Intractable Problems

Part One

Announcements

- Problem Set Five due right now.
  - Solutions will be released at end of lecture.
- Correction posted for “Guide to Dynamic Programming,” sorry about that!

Please evaluate this course on Axess.

Your feedback really makes a difference.

Outline for Today

- **Intractable Problems**
  - What are the limits of efficient computation?
- **Exponential-Time Algorithms**
  - How do you design better (i.e. less atrocious) algorithms for hard problems?

What is an efficient algorithm?

Defining Efficiency

- Classical definition of efficiency:

  An algorithm is efficient iff it runs in polynomial time on a serial computer.

- Runtimes of "efficient" algorithms:

  $$\mathbf{O}(n) \quad \mathbf{O}(n \log n) \quad \mathbf{O}(n^3 \log^2 n)$$

  $$\mathbf{O}(n^{10,000,000,000})$$

- Runtimes of "inefficient" algorithms:

  $$\mathbf{O}(2^n) \quad \mathbf{O}(n!)$$

  $$\mathbf{O}(1.0000001^n)$$

Some Caveats

- **Parallelism:** Some problems can be solved in time $O(\log^k n)$ time on machines with a polynomial number of processors.
  - Are all efficient algorithms parallelizable?
- **Randomization:** Some algorithms can be solved in *expected* polynomial time, or have poly-time Monte Carlo algorithms that work with high probability.
  - Are randomized efficient algorithms efficient solutions?
- **Quantum computation:** Some algorithms can be solved in polynomial time on a quantum computer.
  - Are quantum efficient algorithms efficient solutions?

These are all open problems!

Tractability and Intractability

- A problem is called **tractable** iff there is an efficient (i.e. polynomial-time) algorithm that solves it.
- A problem is called **intractable** iff there is no efficient algorithm that solves it.
- ***Intractable problems are common***. We need to discuss how to approach them when you come across them in practice.

NP-Completeness and **NP**-Hardness

The Complexity Class **NP**

- A **decision problem** is a problem with a yes/no answer.
- The class **NP** consists of all decision problems where "yes" answers can be *verified* efficiently.
- Examples:
  - Is the kth order statistic of A equal to x?
  - Is there a cut in G of size at least k?
  - Is there a dominating set in G of size at most k?
- All tractable decision problems are in **NP**, plus a lot of problems whose difficulty is unknown.

NP-Completeness

- The **NP-complete problems** are (intuitively) the hardest problems in **NP**.
- Either *every* **NP**-complete problem is tractable or *no* **NP**-complete problem is tractable.
  - This is an open problem: the $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ question has a \$1,000,000 bounty!
- As of now, there are no known polynomial-time algorithms for any **NP**-complete problem.

NP-Hardness

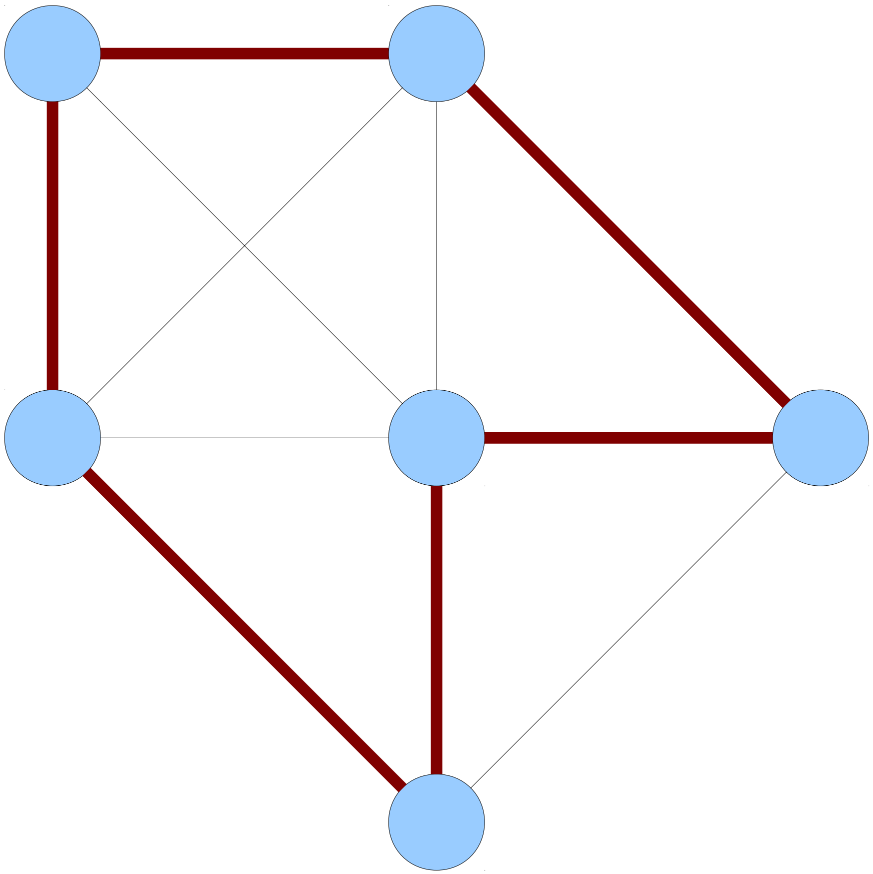
- A problem (which may or may not be a decision problem) is called **NP-hard** if (intuitively) it is at least as hard as every problem in **NP**.
- As before: no polynomial-time algorithms are known for any **NP**-hard problem.
- Vary wildly in difficulty: 3SAT and the halting problem are both **NP**-hard.

Combating **NP**-Hardness

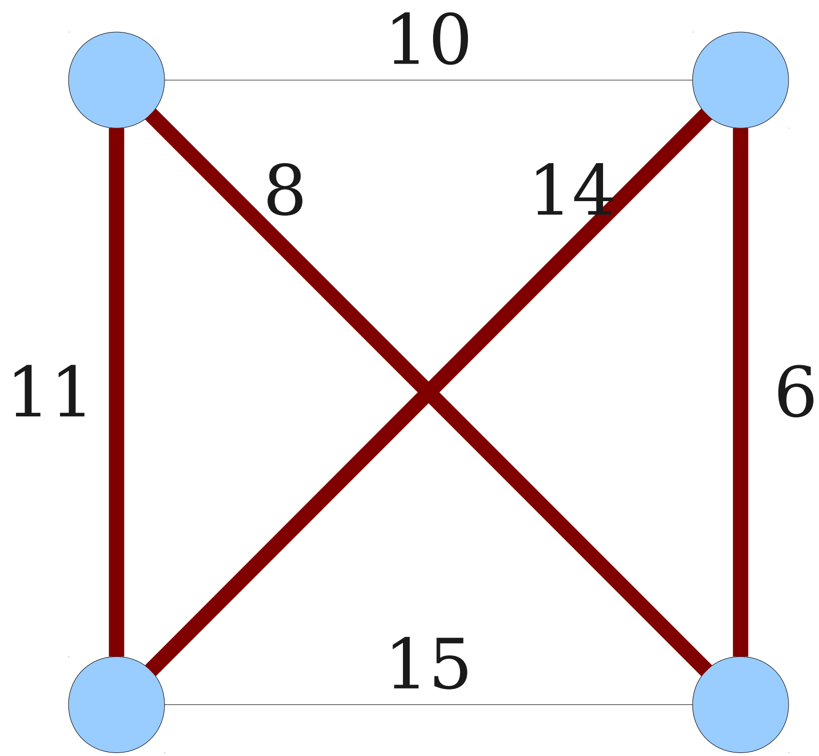
- Under the (commonly-held) assumption that $\mathbf{P} \neq \mathbf{NP}$, all **NP**-hard problems are intractable.
- However:
  - This ***does not*** mean that brute-force algorithms are the only option.
  - This ***does not*** mean that all instances of the problem are equally hard.
  - This ***does not*** mean that it is hard to get approximate answers.

Beating Brute Force:

Traveling Salesperson Problem



A **Hamiltonian cycle** in an undirected graph G is a simple cycle that visits every node in G.



Cost: **39**

(This is the optimal solution)

Given a complete, undirected, weighted graph G, the **traveling salesperson problem** (**TSP**) is to find a Hamiltonian cycle in G of least total cost.

TSP, Formally

- Given as input
  - A complete, undirected graph G, and
  - a set of edge weights, which are positive integers,

  the **TSP** is to find a Hamiltonian cycle in G with least total weight.
- Note that since G is complete, there has to be at least one Hamiltonian cycle. The challenge is finding the least-cost cycle.
- This problem is known to be **NP**-hard.

A Naïve Solution

- Option One: Try all possible Hamiltonian cycles in the graph.
- How many Hamiltonian cycles are there?
  - Answer: **$(n - 1)! / 2$**
- Spend $O(n)$ time processing each cycle.
- Total time: **$\Theta(n!)$**.
- *This is completely impractical!*

A Useful Observation

A Recurrence Relation

- Let OPT(v, S) be the minimum cost of an s – v path that visits exactly the nodes in S. We assume $v \in S$. Let $w(u, v)$ be the weight of the edge (u, v).
- **Claim:** OPT(v, S) satisfies the following recurrence:

$$\text{OPT}(v, S) = \begin{cases} 0 & \text{if } v = s \text{ and } S = \{s\} \\ \infty & \text{if } s \notin S \\ \min\limits_{u \in S - \{v\}} \{\text{OPT}(u, S - \{v\}) + w(u, v)\} & \text{otherwise} \end{cases}$$

Evaluating the Recurrence

$$\mathrm{OPT}(v,S)=\begin{cases} 0 & \text{if } v=s \text{ and } S=\{s\} \\ \infty & \text{if } s\notin S \\ \min\limits_{u\in S-\{v\}}\{\mathrm{OPT}(u,S-\{v\})+w(u,v)\} & \text{otherwise} \end{cases}$$

- Evaluating this recurrence when $|S| = k$ involves evaluating the recurrence on subproblems whose sets are of size k – 1.
- **Idea:** Evaluate the recurrence on sets of size 1, size 2, size 3, …, size n.
- **Note:** There are 2^n possible choices of a set S, of which 2^{n-1} contain s.

Evaluating the Recurrence

$$\mathrm{OPT}(v,S)=\begin{cases} 0 & \text{if } v=s \text{ and } S=\{s\} \\ \infty & \text{if } s\notin S \\ \min\limits_{u\in S-\{v\}} \{\mathrm{OPT}(u,S-\{v\})+w(u,v)\} & \text{otherwise} \end{cases}$$

Let DP be an $n \times 2^{n-1}$ table.

Set DP[s][{s}] = 0

For k = 2 to n:

  For all sets $S \subseteq V$ where $|S| = k$ and $s \in S$:

    For all $v \in S - \{s\}$:

      Set DP[v][S] = $\min_{u \in S - \{v\}}$ {DP[u][S – {v}] + $w(u, v)$}

Return $\min_{v \neq s}$ { DP[v][V] + $w(v, s)$ }

Analyzing the Runtime

Let DP be an $n \times 2^{n-1}$ table.

Set DP[s][{s}] = 0

For $k = 2$ to n:

  For all sets $S \subseteq V$ where $|S| = k$ and $s \in S$:

    For all $v \in S - \{s\}$:

      Set $DP[v][S] = \min_{u \in S - \{v\}} \{DP[u][S - \{v\}] + w(u, v)\}$

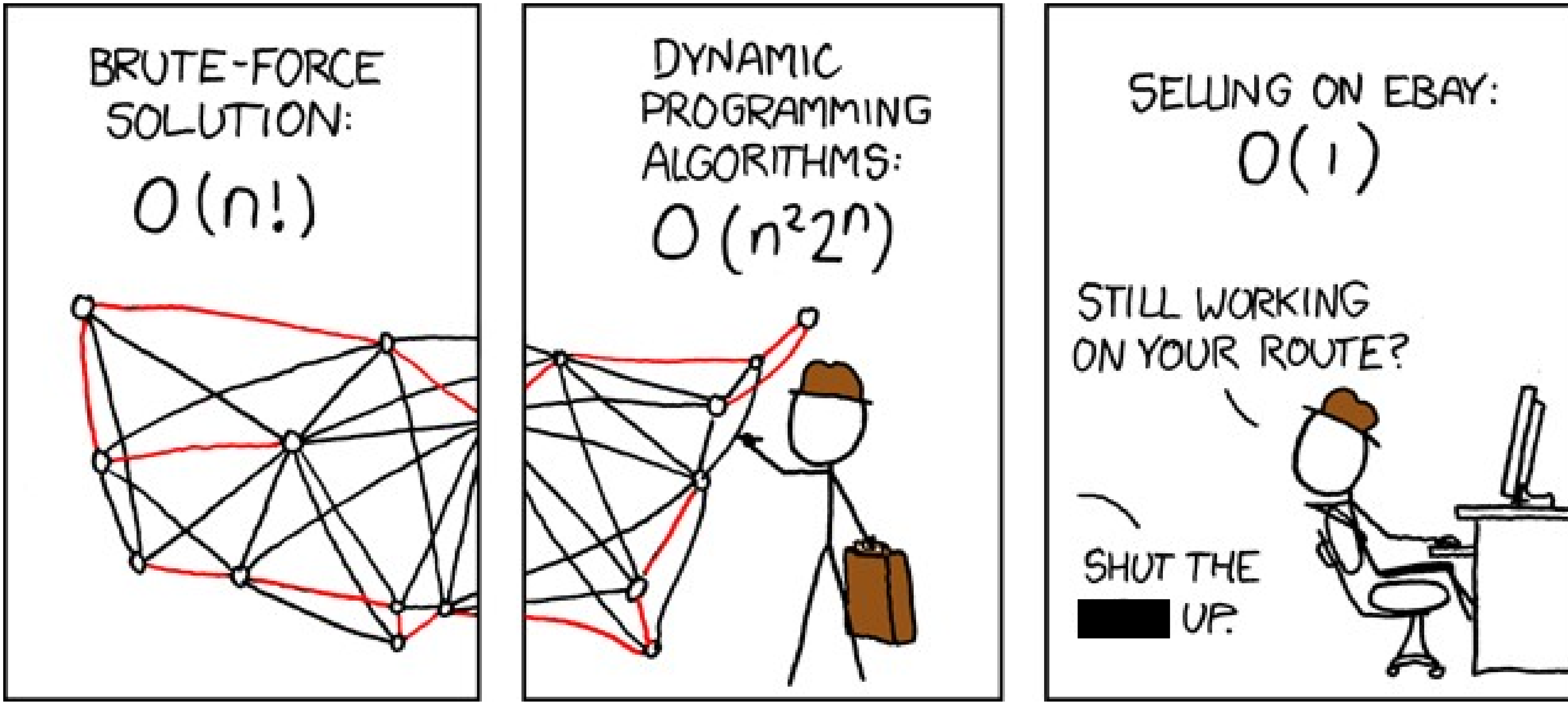
Return $\min_{v \neq s} \{ DP[v][V] + w(v, s) \}$

Storing Sets

- Each subset of V containing s can be mapped to a unique integer in 0, 1, 2, …, $2^{n-1} - 1$.
  - Idea: Treat the number as a bitvector where present elements are 1s and absent elements are 0s. Exclude s from the bitvector.
- Notice: each subproblem depends on many subproblems, but each subproblem references the same set.
- In time O(n), compute the above number and use it to quickly index into the table. This requires only O(n) overhead per subproblem.

To Summarize

- $O(2^n n)$ total subproblems.
- Can generate all subsets in ascending order of size, producing each subset in time $O(n)$.
- Solving each subproblem requires us to look at $O(n)$ different subproblems, doing $O(1)$ work for each.
- Tricky part: need to be able to index subproblems with a set. Can map all subsets of V to numbers in the range $0, 1, 2, \ldots, 2^n - 1$ spending $O(n)$ time per mapping.
- Thus $O(n)$ time per subproblem and $O(2^n n)$ subproblems, so total time is $\mathbf{O(2^n n^2)}$.



http://xkcd.com/399/

Why This Matters

- Compare $15!$ and $2^{15} \cdot 15^2$:

$$15! \approx 1.31 \times 10^{12}$$

$$2^{15} \cdot 15^2 \approx 7.4 \times 10^6$$

- Compare $25!$ and $2^{25} \cdot 25^2$:

$$25! \approx 1.65 \times 10^{25}$$

$$2^{25} \cdot 25^2 \approx 2.1 \times 10^{10}$$

- Compare $30!$ and $2^{30} \cdot 30^2$:

$$30! \approx 2.7 \times 10^{32}$$

$$2^{30} \cdot 30^2 \approx 9.7 \times 10^{11}$$

Why This Matters

- Improving upon brute-force increases the sizes of the problems for which we can get exact answers.
- Problems exist for which we can get exact answers for decently large inputs using optimized exponential-time algorithms.
- You can use the techniques from this course to design exponential-time algorithms!

Next Time

- Parameterized Complexity
- Pseudopolynomial-Time Algorithms