Unit 1A: Computational Complexity

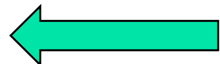
- Course contents:
  - Computational complexity: time, space
  - NP-completeness
  - Algorithmic Paradigms
    - Spanning tree, Steiner tree, Rectilinear spanning/steiner tree
    - Search (exhaustive, branch-and-bound, breadth, depth, dynamic)
      - (supplemented with videos from Stanford AI courses)
    - (welcome to study more about graph theory)

- Readings
  - Chapters 3, 4, and 5

Unit 1A: Computational Complexity

- How to represent complexity – big O
- n is problem size

Time	Big-Oh	$n = 10$	$n = 100$	$n = 10^3$	$n = 10^6$
500	$O(1)$	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec
$3n$	$O(n)$	3×10^{-8} sec	3×10^{-7} sec	3×10^{-6} sec	0.003 sec
$n \log n$	$O(n \log n)$	3×10^{-8} sec	2×10^{-7} sec	3×10^{-6} sec	0.006 sec
n^2	$O(n^2)$	1×10^{-7} sec	1×10^{-5} sec	0.001 sec	16.7 min
n^3	$O(n^3)$	1×10^{-6} sec	0.001 sec	1 sec	3×10^5 cent.
2^n	$O(2^n)$	1×10^{-6} sec	3×10^{17} cent.	∞	∞
$n!$	$O(n!)$	0.003 sec	∞	∞	∞



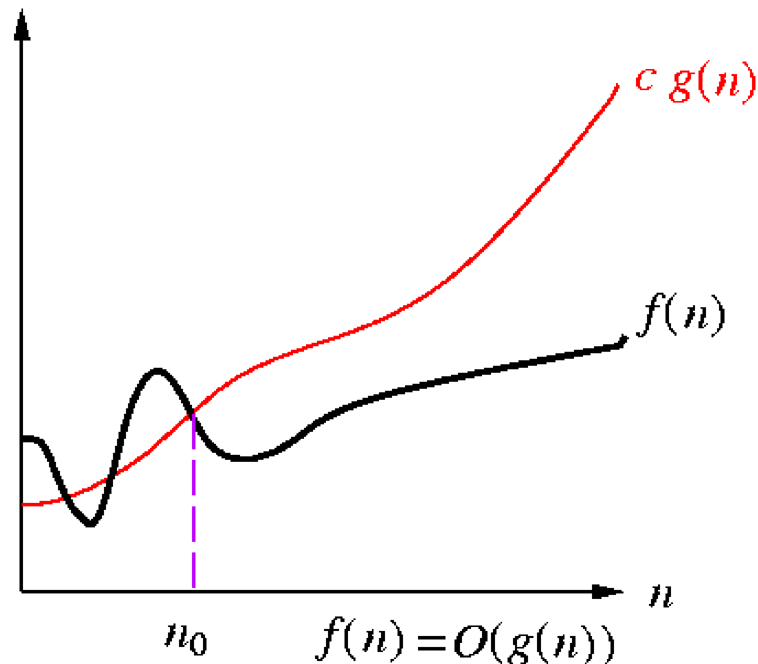
Time taken to produce results

O: Upper Bounding Function

- **Def:** $f(n) = O(g(n))$ if $\exists\, c > 0$ and $n_0 > 0$ such that $0 \le$ ***f*(*n*) ≤*cg*(*n*)** for all $n \ge n_0$.
  - Examples: $2n^2 + 3n = O(n^2)$, $2n^2 = O(n^3)$, $3n \lg n = O(n^2)$

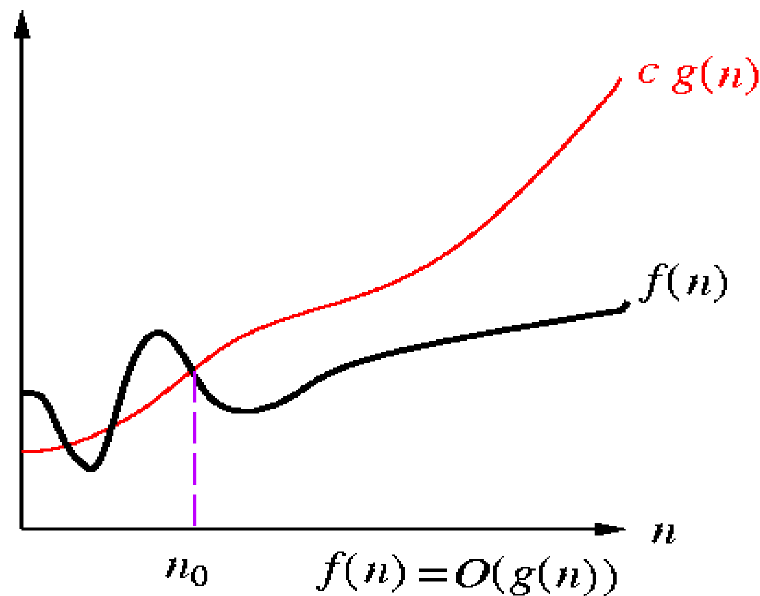
- Intuition: $f(n)$ "≤" $g(n)$ when we ignore constant multiples and small values of n.

$$
\begin{aligned}
\tfrac{1}{3}n^2 &= \mathcal{O}(n^2) \\
0.02n^2 + 127n + 1923 &= \mathcal{O}(n^2) \\
3n \log n + n &= \mathcal{O}(n \log n) \\
5n^3 &= \mathcal{O}(\tfrac{1}{5}n^3) \\
5n^3 &= \mathcal{O}(n^4) \\
5n^3 &= \mathcal{O}(n^3 + n^2)
\end{aligned}
$$



Big-O Notation

- How to show O (Big-Oh) relationships?
  - $f(n) = O(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ for some $\boldsymbol{c \geq 0}$.
- "An algorithm has worst-case running time $O(f(n))$": there is a constant c s.t. for every n big enough, **every execution** on an input of size n takes **at most** $cg(n)$ time.



Computational Complexity

- Computational complexity: an abstract measure of the time and space necessary to execute an algorithm as function of its "input size".
- Input size examples:
  - sort n words of bounded length n
  - the input is the integer n $\lg n$
  - the input is the graph $G(V, E)$ *with* $|V|$ and $|E|$

- Time complexity is expressed in *elementary computational steps* (e.g., an addition, multiplication, pointer indirection).

- Space Complexity is expressed in *memory locations* (e.g. bits, bytes, words).

Asymptotic Functions

- Polynomial-time complexity: $O(n^k)$, where n is the **input size** and k is a constant.
- Example polynomial functions:
  - 999: constant
  - $\lg n$: logarithmic
  - $\sqrt{n}$: sublinear
  - n: linear
  - $n \lg n$: loglinear
  - n^2: quadratic
  - n^3: cubic
- Example non-polynomial functions
  - 2^n, 3^n: exponential
  - $n!$: factorial

Running-time Comparison

- Assume 1000 MIPS (Yr: 200x), 1 instruction /operation

Time	Big-Oh	$n = 10$	$n = 100$	$n = 10^3$	$n = 10^6$
500	$O(1)$	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec
$3n$	$O(n)$	3×10^{-8} sec	3×10^{-7} sec	3×10^{-6} sec	0.003 sec

Running-time Comparison

- Assume 1000 MIPS (Yr: 200x), 1 instruction /operation

Time	Big-Oh	$n = 10$	$n = 100$	$n = 10^3$	$n = 10^6$
$3n$	$O(n)$	3×10^{-8} sec	3×10^{-7} sec	3×10^{-6} sec	0.003 sec
$n \log n$	$O(n \log n)$	3×10^{-8} sec	2×10^{-7} sec	3×10^{-6} sec	0.006 sec
n^2	$O(n^2)$	1×10^{-7} sec	1×10^{-5} sec	0.001 sec	16.7 min
		T	10^2 T	10^4 T	~1000 sec

Running-time Comparison

- Assume 1000 MIPS (Yr: 200x), 1 instruction /operation

Time	Big-Oh	$n = 10$	$n = 100$	$n = 10^3$	$n = 10^6$
n^3	$O(n^3)$	1×10^{-6} sec	0.001 sec	1 sec	3×10^5 cent.
2^n	$O(2^n)$	1×10^{-6} sec	3×10^{17} cent.	∞	∞

century

n=10^4, 10^5

Result in Centuries:

1,000,000,000 s = 0.316887 c

Running-time Comparison

- Assume 1000 MIPS (Yr: 200x), 1 instruction /operation

Time	Big-Oh	$n = 10$	$n = 100$	$n = 10^3$	$n = 10^6$
500	$O(1)$	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec	5×10^{-7} sec
$3n$	$O(n)$	3×10^{-8} sec	3×10^{-7} sec	3×10^{-6} sec	0.003 sec
$n \log n$	$O(n \log n)$	3×10^{-8} sec	2×10^{-7} sec	3×10^{-6} sec	0.006 sec
n^2	$O(n^2)$	1×10^{-7} sec	1×10^{-5} sec	0.001 sec	16.7 min
n^3	$O(n^3)$	1×10^{-6} sec	0.001 sec	1 sec	3×10^5 cent.
2^n	$O(2^n)$	1×10^{-6} sec	3×10^{17} cent.	∞	∞
$n!$	$O(n!)$	0.003 sec	∞	∞	∞

Optimization Problems

- **Problem:** a general class, e.g., "the shortest-path problem for directed acyclic graphs."
- **Instance:** a specific case of a problem, e.g., "the shortest-path problem in a specific graph, between two given vertices."
- **Optimization problems:** those finding a legal configuration such that its cost is minimum (or gain is maximum).
  - Minimum Spanning Tree: Given a graph $G=(V, E)$, find the cost of a minimum spanning tree of G.
- An instance $I = (F, c)$ where
  - F is the set of *feasible solutions*, and
  - c is a *cost function*, assigning a cost value to each feasible solution $c : F \rightarrow R$
- The solution of the optimization problem is the feasible solution with optimal (minimal/maximal) cost
- c.f., **Optimal** solutions/costs, optimal (**exact**) algorithms (Attn: optimal $\neq$ exact in the theoretic computer science community).

Exact vs approximate

c.f.: compare

Min Spanning Tree (MST)

A planar graph and its minimum spanning tree. Each edge is labeled with its weight, which here is roughly proportional to its length.

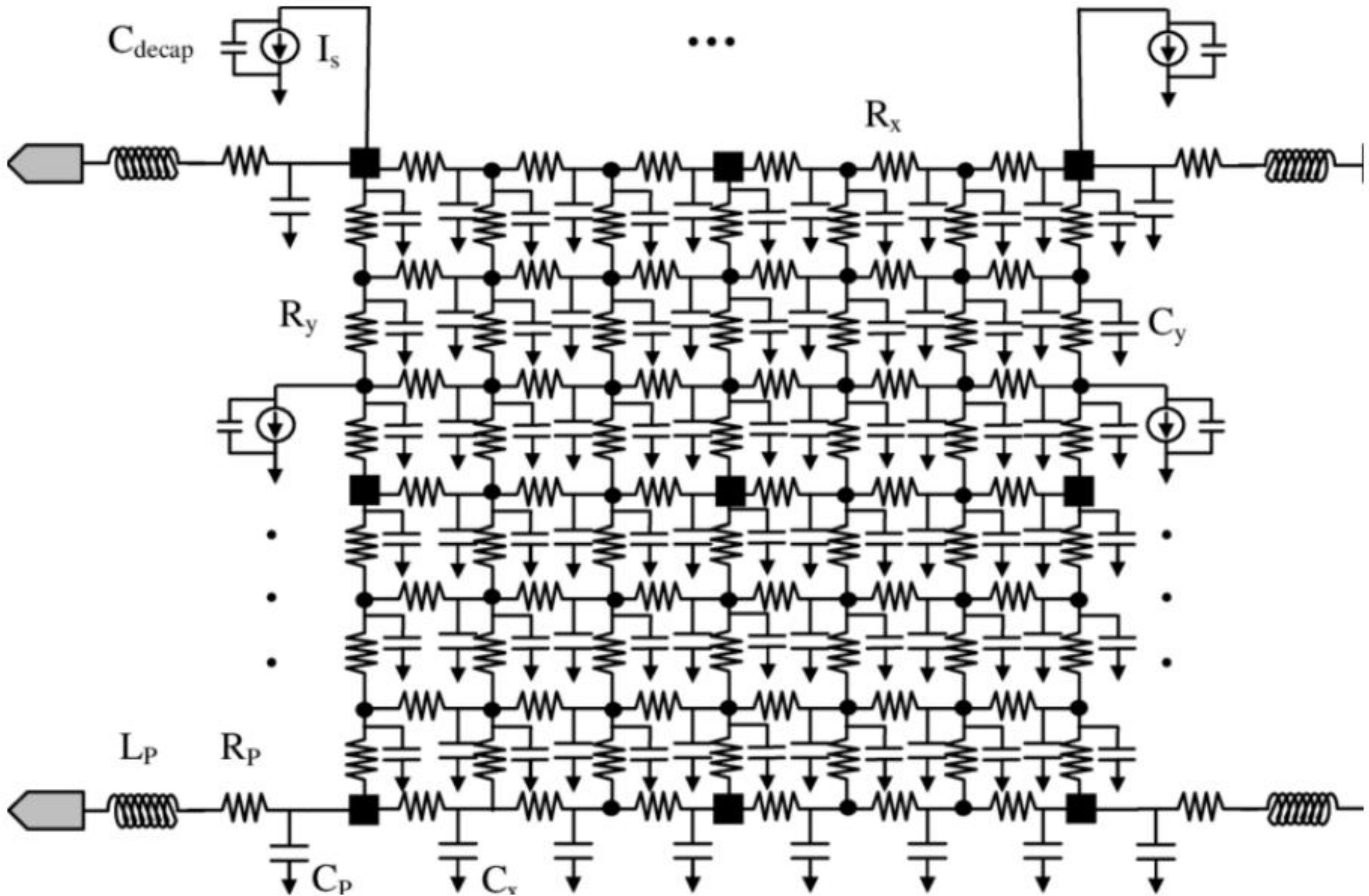
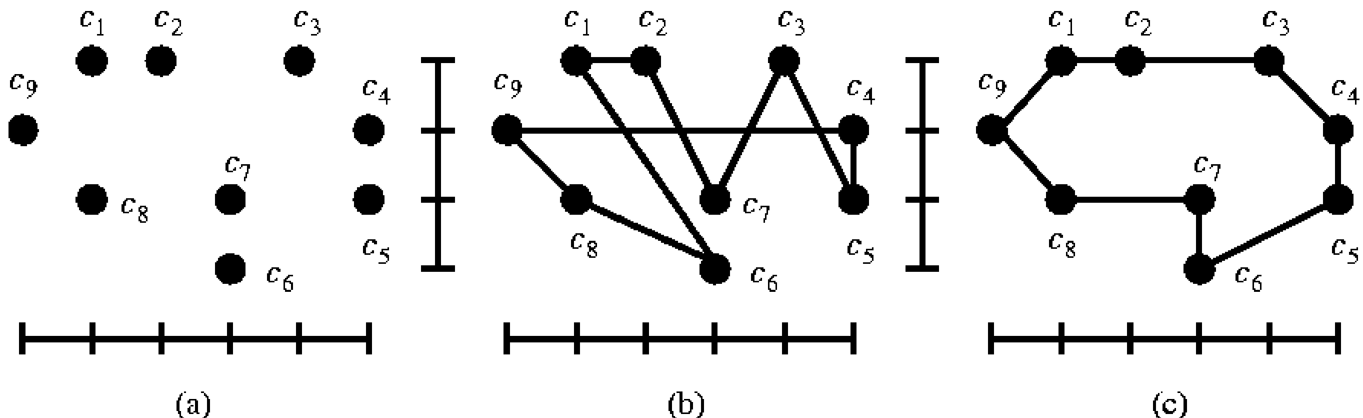


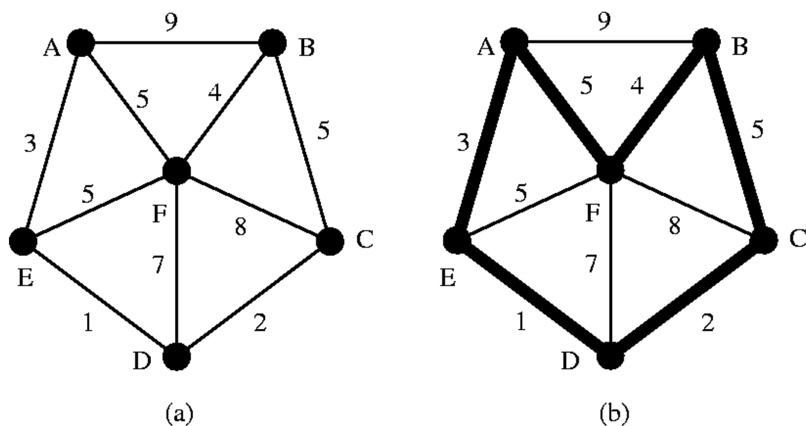
Figure 1: RLC model of supply grid network

Multiple sources → given an instance pin, find LRP (least resistive paths)

The Traveling Salesman Problem (TSP)

- TSP: Given a set of cities and that distance between each pair of cities, find the distance of a "minimum tour" starts and ends at a given city and visits every city exactly once.





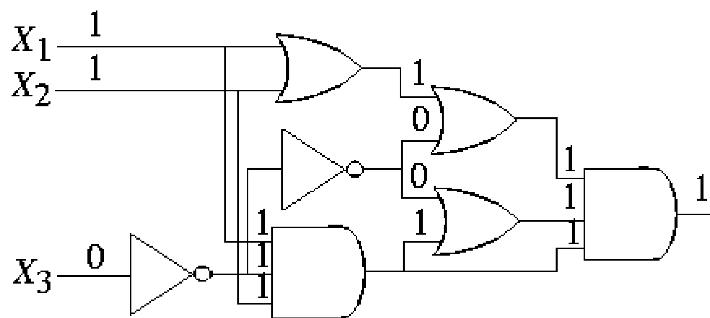
A greedy solution (approximation)

Decision Problem

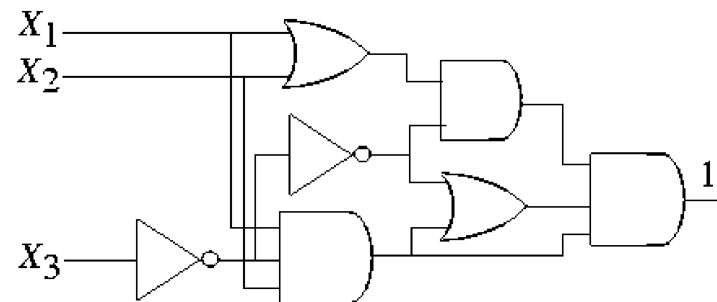
- **Decision problems:** problem that can only be answered with "yes" or "no"
  - MST: Given a graph $G=(V, E)$ and a bound K, is there a spanning tree with a cost at most K?
  - TSP: Given a set of cities, distance between each pair of cities, and a bound B, is there a route that starts and ends at a given city, visits every city exactly once, and has total distance at most B?
- A decision problem Π, has instances: $I = (F, c, k)$
  - The set of instances for which the answer is "yes" is given by Y_Π.
  - A subtask of a decision problem is ***solution checking***: given $f \in F$, checking whether the cost is less than k.
- Could apply binary search on decision problems to obtain solutions to optimization problems.
- NP-completeness is associated with decision problems.

The Circuit-Satisfiability Problem (Circuit-SAT)

- **The Circuit-Satisfiability Problem (Circuit-SAT):**
  - **Instance:** A combinational circuit *C* composed of AND, OR, and NOT gates.
  - **Question:** Is there an assignment of Boolean values to the inputs that makes the output of *C* to be 1?
- A circuit is satisfiable if there exists a set of Boolean input values that makes the output of the circuit to be 1
  - Circuit (a) is satisfiable since <x_1, x_2, x_3> = <1, 1, 0> makes the output to be 1.



(a) satisfiable circuit



(b) unsatisfiable circuit

Complexity Class P

- **Complexity class *P*** contains those problems that can be **solved** in polynomial time in the **size of input**.
  - **Input size:** size of encoded "binary" strings.
  - Edmonds: Problems in P are considered **tractable**.
- MST is in *P*

- The computer concerned is a *deterministic* *Turing machine*
  - *Deterministic* means that each step in a computation is predictable.
  - A *Turing machine* is a mathematical model of a universal computer (any computation that needs polynomial time on a Turing machine can also be performed in polynomial time on any other machine).

Complexity Class NP

- Suppose that ***solution checking*** for some problem can be done in polynomial time on a deterministic machine the problem can be solved in polynomial time on a ***nondeterministic*** *Turing machine.*
  - *Nondeterministic*: the machine makes a guess, e.g., the right one (or the machine evaluates all possibilities in parallel).
- **The class NP** (**N**ondeterministic **P**olynomial): class of problems that can be **verified** in polynomial time in the size of input.
  - NP: class of problems that can be solved in polynomial time on a nondeterministic machine.
- Is TSP $\in$ NP?
  - Need to **check** a solution in polynomial time.
    - Guess a tour.
    - Check if the tour visits every city exactly once.
    - Check if the tour returns to the start.
    - Check if total distance $\leq B$.
  - All can be done in $O(n)$ time, **so TSP $\in$ NP**.

輕鬆談演算法的複雜度分界：
什麼是P, NP, NP-Complete, NP-Hard問題

https://www.ycc.idv.tw/algorithm-complexity-theory.html

如果給予 Turing machine 某個 state 和某個 symbol 下，它的下一步如果只有一種可能，那我們就稱它為 deterministic Turing machine (DTM)。

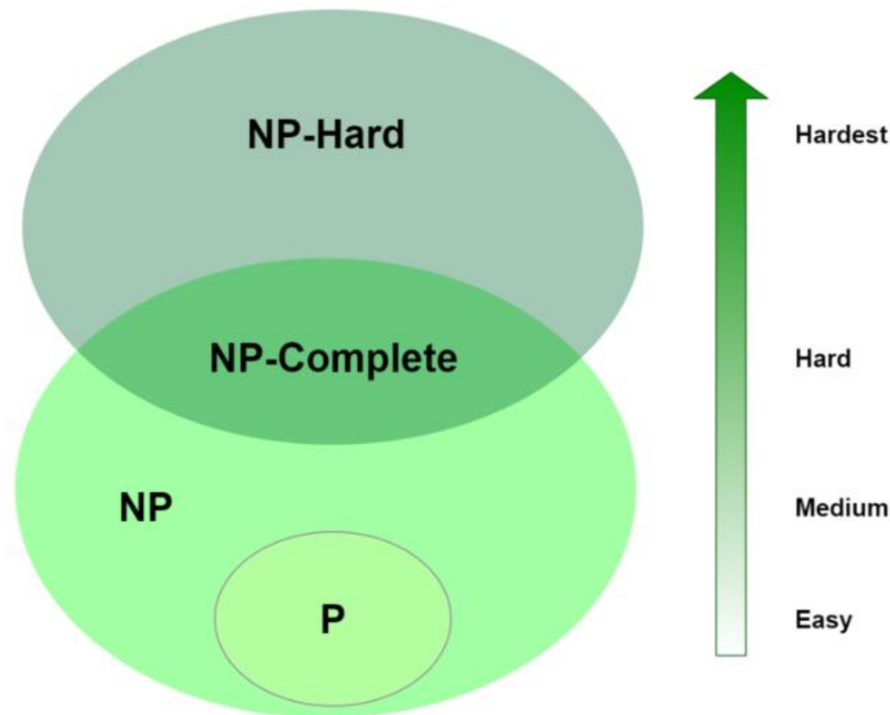
但是 Non-deterministic Turing machine (NTM) 就不拘於此，針對某個 state 和某個 symbol 它的下一步可能會有很多種，它會是一個分支，它可能同時要向右移3格，又同時要向左移動2格，所以你可以想像一下你的**讀寫頭一分為二**，然後再各自進行自己的任務，**這個分支可以有無限多**個，只要最後有某個分支到達 halt state，我們就解完問題並停止計算，這就是 Non-deterministic Turing machine (NTM)。

如果有一群演算法用DTM來做計算所需時間是polynomial time，那這類演算法或問題被稱為P問題，P就是 polynomial-time 的縮寫。

另外如果有一群演算法用NTM來做計算所需時間是polynomial time，那這類問題被稱為NP問題，NP是non-deterministic polynomial-time 的縮寫。

SAT is the first problem that was proven to be NP-complete; see Cook–Levin theorem.

This means that all problems in the complexity class NP, which includes a wide range of natural decision and optimization problems, are at most as difficult to solve as SAT.



只要滿足以下兩個條件的，我們都稱之為NP-Complete：

1. 「問題」本身是一個NP問題
2. 所有的NP問題都可以用DTM在 polynomial time 內化約成為這個「問題」。

A decision problem C is NP-complete if:

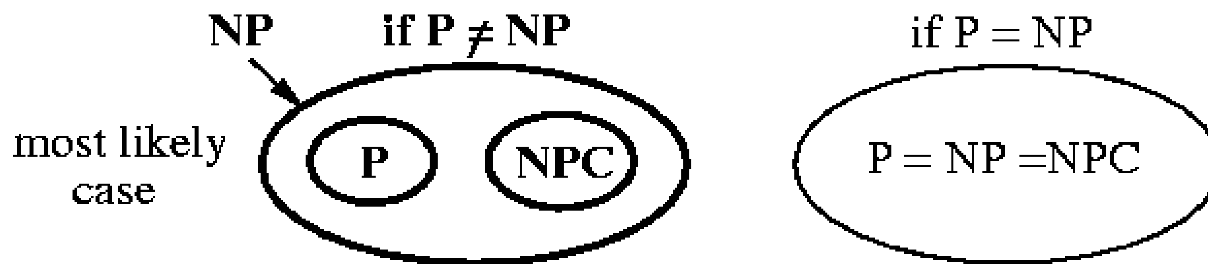
https://en.wikipedia.org/wiki/NP-completeness#Formal_definition

1. C is in NP, and
2. Every problem in NP is reducible to C in polynomial time.[3]

Why powerful?

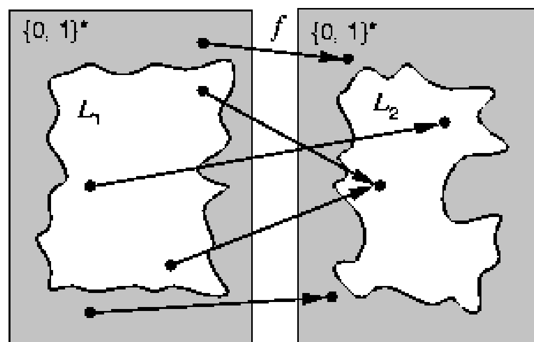
(We know SAT is NP-complete) 這個概念非常強大，假設我證明了SAT是P問題，就等於今天我隨便拿到一個 NP 問題就可以在 polynomial time 內把問題轉換成SAT，然後再用 polynomial time 把 SAT 解掉，所以所有的 NP 問題都只是P問題了→ P=NP，因此NP-Complete問題就是解決 P=NP 的關鍵，如果可以證明NP-Complete問題為P問題，就可以間接證明P=NP。

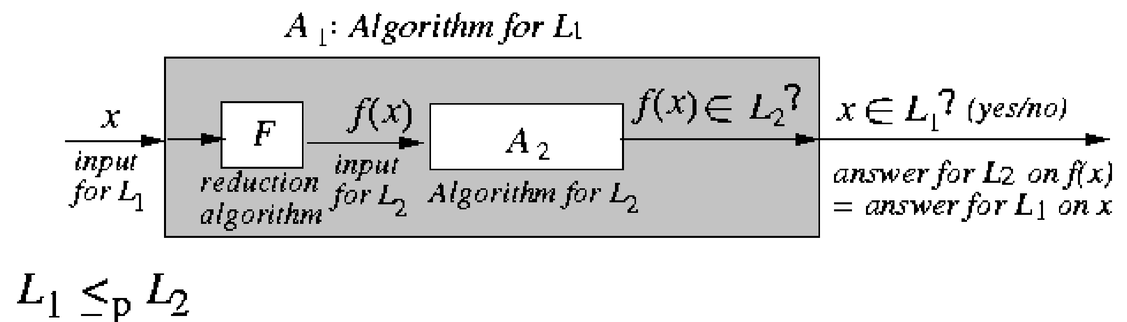
1972年，Richard Karp將這個想法往前推進了一步，他證明了<u>21個不同但都難解的組合數學與圖論問題為NP-Complete問題</u>，一樣的其中的任何一種只要被證明為P問題，都可以間接證明P=NP



Polynomial-time Reduction

- **Motivation:** Let L_1 and L_2 be two decision problems. Suppose algorithm A_2 can solve L_2. Can we use A_2 to solve L_1?
- **Polynomial-time reduction *f* from L_1 to L_2:** $L_1 \leq_P L_2$
  - *f* reduces input for L_1 into an input for L_2 s.t. the reduced input is a "yes" input for L_2 iff the original input is a "yes" input for L_1.
    - $L_1 \leq_P L_2$ if $\exists$ polynomial-time computable function f: $\{0, 1\}^* \rightarrow \{0, 1\}^*$ s.t. $x \in L_1$ iff $f(x) \in L_2$, $\forall\, x \in \{0, 1\}^*$.
    - **L_2 is at least as hard as L_1.**
- *f* is computable in polynomial time.

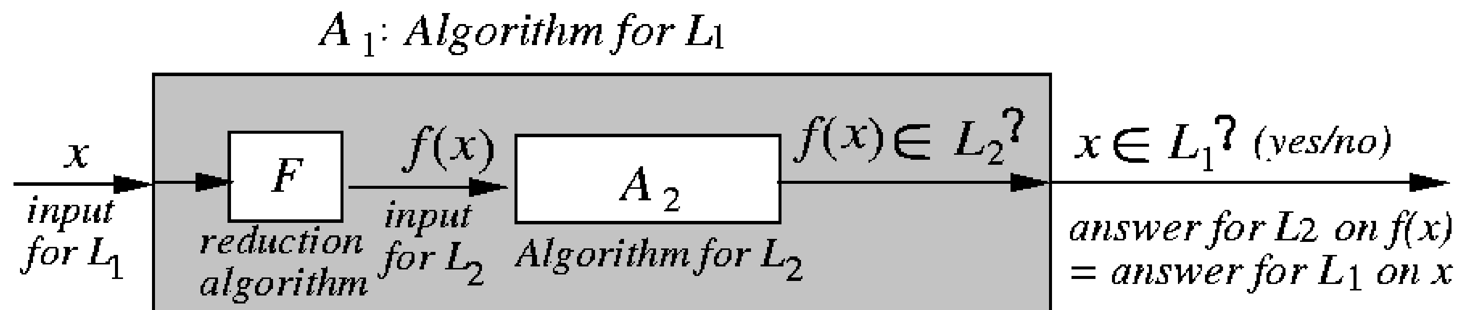




$L_1 \leq_P L_2$

Significance of Reduction

- Significance of $L_1 \leq_P L_2$:
  - $\exists$ polynomial-time algorithm for $L_2 \Rightarrow \exists$ polynomial-time algorithm for L_1 ($L_2 \in$ P $\Rightarrow L_1 \in$ P).
  - $\nexists$ polynomial-time algorithm for $L_1 \Rightarrow$ $\nexists$ ynomial-time algorithm for L_2 ($L_1 \notin$ P $\Rightarrow L_2 \notin$ P).
- $\leq_P$ is transitive, i.e., $L_1 \leq_P L_2$ and $L_2 \leq_P L_3 \Rightarrow L_1 \leq_P L_3$.



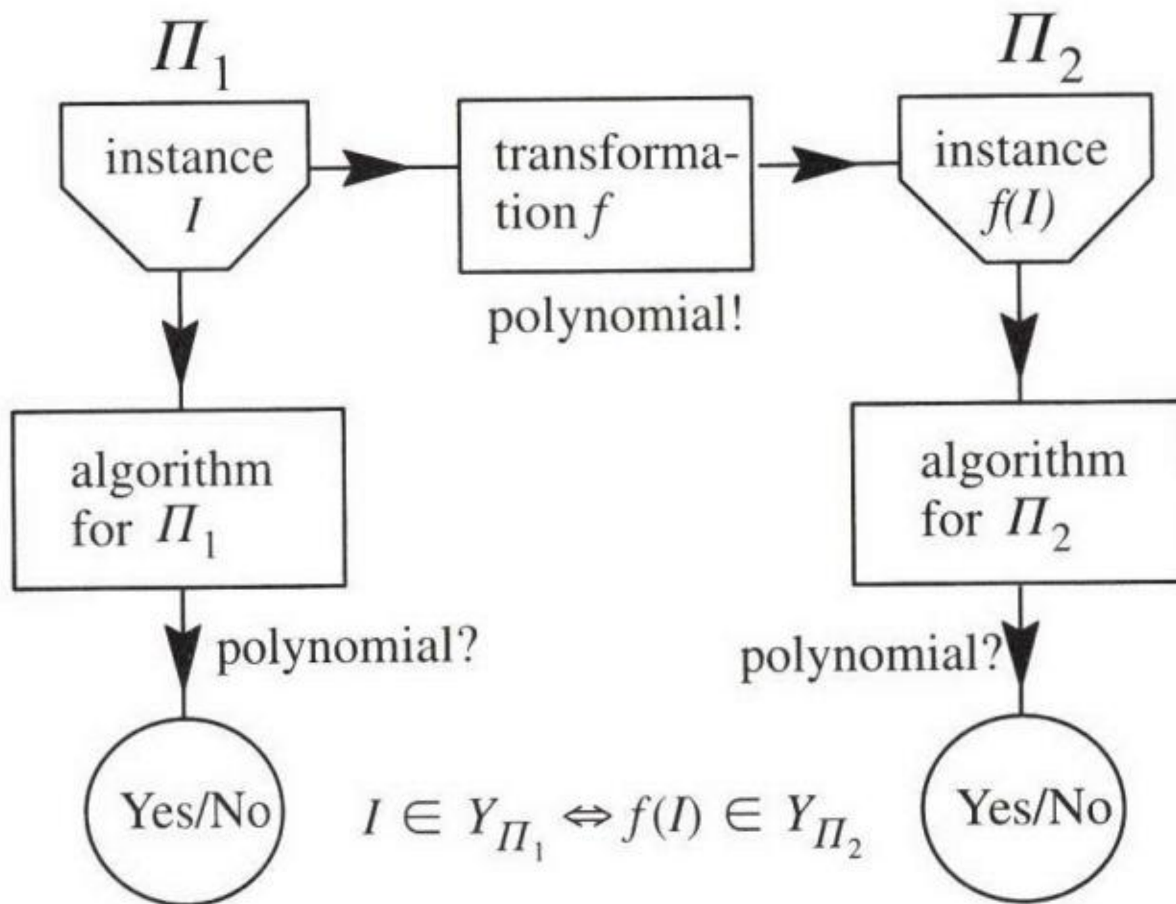


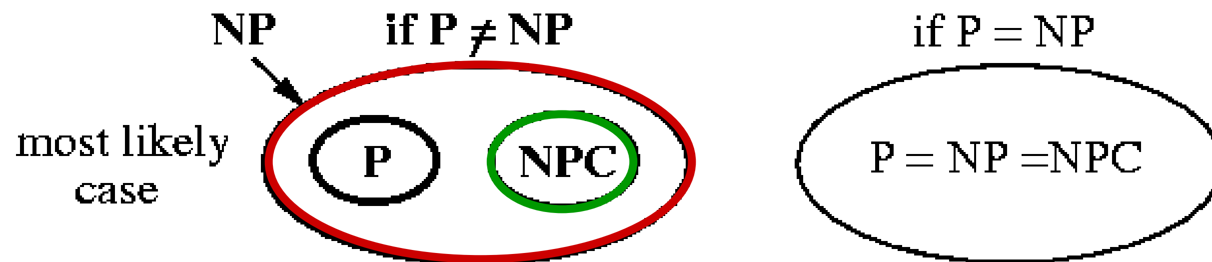
Figure 4.4 The reduction of a problem Π_1 to a problem Π_2.

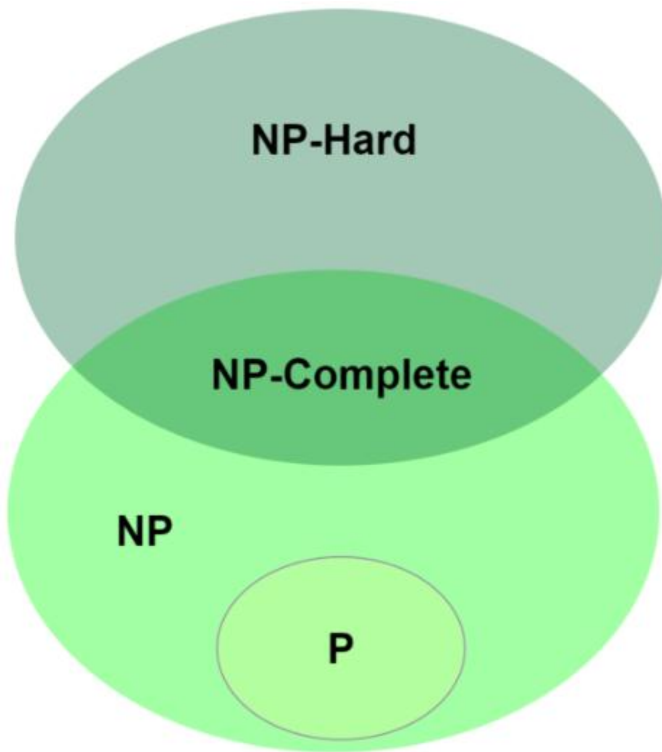
NP-Completeness

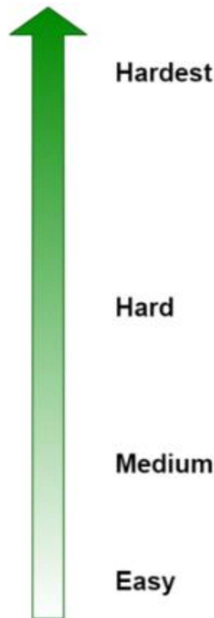
- An issue which is still unsettled:

  $P \subset NP$ or $P = NP$?

- There is a strong belief that $P \neq NP$, due to the existence of NP-complete problems.
- **The class NP-complete (NPC):**
  - Developed by S. Cook and R. Karp in early 1970.
  - All problems in NPC have the same degree of difficulty: **Any** NPC problem can be solved in polynomial time **all** problems in NP can be solved in polynomial time.







https://www.baeldung.com/cs/p-np-np-complete-np-hard

NP-hard
- K-means Clustering
- Traveling Salesman Problem, and
- Graph Coloring

NP-complete
- Traveling Salesman
- Knapsack, and
- Graph Coloring
- Satisfiability (SAT) problem

NP
- Integer Factorization and
- Graph Isomorphism

- Easy $\rightarrow \mathcal{P}$
- Medium $\rightarrow \mathcal{NP}$
- Hard $\rightarrow \mathcal{NP}\text{-}Complete$
- Hardest $\rightarrow \mathcal{NP}\text{-}Hard$

P
- All basic mathematical operations; addition, subtraction, division, multiplication
- Testing for primacy
- **Hashtable lookup**, **string operations**, **sorting problems**
- Shortest Path Algorithms; **Djikstra**, **Bellman-Ford**, Floyd-Warshall
- Linear and **Binary Search Algorithms** for a given set of numbers

- $\mathcal{P}$ problems are quick to solve
- $\mathcal{NP}$ problems are quick to verify but slow to solve
- $\mathcal{NP}$-$\mathit{Complete}$ problems are also quick to verify, slow to solve and can be reduced to any other $\mathcal{NP}$-$\mathit{Complete}$ problem
- $\mathcal{NP}$-Hard problems are slow to verify, slow to solve and can be reduced to any other $\mathcal{NP}$ problem

Example: Consider the optimization problem "find the length of the shortest path in all the instances of the traveling salesman problem."
This is an optimization problem, not a decision problem. It is **NP-hard** because, in particular, if we can solve it then we can use that solution to solve the decision version of TSP, which is NP-complete. And by definition of NP-complete, if we can solve decision-TSP in polynomial time then we can solve any problem in NP in polynomial time.

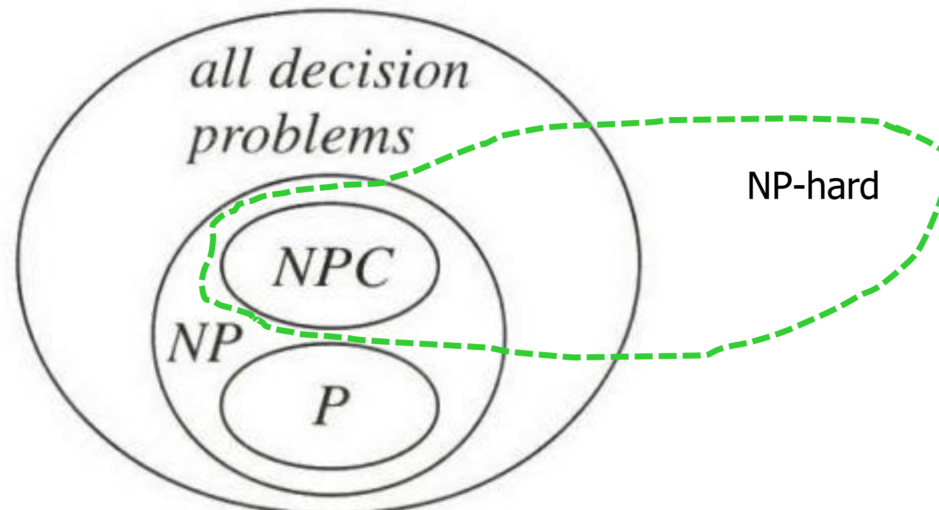
What is difference between NP-complete and NP-hard?

NP-Complete problems are as hard as NP problems.

...

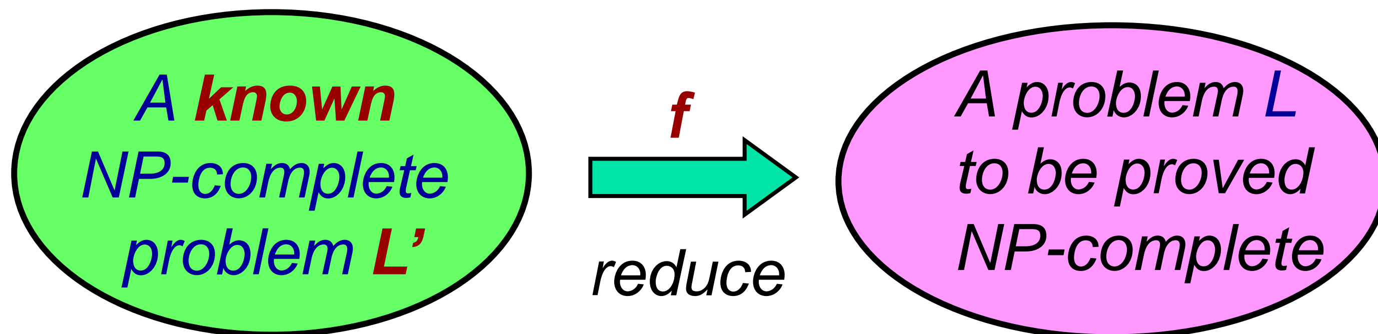
Difference between NP-Hard and NP-Complete:

NP-hard	NP-Complete
To solve this problem, it do not have to be in NP .	To solve this problem, it must be both NP and NP-hard problems.
Do not have to be a Decision problem.	It is exclusively a Decision problem.



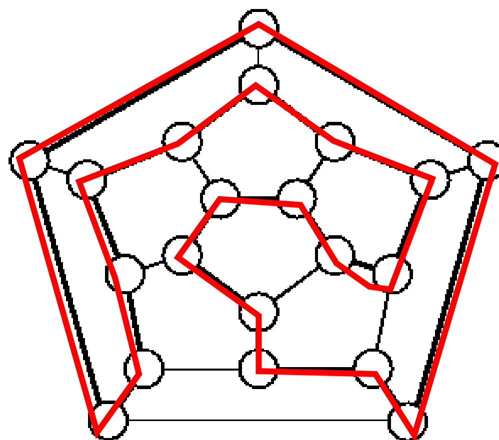
Proving NP-Completeness

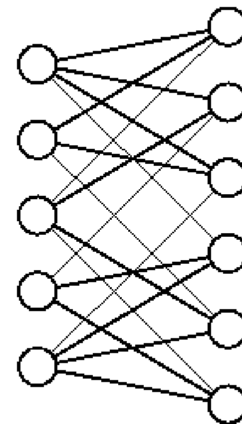
- **Five steps for proving that *L* is NP-complete:**
  1. Prove $L \in$ NP.
  2. Select a known NP-complete problem ***L'***.
  3. Construct a reduction *f* transforming **every** instance of ***L'*** to an instance of *L*.
  4. Prove that $x \in L'$ iff $f(x) \in L$ for all $x \in \{0, 1\}^*$.
  5. Prove that *f* is a polynomial-time transformation.
- We have shown that TSP is NP-complete



Polynomial-time Reduction

- The Hamiltonian Circuit Problem (HC)
  - **Instance:** an undirected graph $G = (V, E)$.
  - **Question:** is there a cycle in G that includes every vertex exactly once?
- TSP (The Traveling Salesman Problem)
- How to show $HC \leq_P TSP$?
  1. Define a function f mapping any HC instance into a TSP instance, and show that f can be computed in polynomial time.
  2. Prove that G has an HC iff the reduced instance has a TSP tour with distance $\leq B$ ($x \in HC \Leftrightarrow f(x) \in TSP$).


Hamiltonian


nonhamiltonian

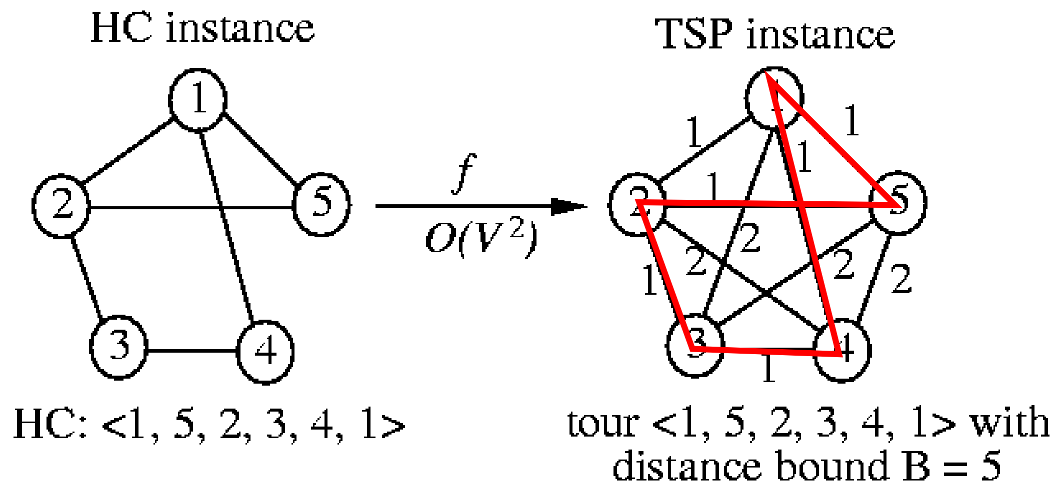
HC $\leq_P$ TSP: Step 1

1. Define a reduction function *f* for HC $\leq_P$ TSP.
   - Given an arbitrary HC instance $G = (V, E)$ with n vertices
     - Create a set of n cities labeled with names in V.
     - Assign distance between u and v

$$d(u,v) = \begin{cases} 1, & \text{if } (u,v) \in E, \\ 2, & \text{if } (u,v) \notin E. \end{cases}$$

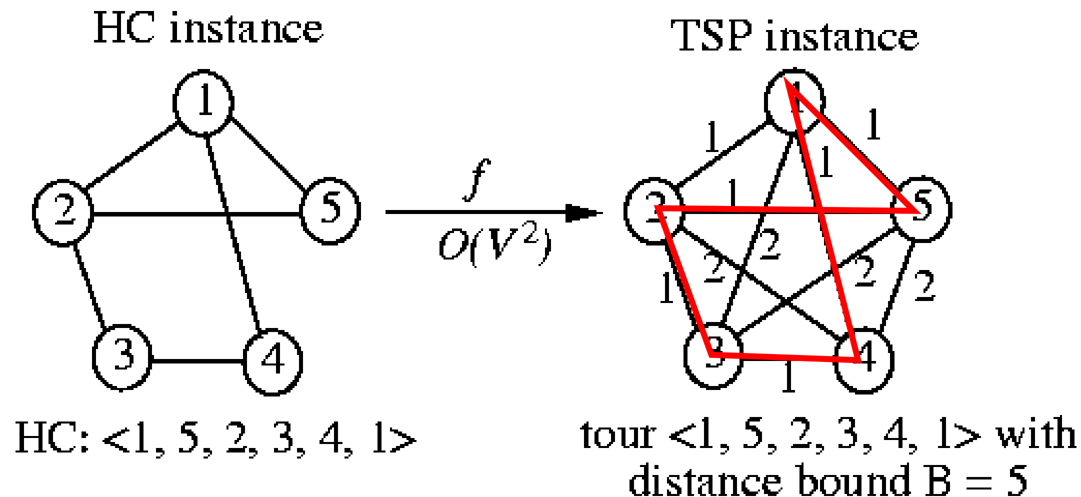
Assign a big number, say 1M

     - Set bound $B = n$.
   - *f* can be computed in $O(V^2)$ time.



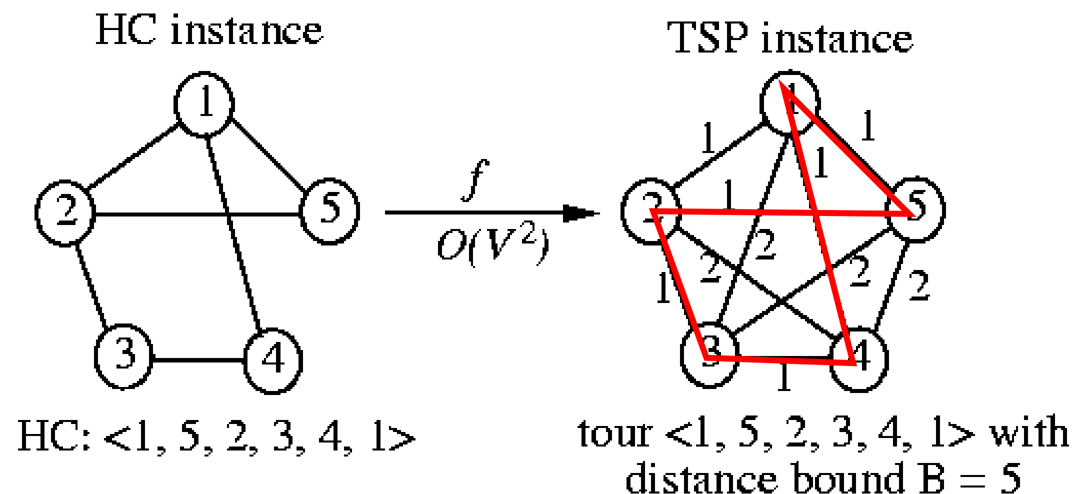
HC $\leq_P$ TSP: Step 2

2. G has an HC iff the reduced instance has a TSP with distance $\leq B$.
   - $x \in$ HC $\Rightarrow f(x) \in$ TSP.
   - Suppose the HC is $h = \langle v_1, v_2, \ldots, v_n, v_1\rangle$. Then, h is also a tour in the transformed TSP instance.
   - The distance of the tour h is $n = B$ since there are n consecutive edges in E, and so has distance 1 in $f(x)$.
   - Thus, $f(x) \in$ TSP ($f(x)$ has a TSP tour with distance $\leq B$).



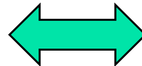
HC $\leq_P$ TSP: Step 2 (cont'd)

2. *G* has an HC iff the reduced instance has a TSP with distance $\leq B$.
   - $f(x) \in$ TSP $\Rightarrow x \in$ HC.
   - Suppose there is a TSP tour with distance $\leq n = B$. Let it be <v_1, v_2, ..., v_n, v_1>..
   - Since distance of the tour $\leq n$ and there are n edges in the TSP tour, the tour contains only edges in E.
   - Thus, <v_1, v_2, ..., v_n, v_1> is a Hamiltonian cycle ($x \in$ HC).



HC → TSP

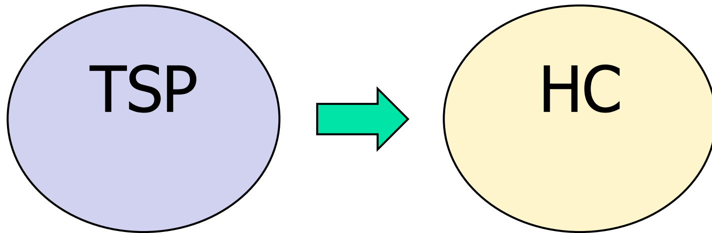
Any graph **A** we want to find a HC

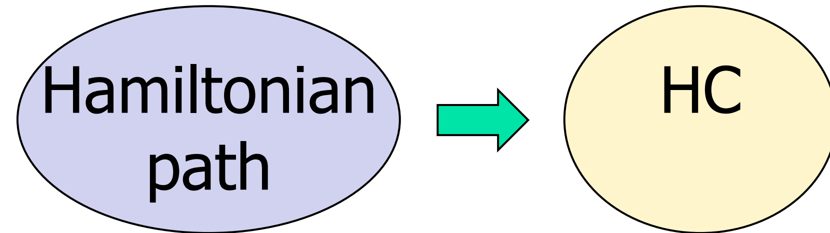


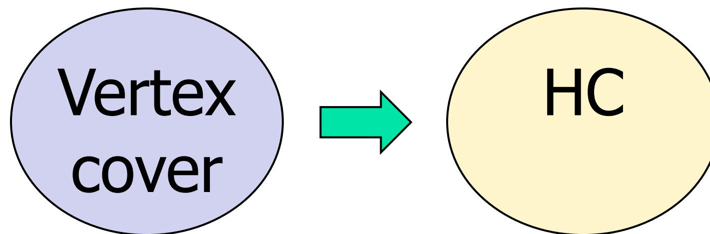
Create a complete graph,
same # of nodes,
edges with weight 1, if those two nodes in **A** are adjacent;
Otherwise, big number.
Bound is set to N.
Then, find a TSP

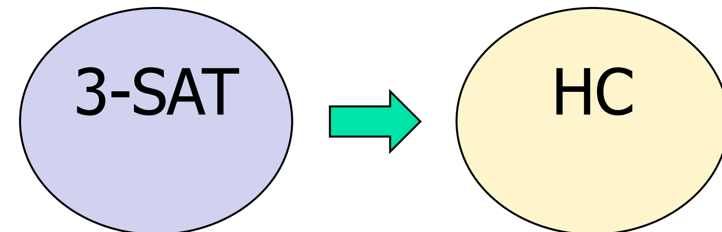
Is Hamiltonian Circuit NP-Complete?

Shall we try to prove TSP -> HC?









NP-Completeness and NP-Hardness

- **NP-completeness: worst-case** analyses for **decision** problems.
- L is **NP-complete** if
  - **$L \in NP$**
  - **NP-Hard: $L' \leq_P L$ for every $L' \in NP$.**
- **NP-hard:** If L satisfies the 2nd property, but not necessarily the 1st property, we say that L is **NP-hard**.

- Suppose $L \in$ NPC.
  - If $L \in P$, then there exists a polynomial-time algorithm for every $L' \in$ NP (i.e., P = NP).
  - If $L \notin P$, then there exists no polynomial-time algorithm for any $L' \in$ NPC (i.e., P $\neq$ NP).

- $\mathcal{P}$ problems are quick to solve
- $\mathcal{NP}$ problems are quick to verify but slow to solve
- $\mathcal{NP}\text{-}Complete$ problems are also quick to verify, slow to solve and can be reduced to any other $\mathcal{NP}\text{-}Complete$ problem
- $\mathcal{NP}\text{-}Hard$ problems are slow to verify, slow to solve and can be reduced to any other $\mathcal{NP}$ problem

Studying algorithms is exciting.
It can be applied to lots of areas.
Down the road you can make a difference in the environment/industry you are in.

Coping with NP-hard problems

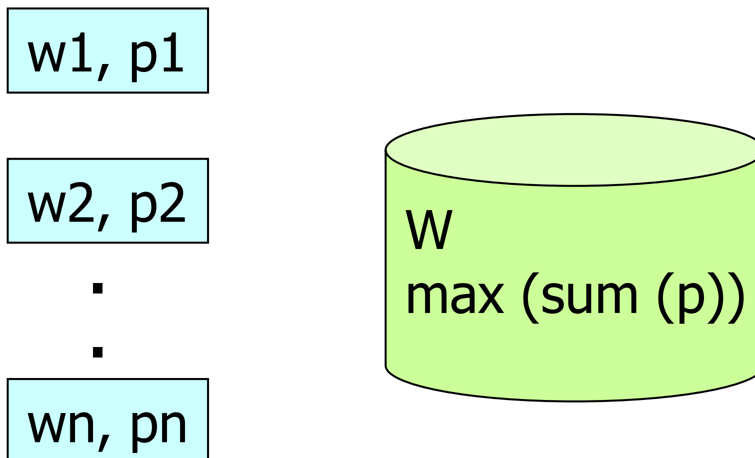
- **Approximation algorithms**
  - Guarantee to be a fixed percentage away from the optimum.
  - E.g., Min-Spanning-T for the **minimum Steiner tree** problem.
- **Pseudo-polynomial time algorithms**
  - Has the form of a polynomial function for the complexity, but is not to the problem size.
  - E.g., $O(nW)$ for the 0-1 knapsack problem.
- **Restriction**
  - Work on some subset of the original problem.
  - E.g., the longest path problem in directed acyclic graphs.
- **Exhaustive search/Branch and bound**
  - Is feasible only when the problem size is small.
- **Local search:**
  - Simulated annealing (hill climbing), genetic algorithms, etc.
- **Heuristics:** No guarantee of performance.

0/1 Knapsack Problem

Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it].

The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

- An item is in the bag or not at all (0/1 problem.)



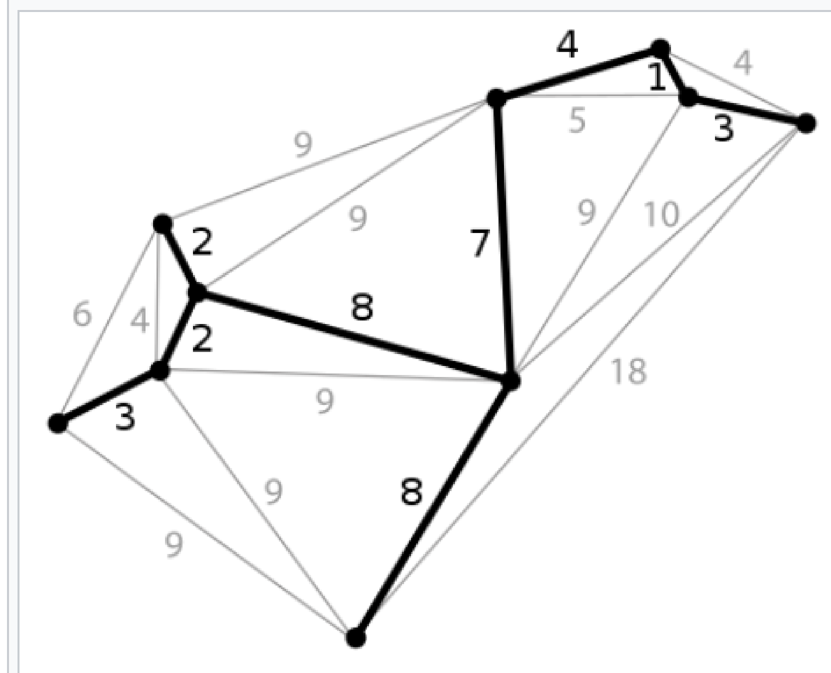
https://web.ntnu.edu.tw/~algo/KnapsackProblem.html

https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/

(For TsingHua University only)

Minimum Spanning Tree

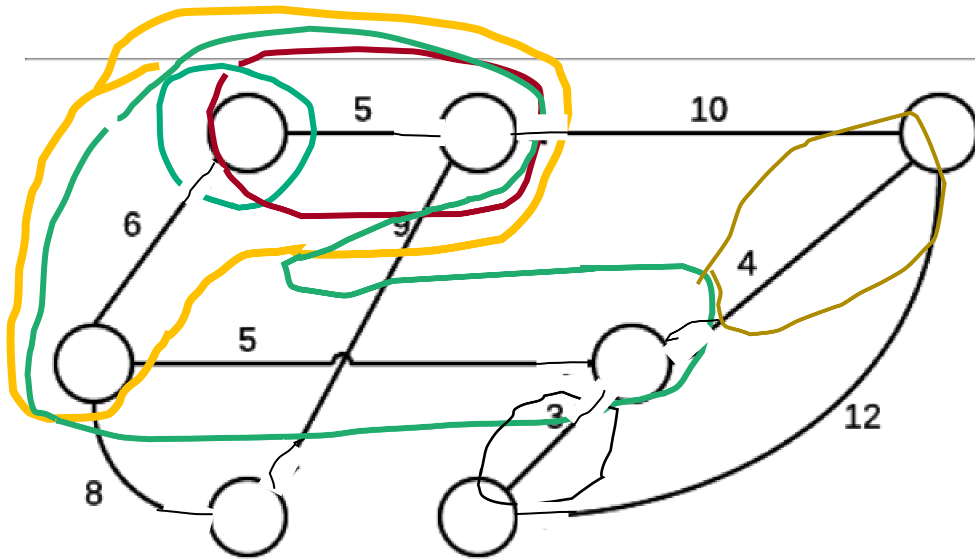
A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted **undirected graph** that connects all the vertices together, without any cycles and with the minimum possible total edge weight.



A planar graph and its minimum spanning tree. Each edge is labeled with its weight, which here is roughly proportional to its length.

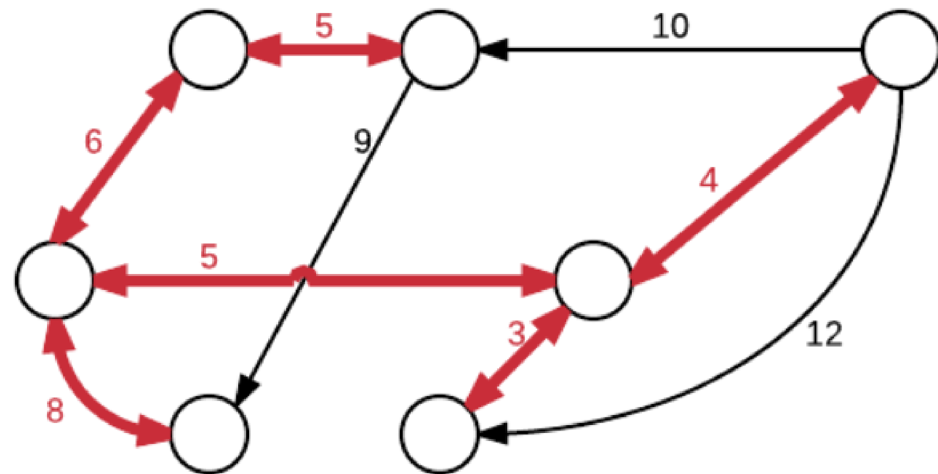
Uniqueness. If each edge has a distinct weight then there will be only one, unique minimum spanning tree.

1. Find the minimum spanning tree for the graph below. What is its total weight? (please ignore direction)



The minimum spanning tree is shown below. Its total weight is 31.

A second algorithm is Prim's algorithm, which was invented by Vojtěch Jarník in 1930 and rediscovered by Prim in 1957 and Dijkstra in 1959. Basically, it grows the MSpanningT (*T*) one edge at a time. Initially, *T* contains an arbitrary vertex. In each step, *T* is augmented with a least-weight edge (*x*,*y*) such that *x* is in *T* and *y* is not yet in *T*. By the Cut property, all edges added to *T* are in the MST. Its run-time is either $O(E \log V)$ or $O(E + V \log V)$, depending on the data-structures used.



https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

https://en.wikipedia.org/wiki/Minimum_spanning_tree#Algorithms

https://en.wikipedia.org/wiki/Prim%27s_algorithm

Search

Backtracking Search - Exhaustive

Search stopped due to visiting a city twice

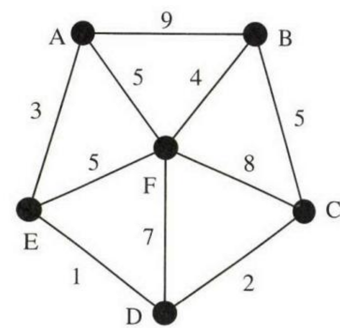


Figure 5.3 An example traveling salesman problem instance.

5. General-purpose Methods for Combinatorial Optimization 59

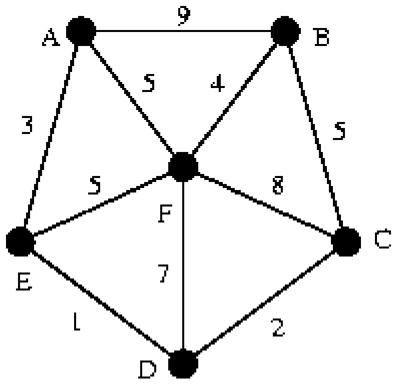
Figure 5.4 The search tree obtained by the exhaustive search backtracking algorithm for the example of Figure 5.3.

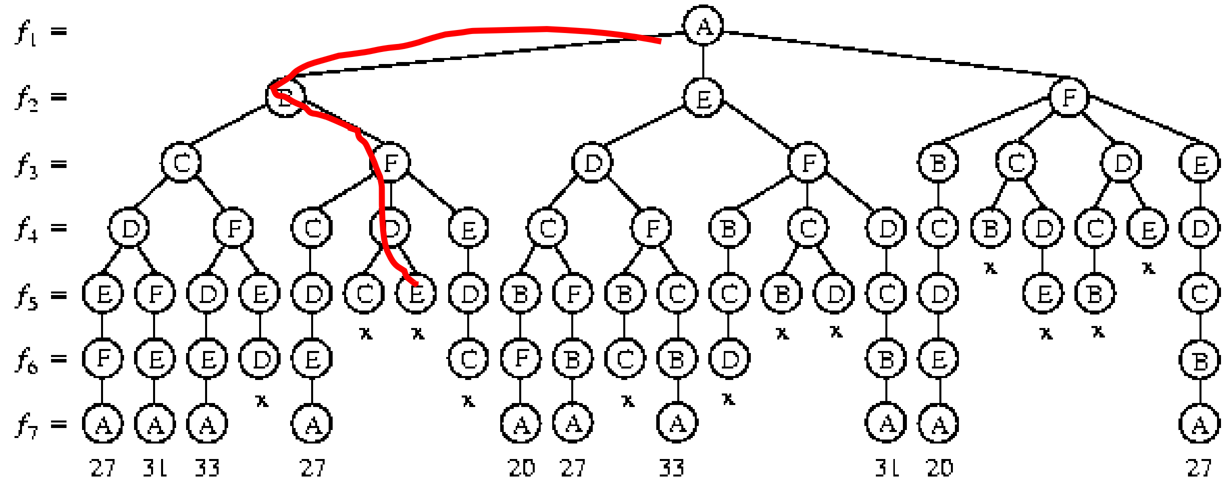
Another Stopping Criterion

The partial path total cost/distance is >= current best cost found

Exhaustive Search v.s. Branch and Bound

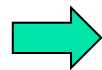
- TSP example

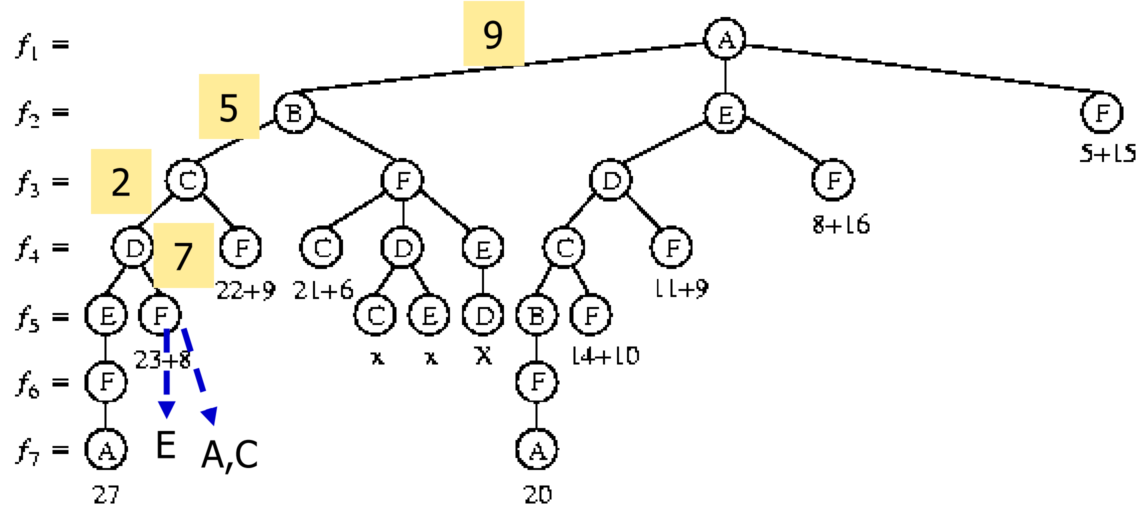




Backtracking/exhaustive search

The modification of the backtracking algorithm that provides in killing partial solutions is called branch-and-bound

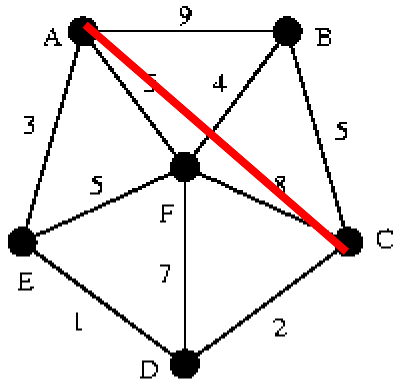


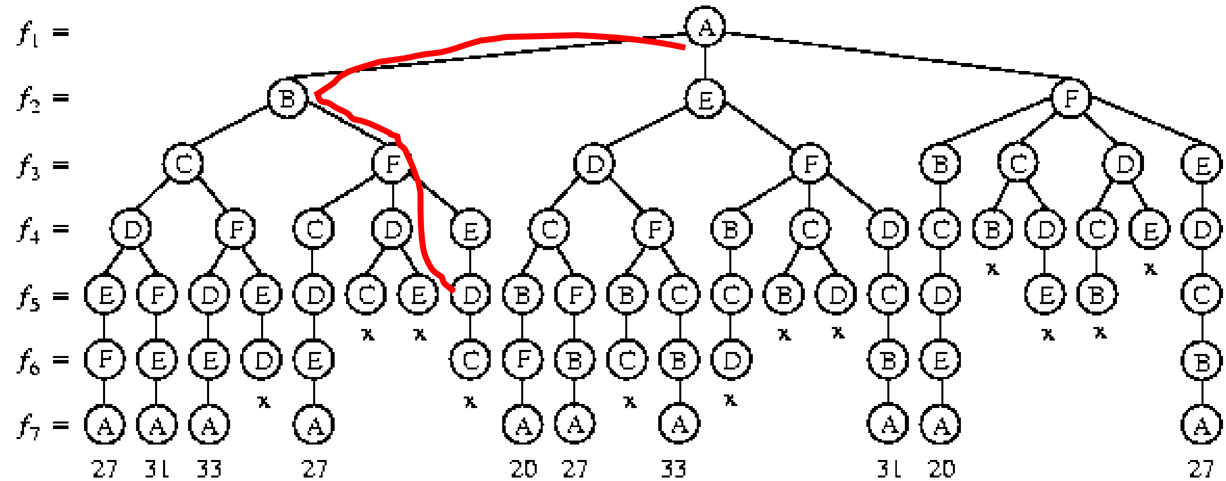


Branch and bound

Exhaustive Search v.s. Branch and Bound

- TSP example

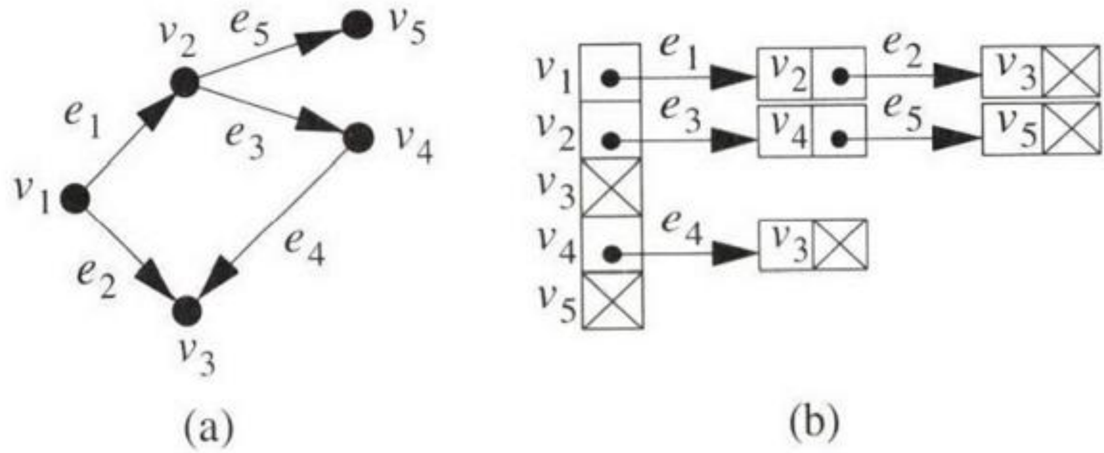




What if some edge weights are negative?

Backtracking/exhaustive search

Some programs require the graph to be complete. One way to "fool" them is to assign "infinite" weight to the "missing" edges

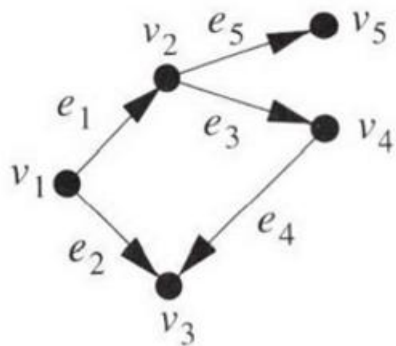


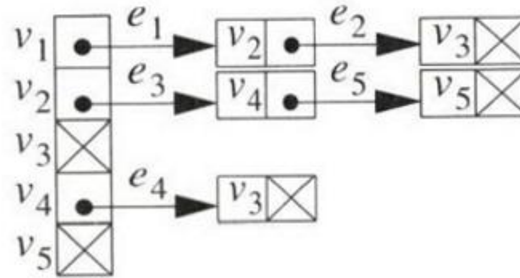
Depth first search

Figure 3.9 A directed graph (a) and its adjacency-list representation (b).

dfs(v_1)			
$\rightarrow e_1 = (v_1, v_2)$	dfs(v_2)		
	$\rightarrow e_3 = (v_2, v_4)$	dfs(v_4)	
		$\rightarrow e_4 = (v_4, v_3)$	dfs(v_3)
	$\rightarrow e_5 = (v_2, v_5)$	dfs(v_5)	
$\rightarrow e_2 = (v_1, v_3)$			

Figure 3.10 The different steps of the depth-first search algorithm applied to the graph of Figure 3.9(a).





Breadth first search

Q	w	edges processed
(v_1)	v_1	$e_1 = (v_1, v_2)$, $e_2 = (v_1, v_3)$
(v_2, v_3)	v_2	$e_3 = (v_2, v_4)$, $e_5 = (v_2, v_5)$
(v_3, v_4, v_5)	v_3	-
(v_4, v_5)	v_4	$e_4 = (v_4, v_3)$
(v_5)	v_5	-

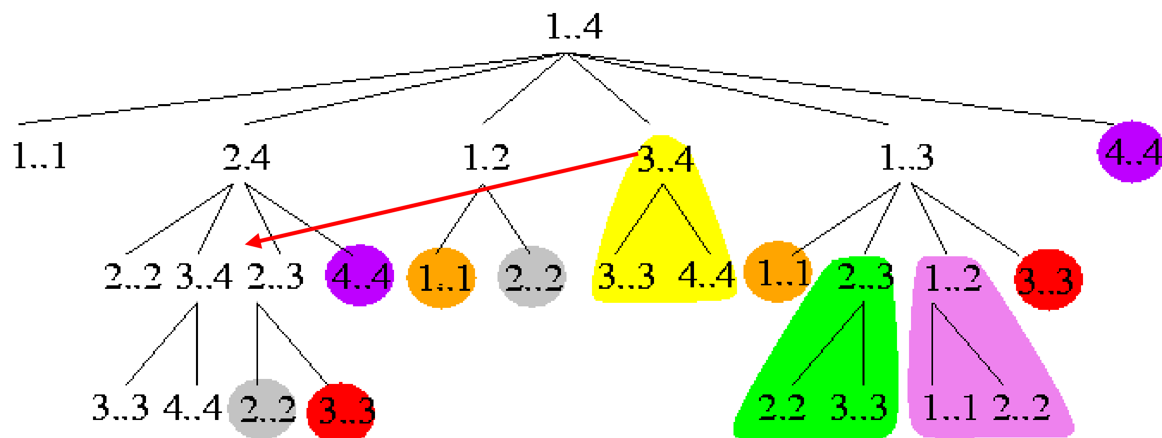
Figure 3.12 The different steps of the breadth-first search algorithm applied to the graph of Figure 3.9(a).

Algorithmic Paradigms

- **Exhaustive search:** Search the entire solution space.
- **Branch and bound:** A search technique with pruning.
- **Greedy method:** Pick a locally optimal solution at each step.
- **Dynamic programming:** Partition a problem into a collection of sub-problems, the sub-problems are solved, and then the original problem is solved by combining the solutions. (Applicable when the sub-problems are **NOT independent**).
- **Hierarchical approach:** Divide-and-conquer.
- **Mathematical programming:** A system of solving an objective function under constraints.
- **Simulated annealing:** An adaptive, iterative, non-deterministic algorithm that allows "uphill" moves to escape from local optima.
- **Tabu search:** Similar to simulated annealing, but does not decrease the chance of "uphill" moves throughout the search.
- **Genetic algorithm:** A population of solutions is stored and allowed to evolve through successive generations via mutation, crossover, etc.

Dynamic Programming (DP) v.s. Divide-and-Conquer

- Both solve problems by combining the solutions to subproblems.
- Divide-and-conquer algorithms
  - Partition a problem into **independent** subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.
    - Such as Quicksort (https://en.wikipedia.org/wiki/Quicksort)
  - Inefficient if they solve the same subproblem more than once.
- Dynamic programming (DP)
  - Applicable when the subproblems are **not independent**.
  - DP solves each subproblem just once.



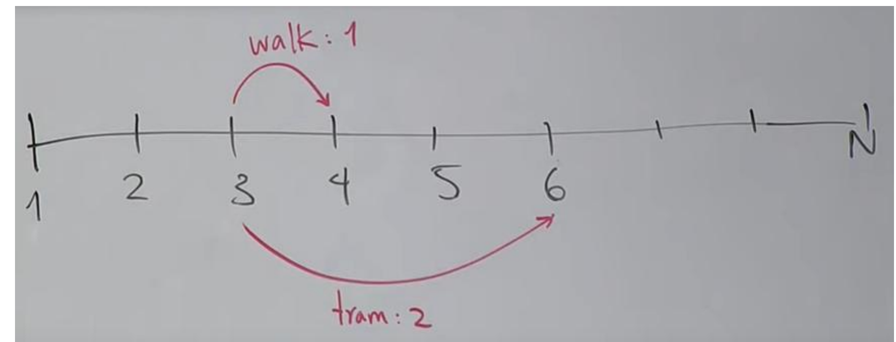
https://www.youtube.com/watch?v=aIsgJJYrlXk

Transportation example

Example: transportation

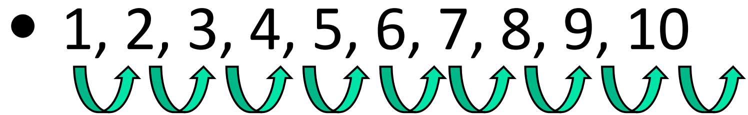
Street with blocks numbered 1 to n.
Walking from s to $s+1$ takes 1 minute.
Taking a magic tram from s to $2s$ takes 2 minutes.
How to travel from 1 to n in the least time?

- Backtracking search
- Depth first search
- Breadth first search
- Dynamic programming
- Uniform cost search



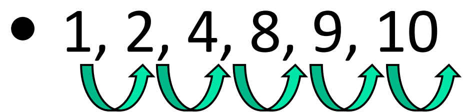
Exploring Solution Space

- Say N=10
- 1, 2, 3, 4, 5, 6, 7, 8, 9, 10



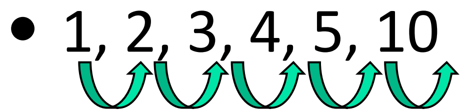
  - Total minutes: 9 = (1+…+1)

- 1, 2, 4, 8, 9, 10



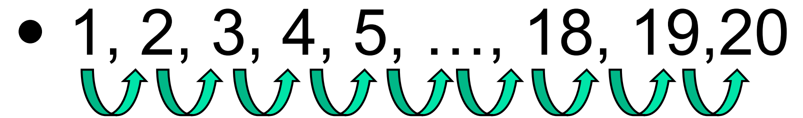
  - Total minutes: 7 = (1+2+2+1+1)

- 1, 2, 3, 4, 5, 10



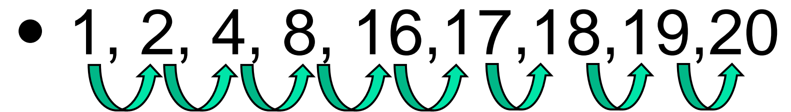
  - Total minutes: 6 = (1+1+1+1+2)

- Say N=20
- 1, 2, 3, 4, 5, …, 18, 19,20



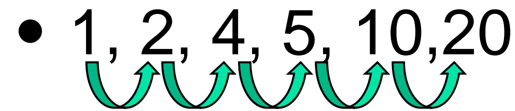
  - Total minutes: 19 = (1+…+1)

- 1, 2, 4, 8, 16,17,18,19,20



  - Total: 13 =(1+2+2+2+1+1+1+1)

- 1, 2, 4, 5, 10,20



  - Total minutes: 8 =(1+2+1+2+2)

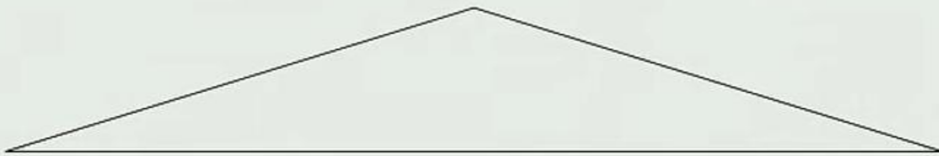
Video For Backtracking Programming

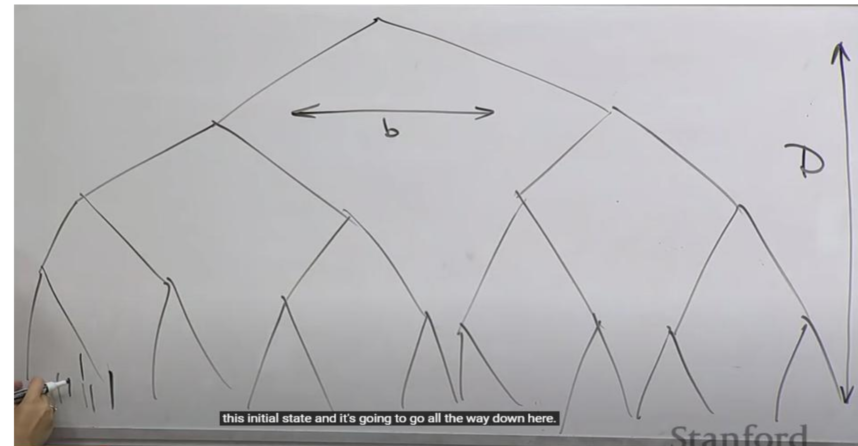
https://www.youtube.com/watch?v=aIsgJJYrlXk

From time 18:10, talk about transportation problem

•**Backtracking** search: traverse tree, down to leaf level, evaluate all the choices - exhaustive (from time: 23:20)

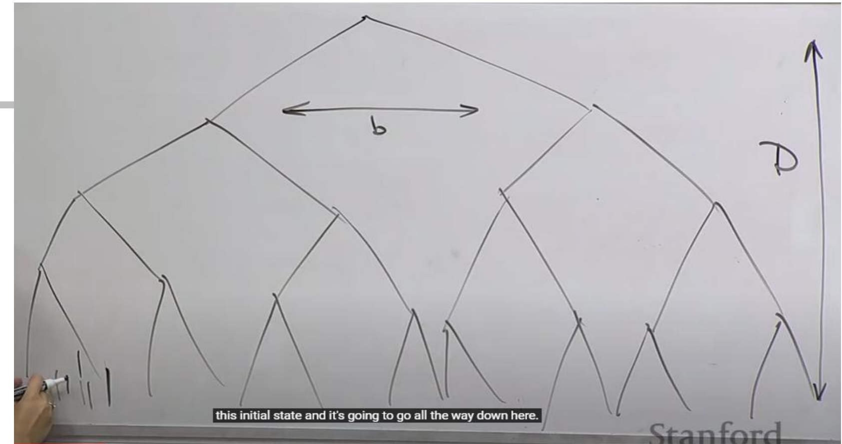
From time 50:57, it talks about dynamic programming
Running python program 'trans1.py'
(can comment out recursion limit -> then, N=400 will fail)





Backtracking search

[whiteboard: search tree]



If b actions per state, maximum depth is D actions:

- Memory: $O(D)$ (small)
- Time: $O(b^D)$ (huge) $[2^{50} = 1125899906842624]$

Backtracking search

Algorithm: backtracking search

def backtrackingSearch(s, path):
    If $\text{IsEnd}(s)$: update minimum cost path
    For each action $a \in \text{Actions}(s)$:
        Extend path with $\text{Succ}(s, a)$ and $\text{Cost}(s, a)$
        Call backtrackingSearch($\text{Succ}(s, a)$, path)
    Return minimum cost path

Breadth First Search & Depth First Search

- BFS

5. General-purpose Methods for Combinatorial Optimization 59

Figure 5.4 The search tree obtained by the exhaustive search backtracking algorithm for the example of Figure 5.3.

Algorithms	Cost	Time	Space
Backtracking Search	Any	$O(b^D)$	$O(D)$
DFS	0	$O(b^D)$	$O(D)$
BFS	$c \geq 0$	$O(b^d)$	$O(b^d)$

- DFS

5. General-purpose Methods for Combinatorial Optimization 59

Figure 5.4 The search tree obtained by the exhaustive search backtracking algorithm for the example of Figure 5.3.

DFS cost=0 was given in this problem

d = solution depth

Tree search algorithms

Legend: b actions/state, solution depth d, maximum depth D

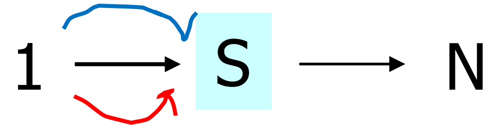
Algorithm	Action costs	Space	Time
Backtracking	any	$O(D)$	$O(b^D)$
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$

- Always exponential time
- Avoid exponential space with DFS-ID

DFS on **iterative deepening** with prefixed depth (kind of BFS)

https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/

Something We Notice



- From state-1 to S, there are many possible ways. Once from a path reaching S, we compute the cost from S to reach N, we can save the result for S. Then, from a new path 1->S, we do not need to recompute the cost from S->N

Video For Dynamic Programming

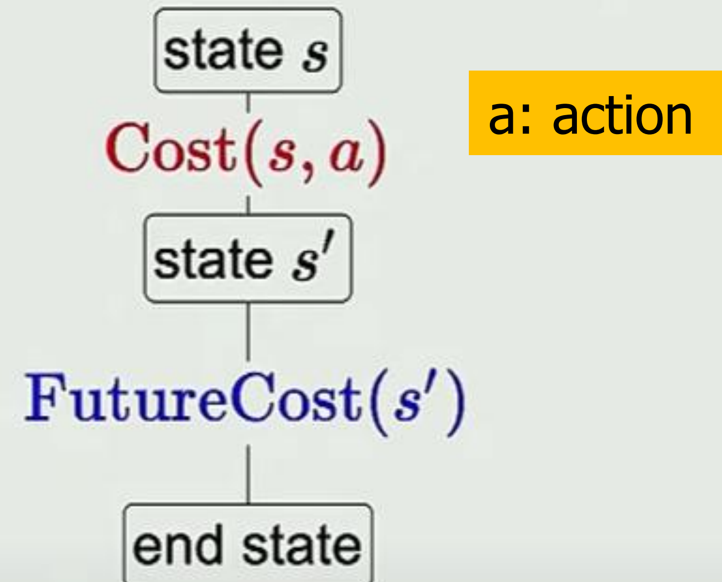
https://www.youtube.com/watch?v=aIsgJJYrlXk

From time 50:57, dynamic programming starts
Maybe running python program 'trans1.py'
(can comment out recursion limit -> then, N=400 will fail)

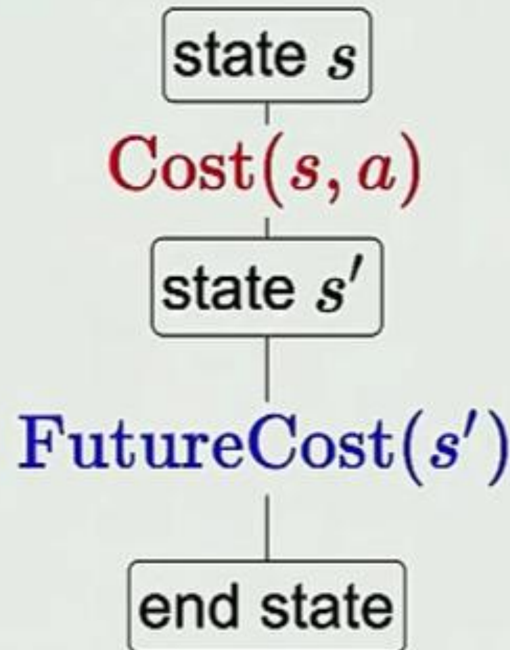
Example: transportation

Street with blocks numbered 1 to n.
Walking from s to $s+1$ takes 1 minute.
Taking a magic tram from s to $2s$ takes 2 minutes.
How to travel from 1 to n in the least time?

[semi-live solution: `TransportationProblem`]



Dynamic programming



Minimum cost path from state s to a end state:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if IsEnd}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

a: action

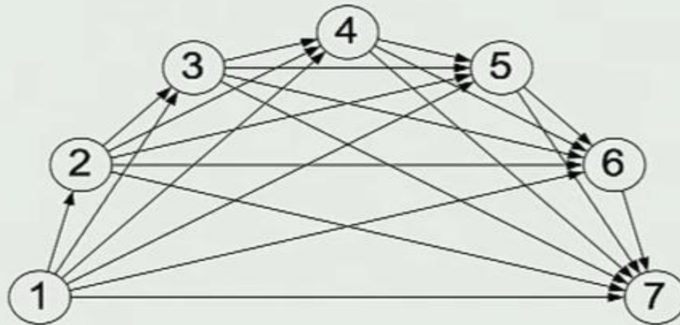
Succ: successor after taking action a

```
def dynamicProgramming (problem):
    cache = {}  # state -> futureCost (state)

    def futureCost(state):
        # base case
        if problem.isEnd (state):
            return 0

        if state in cache:  # exponential saving
            return cache [state]

        # acutally doing work
        result = min (cost + futureCost (newState) \
                for action, newState, cost in problem.succAndCost (state))

        cache [state] = result
        return result

    return (futureCost (problem.startState()), [])
```

Dynamic Programming

Dynamic programming

State: ~~past sequence of actions~~ current city



Exponential saving in time and space!

Key idea: state

A **state** is a summary of all the past actions sufficient to choose future actions **optimally**.

Similar to causality in control theory

past actions (all cities)	1 3 4 6
state (current city)	1 3 4 6

Video For Dynamic Programming

https://www.youtube.com/watch?v=aIsgJJYrlXk
From time 50:57, it is DP
Maybe running python program 'trans1.py'
(can comment out recursion limit -> then, N=400 will fail)

Algorithm: dynamic programming

def $\text{DynamicProgramming}(s)$:

  If already computed for s, return cached answer.

  If $\text{IsEnd}(s)$: return solution

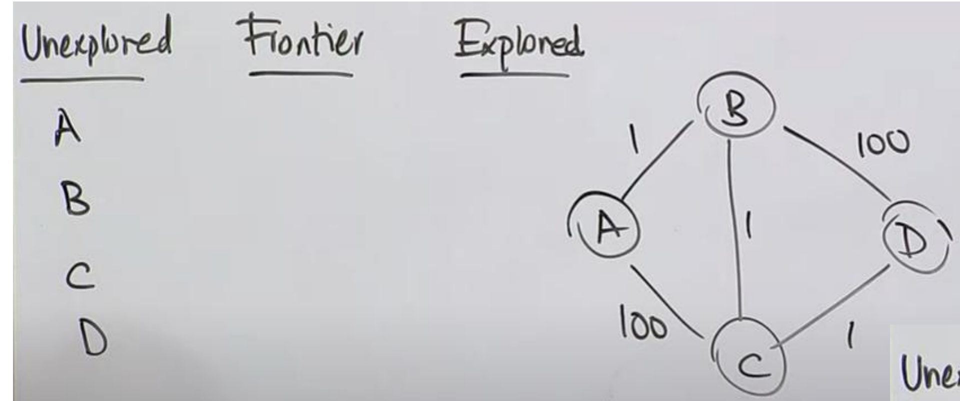
  For each action $a \in \text{Actions}(s)$: ...

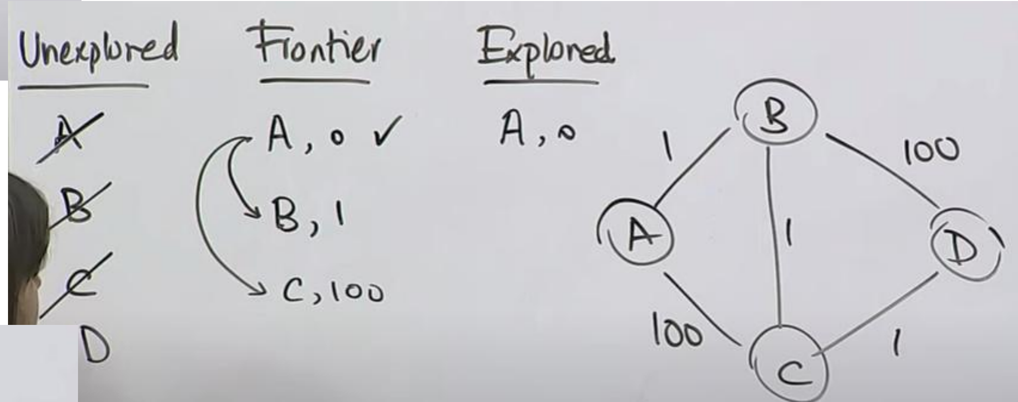
[semi-live solution: `Dynamic Programming`]

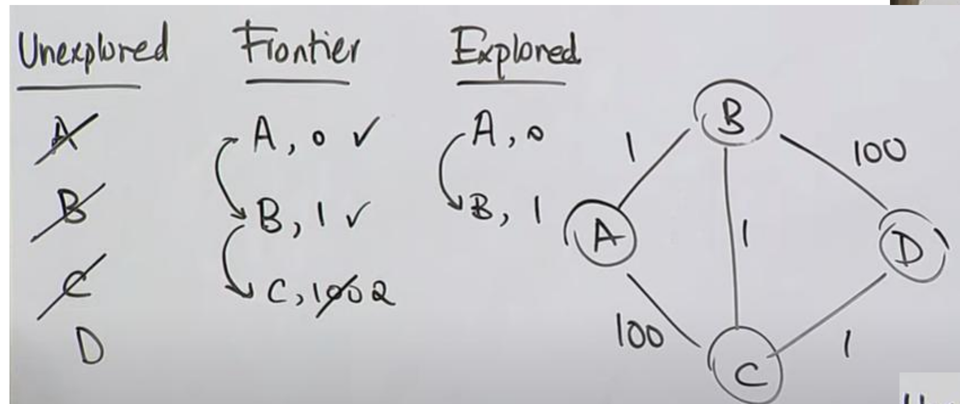
Assumption: acyclicity

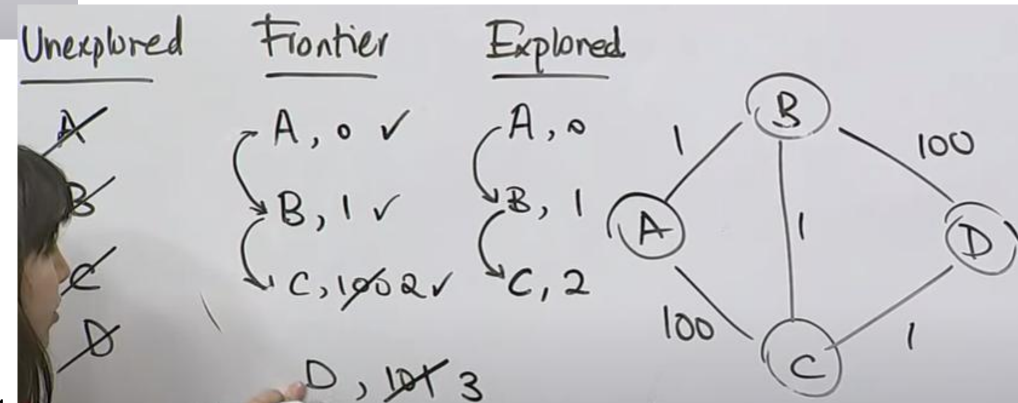
The state graph defined by $\text{Actions}(s)$ and $\text{Succ}(s, a)$ is acyclic.

When Cycles Exist









Uniform cost search (UCS)

Algorithm: uniform cost search [Dijkstra, 1956]

Add s_{start} to **frontier** (priority queue)
Repeat until frontier is empty:
    Remove s with smallest priority p from frontier
    If $\text{IsEnd}(s)$: return solution
    Add s to **explored**
    For each action $a \in \text{Actions}(s)$:
        Get successor $s' \leftarrow \text{Succ}(s, a)$
        If s' already in explored: continue
        Update **frontier** with s' and priority $p + \text{Cost}(s, a)$

cs221.stanford.edu/q

Question

Suppose we want to travel from city 1 to city n (going only forward) and back to city 1 (only going backward). It costs $c_{ij} \geq 0$ to go from i to j. Which of the following algorithms can be used to find the minimum cost path (select all that apply)?

- depth-first search
- breadth-first search
- dynamic programming
- uniform cost search

activate deactivate reset report

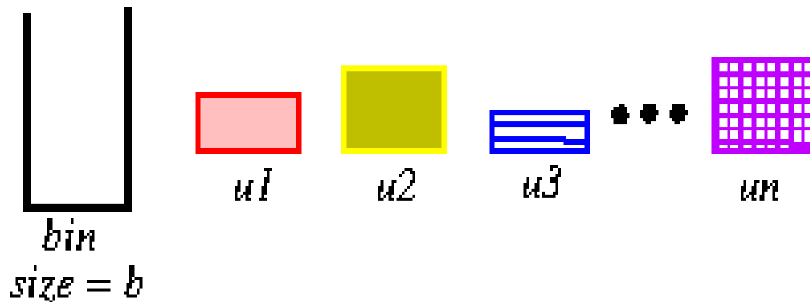
CS221 / Autumn 2019 / Liang & Sadigh

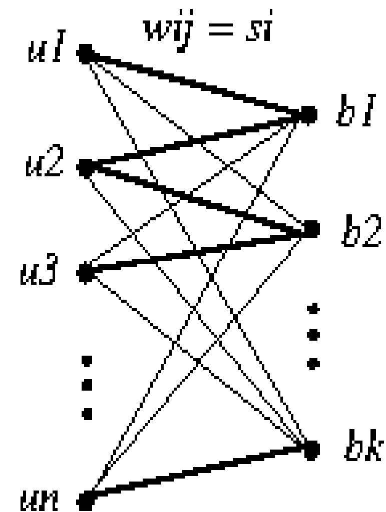
Stanford

Can not use DFS and BFS; But can use DP (just need to split the problem into two: forward to city-n; back to city-1, even though it kind of has loops)

Example: Bin Packing

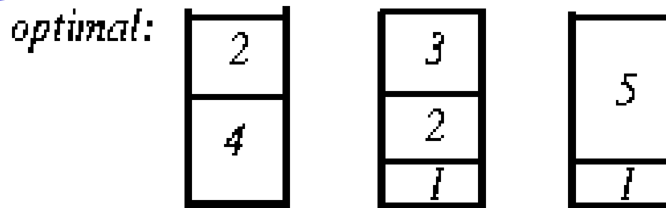
- **The Bin-Packing Problem Π :** Items $U = \{u_1, u_2, \ldots, u_n\}$, where u_i is of an integer size s_i; set B of bins, each with capacity b.
- **Goal:** Pack all items, minimizing # of bins used. (**NP-hard!**)





$Exp: b = 6,$ $\vec{S}$ = **(1, 4, 2, 1, 2, 3, 5)**

These are the sizes of items, not item's index.
Total 7 items



Example Applications

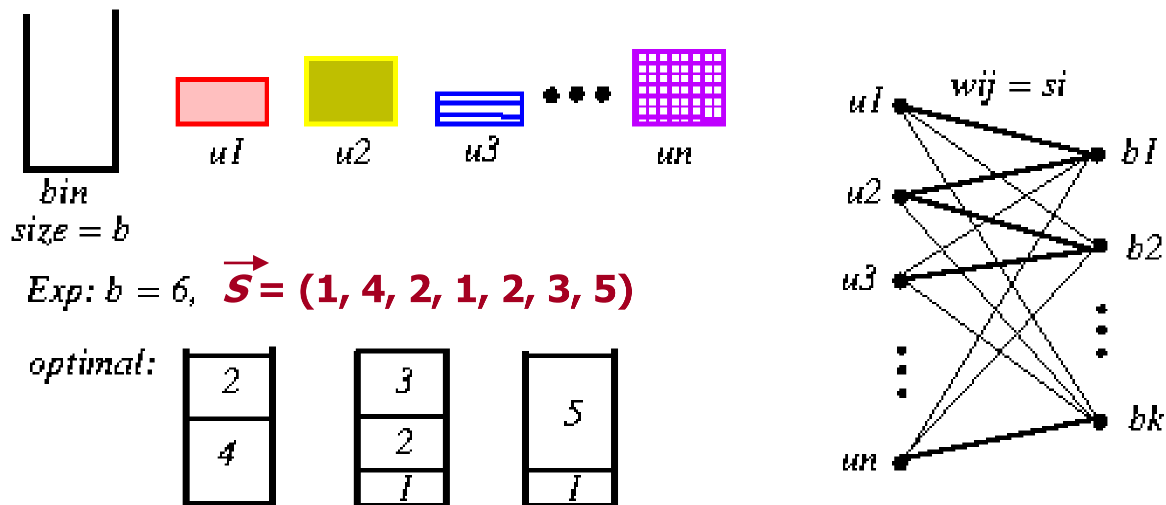
Filling recycle bins

Loading trucks

Historical Application

- Mix tapes

Algorithms for Bin Packing



- Greedy approximation alg.: First-Fit Decreasing (FFD)
  - *FFD*(Π) ≤ 11 OPT(Π)/9 + 4)
  - Order the items from largest to smallest; then, take items 1-by-1 to see which bin it can fit.
- Dynamic Programming? Hierarchical Approach? Genetic Algorithm? …
- Mathematical Programming: Use **integer linear programming (ILP)** to find a solution using |*B*| bins, then search for the smallest feasible |*B*|.

First-Fit Decreasing

$Exp: b = 6,\ \vec{S} = (1, 4, 2, 1, 2, 3, 5)$

First, we sort → 5, 4, 3, 2, 2, 1, 1
Take the first one not yet assigned: 5
Next is 4;
Next: 3
Next: 2 → can go to Bin2
Next: 2 → can go to Bin3
Next: 1 → can go to Bin1
Next: 1 → can go to Bin3

B1: 5 | (1)

B2: 4 | (2)

B3: 3 | (3)

B2: 4, 2 | (0)

B3: 3, 2 | (1)

B1: 5, 1 | (0)

B3: 3, 2, 1 | (0)

From Wikipedia

https://www.youtube.com/watch?v=qbuMPi44bVQ

Bin Packing Algorithms

- Online heuristics
  - Next fit
  - Next-k fit
  - First fit
  - Best fit
  - Worst fit
  - Almost worst fit
  - Refined first fit
  - ...
- Offline heuristics
  - Multiplicative approximation
    - First fit decreasing
    - Next fit decreasing
    - Modified first fir decreasing
    - ...

What if some items can not mixed in the same bin?

Sometimes we can not sort those items beforehand

ILP Formulation for Bin Packing

- 0-1 variable: x_{ij}=1 if item u_i is placed in bin b_j, 0 otherwise.

$$\max \sum_{(i,j)\in E} w_{ij}x_{ij}$$

subject to

$$\sum_{\forall i\in U} w_{ij}x_{ij} \leq b_j, \forall j \in B \quad /* \textit{ capacity constraint } */ \quad (1)$$

$$\sum_{\forall j\in B} x_{ij} = 1, \forall i \in U \quad /* \textit{ assignment constraint } */ \quad (2)$$

$$\sum_{ij} x_{ij} = n \quad /* \textit{ completeness constraint } */ \quad (3)$$

$$x_{ij} \in \{0,1\} \quad /* \textit{ 0, 1 constraint } */ \quad (4)$$

- **Step 1:** Set $|B|$ to the lower bound of the # of bins.
- **Step 2:** Use the ILP to find a feasible solution.
- **Step 3:** If the solution exists, the # of bins required is $|B|$. Then exit.
- **Step 4:** Otherwise, set $|B| \leftarrow |B| + 1$. Goto Step 2.

How to guess the lower bound?

Recap

- Go over how to reduce from one graph's HC problem to TSP
- Describe minimum spanning tree algorithm
- Using a TSP problem to describe
  - Exhaustive search, backtracking (BT), branch-and-bound
  - Depth first, breadth first, dynamic programming (DP), uniform cost(UC)
- Using a transportation problem for the above BT, DP, UC
  - Python codes is shown; welcome to study those codes
- Bin packing

Explaining hw