \\ 2x_1 - x_2 \leq 4 \\ x_1 + 2x_2 \leq 9 \\ -x_1 + x_2 \leq 3 \\ x_1, x_2 \geq 0 \end{gathered}$$

# Integer Linear Programming

$$2x_1 + 5x_2 \leq g$$
$$2x_1 - x_2 \leq 4$$
$$x_1 + 2x_2 \leq 9$$
$$-x_1 + x_2 \leq 3$$
$$x_1, x_2 \geq 0$$

- But there is a redundancy here:
    - the last constraint $c \cdot \mathbf{x} \geq g$ is itself a linear inequality and
    - can be absorbed into $A\mathbf{x} \leq b$.

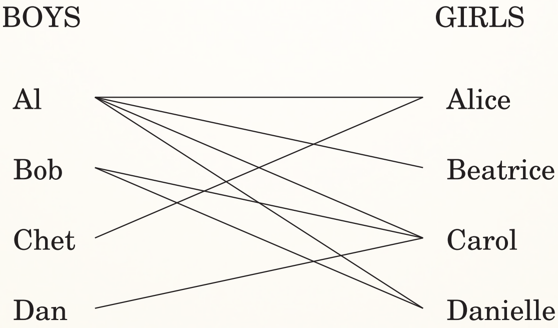
# Integer Linear Programming

So, we define ILP to be following search problem:

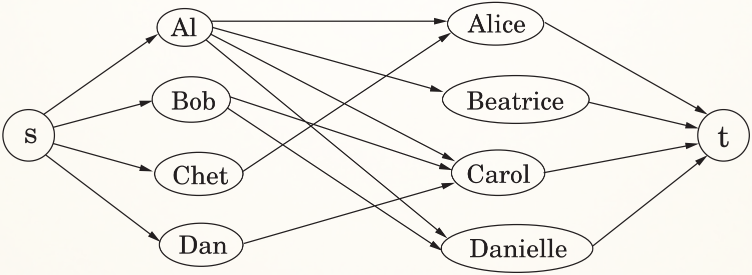
Given $A$ and $b$, find a nonnegative integer vector $\mathbf{x}$ satisfying the inequalities $A\mathbf{x} \leq b$.

# 3D-Matching

# Bipartite Matching

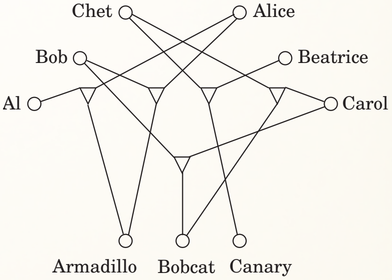


# Bipartite Matching



# Three-Dimensional Matching

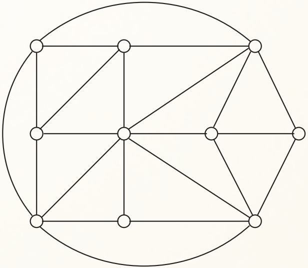
- 3D matching: There are $n$ boys and $n$ girls, but also $n$ pets, and the compatibilities among them are specified by a set of triples, each containing a boy, a girl, and a pet.
- Intuitively, a triple $(b, g, p)$ means that boy $b$, girl $g$, and pet $p$ get along well together.
- We want to find $n$ disjoint triples and thereby create $n$ harmonious households.



# Independent Set

# Independent Set, Vertex Cover, and Clique

- Independent set: Given a graph and an integer $g$, find $g$ vertices, no two of which have an edge between them.
- Vertex cover: Given a graph and an integer $b$, find $b$ vertices cover (touch) every edge.
- Clique: Given a graph and an integer $g$, find $g$ vertices such that all possible edges between them are present.



# Longest Path

# Longest Path

- Longest path: Given a graph $G$ with nonnegative edge weights and two distinguished vertices $s$ and $t$, along with a goal $g$.
- We are asked to find a path from $s$ to $t$ with total weight at least $g$.
- To avoid trivial solutions we require that the path be simple, containing no repeated vertices.

# Knapsack

# 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.

# Knapsack

- Is there a polynomial algorithm for Knapsack? Nobody knows of one.
- A variant of the Knapsack problem is that the integers are coded in unary.
    - e.g., by writing *IIIIIIIIIIII* for 12.
    - It defines a legitimate problem, which we could call Unary knapsack.
    - It has a polynomial algorithm.
- A different variation:
    - Suppose now that each item's value is equal to its weight (all given in binary), the goal $g$ is the same as the capacity $W$.
    - This special case is tantamount to finding a subset of a given set of integers that adds up to exactly $W$.
    - Q: Could it be polynomial?

# Subset Sum

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

# Referred Materials

- Content of this lecture comes from Section 8.1, 5.1, and 3.4 in [DPV07].
- Suggest to read Chapter 34 of [CLRS09].