香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

Complexity Classes

Complexity Classes

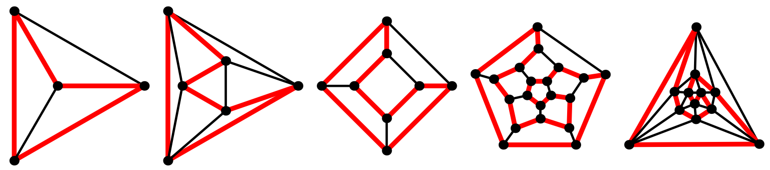
- What happens if you can't find an efficient algorithm? Is it your "fault" or the problem's?

- Showing that a problem has an efficient algorithm is, relatively, easy. "All" that is needed is to demonstrate an algorithm.

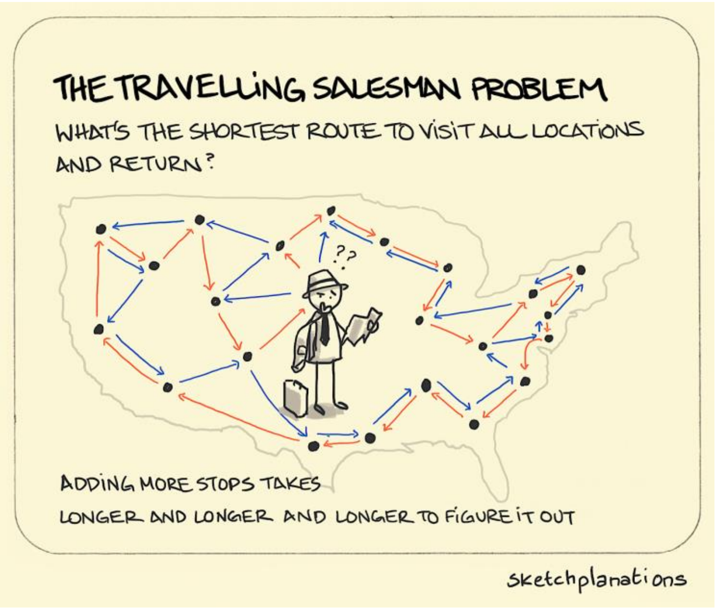
- Proving that no efficient algorithm exists for a particular problem is difficult. How can we prove the non-existence of something?

Decision vs Optimization Problem

- Decision problem: A decision problem is a problem that has two possible answers, yes and no.
  - E.g. Hamiltonian Cycle Problem: Given a graph, is there a cycle that visits every vertex exactly once.



- Optimization Problem: An optimization problem requires an answer that is an optimal configuration.
  - Traveling Salesman Problem: Given a weighted graph, find a Hamiltonian cycle with the smallest total weight



Decision vs Optimization Problem

- Optimization problems have decision versions.
  - Traveling Salesperson Problem: Given a weighted graph and a value W, is there a Hamiltonian cycle with a total weight $\leq W$?

- Complexity classes are usually defined for decision problems. Hard decision version implies hard optimization version.

Complexity Classes

The Theory of Complexity deals with

- the classification of certain "decision problems" into several classes:
  - the class of "easy" problems,
  - the class of "hard" problems,
  - the class of "hardest" problems;
- relations among the three classes;
- properties of problems in the three classes.

Question: How to classify decision problems

Answer: Use "polynomial-time algorithms."

Input Size

- Input size of problems: The input size of a problem is the minimum number of bits ({0, 1}) needed to encode the input of the problem

- Examples:
  - Given a positive integer n, are there integers $j, k > 1$, such that $n = jk$?
  - Sort n integers $a_1, a_2, \ldots, a_n$

Input size?

Input Size

- Composite Number
  - In a binary number system, any integer $n > 0$ can be represented as

$$\sum_{i=0}^{k} c_i 2^i, \text{ where } k = \lceil \log_2(n+1) - 1 \rceil$$

and hence represented by the string $c_0 c_1 \dots c_k$ of length $\lceil \log_2(n+1) \rceil$.

- Sorting
  - Fixed length encoding writes a_i as a binary string:

$$m = \left\lceil \log_2 \max_i (a_i + 1) \right\rceil$$

  - Input size: nm

- Running times of algorithms, unless otherwise specified, should be expressed in terms of input size.

Input Size

- Two positive functions $f(n)$ and $g(n)$ are of the same type if
$$c_1 g(n^{a_1})^{b_1} \leq f(n) \leq c_2 g(n^{a_2})^{b_2}$$
for all large n, where $a_1, b_1, c_1, a_2, b_2, c_2$ are some positive constants.

- E.g., Polynomials are of the same type, but polynomials and exponentials are of different types.

- s: actual input size. Similarly, any t satisfying $s^{a_1} \leq t \leq s^{a_2}$ for some positive constants a_1 and a_2, can also be used as a measure of the input size of the problem.

Input Size

- E.g.,

- Graph problems: For many graph problems, the input is a graph $G = (V, E)$. What is the input size?
- A natural choice: There are n vertices and e edges. So we need to encode $n + e$ objects. With fixed length coding, the input size is

$$(n + e)\lceil \log_2(n + e + 1)\rceil.$$

- Since

$$[(n + e)\lceil \log_2(n + e + 1)\rceil]^{1/2} \le n + e \le (n + e)\lceil \log_2(n + e + 1)\rceil,$$

we may use $n + e$ as the input size.

Polynomial-Time Algorithms

Definition: An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem that the algorithm solves.

Examples:

The standard multiplication algorithm learned in school has time $O(m_1 m_2)$ where m1 and m_2 are, respectively, the number of digits in the two integers

DFS has time $O(n + e)$.

Kruskal's MST algorithm runs in time $O((e+n)\log n)$.

Polynomial-time Algorithm

- Definition: An algorithm is polynomial-time if its running time is $O(n^k)$, where k is a constant independent of n, and n is the input size of the problem.

- Examples: The integer multiplication problem, and the cycle detection problem for undirected graphs.

- Tractable Problems: Problems that can be solved in polynomial time.

Nonpolynomial-Time Algorithms

- Definition: An algorithm is non-polynomial-time if the running time is not $O(n^k)$ for any fixed k ≥ 0.

- Example: Counting up to m
  - Input: n bits. Problem: count from 1 up to the number m represented by these n bits.
  - Needs $\Theta(m) = \Theta(2^n)$ time.

- Intractable Problems: Problems that cannot be solved in polynomial time.

Polynomial- vs. Nonpolynomial-Time

Nonpolynomial-time algorithms are *impractical*

For example, to run an algorithm of time complexity $O(2^n)$ for $n = 100$ on a computer which does 1 Terraoperation (10^{12} operations) per second: It takes $2^{100}/10^{12} \approx 10^{18.1}$ seconds $\approx 4 \cdot 10^{10}$ years

For the sake of our discussion of complexity classes Polynomial-time algorithms are ”practical”

Note: in reality an $O(n^{20})$ algorithm is not really practical.

The Class P

- Definition: The class P consists of all decision problems that are solvable in polynomial time
- How to prove that a decision problem is in P?

  You need to find a polynomial-time algorithm for this problem
- How to prove that a decision problem is not in P?

  You need to prove there is no polynomial-time algorithm for this problem (much harder).

Unsolvable Problems?

- Are there computational problems that **cannot be solved**?

- HALTING: Does python program P on input X terminate?
  - `while True: continue` → does not terminate for any input
  - `print "Hello World!”` → terminates for any input.
- Suppose there exists an algorithm H solving HALTING, i.e.,
  - H(P,X)=1 if program P on input X terminates.
  - H(P,X)=0 if program P on input X runs forever.

Programs as Inputs

Programs can be inputs to programs

- P= `print 'Hello, world!’.`
- P=01101011101001010110100010101001……

- A(X): the result of a program A that runs on X = input.
- A(X) is well-defined as a function of both A, X.
  - A(X) might look at a piece of the bit sequence X.
  - Then A(P) is also well defined, P = program.

Halting Problem

- Can we ever build this program H?
  - H(P,X) = 1 if program P on input X terminates.
  - H(P,X) = 0, if program P on input X runs forever.

- Suppose we do! We show that this will lead to a contradiction.
- Assume that H(P,X) always gives the correct result.

Halting Problem

- H(P,X)=1, if program P on input X terminates.
- H(P,X)=0, if program P on input X runs forever.

Program A(P)

```
# Input: Program P
# Uses the code of program H

if H(P,P)=1, then enter infinite loop
else if H(P,P)=0, then stop
```

Halting Problem

- H(P,X)=1, if program P on input X terminates.
- H(P,X)=0, if program P on input X runs forever.

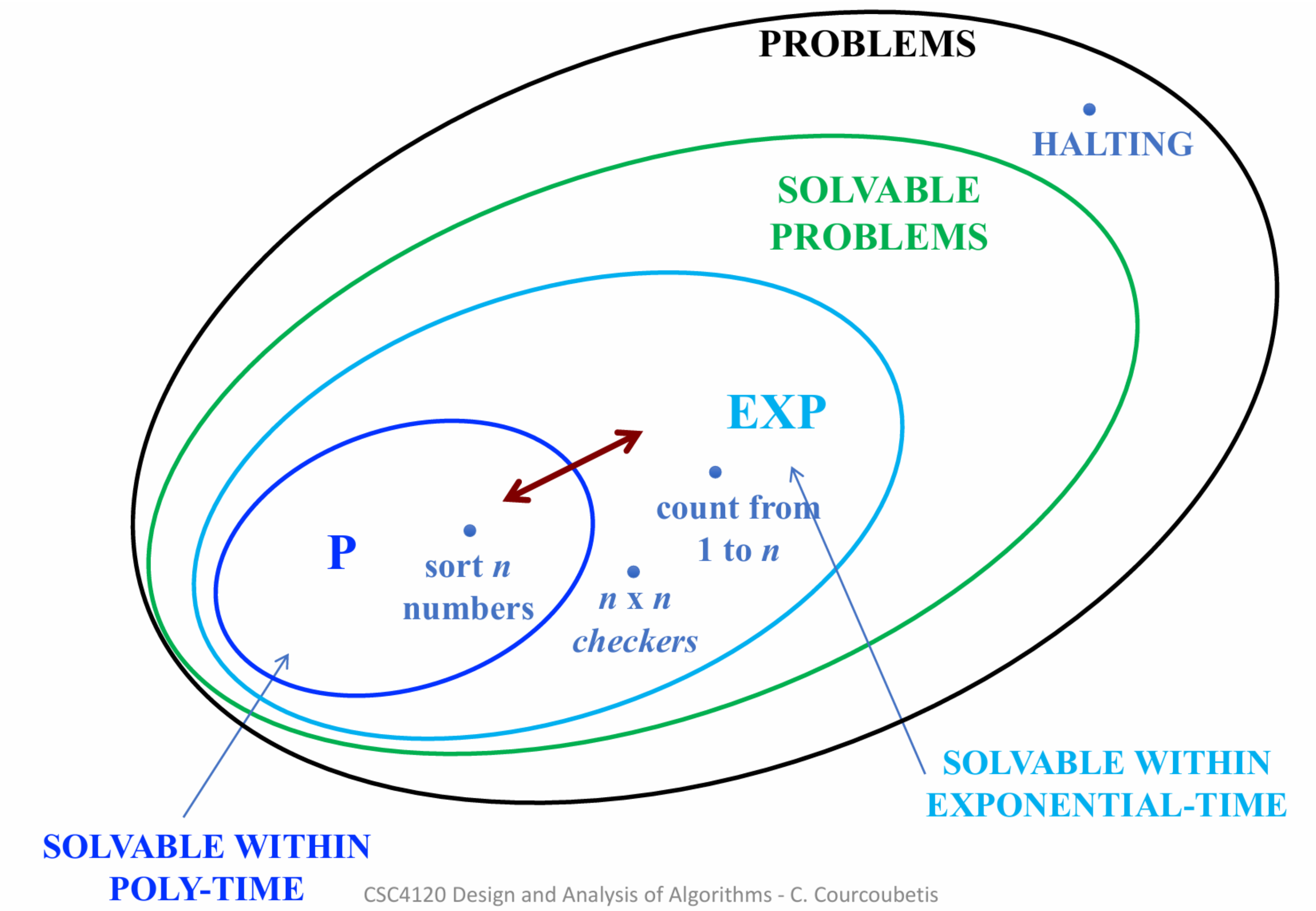
```
Program A(P)

if H(P,P)=1, then enter infinite loop
else if H(P,P)=0, then stop
```

- Question: Does A(A) terminate?
  - If it does, H(A,A)=0 -> A(A) runs forever.
  - If it does not, H(A,A)=1, A(A) terminates.

Contradiction!

Classification of Problems



Certificates and Verifying Certificates

- Almost ready to introduce NP. Some important concepts:

- A decision problem is usually formulated as

  Is there an object satisfying some conditions?

- A Certificate is a specific object satisfying the conditions (exists only for yes-inputs by definition).

- Verifying a certificate: Check that the given object (certificate) satisfies the conditions (that is, verifying that the input is yes-input).

Certificates and Verifying Certificates

香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

- Example:

Is given positive integer n composite

Certificate: an integer a dividing n such that 1 < a < n

Verifying a certificate: Given a certificate a, check whether a divides n. This can be done in time $O((\log_2 n)^2)$(recall that input size is $\log_2 n$ so this is polynomial in input size)

Certificates and Verifying Certificates

- Hamiltonian Cycle: Input is a graph G = (V, E). A cycle of graph G is called Hamiltonian if it contains every vertex exactly once

- Decision problem: DHamCyc

  Does G have a Hamiltonian cycle?

- Certificate: an ordering of the n vertices in G (corresponding to their order along the Hamiltonian Cycle)

- Verification: Given a certificate the verification algorithm checks whether it is a Hamiltonian cycle of G by simply checking whether all of the edges appear in the graph.

The Class NP

- Definition: The class NP consists of all decision problems such that, for each yes-input, there exists a certificate that can be verified in polynomial time.

- Example: DHamCyc ∈ NP. (As shown earlier, there is a polynomial time algorithm to verify a certificate.)

NP stands for "Nondeterministic Polynomial time".

P = NP?

One of the most important problems in computer science:

$$P = NP \text{ or } P \neq NP$$

Put another way, is every problem that can be verified in polynomial time also decidable in polynomial time?

At first glance it seems “obvious” that P ≠ NP; after all, deciding a problem is much more restrictive than verifying a certificate

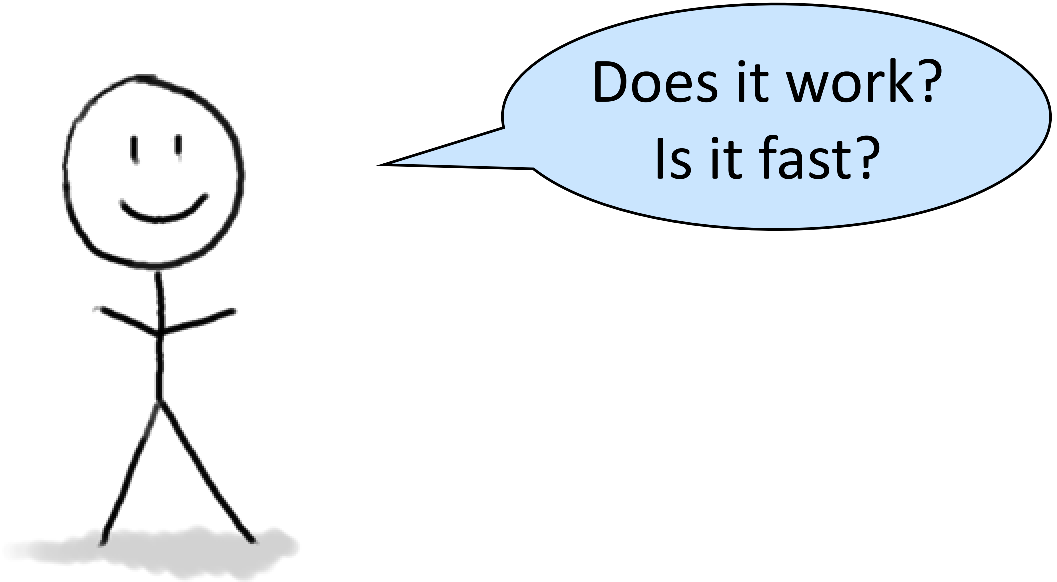
Still do not know the answer…

Course Review

Course Review

So far we have learned:

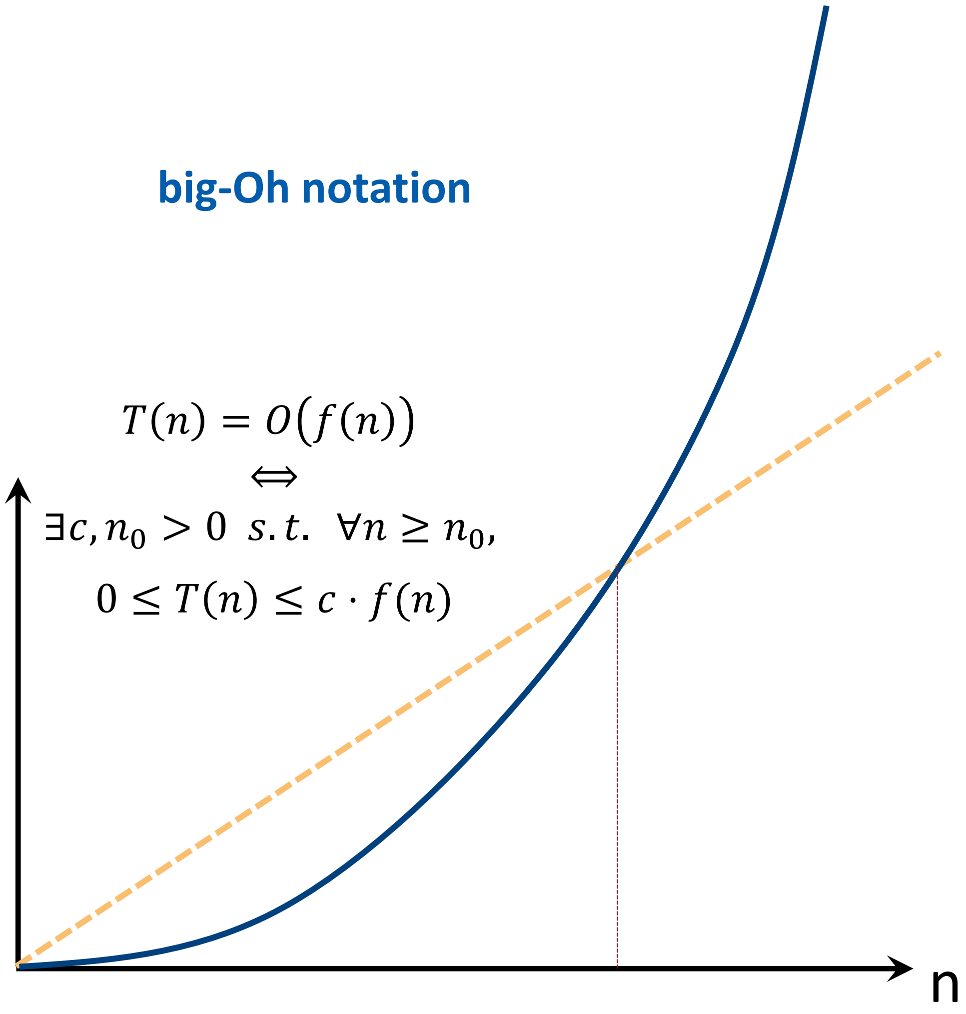
Can I do better?



Algorithm designer

big-Oh notation

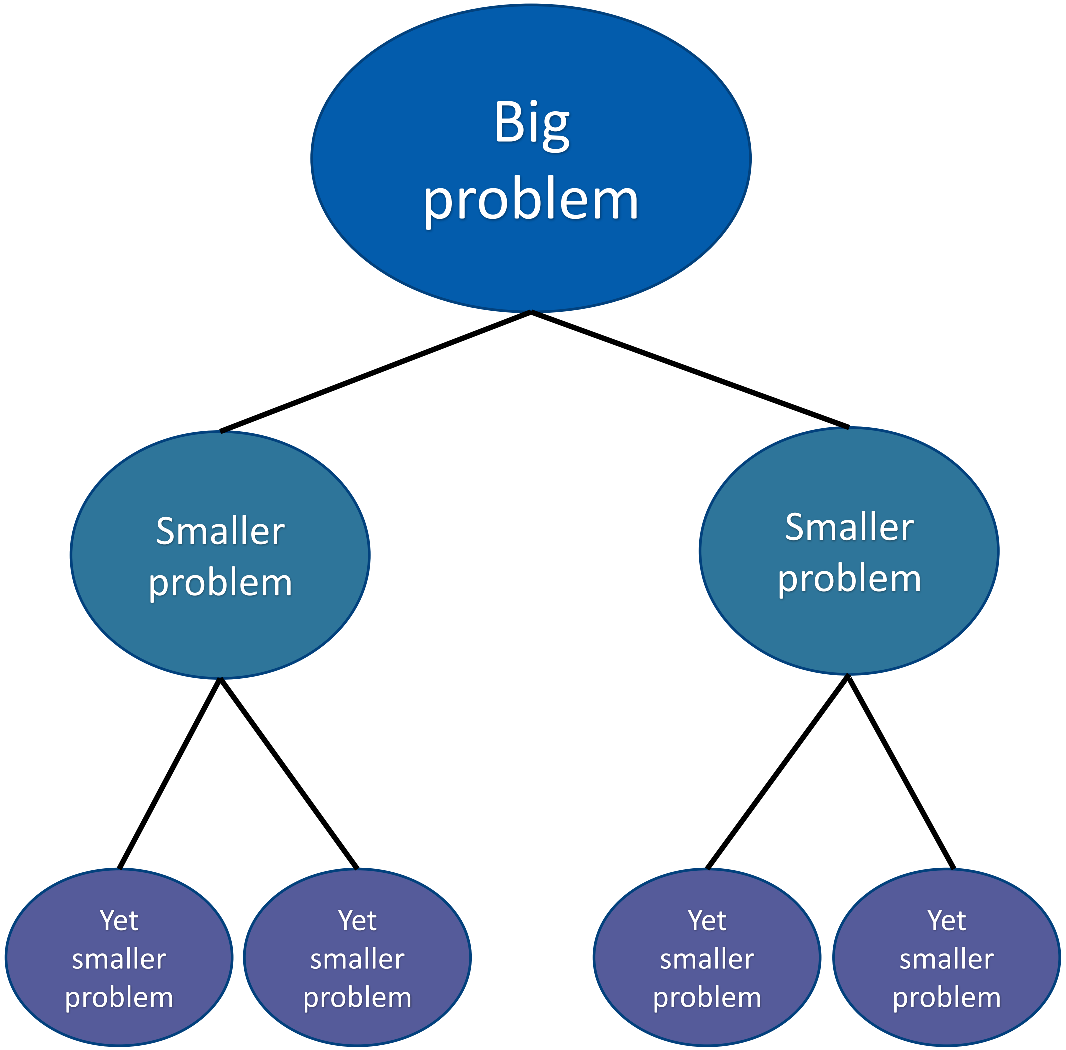
$$T(n) = O\big(f(n)\big)$$
$$\Leftrightarrow$$
$$\exists c, n_0 > 0 \;\; s.t. \;\; \forall n \geq n_0,$$
$$0 \leq T(n) \leq c \cdot f(n)$$



Algorithm Design Paradigm

Divide and Conquer

Like MergeSort



Algorithm Design Paradigm

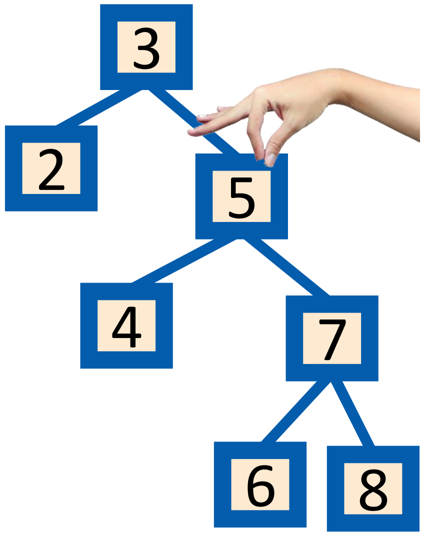
香港科技大学(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

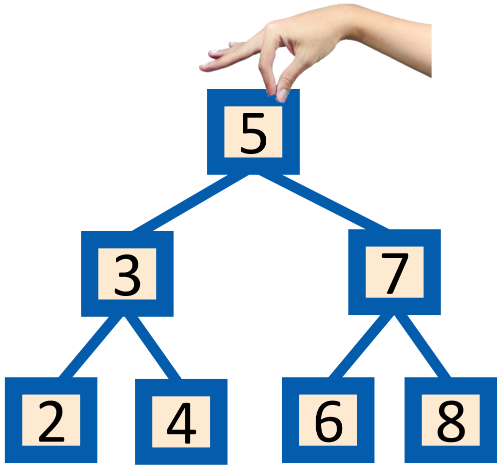
Why not use randomness?

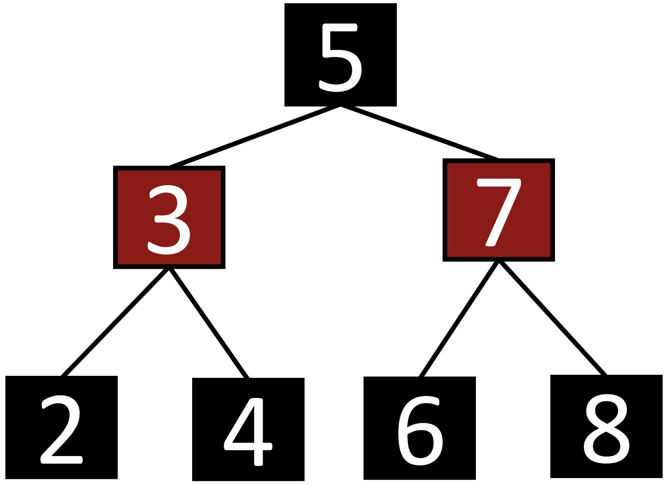
- QuickSort
- Still worst-case input, but we use randomness after the input is chosen.
- Always correct, usually fast.

Beyond Sorted Arrays / Linked Lists

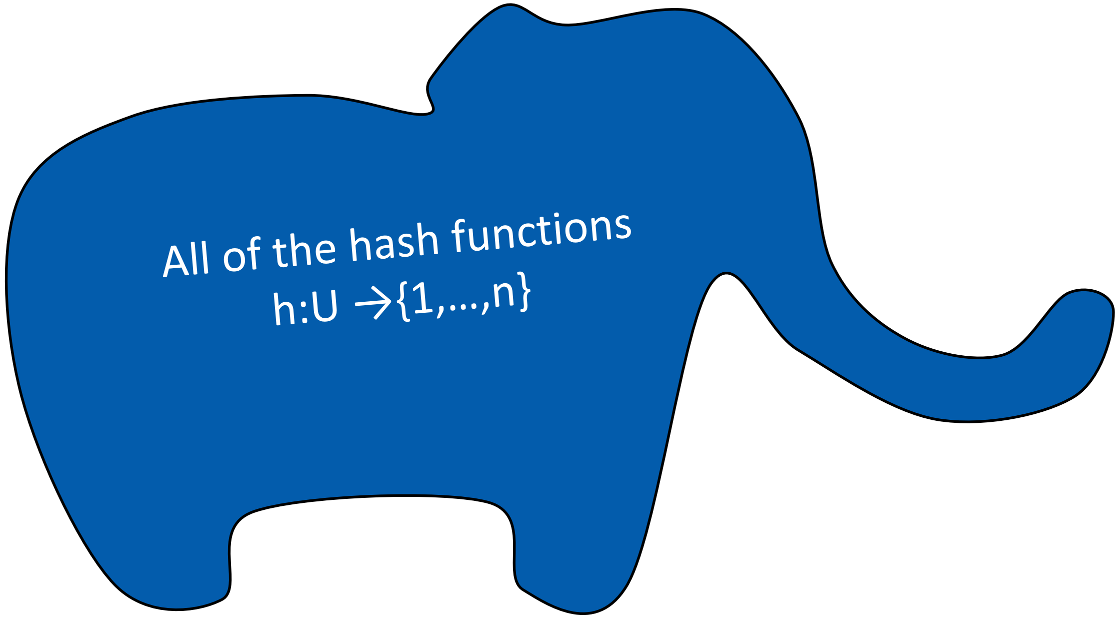
- Binary Search Trees
- AVL Trees
- RB Trees







Another Way to Store Things



Choose h randomly from a universal hash family.

It's better if the hash family is small! Then it takes less space to store h.

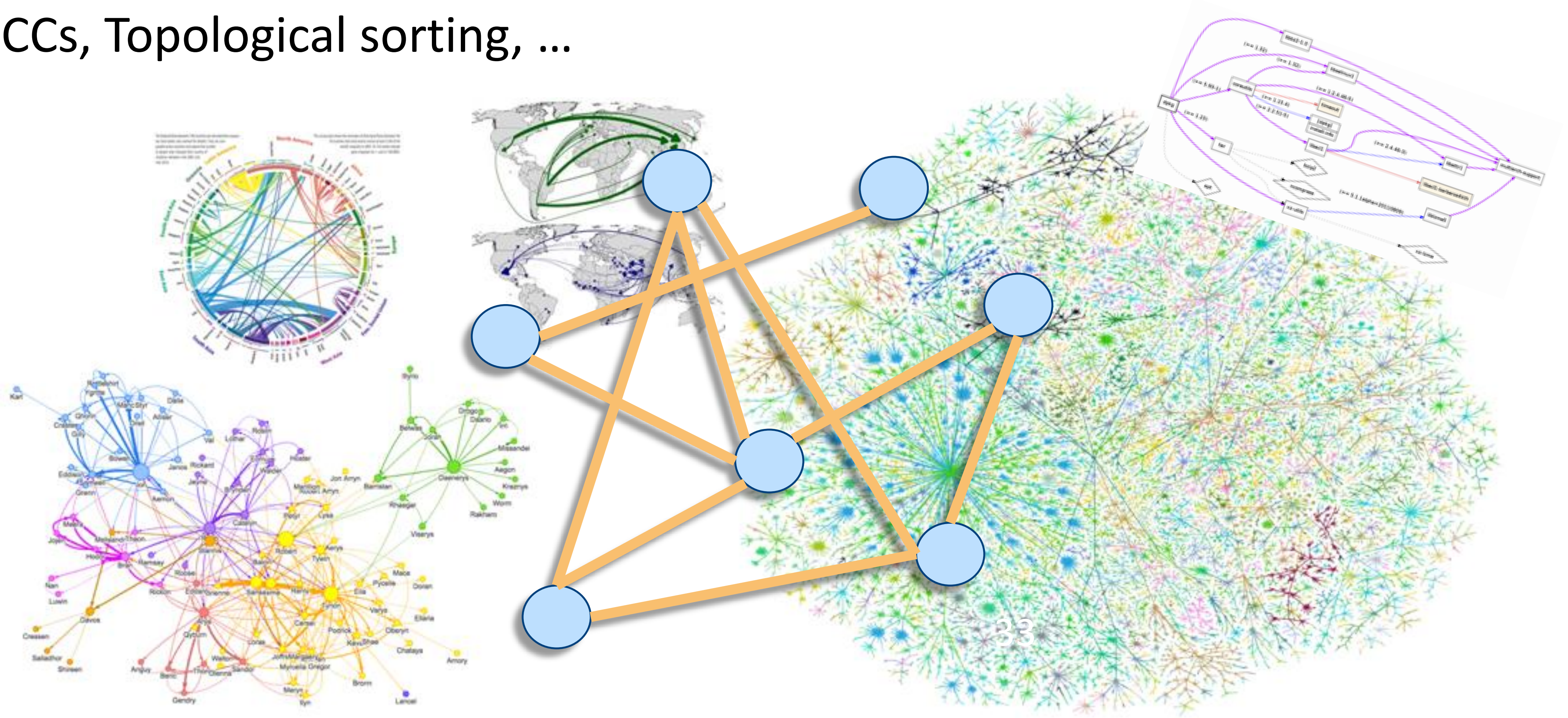
The universe

hash function h

Some buckets

Graphs

- BFS, DFS, and applications!
- SCCs, Topological sorting, …

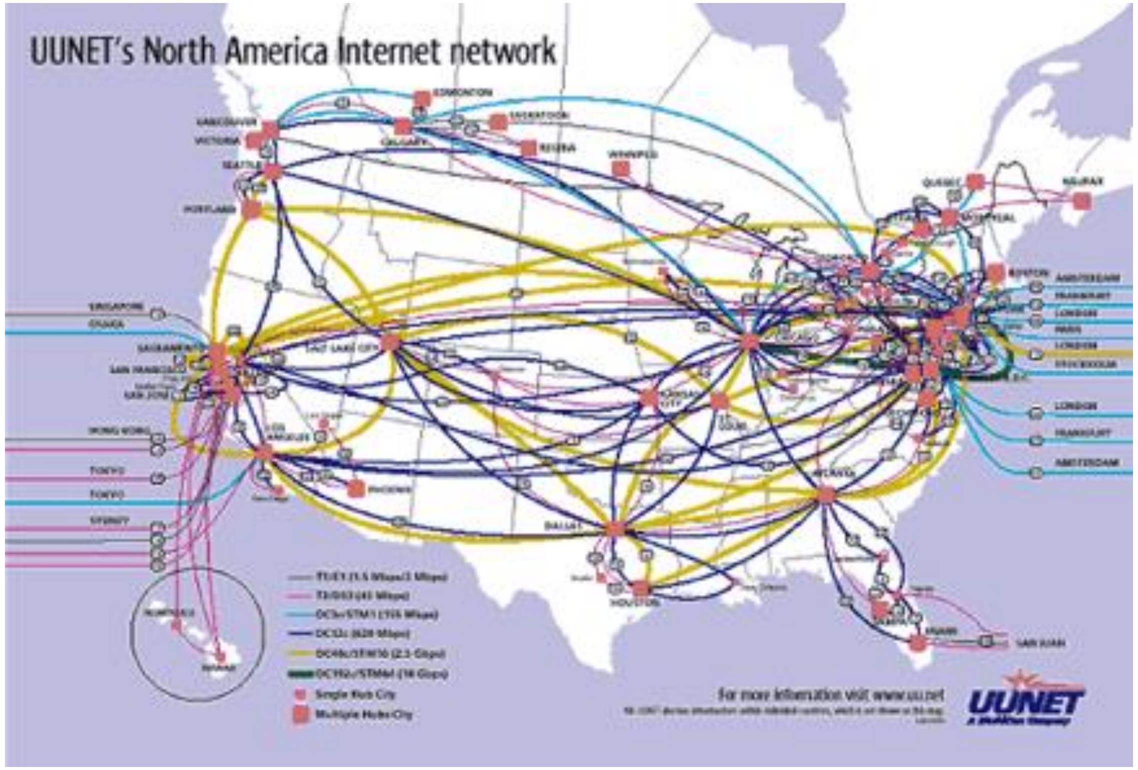


Graphs

- A fundamental graph problem:

Shortest Paths

- E.g., transit planning, packet routing, …
- Dijkstra!
- Bellman-Ford!
- Floyd-Warshall!



Dynamic Programming

Bellman-Ford and Floyd-Warshall were examples of…

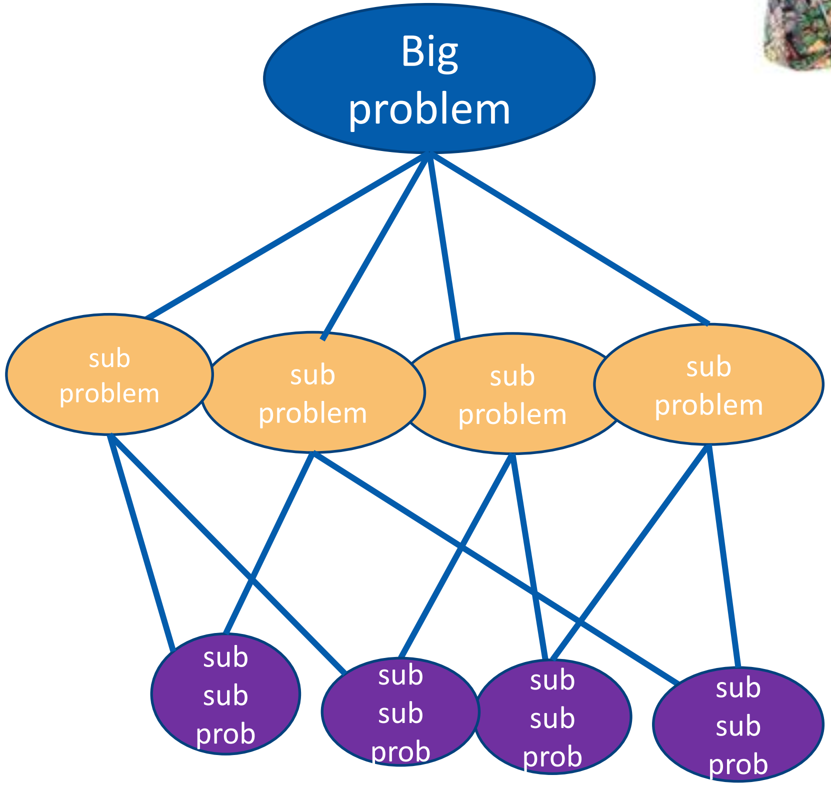
- Not programming in an action movie.

Instead, an algorithmic paradigm!

Dynamic Programming!

We saw many other examples, including Longest Common Subsequence and Knapsack Problems.

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a
  for the value of the optimal solution.
- **Steps 3-5:** Use dynamic programming: fill in a table to find the answer!

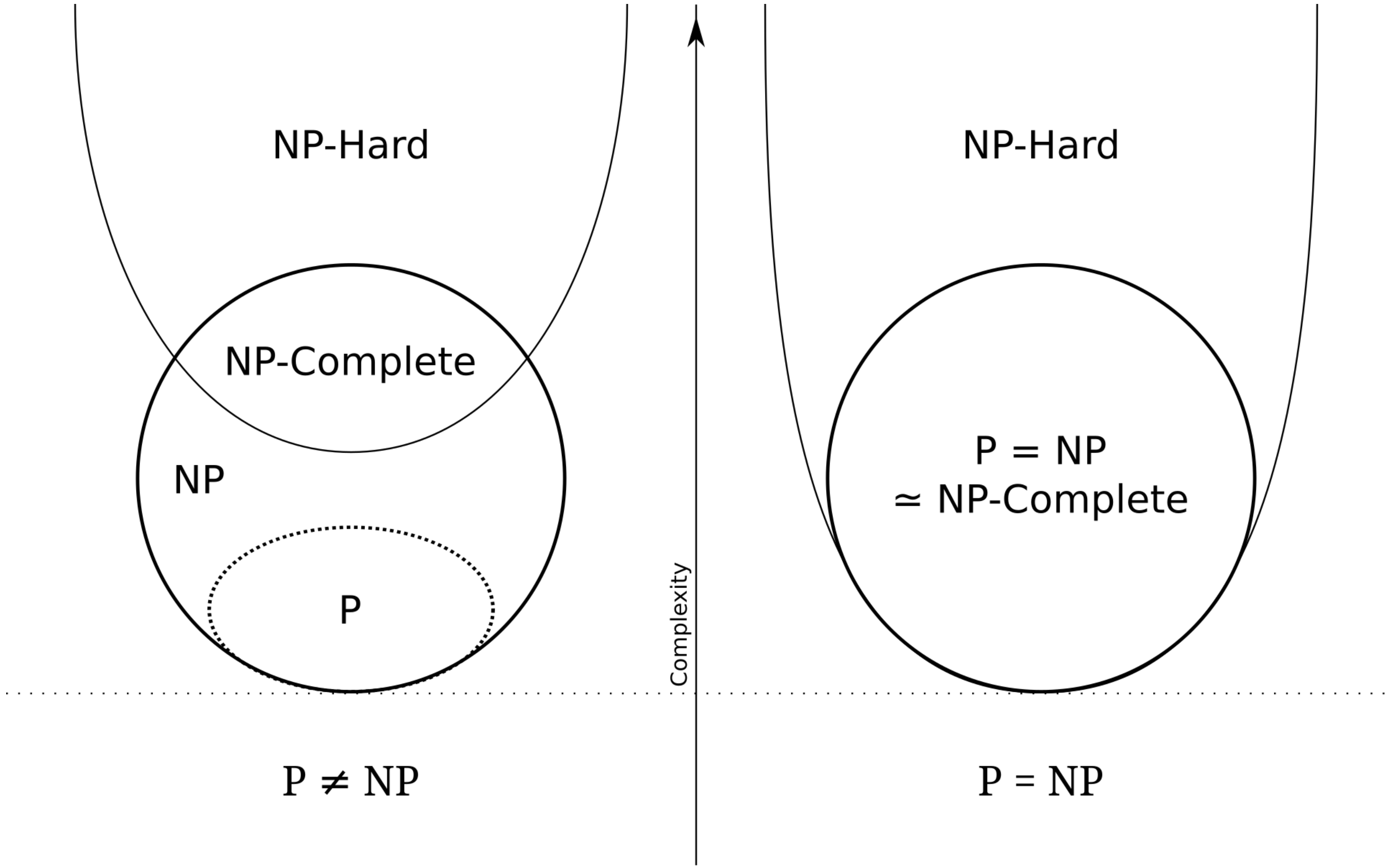


Greedy

- Make a series of choices, and commit!

- Intuitively we want to show that our greedy choices never rule out success.
- Rigorously, we usually analyzed these by induction.
- Examples!
  - Activity Selection
  - Job Scheduling
  - Minimum Spanning Trees

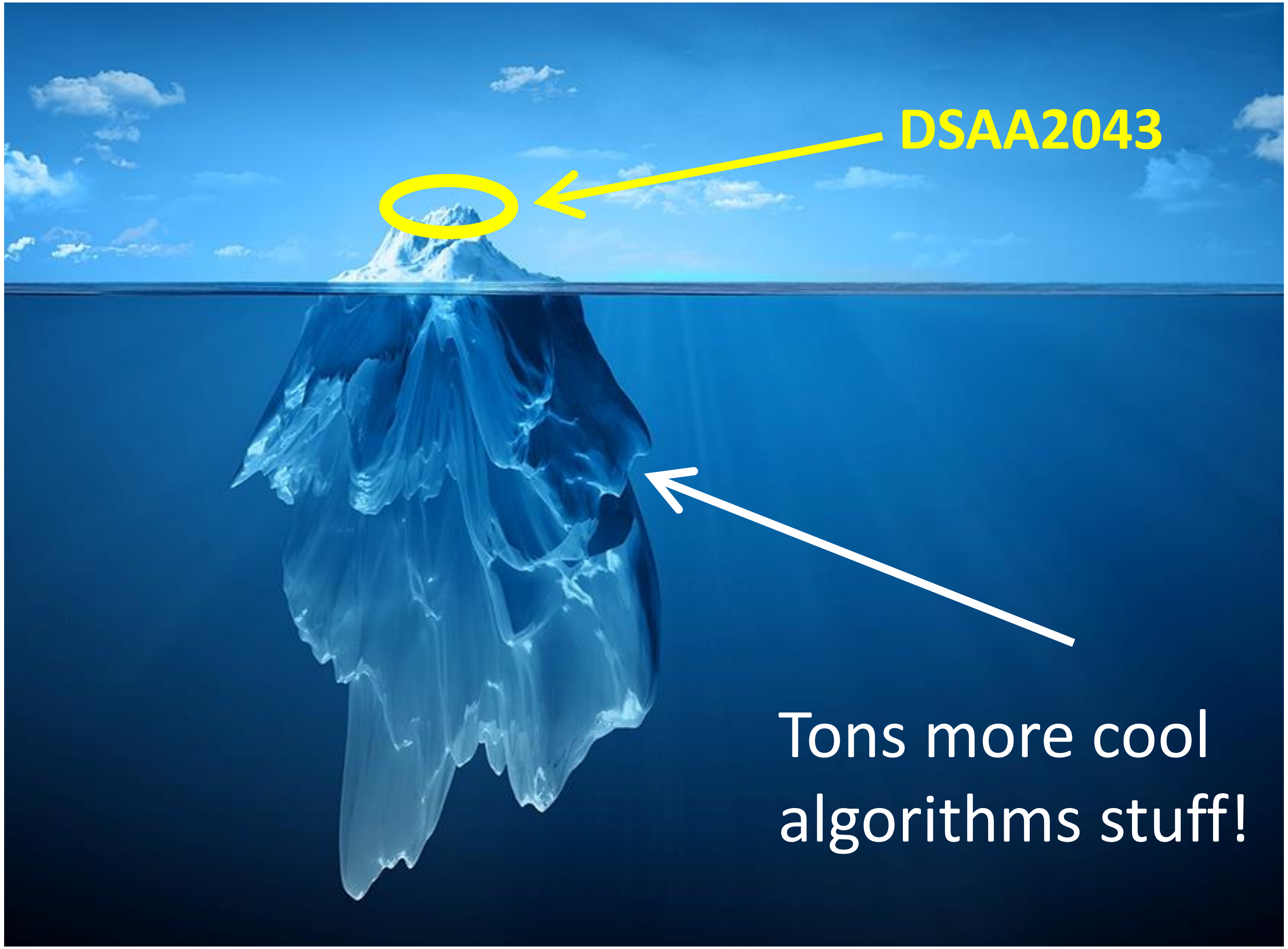
P = NP?


NP-Hard
NP-Complete
NP
P
P ≠ NP
Complexity
NP-Hard
P = NP
≃ NP-Complete
P = NP

What have we learned

- A few algorithm design paradigms:
  - Divide and conquer, dynamic programming, greedy
- A few analysis tools:
  - Worst-case analysis, asymptotic analysis, recurrence relations, probability tricks, proofs by induction
- A few common objects:
  - Graphs, arrays, trees, hash functions
- A LOT of examples!

What have we learned



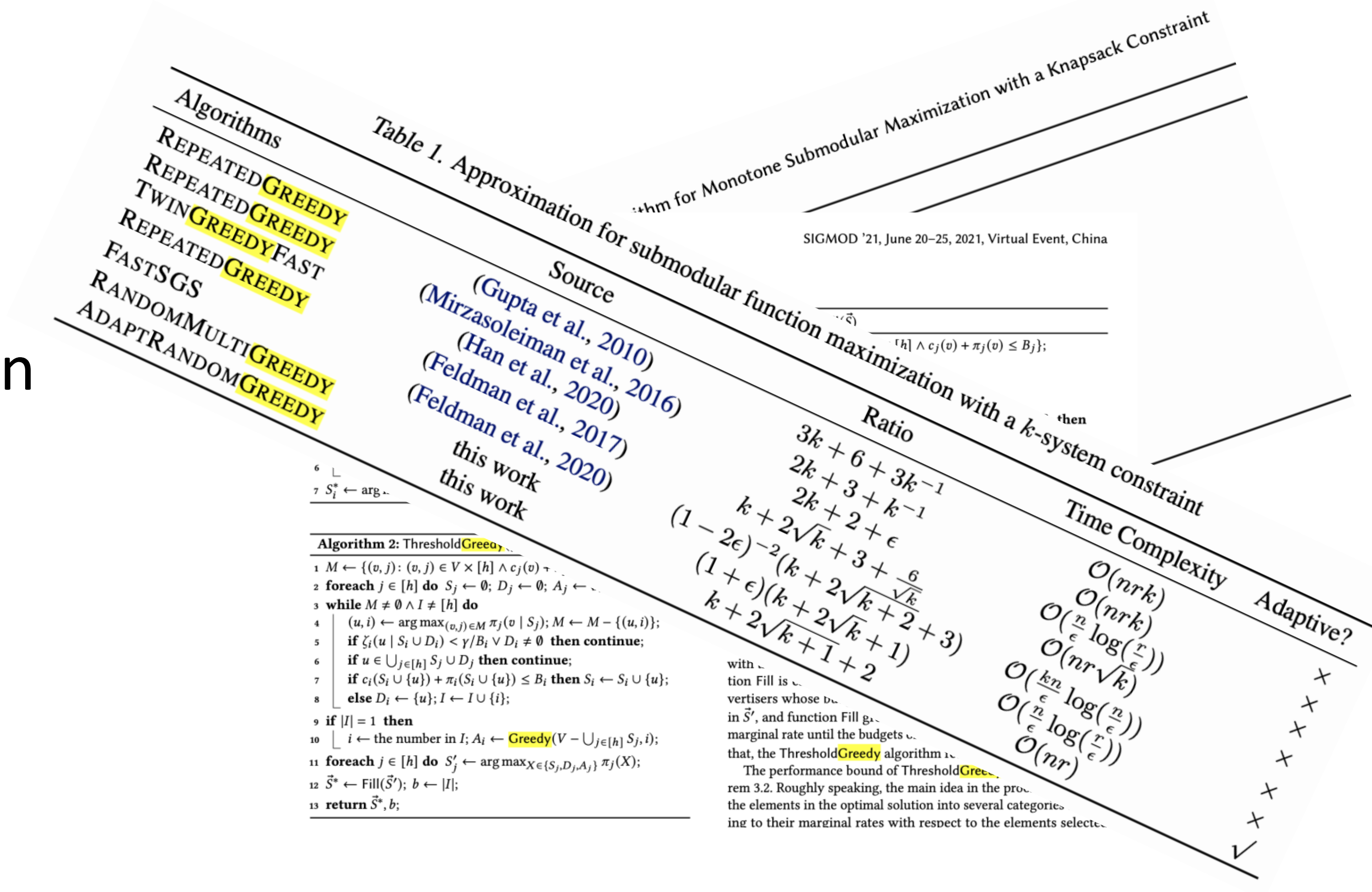
The Road Ahead – Advanced Topics

Greedy Approximation and Submodular Optimization

Previous, we use greedy to obtain optimal solution.

Greedy is more powerful than you can think of.

An important strategy to design approximation algorithms.



Submodular Function

Consider a function f defined over all subsets of a set X, i.e., 2^X. f is called a *submodular function* if for any two subsets A and B of X,

$$f(A) + f(B) \geq f(A \cup B) + f(A \cap B).$$

f is said to be monotone nondecreasing if for any two subsets A and B of X,

$$A \subset B \rightarrow f(A) \leq f(B)$$

香港科技大學(广州)
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY (GUANGZHOU)

Another expression

A set function $f: 2^X \rightarrow R$ is submodular if and only if for any two subsets A and B with $A \subseteq B$ and for any $x \in X \setminus B$,

$$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B).$$

Prove it!

Another expression

Proof

f being submodular $\rightarrow f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B)$.

Assume that f is submodular. Consider two subsets A and B with $A \subseteq B$ and an element $x \in X \setminus B$. We have

$$\begin{aligned} f(A \cup \{x\}) + f(B) &\geq f(A \cup \{x\} \cup B) + f\big((A \cup \{x\}) \cap B\big) \\ &= f(B \cup \{x\}) + f(A) \end{aligned}$$

Rearranging the terms leads to the result.

Greedy Approximation and Submodular Optimization

Proof ctd.

$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\}) - f(B) \to f$ is submodular.

Consider two subsets U and V. Suppose $U \setminus V = \{y_1, y_2, \dots, y_k\}$.
Denote $Y_i = \{y_1, \dots, y_i\}$.

$$f(U) - f(U \cap V) = \sum_{i=0}^{k-1} f\big((U \cap V) \cup Y_i \cup \{y_{i+1}\}\big) - f\big((U \cap V) \cup Y_i\big)$$

$$\geq \textstyle\sum_{i=0}^{k-1}\big(f(V \cup Y_i \cup \{y_{i+1}\}) - f(V \cup Y_i)\big) = f(U \cup V) - f(V).$$

Rearranging the terms obtains the result.

Submodular Maximization

$$
\begin{aligned}
\max \quad & f(S) \\
\text{subject to} \quad & |S| \leq k, \\
& S \in 2^X,
\end{aligned}
$$

where f is a monotone nondecreasing submodular function over 2^X with $f(\emptyset) = 0$.

An example:

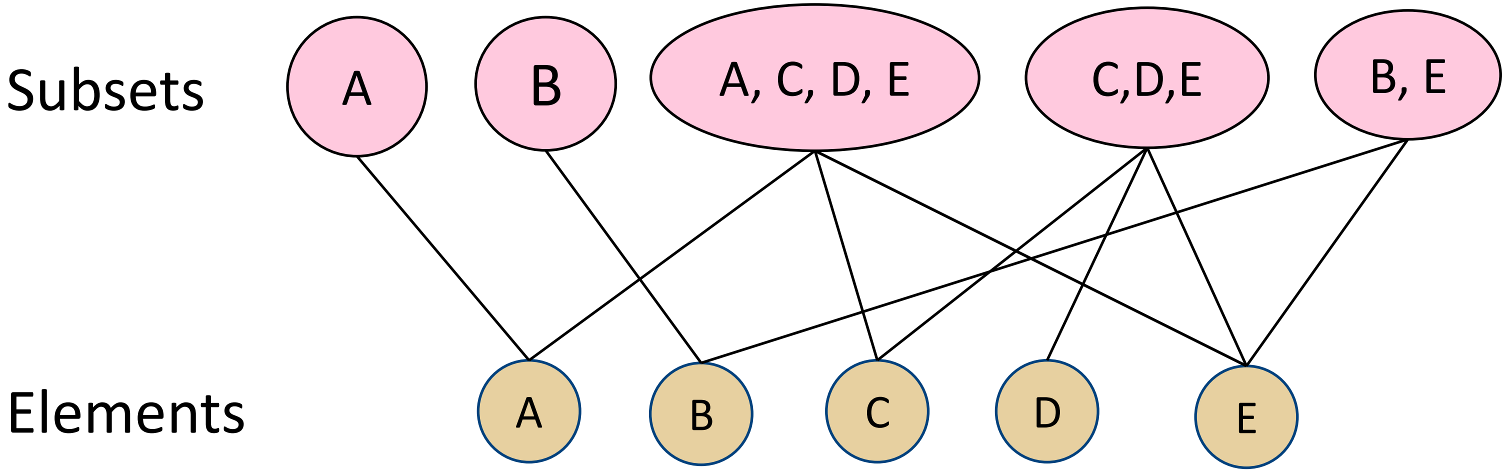
Maximum Set Coverage

Given a collection $\mathcal{C}$ of subsets of a finite set X and a positive integer k, find k subsets from $\mathcal{C}$ to cover the maximum number of elements in X.

For any subcollection $\mathcal{A}$ of $\mathcal{C}$, define

$$\mu(\mathcal{A}) = |\cup_{S\in\mathcal{A}} S|$$

$$\begin{aligned} \max \quad & \mu(\mathcal{A}) \\ \text{subject to} \quad & |\mathcal{A}| \le k, \\ & \mathcal{A} \in 2^{\mathcal{C}}, \end{aligned}$$



Submodular Maximization

Algorithm 1 Greedy(k, f): general greedy algorithm.

Input: k: size of returned set; f: monotone and submodular set function

Output: selected subset

1: initialize $S \leftarrow \emptyset$
2: **for** $i = 1$ to k **do**
3: $\quad u \leftarrow \mathrm{argmax}_{w \in V \setminus S} (f(S \cup \{w\}) - f(S))$
4: $\quad S \leftarrow S \cup \{u\}$
5: **end for**
6: **return** S

Theorem

Let $S^* = \text{argmax}_{|S| \le k} f(S)$ be the set maximizing $f(S)$ among all sets with size at most k. If $f(\cdot)$ is monotone and submodular and $f(\emptyset) = 0$, then for the set S_g computed by Greedy(k, f), we have

$$f(S_g) \ge \left(1 - \frac{1}{e}\right) f(S^*)$$

Submodular Maximization

Proof.

Define $\Delta(x|S) = f(S \cup \{x\}) - f(S)$

Suppose $x_1, x_2, \dots, x_k$ are obtained by the greedy algorithm.

$S_i = \{x_1, x_2, \dots, x_i\}$ for $i = 1, \dots, k$ and $S_0 = \emptyset$.

Assume that the optimal solution is $S^* = \{u_1, u_2, \dots, u_k\}$.

For $i = 0, 1, \dots, k-1$, we have

$$f(S^*) \le f(S^* \cup S_i)$$

Submodular Maximization

$$
\begin{aligned}
f(S^*) &\le f(S^* \cup S_i) \\
&= f(S_i) + \Delta(u_1|S_i) + \cdots + \Delta(u_k|S_i \cup \{u_1, \ldots, u_k\}) \\
&\le f(S_i) + \Delta(u_1|S_i) + \Delta(u_2|S_i) + \cdots + \Delta(u_k|S_i) \\
&\le f(S_i) + k \cdot \Delta(x_{i+1}|S_i)
\end{aligned}
$$

Submodular Maximization

$$f(S_k) \geq \left(1 - \frac{1}{k}\right) f(S_{k-1}) + \frac{f(S^*)}{k}$$

$$\begin{aligned} f(S^*) - f(S_k) &\leq \left(1 - \frac{1}{k}\right)\left(f(S^*) - f(S_{k-1})\right) \\ &= \left(1 - \frac{1}{k}\right)^k \left(f(S^*) - f(S_0)\right) \\ &\leq \left(e^{-1/k}\right)^k \left(f(S^*) - f(S_0)\right) \end{aligned}$$

$$f(S^*) \geq \left(1 - \frac{1}{e}\right) f(S^*)$$

Submodular Maximization

From cardinality constraint to knapsack constraint

$$|S| \leq k \rightarrow \sum_{x \in S} b(x) \leq B$$

Submodular Maximization

- Modified Greedy

Algorithm 1: MGreedy

initialize $S_{\mathrm{g}} \leftarrow \emptyset, V' \leftarrow V$;

while $V' \neq \emptyset$ **do**

  find $u \leftarrow \arg\max_{v \in V'} \left\{ \frac{f(v|S_{\mathrm{g}})}{c(v)} \right\}$;

  if $c(S) + c(u) \leq b$ **then**

    $S_{\mathrm{g}} \leftarrow S_{\mathrm{g}} \cup \{u\}$;

  update the search space $V' \leftarrow V' \setminus \{u\}$;

$v^* \leftarrow \arg\max_{v \in V} f(v)$;

$S_{\mathrm{m}} \leftarrow \arg\max_{S \in \{\{v^*\}, S_{\mathrm{g}}\}} f(S)$;

return S_{m};

Submodular Maximization

Theorem

Let $S^* = \operatorname{argmax}_{|S| \leq k} f(S)$ be the set maximizing $f(S)$ among all sets with size at most k. If $f(\cdot)$ is monotone and submodular and $f(\emptyset) = 0$, then for the set S_g computed by Greedy(k, f), we have

$$f(S_g) \geq \left(\frac{1 - 1/e}{2}\right) f(S^*)$$

Submodular Maximization

Proof.

Define $\Delta(x|S) = f(S \cup \{x\}) - f(S)$

Suppose $x_1, x_2, \dots, x_k, x_{k+1}$ are obtained by the greedy algorithm.

$S_i = \{x_1, x_2, \dots, x_i\}$ for $i = 1, \dots, k$ and $S_0 = \emptyset$.

Assume that the optimal solution is $S^* = \{u_1, u_2, \dots, u_h\}$ and denote $S_j^* = \{u_1, u_2, \dots, u_j\}$.

For $i = 0, 1, \dots, k-1$, we have

$$f(S^*) \le f(S^* \cup S_i)$$

Submodular Maximization

$$
\begin{aligned}
f(S^*) &\leq f(S^* \cup S_i) \\
&= f(S_i) + \Delta(u_1|S_i) + \cdots + \Delta(u_h|S_i \cup S^*_{h-1}) \\
&\leq f(S_i) + \Delta(u_1|S_i) + \Delta(u_2|S_i) + \cdots + \Delta(u_h|S_i) \\
&\leq f(S_i) + \sum_{j=1}^{h} \frac{b(u_j)}{b(x_{i+1})} \cdot \Delta(x_{i+1}|S_i) \\
&= f(S_i) + \frac{b(S^*)}{b(x_{i+1})} \cdot \big(f(S_{i+1}) - f(S_i)\big)
\end{aligned}
$$

Submodular Maximization

$$f(S^*) - f(S_i) \le \frac{b(S^*)}{b(x_{i+1})}\Big(\big(f(S^*) - f(S_i)\big) - \big(f(S^*) - f(S_{i+1})\big)\Big)$$

$$f(S^*) - f(S_{i+1}) \le \left(1 - \frac{b(x_{i+1})}{b(S^*)}\right)\big(f(S^*) - f(S_i)\big)$$

$$\le e^{-\frac{b(x_{i+1})}{b(S^*)}} \cdot \big(f(S^*) - f(S_i)\big).$$

Hence,

$$f(S^*) - f(S_{k+1}) \le \big(f(S^*) - f(S_0)\big) \cdot e^{-\frac{b(S_{k+1})}{b(S^*)}} \le \frac{f(S^*)}{e}$$

Submodular Maximization

- Therefore,

$$\left(1-\frac{1}{e}\right)\cdot f(S^*) \le f(S_{k+1}) = f(S_k) + f(x_{k+1}|S_k)$$
$$\le f(S_k) + f(\{x_{k+1}\}) \le f(S_k) + f(\{x^*\})$$

Finally,

$$\max\{f(S_k), f(\{x^*\})\} \ge \frac{1-1/e}{2}\cdot f(S^*)$$

The End