Efficient and Not-So-Efficient Algorithms

Problem spaces tend to be big:
- A graph on \( n \) vertices can have up to \( n^{n-2} \) spanning trees.
- A graph on \( n \) vertices can have \( \Theta(2^n) \) many paths between verts \( s \) and \( t \).
- Etc.

The Good News
In our previous work, out of problems with \( \Theta(2^n) \) (or worse) many choices, we have found the right answer in time \( O(n^k) \) for some \( k \).

The Bad News
Not all problems are so nice.

Search Problems
Search problems are those of the form:

**Given:** …
**Find:** … (usually within a large search space)

Satisfiability (as a search problem)
**Given:** A boolean formula in conjunctive normal form (CNF).
**Find:** A satisfying assignment for it (if it has one).

We need to define some terms.

Propositional Logic
- The *formulas* of propositional logic are given by the grammar:
  \[
  P \ ::= \ Var \mid \neg P \mid P \land P \mid P \lor P
  \]
  \[
  Var \ ::= \text{the syntactic category of variables}
  \]
- A *truth assignment* is a function \( I : \text{Variables} \to \{ \text{False}, \text{True} \} \).
- \( I \), a truth assignment, determines the value of a formula by:
  \[
  I[x] = \text{True} \iff I(x) = \text{True} \quad (x \text{ a variable})
  \]
  \[
  I[\neg p] = \text{True} \iff I[p] = \text{False}
  \]
  \[
  I[p \land q] = \text{True} \iff I[p] = I[q] = \text{True}
  \]
  \[
  I[p \lor q] = \text{True} \iff I[p] = \text{True} \quad \text{or} \quad I[q] = \text{True}
  \]
- A *satisfying assignment* for a formula \( p \) is an \( I \) with \( I[p] = \text{True} \).
- The only known algorithms for SAT run in exponential time (in the worst case).
Digression: DeMorgan’s Laws

\( \neg(P \lor Q) \iff (\neg P) \land (\neg Q) \)

\( \neg(P \land Q) \iff (\neg P) \lor (\neg Q) \)

Digression: Distributive Laws

\( A \land (B \lor C) \iff (A \land B) \lor (A \land C) \)

\( A \lor (B \land C) \iff (A \lor B) \land (A \lor C) \)

* Exercise for the reader.

Conjunctive Normal Form

- Instead of writing \( \neg x \) we write \( \overline{x} \).
- A variable \( x \) and the negation of a variable \( \overline{x} \) are called literals.
- A clause is a disjunction of literals.
  - E.g.: \( x \lor \overline{y} \lor z \).
- A conjunctive normal form formula is a conjunction of clauses.
  - E.g.: \( (x \lor y \lor z) \land (x \lor \overline{y} \lor z) \land (z \lor \overline{x}) \land (x \lor \overline{y} \lor \overline{z}) \)

Satisfiability (as a search problem)

**Given:** A boolean formula in conjunctive normal form (CNF).
**Find:** A satisfying assignment for it (if it has one).

- Note the differences with the boolean circuit evaluation problem.
- If a CNF formula has \( n \) variables, there are \( 2^n \) possible assignments.

Digression: Translating Boolean Formulas to CNF

- A formula is in *negation normal form* (NNF) iff the only place a negation symbol appears in \( F \) is in front of a variable.

**Step 1**
Given a formula \( F \) translate it to an equivalent NNF formula using DeMorgan’s Laws.

**Step 2**
Given a NNF formula \( F \) translate it to an equivalent CNF formula using the distributive law \( A \lor (B \land C) \iff (A \lor B) \land (A \lor C) \).
### Elements of a Search Problem

- **I**: an instance of the problem
- **S**: a possible solution for I
- **C**: (Instances) $\times$ (Potential Solutions) $\rightarrow \{ \text{True}, \text{False} \}$
- An efficient checking algorithm for C is an algorithm for C that, on input $(I, S)$ runs in $O(|I|^k)$-time for some $k$. (Implies that $|S|$ cannot be too large.)

For SAT:
- An instance: a CNF formula $(x \lor y) \land (y \lor \neg x)$
- A potential solution: a truth assignment $x \mapsto \text{True}, y \mapsto \text{True}$
- Efficient checker: boolean circuit evaluation

### Traveling Salesman

**Traveling Salesman (TS) as a search problem**

**Given**: $n$ vertices and all $n \cdot (n - 1)/2$-many distances between them, $b$ a budget (number)  
**Find**: An ordering of $1, \ldots, n$: $\pi(1), \pi(2), \ldots, \pi(n)$ (a tour) so that 
$$d_{\pi(1),\pi(2)} + d_{\pi(2),\pi(3)} + \cdots + d_{\pi(n),\pi(1)} \leq b.$$  

**Traveling Salesman (TS) as an optimization problem**

**Given**: $n$ vertices and all $n \cdot (n - 1)/2$-many distances between them.  
**Find**: An ordering of $1, \ldots, n$: $\pi(1), \pi(2), \ldots, \pi(n)$ so that the tour’s cost 
$$d_{\pi(1),\pi(2)} + d_{\pi(2),\pi(3)} + \cdots + d_{\pi(n),\pi(1)}$$ is minimal.

The two problems are equivalent:
- A solution to the optimization problem, solves the search problem.  
- Given a way to solve the search problem, you can construct a solution to the opt. problem via binary search.  
- The only known algorithms for these problems are exponential time.  
- TS is a restriction of the Minimal Spanning Tree problem in which the MST is allowed no branches.

### Euler and Hamiltonian Paths, 1

**Definition**

A path in an undirected graph is an Euler path when it uses each edge of the graph exactly once. (The path may pass through a vertex many times.) If the path is a cycle, then it is called an Euler Tour or an Euler Circuit.

**Theorem (Euler)**

$G$ has an Euler path $\iff$ $G$ is connected and has at most two vertices of odd degree.
Euler and Hamiltonian Paths, 2

Definition
A path in an undirected graph is a Rudrata path (or more usually a Hamiltonian path) when it uses each vertex of the graph exactly once.

If the path is a cycle, then it is called a Rudrata Cycle or Hamiltonian Cycle.

There is a nice poly-time algorithm for the Euler Path Search problem. (See http://en.wikipedia.org/wiki/Eulerian_path.)

All known algorithms for the Hamiltonian Path Search Problem are exponential-time.

Integer Linear Programming

Given: constraints $A\vec{x} \leq \vec{b}$ and objective function $\vec{c}^T \cdot \vec{x}$ and goal: $g$

Find: A vector of integers $\vec{x}$ satisfying $A\vec{x} \leq \vec{b}$ and $\vec{c}^T \cdot \vec{x} \geq g$.

--- or equivalently ---

Given: constraints $A\vec{x} \leq \vec{b}$

Find: A vector of integers $\vec{x}$ satisfying $A'\vec{x} \leq \vec{b}'$.

$(\vec{c}^T \cdot \vec{x} \geq g$ is incorporated into the constraints.)

Zero-One Equations (ZOE)

Given: constraints $A\vec{x} \leq \vec{b}$

Find: A vector of 0's and 1's $\vec{x}$ satisfying $A'\vec{x} \leq \vec{b}'$.

- ILP and ZOE show up in lots of optimization work.
- The only known algorithms for ILP and ZOE are exponential time.

Cuts and bisections

Definition
A cut in a graph is a set of edges which, if removed, disconnect the graph.

A minimum cut is a cut of smallest size.

Minimum Cut Problem
Given: An undirected graph $G$ and a budget $b$ (a number),
Find: A cut of $G$ of at most $b$ edges.

Balanced Cut Problem
Given: An undirected graph $G = (V,E)$ and a budget $b$ (a number),
Find: A partition of $V$ into sets $S$ and $T$ with $|S|, |T| \geq |V|/3$ such that the number of edges between $S$ and $T$ is at most $b$.

- You can use Ford-Fulkerson to solve Min-Cut in poly-time.
- The only known algorithms for Balanced-Cut are exponential time.
- Balanced-Cuts are important in clustering. (See DPV.)

Three-dimensional matching

3D Matching
Given: $R \subseteq A \times B \times C$ where $|A| = |B| = |C| = n$.
Find: A subset $M \subseteq R$ of $n$ many triples such that if $(a,b,c)$ and $(a',b',c')$ are distinct elements of $M$, then $a \neq a'$, $b \neq b'$, and $c \neq c'$.

- 2D matching is poly-time (via Ford-Fulkerson).
- The only known algorithms for 3D Matching are exponential time.
Independent set, vertex cover, and clique, 1

**Definition**

Suppose \( G = (V, E) \) is an undirected graph and \( U \subseteq V \).

- \( U \) is *independent* when for each \( u, v \in U \), \((u, v) \notin E \).
- \( U \) is a *vertex cover* when each edge of \( E \) has at least one endpoint in \( U \).
- \( U \) is a *clique* when for each distinct \( u, v \in U \), \((u, v) \in E \).

![Graph with independent set, vertex cover, and clique](image)

The blue vertices are a max-sized independent set for the graph.
The red vertices are a min-sized vertex cover for the graph.
The red vertices are a max-sized clique for the graph.

---

Independent set, vertex cover, and clique, 2

**Definition**

Suppose \( G = (V, E) \) is an undirected graph and \( U \subseteq V \).

- \( U \) is *independent* when for each \( u, v \in U \), \((u, v) \notin E \).
- \( U \) is a *vertex cover* when each edge of \( E \) has at least one endpoint in \( U \).
- \( U \) is a *clique* when for each distinct \( u, v \in U \), \((u, v) \in E \).

**Independent Set Problem**

*Given:* \( G \) and \( b \).
*Find:* An independent set for \( G \) of size \( \geq b \).

**Vertex Cover Problem**

*Given:* \( G \) and \( b \).
*Find:* A vertex cover for \( G \) of size \( \leq b \).

- The only known algorithms for these problems are exponential time.

---

Longest Path, Knapsack, Subset Sum

**The Longest Path Problem**

*Given:* A undirected graph \( G \).
*Find:* A longest simple path in \( G \). *(Simple path \equiv a path with no repeated vertices.)*

**Knapsack**

*Given:* Weights \( w_1, \ldots, w_n \), values \( v_1, \ldots, v_n \), Total Capacity: \( W \), and Goal: \( G \) (all positive integers).
*Find:* A selection of items with total weight \( \leq W \) and total value \( \geq G \).

**Subset Sum**

*Given:* A multiset of integers \( M \) and goal \( G \).
*Find:* An \( \{x_1, \ldots, x_k\} \subseteq M \) such that \( G = x_1 + \cdots + x_k \).

- The only known algorithms for these problems are exponential time.
  But didn’t we have an LP solution to Knapsack?  *Yes, but …*

---

Problems: Hard and Easy

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

- All of the hard problems above are hard for the same reason.
- In fact, they are all “the same problem” in disguise.
Recall: Elements of a Search Problem

- $I$: an instance of the problem and $S$: a possible solution for $I$
- $C: \text{ (Instances)} \times \text{(Pot. Solutions)} \rightarrow \{\text{ True, False }\}$; $C(I, S) = \begin{cases} \text{True, } &\text{if } S \text{ solves } I; \\ \text{False, } &\text{otherwise.} \end{cases}$

Definition (NP and P: As search problems)

- An efficient checking algorithm is an algorithm that computes $C$ as above in $O(|I|^k)$-time for some $k$. (This implies that $|S|$ cannot be too large.)
- Each $C: \text{ (Instances)} \times \text{(Pot. Solutions)} \rightarrow \{\text{ True, False }\}$ that is computable in $O(|I|^{O(1)})$ time determines an (artificial) search problem: $[S \text{ is a solution for } I] \iff [C(I, S) = \text{True}]$.
- $NP =$ the class of all search problems that have efficient checking algorithms.
- $P =$ the class of all search problems for which one can find solutions (or determine there are none) in polynomial time.

**Formalizing Reductions, 1**

Q: What is the basis for believing all those problems we just listed are hard?
Reducing problem $A$ to problem $B \approx$ rephrasing $A$ in $B$’s language
E.g., The rephrasing the Boolean Circuit Eval Problem as an IP problem.

Definition

A problem $A$ is reducible to problem $B$ (written: $A \rightarrow B$ [DPV] or $A \leq B$ [these slides]) when there are two polytime computable functions $f$ and $h$:
- $f$ transforms an $A$-instance $I$ into a $B$-instance $f(I)$.
- $h$ transforms a $B$-solution $S$ of $f(I)$ for into an $A$-solution $h(S)$ of $I$.

Algorithm for $A$

<table>
<thead>
<tr>
<th>Instance $I$</th>
<th>$f(I)$</th>
<th>Algorithm for $B$</th>
<th>Solution $h(S)$ of $I$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No solution to $f(I)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solution $S$ of $f(I)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No solution to $I$</td>
<td></td>
</tr>
</tbody>
</table>

P and NP, 2

**Definition**

A decision problem is a problem with an Yes/No answer.

SAT as a decision problem: Does CNF formula $F$ have satisfying assignment?
Hamiltonian Path as a decision problem: Does $G$ have a Hamiltonian Path?

**Definition (NP and P: As decision problems)**

- $NP = \text{ the class of all decision problems that have eff. checking algorithms.}$
- $P = \text{ the class of all decision problems that have polytime algorithms.}$

In Algorithms: search problems are a bit more natural than decision problems.
In Complexity Theory: the reverse
Which is better? The search & decision forms of NP problems are roughly of equivalent hardness.

The $1,000,000,000,000$ Question: $P \not\equiv NP$.
Collect your prize here: http://www.claymath.org/millennium/

**Formalizing Reductions, 2**

Definition

- A problem is $NP$-hard when all $NP$-problems reduce to it.
- A problem is $NP$-complete when it is in $NP$ and $NP$-hard.

Suppose $A \leq B$. Then:
- $B$ is easy $\implies A$ is also easy. (Why?)
- $A$ is hard $\implies B$ is also hard. (Why?)

Algorithm for $A$

<table>
<thead>
<tr>
<th>Instance $I$</th>
<th>$f(I)$</th>
<th>Algorithm for $B$</th>
<th>Solution $h(S)$ of $I$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>No solution to $f(I)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Solution $S$ of $f(I)$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>No solution to $I$</td>
<td></td>
</tr>
</tbody>
</table>

Image from http://www.scspace.co.uk/auv/PINP.html
Formalizing Reductions, 2

**Definition**
- A problem is **NP-hard** when all NP-problems reduce to it.
- A problem is **NP-complete** when it is in NP and NP-hard.

Suppose $A \leq B$. Then:
- $B$ is easy $\implies$ $A$ is also easy. (Why?)
- $A$ is hard $\implies$ $B$ is also hard. (Why?)

$P \subseteq Q \iff \neg Q \implies \neg P$.

Formalizing Reductions, 3

- **Reductions compose**
  - i.e., $A \leq B$ and $B \leq C$ implies that $A \leq C$.
  - $\therefore$ If $A \leq B$ and $A$ is NP-hard, then so it $B$.
  - $\therefore$ If $A$ is NP-complete and $B$ is an NP-problem with $A \leq B$, then $B$ is also NP-complete. (Why is this handy?)

The Plan of §8.3

- All of NP
  - SAT
  - 3SAT
  - INDEPENDENT SET
  - 3D MATCHING
  - VERTEX COVER
  - CLIQUE
  - ZOE
  - SUBSET SUM
  - ILP
  - RUDRA CYCLE
  - TSP