Lecture10
Dynamic Programming
NP-Complete

陆清怡

2022.12.16

991《数据结构与算法》考纲

9、算法基础

（1）字符串模式匹配算法。

（2）贪心法、分治法、动态规划的基本概念。

（3）计算复杂度类别的基本概念，NP-Complete 问题。

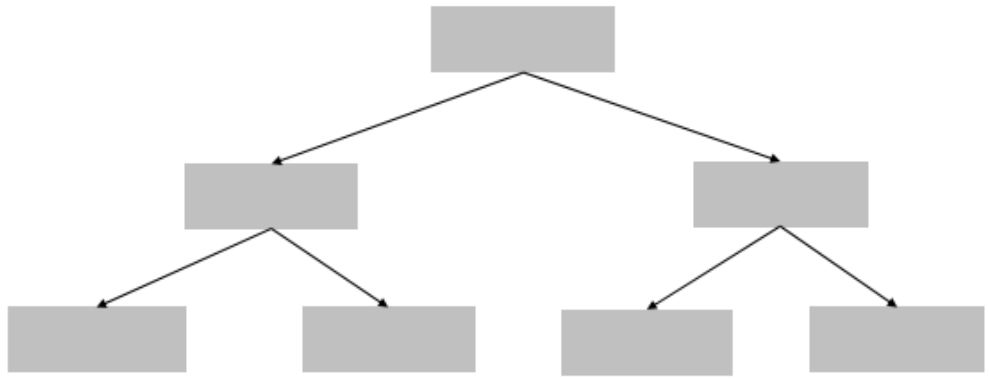
Dynamic Programming

Algorithmic paradigms

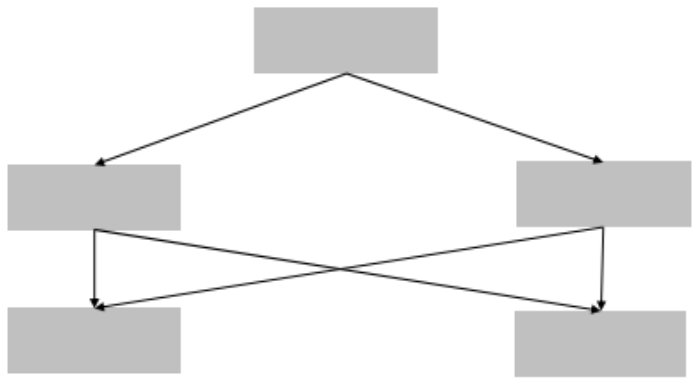
- Greedy Build up a solution incrementally, myopically optimizing some local criterion.
- Divide and conquer Break up a problem into a few sub-problems, solve each sub-problem independently and recursively, and combine solution to sub-problems to form solution to original problem.
- Dynamic programming Break up a problem into a series of overlapping sub-problems, and build up solutions to larger and larger sub-problems.
  - Very powerful and widely used technique in CS, OR, info and control theory.
  - Efficiently solves problems that otherwise seem intractable.
  - Name comes from dynamic “schedule” of subproblems the algorithm produces.

Algorithmic paradigms

Divide and conquer

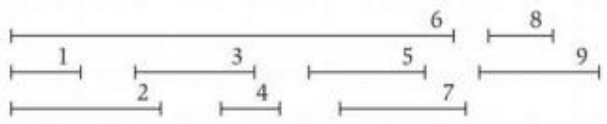


Dynamic programming

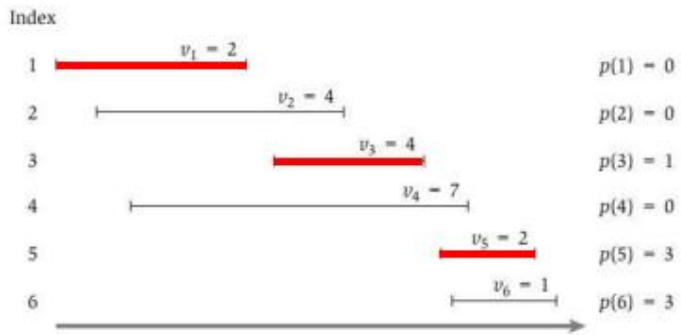


Weighted interval scheduling

- Recall the interval scheduling problem
  - Given a set of intervals, pick the largest set of nonoverlapping intervals.
  - For n intervals, solvable by a greedy algorithm in O(n log n) time.
- Weighted interval scheduling generalizes the problem so the intervals have weights.
  - Pick a set of nonoverlapping intervals with the largest combined weight.
  - No known natural greedy algorithm to solve this.

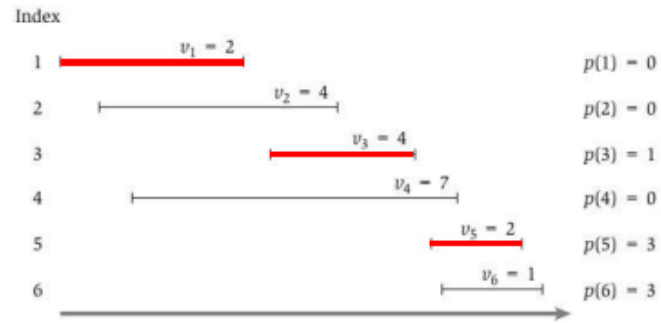


Source: Algorithm Design. Kleinberg, Tardos



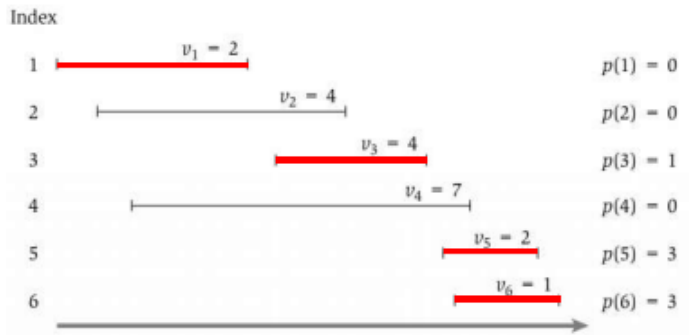
Compatible intervals

- Order the intervals by nondecreasing finishing times, as $I_1, I_2, \ldots, I_n$.
- Given interval I_j, let $p(j)$ be the maximum index k s.t. I_k finishes before I_j starts.
  - If no I_k finishes before I_j starts, let $p(j) = 0$.
- Suppose I_j and I_k are both used in the schedule, and $k < j$, then $k \leq p(j)$.
  - Otherwise I_k overlaps with I_j.



A recursive solution

- Let S^* be an optimal solution. Then either $I_n \in S^*$ or $I_n \notin S^*$.
- If $I_n \in S^*$
  - $I_j \notin S^*$ for all $j \in (p(n), n)$.
  - $S^* - \{I_n\}$ is an optimal solution for the intervals $I_1, I_2, \ldots, I_{p(n)}$.
    - I.e. the intervals in S^* besides I_n are a max weight set of non-overlapping intervals from $I_1, I_2, \ldots, I_{p(n)}$.
- If $I_n \notin S^*$
  - S^* is an optimal solution for the intervals $I_1, I_2, \ldots, I_{n-1}$.



Optimal substructure

- Optimal substructure property.
  - After making a decision, the rest of the solution should be optimal for the rest of the problem.
  - Ex After deciding whether to include I_n in S^*, the remaining solution $S^* - \{I_n\}$ is optimal for the remaining problem (either $I_1, I_2, \ldots, I_{p(n)}$, or $I_1, I_2, \ldots, I_{n-1}$).
- Optimal substructure is the key feature of dynamic programming.
  - Allows combining current partial solution and optimal subproblem solution to form optimal overall solution.
- Not all problems have optimal substructure.
  - For some problems, the current solution can't be combined with an optimal solution to a subproblem.

A recursive solution

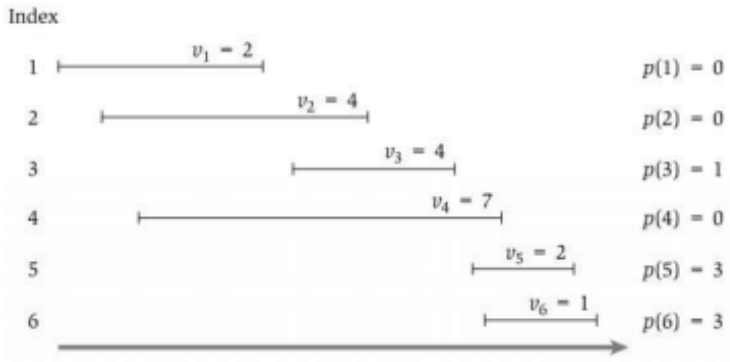
- Let $OPT(j)$ be the weight of a max weight non-overlapping subset of $I_1, \ldots, I_j$.
  - We want to find $OPT(n)$.
- Optimal substructure implies, for any $j \leq n$
  - If $I_j \in S^*$, then $\{I_k \in S^* \mid k < j\}$ is an optimal solution for the intervals $I_1, I_2, \ldots, I_{p(j)}$.
  - If $I_j \notin S^*$, then $\{I_k \in S^* \mid k < j\}$ is an optimal solution for the intervals $I_1, I_2, \ldots, I_{j-1}$.
- Write these as

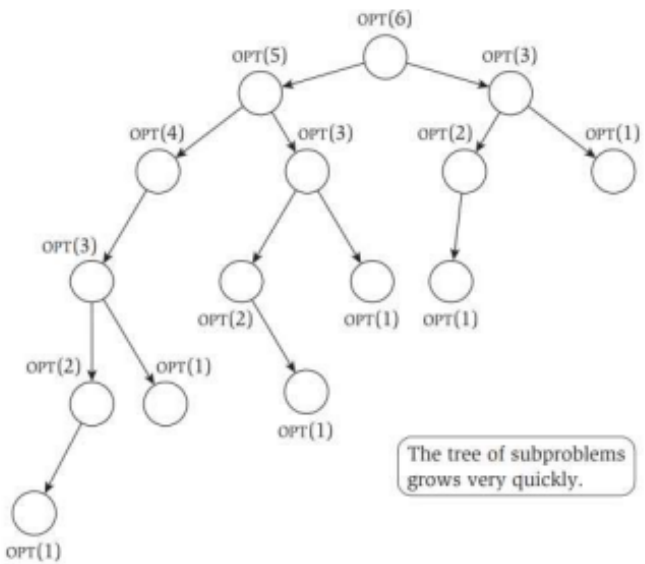
$$OPT(j) = \max\left(v_j + OPT\left(p(j)\right), OPT(j-1)\right)$$

  - First part of expression is when $I_j \in S^*$, and second part is when $I_j \notin S^*$.

A recursive solution

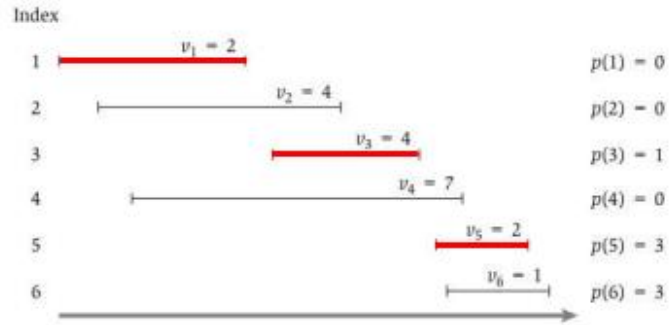
- Can use following recursive algorithm.
  - $OPT(j) = \max\left(v_j + OPT\left(p(j)\right), OPT(j-1)\right)$, for $j \geq 1$.
  - $OPT(0) = 0$.
- However, the number of subproblems increases exponentially, so this algorithm takes exponential time.
- But notice many of the calls are the same, e.g. we call $OPT(1), OPT(2), \ldots$ multiple times.
- Instead of computing $OPT(1), OPT(2), \ldots$ multiple times, we can compute them once and store their values.
  - If already computed $OPT(j)$, then when $OPT(k)$ needs to use $OPT(j)$, look up its value instead of running $OPT(j)$.
  - This is called memoization (notice there's no "r"), or the table method.



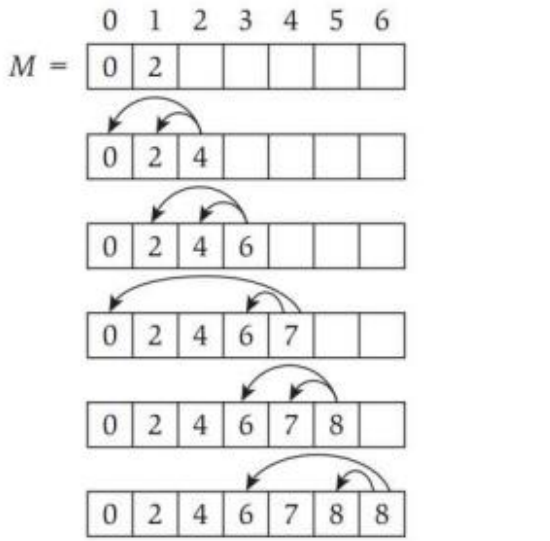


An iterative solution

```
Iterative-Compute-Opt
  M[0] = 0
  For j = 1, 2, ..., n
    M[j] = max(v_j + M[p(j)], M[j − 1])
  Endfor
```



- Use an array M to store the solutions of subproblems we've already solved.
- Solve the subproblems from smallest to largest, i.e. in increasing value of $M[j]$.
- For n intervals, takes $O(n)$ time if we know all the $p(j)$ values.
  - Can compute all the $p(j)$ values by sorting and binary search in $O(n \log n)$ time.



Matrix multiplication

$$\begin{bmatrix} 2 & 1 & 0 & 3 \\ 1 & 4 & 4 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 11 & 11 \\ 33 & 32 \end{bmatrix}$$

- Let A be a p × q matrix, B be a q × r matrix. Let C = A × B.
  - C is a p × r matrix.
- C_{ij} is the entry in the i'th row and j'th column of C.
  - It's the dot product of the i'th row of A and j'th column of B.

$$C_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

- Computing C takes O(pqr) time.
  - Time counted as number of multiplications.
  - C has pr entries.
  - Each entry formed by computing q products, then summing.

Multiplying many matrices

- Suppose we want to multiply A_1, A_2, A_3, A_4. This can be done in several ways.
  - $(A_1(A_2(A_3A_4)))$.
  - $(A_1((A_2A_3)A_4))$.
  - $((A_1A_2)(A_3A_4))$.
  - …
  - All give the same answer.
    - Because matrix multiplication is associative.
  - But they can take different amounts of time!

Same answer, different costs

- Multiplying a $p \times q$ matrix by a $q \times r$ matrix takes O(pqr) time.
- Let A_1, A_2, A_3 be 3 matrices with dimensions 10×100, 100×5, 5×50.
- Cost of $((A_1A_2)A_3)$ is 7500.
  - 10*100*5=5000 for A_1A_2, producing a 10×5 matrix.
  - Then another 10*5*50=2500 to multiply by A_3.
- Cost of $(A_1(A_2A_3))$ is 75,000.
  - 100*5*50=25000 for A_2A_3, producing a 100×50 matrix.
  - Then another 10*100*50=50000 to multiply by A_1.
- Same answer, but 10 times the cost!

Matrix-chain multiplication problem

- Given a sequence $A_1, A_2, \ldots, A_n$ of matrices, where A_i has dimensions $p_{i-1} \times p_i$, for i=1,…n, compute the product $A_1 \times A_2 \times \cdots \times A_n$ in a way that minimizes the cost.
- $A_1 \times A_2 \times \cdots \times A_n$ has dimensions $p_0 \times p_n$.
  - Same as $(((A_1 \times A_2) \times A_3) \ldots \times A_n)$.
  - $(A_1 \times A_2)$ has dimensions $p_0 \times p_2$.
  - $((A_1 \times A_2) \times A_3)$ has dimensions $p_0 \times p_3$.
  - $(((A_1 \times A_2) \times A_3) \times A_4)$ has dimensions $p_0 \times p_4$.
  - Etc.

Breaking into subproblems

- Suppose we want to multiply $A = A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7 \times A_8$ efficiently.
- Say we first compute $B = A_1 \times A_2 \times A_3 \times A_4$, then $C = A_5 \times A_6 \times A_7 \times A_8$, and then $A = B \times C$.
- B has dimensions $p_0 \times p_4$, C has dimensions $p_4 \times p_8$, so computing $B \times C$ takes $O(p_0 p_4 p_8)$ time.
- Let M(1,4) be the smallest cost to compute $A_1 \times A_2 \times A_3 \times A_4$, and M(5,8) the smallest cost to compute $A_5 \times A_6 \times A_7 \times A_8$.
- Then computing A by breaking it apart into B and C takes $M(1,4) + M(5,8) + O(p_0 p_4 p_8)$ time.
- Since we split A into two parts following A_4, we call this breaking at A_4.

Breaking into subproblems

- Alternatively, we can break at A_3.
- Compute $B' = A_1 \times A_2 \times A_3$, then $C' = A_4 \times A_5 \times A_6 \times A_7 \times A_8$, and then $A = B' \times C'$.
- B' has dimensions $p_0 \times p_3$, C' has dimensions $p_3 \times p_8$, so computing B'×C' takes $O(p_0 p_3 p_8)$ time.
- Let M(1,3) be the smallest cost to compute $A_1 \times A_2 \times A_3$, and M(4,8) the smallest cost to compute $A_4 \times A_5 \times A_6 \times A_7 \times A_8$.
- Then computing A by breaking it apart into B' and C' takes $M(1,3) + M(4,8) + O(p_0 p_3 p_8)$ time.

Breaking into subproblems

- Since there are 8 matrices, there are 7 ways we can break A into subproblems this way.
  - Breaking at A_1 has cost $C_1 = M(1,1) + M(2,8) + p_0p_1p_8$.
  - Breaking at A_2 has cost $C_2 = M(1,2) + M(3,8) + p_0p_2p_8$.
  - Breaking at A_3 has cost $C_3 = M(1,3) + M(4,8) + p_0p_3p_8$.
  - Breaking at A_4 has cost $C_4 = M(1,4) + M(5,8) + p_0p_4p_8$.
  - Breaking at A_5 has cost $C_5 = M(1,5) + M(6,8) + p_0p_5p_8$.
  - Breaking at A_6 has cost $C_6 = M(1,6) + M(7,8) + p_0p_6p_8$.
  - Breaking at A_7 has cost $C_7 = M(1,7) + M(8,8) + p_0p_7p_8$.

Breaking into subproblems

- Which split is best?
  - We don't know.
  - But one of the splits gives the best way to multiply $A_1 \times A_2 \times \cdots \times A_8$.
  - Because no matter how we parenthesize $A_1 \times A_2 \times \cdots \times A_8$, there's some multiplication that happens last.
    - This corresponds to the split position.
    - E.g. if we parenthesize as $\big(A_1 \times (A_2 \times A_3)\big) \times \Big((A_4 \times A_5) \times \big(A_6 \times (A_7 \times A_8)\big)\Big)$, the split occurs after A_3.
- So, the minimum cost M(A) to multiply $A_1 \times A_2 \times \cdots \times A_8$ is the minimum cost from one of the splittings.
- So $M(A) = \min(C_1, C_2, C_3, C_4, C_5, C_6, C_7)$.

Dynamic programming equation

- Let M(i,j) be the smallest time to multiply matrices of $A_i \times A_{i+1} \times \cdots \times A_j$. Then

$$M(1,n) = \min_{1 \le j \le n-1} (M(1,j) + M(j+1,n) + p_0 p_j p_n)$$

Cost of multiplying all the matrices $A_1, \ldots, A_n$.

Choose the best break point j

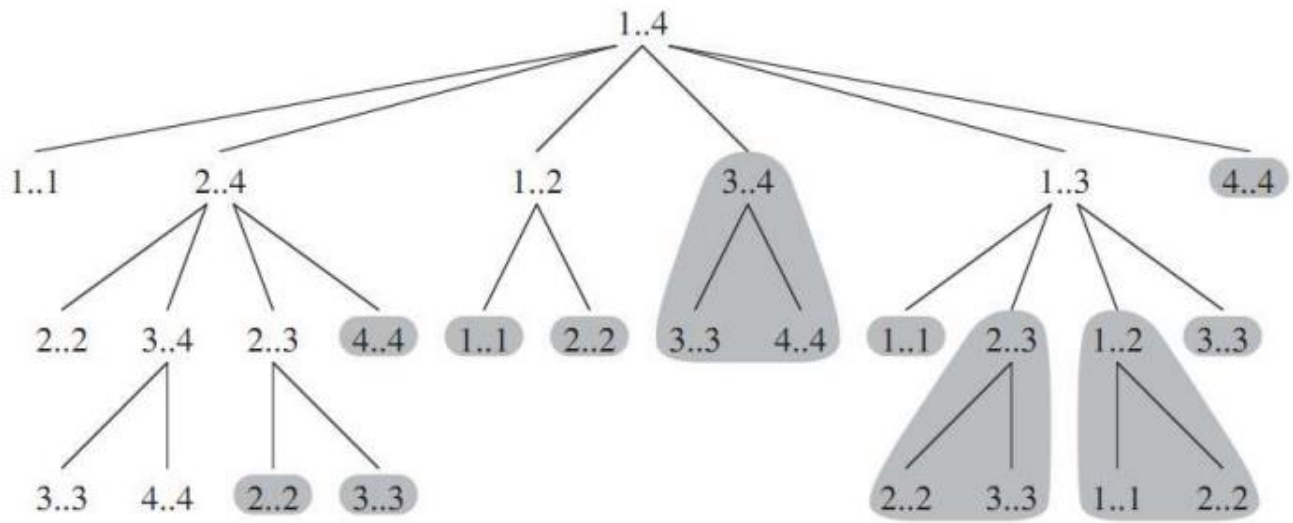
Cost of the first part

Cost of the second part

Cost of multiplying the matrices produced by the first and second parts

DP equation and subproblems

- Solving the main problem M(1,n) requires solving subproblems M(i,j), for $1 \le i,j \le n$.
- Solving each M(i,j) in turn requires solving smaller subproblems.
- Work bottom up. Solve smallest subproblems first, then combine the solutions to solutions of bigger subproblems.
- In the base case we have M(i,i)=0, for all i.
  - Because we just have matrix A_i, with no multiplications.

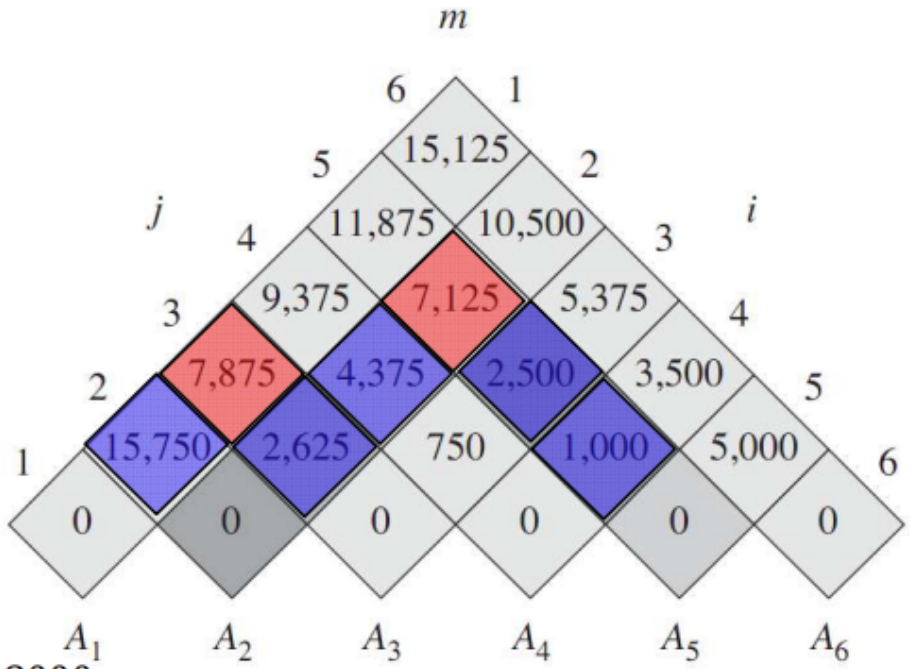


Source: Introduction to Algorithms, Cormen et al

The table method

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

- Compute bottom up, row by row.
- Final answer we want is M(1,6).
- Bottom row is all 0.
- In second to bottom row, $A(i, i+1) = p_{i-1}p_i p_{i+1}$ because we multiply A_i and A_{i+1}.
- Values in each row only depend on values in rows below, which we've already computed.



$$m[1,3] = \min \begin{cases} m[1,2] + m[3,3] + 30 \times 15 \times 5 = 18000 \\ m[1,1] + m[2,3] + 30 \times 35 \times 5 = 7875 \end{cases}$$

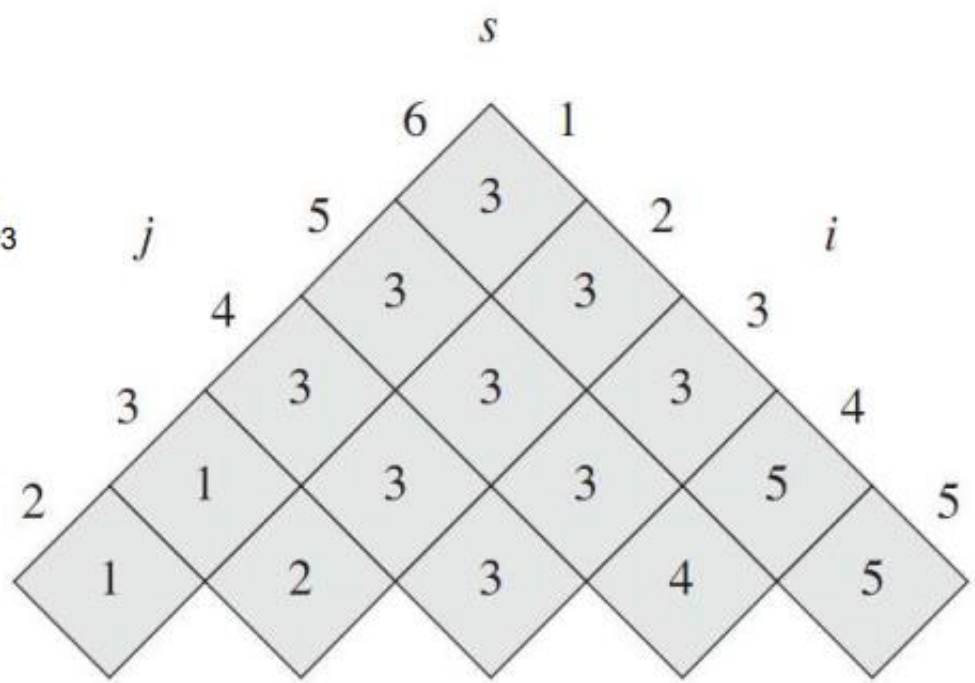
$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$

$$= 7125\,.$$

The table method

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

- The s table says where the breaks should happen.
- This comes from which term produced the min cost in the m table.
- Ex To compute m[2,5], we saw breaking at A_3 gave the min cost. So s[2,5]=3.
- Ex To compute m[1,6] first break at 3.
- So the subproblems are m[1,3] and m[4,6].
- To compute m[1,3], break at 1.
- So the subproblems are m[1,1] and m[2,3].
- To compute m[4,6], break at 5.
- So the subproblems are m[4,5] and m[6,6].
- So altogether the optimal sequence is $(A_1(A_2A_3))((A_4A_5)A_6)$.



Cost of the table method

- We have three nested loops, each of size O(n). So the total time cost is $O(n^3)$.
- More intuitively, we have n^2 table entries to fill in, where entry

  $$M(1,n) = \min_{1 \le j \le n-1} (M(1,j) + M(j+1,n) + p_0 p_j p_n)$$

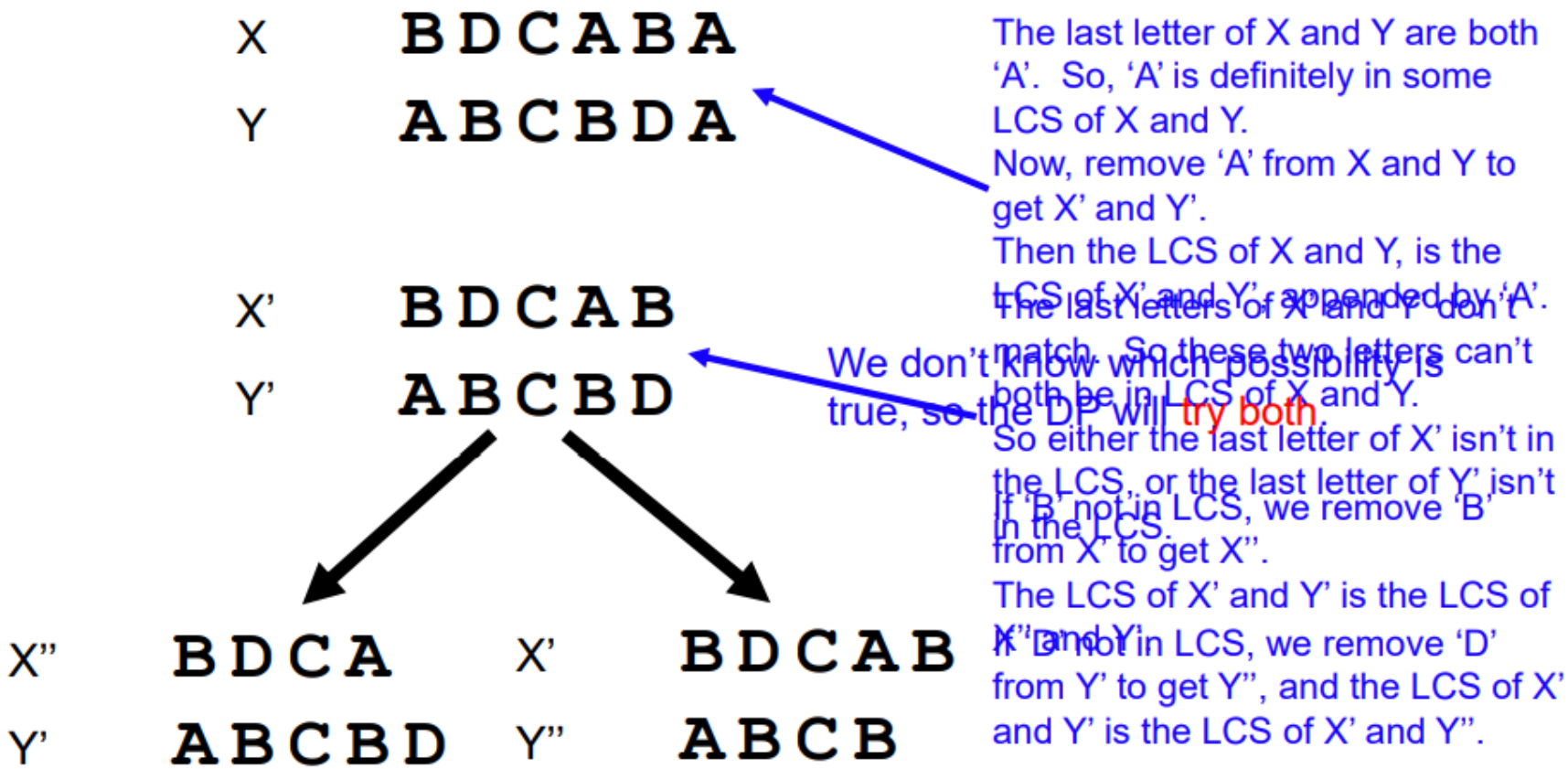
  involves checking O(n) other table entries. So the cost is $n^2 \times O(n) = O(n^3)$.
- Space complexity is $O(n^2)$, because we use two tables of size n^2.

Longest common subsequence

- Given a string X, a subsequence of X is any string formed by removing some letters of X.
  - Ex “let” is a subsequence of “letters”.
    - So are “les” and “ees”.
  - Subsequence isn’t necessarily consecutive.
- A common subsequence of strings X and Y is a subsequence of both X and Y.
  - E.g. “ees” is a common subsequence of both “letters” and “cheers”.
  - It’s not the longest common subsequence (LCS).
  - The longest common subsequence (LCS) of “letters” and “cheers” is “eers”.
  - There may be several LCS’s for some strings.

Subproblems in LCS

In dynamic programming, we divide a problem into smaller subproblems.
For LCS, we divide the problem on a long string to the problem on a shorter string, by removing the last letter of the long string.



The last letter of X and Y are both 'A'. So, 'A' is definitely in some LCS of X and Y.
Now, remove 'A' from X and Y to get X' and Y'.
Then the LCS of X and Y, is the LCS of X' and Y', appended by 'A'.

The last letters of X' and Y' don't match. So these two letters can't both be in LCS of X and Y.
So either the last letter of X' isn't in the LCS, or the last letter of Y' isn't in the LCS.

We don't know which possibility is true, so the DP will try both.

If 'B' not in LCS, we remove 'B' from X' to get X".
The LCS of X' and Y' is the LCS of X" and Y'.

If 'D' not in LCS, we remove 'D' from Y' to get Y", and the LCS of X' and Y' is the LCS of X' and Y".

A dynamic program for LCS

- Let S[1,i] be the first i letters of a string S, and S[i] be the i'th letter of S.
  - Let S[i,0] be the empty string, for any i.
- Let LCS(X[1,i], Y[1,j]) be the LCS of X[1,i] and Y[1,j].
- Let c(i,j) be the length of LCS(X[1,i], Y[1,j]).
- We have the following dynamic programming equations.

$$c[i,0] = 0 \text{ for all } i$$
$$c[0,j] = 0 \text{ for all } j$$

c[i,0] is LCS of X[1,i] and Y[1,0]. Since Y[1,0] is empty string, it has no LCS with X[1,i]. Similarly for c[0,j].

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i-1,j], c[i, j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$

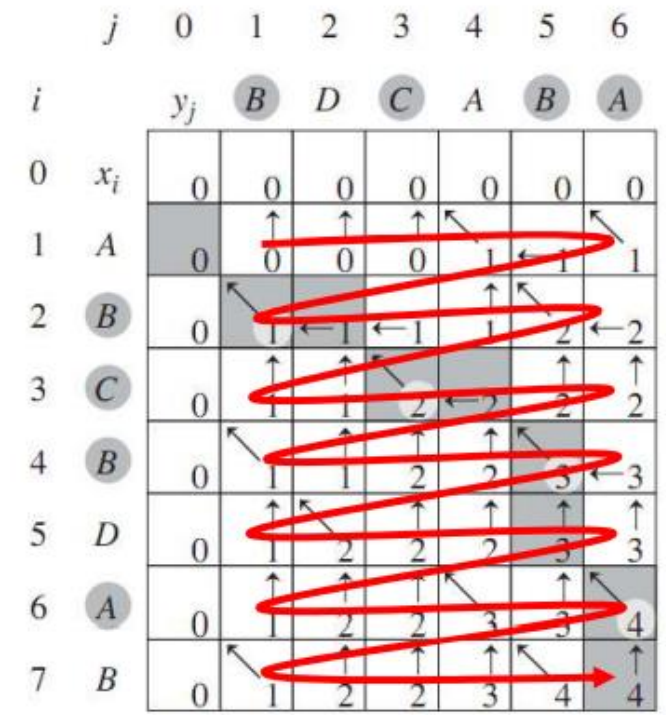
From argument on last slide.

The table method for LCS

- Make a table to record the c[i,j] values.
- Row is prefixes of Y, column is prefixes of X.
- c[i,j] is length of LCS(X[1,i],Y[1,j]).
- Start with the 0 column and 0 row. Fill in all 0's.
- Fill in rest of table from left to right, and from top to bottom. I.e. (1,1), (1,2),(1,3),…, (2,1),(2,2),…
  - Because a cell's value depends on vals in cells to the left, left-up, and up.

$$c[i,0] = 0 \text{ for all } i.$$
$$c[0,j] = 0 \text{ for all } j.$$

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i-1,j], c[i, j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$



Source: Introduction to Algorithms, Cormen et al.

The table method for LCS

- If the letters in row i and column j match, val of cell (i,j) = val of cell (i-1,j-1)+1.
  - Also, make a diagonal arrow, indicating LCS(X[1,i],Y[1,j]) is LCS(X[1,i-1],Y[1,j-1]) plus X[i] (or Y[j]).
- If letters don't match, val of cell (i,j) = max of vals in cells (i-1,j), (i,j-1).
  - Make an arrow to whichever cell gives has higher val (break ties arbitrarily).
  - Arrow indicates which way to go to find LCS.
- Length of LCS is in bottom right cell (4 in the example).
- LCS string given by following arrows starting from bottom right.
  - Each diagonal arrow corresponds to a matching letter.
  - The LCS is BCBA in the example.

c[i,0] = 0 for all i.
c[0,j] = 0 for all j.

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i-1,j], c[i, j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Cost of the LCS DP

- There are $O(mn)$ entries in the c table.
- Filling each entry requires looking at 1 or 2 adjacent entries.
- So running time $O(mn)$.
- Amount of memory needed (space complexity) is also $O(mn)$, since tables have mn entries.

NP-Complete

The class P

- A polynomial time (polytime) algorithm is one that runs in $O(n^k)$ time, for some constant k, when input has size n.
- P is the set of all problems that can be solved by a polytime algorithm.
  - These problems are called "efficiently computable", because a polytime algorithm is considered efficient.
  - In practice though, an e.g. $O(n^3)$ algorithm is quite slow, even for moderate sized n.
- If a problem takes $\omega(n^k)$ time, for any constant k, it's considered not efficiently solvable.
  - Ex An $\Omega(2^n)$ time or $\Omega(n!)$ time algorithm isn't efficient.
  - We only know how to 3-color a map in $\Omega(3^n)$ time (more or less), so 3-coloring (currently) can't be solved efficiently.
  - An $\Omega(3^n)$ time algorithm is much slower than an $O(n^3)$ algorithm.
    - Ex If n=10000, then $n^3 = 10^{12}$, but $3^n = 1.6 \times 10^{4771}$.

The class NP

- NP = Nondeterministic polynomial time.
- Def An instance of a problem consists of an input for the problem.
  - Ex An instance of the sorting problem is a set {3,1,2,4} that we want to sort.
  - Ex An instance of the SSSP problem is a weighted graph along with a source node.
- P is the class of problems for which all instances can be solved in polynomial time by some algorithm.
- NP is the class of problems for which the solvability of an instance can be verified in polynomial time.
  - The verification is done by a "verifier" algorithm.
  - The verifier needs an additional "hint" to work correctly.
    - The hint is also called a "witness" or "certificate".
  - The verifier doesn't find a solution to a problem instance, but only checks that the instance has been solved.

The class NP

- The verifier has the following properties.
  - The verifier's input is a problem instance x, and a certificate y.
  - The verifier's output is either "accept" or "reject".
  - If x has a solution, then if y is a "good" certificate, the verifier will output accept.
    - If y is not a "good" certificate, the verifier can either accept or reject.
  - If x has no solution, the verifier rejects no matter what y is.
  - Intuitively, the certificate y indicates x is solvable.
    - For example, y can be a solution to x.
    - But y can also be an indirect representation of a solution.
  - The verifier is efficient, i.e. runs in polynomial time.

The class NP, formally

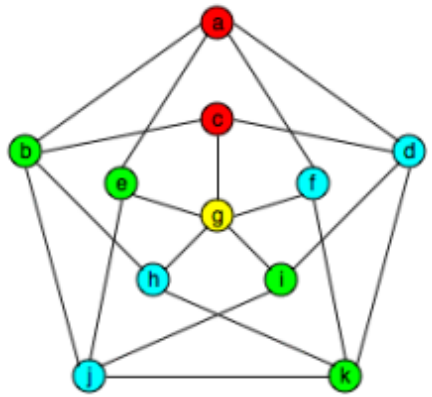
- Def A decision problem is a problem with a yes / no answer.
  - Ex Given a graph, is there a path from node s to t?
  - Ex Given a map, is there a way to 3-color it?
  - Ex Given a number, is it prime?
- Def Given a decision problem, the set of yes (resp. no) instances are the instances of the problem for which the answer is yes (resp. no).
  - Ex 11 is a yes instance to the prime problem, 10 is a no instance.
- Def Given a decision problem A, a polynomial time verifier V for A is an algorithm that does the following
  - V's input is an instance x of A, and a certificate string y.
  - $V(x, y) \in \{0,1\}$, representing "reject" and "accept", resp.
  - If x is a yes instance, there exists a y for which V outputs 1, i.e. $\exists y: V(x, y) = 1$.
  - If x is a no instance, every y makes V output 0, i.e. $\forall y: V(x, y) = 0$.
  - V runs in polynomial time.
- NP is the set of all decision problems with polytime verifiers.

Showing a problem is in P or NP

- To show a problem is in P, give an algorithm solving the problem that runs in polynomial time.
- To show a decision problem is in NP, give a polynomial time verifier for the problem satisfying the properties on the previous slide.
  - This requires specifying what the cerficates are, and how the verifier operates, given an instance of the problem and a certificate.

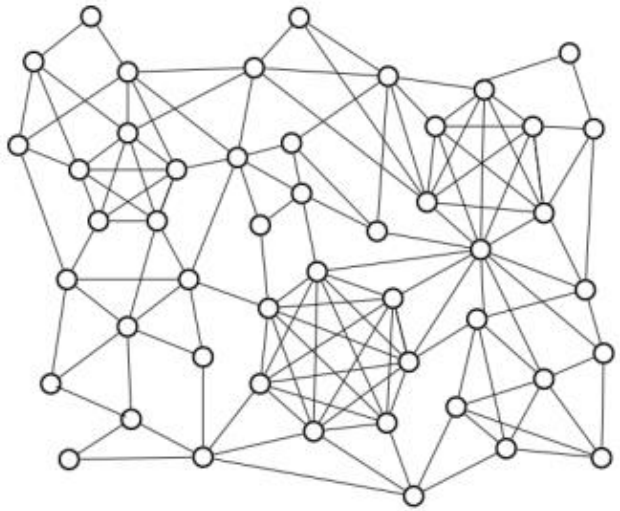
4-coloring is in NP

- Given a graph, can we assign each vertex one of 4 colors, such that adjacent vertices have different colors?
- Verifier
  - Certificate y is an assignment of colors to the vertices of graph x.
  - Check y uses at most 4 colors. If not, output no.
  - Go through all edges of x, and checks endpoints of each edge have different colors.
  - If true for all edges, output 1. Else output 0.
- If x has solution
  - Then x is 4-colorable.
  - So there's way to assign each vertex one of 4 colors s.t. endpoints of each edge have different colors.
  - Let y be this assignment, and give y to V.
  - Clearly V outputs 1.
- x has no solution
  - Then x is not 4-colorable.
  - So no matter how we assign 4 colors to vertices of x, some edge has endpoints with the same order.
  - So V outputs 0, for any input y.
- V runs in polytime.
  - If x has n vertices, then it has $O(n^2)$ edges, so V runs in $O(n^2)$ time.



k-Clique is in NP

- Given a graph with n nodes and a number k, are there k nodes that form a clique, i.e. that are all connected to each other?
- Verifier
  - Certificate y is a set of k nodes in x.
  - Check each pair of the k nodes is connected by an edge. If so, output 1. Otherwise output 0.
- If x has a solution.
  - Then there are k nodes that are mutually connected.
  - Call this set y and give it to V.
  - Clearly V outputs 1.
- If x has no solution.
  - Then in any set of k nodes, some 2 nodes aren't connected.
  - So V outputs 0, no matter what set of k nodes it gets.
- V runs in polytime.
  - Checking k nodes are mutually connected takes $O(k^2)$ time.

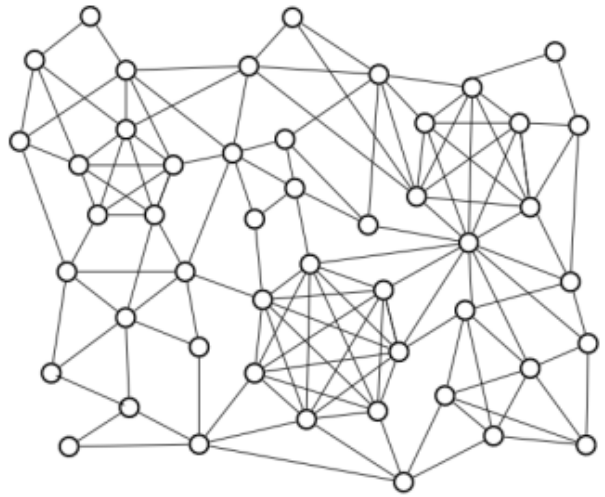


All problems in P are in NP

- Let A be a problem in P. I.e. there's a polytime algorithm S s.t. on every instance x of A
  - If x has a solution, S returns a solution.
  - If x has no solution, S returns fail.
- Verifier
  - V runs S. If S finds a solution, V outputs 1. Otherwise V outputs 0.
- If x has a solution.
  - S finds a solution, so V outputs 1.
- If x has no solution.
  - S returns fail, so V outputs 0.
- V runs in polytime.
  - Because V just runs S, which runs in polytime.
- Notice that for problems in P, V doesn't need a certificate y.
  - For problems in P, it's easy to determine if they're solvable or not.
- But for hard problems (not in P), V isn't powerful enough to determine solvability by itself.
  - So it needs a hint / witness / certificate.
- Ex In factoring, a polytime verifier isn't powerful enough to find a nontrivial factor of an input.
  - But if it's given a nontrivial factor, it can check the factor works in polytime, and therefore verify the input is composite.

Incorrect verifiers

- We showed k-Clique is in NP by giving a correct verifier.
- Let's see some incorrect verifiers.
  - None of these verifiers can be used to prove k-Clique is in NP.
- Verifier 1 Always outputs 1, regardless of y.
  - Wrong, because when graph doesn't contain a k-clique, V is supposed to output 0.
- Verifier 2 Always output 0, regardless of y.
  - Wrong, because when the graph does contain a k-clique, V is supposed to output 1, for some y.
- Verifier 3 Check all subsets of k nodes. If any form a clique, output 1, else output 0.
  - Seems OK. When x has a k-clique, V outputs 1, and when x doesn't, it outputs 0.
  - But V is still wrong, because it doesn't run in polytime.
    - There are $O(n^k)$ subsets of k nodes, and V checks all of them.



P vs NP

- Does P=NP?
  - I.e. suppose there's a problem for which we can verify solvability in polynomial time. Does that mean we can actually find a solution in polynomial time?
- This is the arguably the most important question in computer science.
  - The other would be to produce general AI.
- Many real-world problems are in NP. If P=NP, we can solve them efficiently. If P≠NP, then we can't.
- Every P problem is in NP, as we saw. So $P \subseteq NP$.
- Is every NP problem in P, i.e. $NP \subseteq P$?
- After 50 years, nobody knows.
  - Most, but not all researchers think not all NP problems are in P.
  - There are probably problems we can efficiently verify but not efficiently solve.
  - Ex Factoring is something we can efficiently verify, but not solve.
- If you can prove $P \neq NP$, or even better, $P = NP$, then
  - you ≥ Newton ≥ Einstein ≥ …
  - You also get $1M from the Clay Math Institute.
- Answering this question has vast and profound implications for CS, AI, math, physics, etc.

NP-completeness

- Out of all the NP problems, there's a subset of NP problems called NP-complete (NPC) problems that are the "hardest" NP problems.
- To determine whether P=NP, it suffices to know whether P=NPC.
  - If the hardest problems can be solved in polytime, then all NP problems can be solved in polytime. I.e. P=NP.
- So the study of P vs NP focuses on NPC problems.

Hardness and reductions

- What does it mean to say problem B is harder than problem A?
- It means if you can solve B, you can also solve A.
  - Ex Algebra is harder than arithmetic, because if you can do algebra, you can also do arithmetic.
  - So if I have an algorithm for solving B, I can use it to solve A.
- We say A reduces to B.
  - Write $A \leq_R B$.
  - Read this as “A is equally or less difficult than B”.

Example

- FACTOR-ALL(n) finds all the factors of a number n.
- FACTOR-1(n) finds one factor.
- Of course, FACTOR-1 $\leq_R$ FACTOR-ALL.
  - If we can find all the factors, we can certainly find one.
- FACTOR-ALL $\leq_R$ FACTOR-1.
  - We use FACTOR-1(n) to find one factor m of n.
  - Then divide n by m, and run FACTOR-1 on the result, to find another factor of n.
  - Keep repeating the previous steps until we get all the factors.
- The hard part about factoring a number, is just to find one factor.
  - Since FACTOR-1 $\leq_R$ FACTOR-ALL, FACTOR-ALL $\leq_R$ FACTOR-1, these problems have the same hardness.

Reductions, formally

- Let A and B be two decision problems.
- Let X and Y be the set of yes instances for A and B, resp.
- Ex Say A = PRIME and B = k-CLIQUE.
  - X is the set of prime numbers.
  - Y is the set of graphs containing a k-clique.
- Let f be a function that maps instances of A to instances of B.
- Def A reduces to B if there exists $f: A \rightarrow B$ s.t. for all instances x of A, $x \in X \Leftrightarrow f(x) \in Y$.
  - We write $A \leq_R B$.
- To show $A \leq_R B$, just give the mapping f.
- If $A \leq_R B$, then we can use an algorithm for B to solve A.
  - To solve an instance of A, first map it to an instance of B using f.
  - Then run the B algorithm.
  - Return the same answer for A as the B algorithm gives.
  - By definition, A is true ⇔ f(A) is true.

Example

- Suppose we want to show PRIME $\leq_R$ k-CLIQUE.
- This means there's some mapping f such that.
  - Given an instance of PRIME, i.e. a number n.
  - f(n) is an instance of k-CLIQUE, i.e. f(n) is a graph G.
  - n is prime if and only if f(n) contains a k-clique.
- If we have an algorithm to solve k-CLIQUE, we can use it solve PRIME.
  - To tell if n is prime, map n to a graph G and run the k-CLIQUE algorithm on G.
  - If it returns true, n is prime. Otherwise n isn't.

Polynomial time reductions

- If the mapping function from A to B runs in polynomial time, then it's a polynomial time reduction, and we write $A \leq_P B$.
  - Ex If we're reducing PRIME to k-CLIQUE, then the function to generate a graph from a number must run in polytime.
- Thm 1 Let A, B and C be three problems, and suppose $A \leq_P B$ and $B \leq_P C$. Then $A \leq_P C$.
- Proof Since $A \leq_P B$, there's a polytime mapping f from instances of A to instances of B.
  - Since $B \leq_P C$, there's a polytime mapping g from instances of B to instances of C.
  - Given an instance X of A, let $Y = f(X)$, and $Z = g(Y) = g(f(X))$.
  - Then X is a yes instance of A ⇔ Y is a yes instance of B ⇔ Z is a yes instance of C.
  - So $g \circ f$ is a valid mapping of A to C.
  - Since f and g are both polytime, $g \circ f$ is also polytime.

NP-completeness

- Def A problem A is NP-complete (NPC) if the following are true.
  - $A \in NP$.
  - Given any other problem $B \in NP$, $B \leq_P A$.
- Thus, a NP-complete problem is an NP problem that can be used to solve any other NP problem.
  - It's a "hardest" NP problem.

NP-completeness and SAT

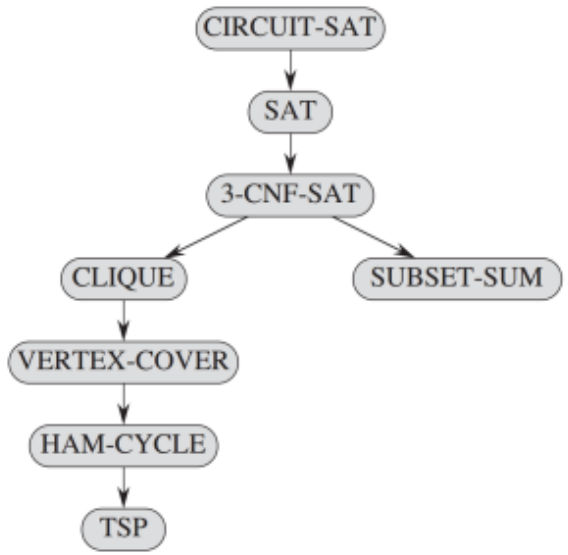
- Do NP-complete problems really exist?
  - Can we really find an NP problem that can be used to solve every other NP problem?
  - One problem to rule them all?
- Yes! Steve Cook and Leonid Levin proved around 1970 that SAT is NP-complete.
- SAT = satisfiable Boolean formulas.
  - Given a Boolean formula, is there any setting for the variables which makes the formula true?
  - Ex $(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee \neg B \vee \neg D) \in SAT$.
    - Setting A=B=C=true, D=false makes the formula true.
  - Ex $A \wedge \neg A \notin SAT$.
    - The formula's false for all settings of A.

NP-completeness and SAT

- Cook-Levin theorem says 2 things.
  - SAT$\in$NP.
    - Prove this yourself.
  - Every NP problem reduces to SAT. I.e. every problem A in NP can be mapped to an SAT formula ϕ in polytime, such that
    - If A is true, then ϕ is satisfiable.
    - If A is false, then ϕ is not satisfiable.
- Basic idea of the theorem is to use the logical operations in a SAT formula to emulate the logical operations in any algorithm.
  - Any NP problem X has a polytime verifier V. The Cook-Levin theorem uses a SAT formula ϕ to emulate the verifier's operations.
  - For a yes instance of X, there's some certificate making V return 1.
    - The certificate can be transformed to a satisfying truth setting for ϕ.
  - Any certificate making V return 0 corresponds to a non-satisfying truth setting for ϕ.
  - So $\phi \in SAT$ if and only if X is a yes instance, and $X \leq_P SAT$.

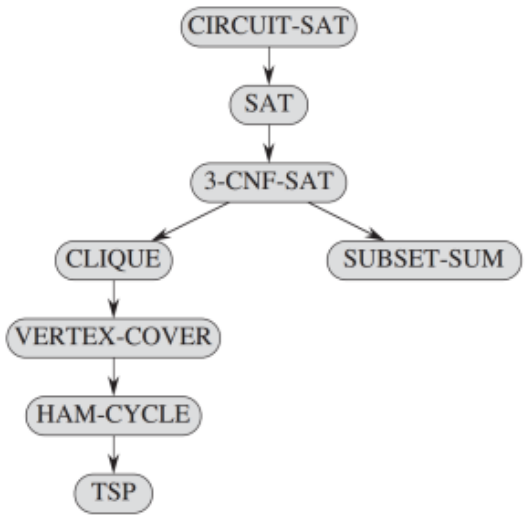
The web of NP-completeness

- For every problem in the picture, if A points to B, it means $A \leq_P B$.
  - So A can be solved using B.
- CIRCUIT-SAT was the original problem that Cook-Levin proved was NP-complete.
- So every problem in NP can be solved using CIRCUIT-SAT.
- But CIRCUIT-SAT can be solved using SAT, because CIRCUIT-SAT $\leq_P$SAT.
  - So every problem in NP can be solved using SAT.
  - So SAT is also NP-complete!
- SAT can be solved using 3-CNF-SAT.
  - So every NP problem can be solved using 3-CNF-SAT.
  - So 3-CNF-SAT is also NP-complete.
- All problems in the diagram are NP-complete.
- Of course, each of the reductions requires a proof, which is sometimes tricky.
  - We'll see some reduction proofs next lecture.
- There are thousands of other NPC problems.



The web of NP-completeness

- Thm Given two NP problems A and B, suppose A is NP-complete, and A $\leq_P$ B. Then B is also NP-complete.
- Proof Let C be any NP problem. Then C $\leq_P$ A, since A is NP-complete.
  - Since $A \leq_P B$, then by Theorem 1, we have $C \leq_P A \leq_P B$.
  - Since also $B \in NP$, then B is NPC.
- To prove a problem B is NP-complete
  - Take a problem A you know is NPC, and prove $A \leq_P B$.
  - E.g., A can be any problem in the previous diagram.
  - To prove $A \leq_P B$, you need to give a polytime reduction from A to B.
    - This can sometimes be quite challenging.
  - You also have to prove $B \in NP$, but that's usually not hard.

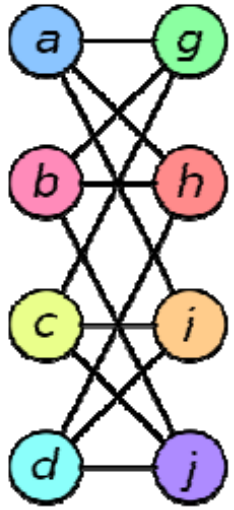


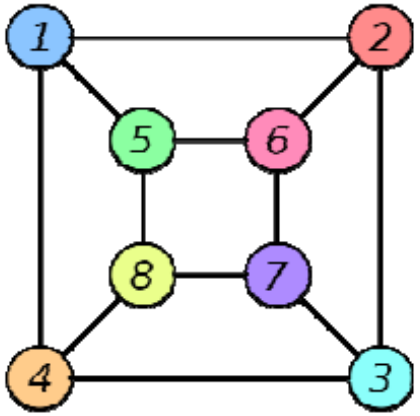
NP-completeness and P vs NP

- Thm 2 Suppose a problem A is NP-complete, and $A \in P$. Then P=NP.
- Proof Consider any other NP problem B. We'll show $B \in P$.
  - Since A is NPC, there's a polytime mapping f from B to A.
  - Given an instance X of B, run f on X to get an instance Y of A.
  - Since $A \in P$, there's a polytime algorithm g to solve A.
  - Run $g(Y)$, and return the same answer for X.
  - By the definition of $\leq_P$, $g(Y)$ is true $\Leftrightarrow$ X is true.
  - Running f and g both take polytime. So we can solve B in polytime.
- Cor Suppose a problem A is NP-complete, and $A \notin P$. Then for any NP-complete problem B, $B \notin P$.
  - If $B \in P$, then since B is NPC, we have P = NP by Theorem 2. So since $A \in NP$, we have $A \in P$, a contradiction.
- To prove $P \neq NP$ (which is what most people think), it's enough to show one NPC problem is not solvable in polytime, by the corollary.
  - But after 50 years, no one has any such proof.
  - Nor has anyone shown a polytime algorithm for any NPC problem.

Beyond NP

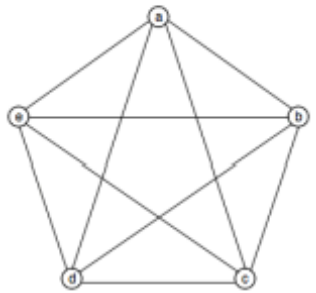
- NP includes many important and practical problems.
- But not all problems are in NP.
- In the graph isomorphism (GRAPH-ISO) problem, we ask whether two graphs simply relabelings of each other.
  - I.e. Given two graphs G = (V,E) and G'=(V, E'), is there a permutation $f: V \to V$ s.t. $(u, v) \in E \Leftrightarrow \big(f(u), f(v)\big) \in E'$.
  - This is in NP, because on "yes" instances, we just give the relabeling to the verifier.
- The graph non-isomorphism (GRAPH-NONISO) problem is the opposite: are two graphs really different, and not relabelings of each other.
  - We don't know if GRAPH-NONISO ∈ NP.
  - If two graphs are really different, how do we produce a certificate to prove this to a polytime verifier?
  - We could give the verifier a list of all possible relabelings, and show the graphs are different under each.
    - But this isn't polytime because there are n! relabelings.
  - We also don't know if GRAPH-NONISO ∈ P.





CLIQUE is NP-complete

- CLIQUE Given a graph with n nodes, is there a clique with n/3 nodes?
  - I.e., are there n/3 nodes s.t. they're all connected to each other?
  - Actually, the CLIQUE problem asks if there's a k-clique, for an arbitrary k. But we consider k=n/3 for simplicity.
- CLIQUE$\in$ NP.
  - The witness is a purported n/3-clique.
  - The verifier just checks there are n/3 nodes, and they're are all connected.



CLIQUE is NP-complete

- Show some NP-complete problem reduces to CLIQUE.
  - The problem you reduce from has to be NP-complete, not just in NP.
  - Note, you're reducing from the NPC problem to your problem, not the other way around.
  - You can choose any NP-complete problem to reduce from.
    - Decide on the right problem can make the task a lot easier. But this takes careful thought.

CLIQUE is NP-complete

- 3-CNF-SAT
  - Given a Boolean formula that's an AND of ORs, where each OR has 3 literals, is it satisfiable?
  - $(A \vee B \vee \neg C) \wedge (A \vee \neg B \vee C) \wedge (\neg A \vee \neg B \vee \neg D) \in 3 - CNF - SAT.$
    - Set A=B=C=true, D=false.
  - Each OR unit is called a clause. Each literal is either a variable or its negation.
  - A special kind of SAT. SAT allows other formula types, besides ANDs of ORs, and allows any number of variables per clause.
- Assume we've already proven 3-CNF-SAT is NP-complete.
- We show 3-CNF-SAT $\leq_P$ CLIQUE.
- The reduction says that given a 3-CNF-SAT formula ϕ, we can create in polytime a graph G, such that ϕ is satisfiable if and only if G has an n/3-clique.
  - This is actually quite remarkable. Why should a graph be related to a formula?
  - But we'll see how to construct a special graph that captures the satisfiability of a 3-CNF formula.

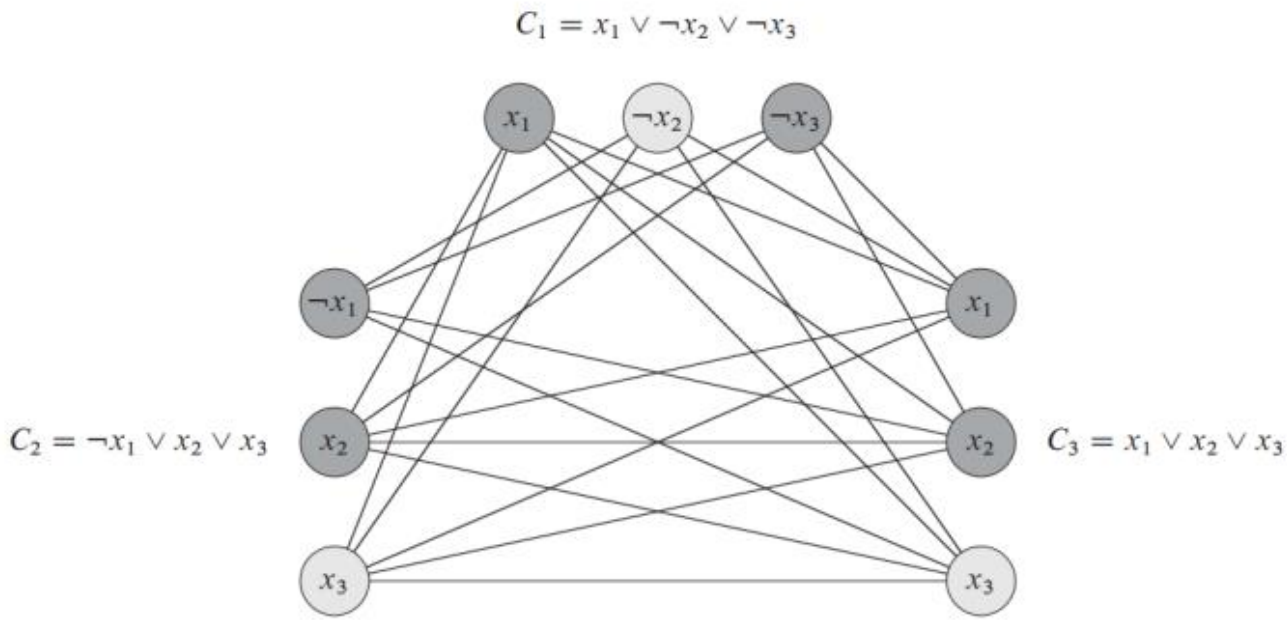
Reducing 3-CNF-SAT to CLIQUE

- Let ϕ be a 3-CNF formula with m clauses.
- Let C be a clause in ϕ. Then C has 3 literals.
  - Make 3 vertices in G corresponding to the literals.
    - So G has 3m vertices total.
    - Let n be the number of nodes in G. Then m = n/3.
- Now, add in an edge between two vertices u, v if both conditions below hold.
  - u, v correspond to literals from different clauses of ϕ.
  - The literals corresponding to u and v are not negations of each other.
    - We say u and v are consistent.

Reducing 3-CNF-SAT to CLIQUE

- 3 vertices for each clause.
- For vertices u,v, add edge (u,v) if u,v are from different clauses, and are consistent (not negations of each other).

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$$



Source: Introduction to Algorithms, Cormen et al

Proving the reduction works

- We first need to show the reduction runs in polytime.
  - Yes. If there are n clauses, the reduction takes $O(n^2)$ time.
- Recall the graph has $n = 3m$ nodes, so $m = n/3$.
- Show $\phi \in$ 3-CNF-SAT $\Leftrightarrow$ G $\in$ m-CLIQUE.
  - ($\Rightarrow$) If ϕ has a satisfying assignment, then G has an m clique.
  - ($\Leftarrow$) If G has an m clique, then ϕ is satisfiable.

$\exists$ sat. assignment $\Rightarrow \exists$ m clique

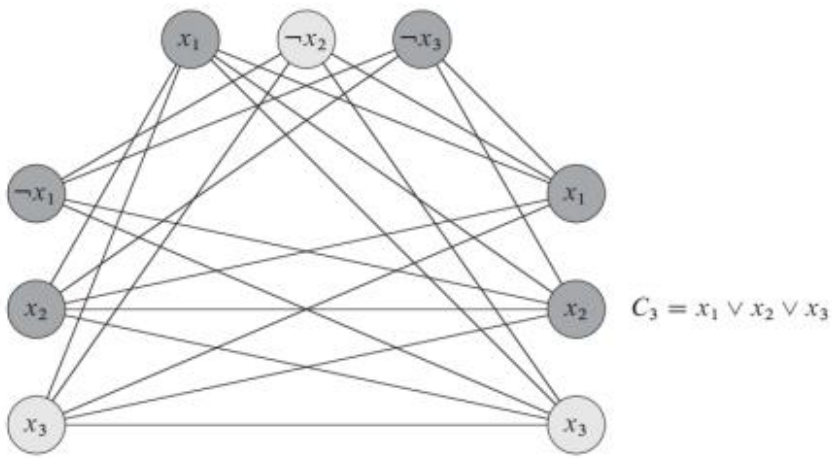
- In the satisfying assignment, every clause has to be true, since we AND them.
- In each clause, at least one literal has to be true, since we OR them.
- So for each clause, pick a true literal.
  - We pick m = n/3 literals.
- The true literal corresponds to a vertex in the graph.
  - Pick m vertices corresponding to the m literals we picked.
- Claim The selected vertices form an m-clique.
- Proof Consider any 2 vertices u, v we selected.
  - u,v come from different triples.
    - Because they come from literals from different clauses.

$\exists$ sat. assignment $\Rightarrow \exists$ m clique

- Proof ctd u,v are consistent. I.e. they don't correspond to a literal in one clause, and its negation in another clause.
  - Because we only picked true literals.
  - So there's an edge (u,v), by construction.
  - So any 2 of the m selected vertices are connected. So the vertices are an m-clique.
- Ex ϕ has a satisfying assignment $x_1 = x_2 = x_3 = T$.
  - The corresponding nodes form a 3-clique.

$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$

$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$



$C_2 = \neg x_1 \vee x_2 \vee x_3$

$C_3 = x_1 \vee x_2 \vee x_3$

$\exists$ m clique $\Rightarrow$ $\exists$ sat. assignment

- Consider the m vertices in the clique.
- None of the vertices come from literals in the same clause in ϕ.
  - For any pair of vertices, they're connected.
  - There are no edges between vertices from the same clause.
- None of the vertices correspond to a literal and its negation in ϕ.
  - We don't add edges between such vertices.
- For the literals corresponding to the clique vertices, set all of them to be true in the formula.
  - This is a valid assignment, since we never set a literal and its negation both to true.
  - We have one true literal per clause.
  - So every clause is true.
  - So the formula is true.

CLIQUE is NP-complete

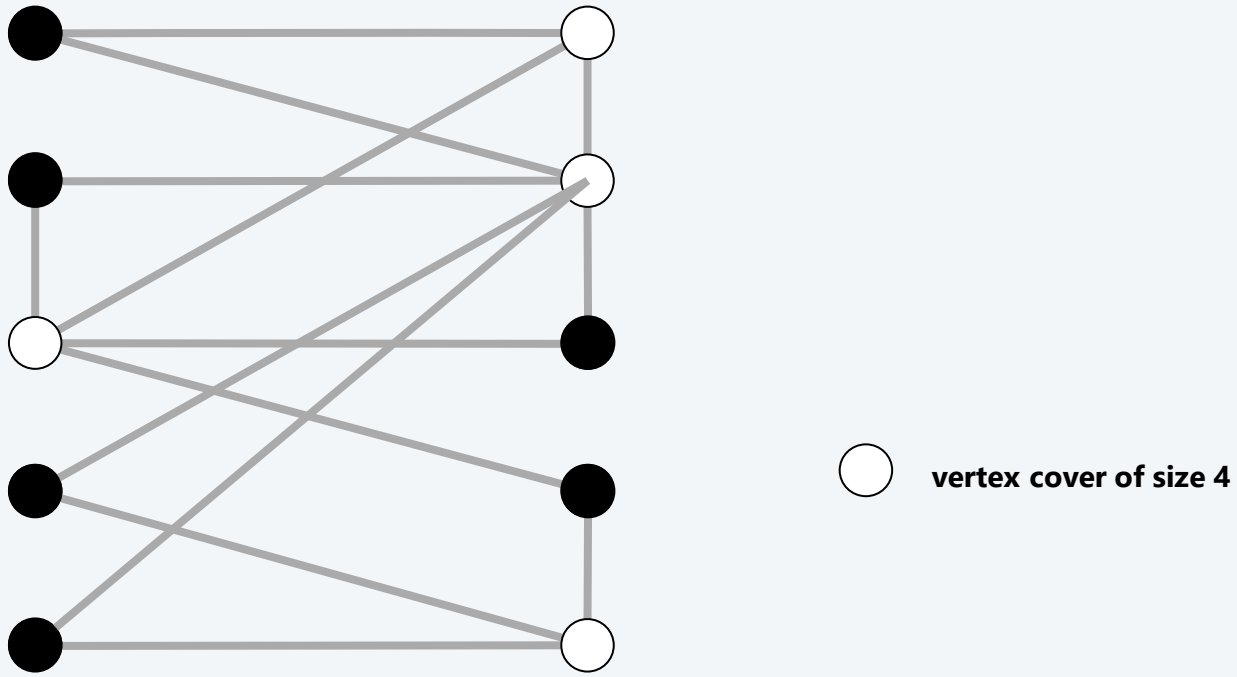
- We've shown CLIQUE $\in$ NP.
- We've shown 3-CNF-SAT $\leq_P$ CLIQUE.
  - We found a polytime reduction, constructing a graph G s.t. for every 3-CNF-SAT formula ϕ
    - If ϕ is satisfiable, G has an n/3-clique.
    - If G has an n/3-clique, then ϕ is satisfiable.
- So CLIQUE is NP-complete.

Vertex cover

VERTEX-COVER. Given a graph $G = (V, E)$ and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?

Ex. Is there a vertex cover of size ≤ 4 ?

Ex. Is there a vertex cover of size ≤ 3 ?



vertex cover of size 4

Set cover

SET-COVER. Given a set U of elements, a collection S of subsets of U, and an integer k, are there $\leq k$ of these subsets whose union is equal to U?

Sample application.

- m available pieces of software.
- Set U of n capabilities that we would like our system to have.
- The i^{th} piece of software provides the set $S_i \subseteq U$ of capabilities.
- Goal: achieve all n capabilities using fewest pieces of software.

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

$k = 2$

a set cover instance

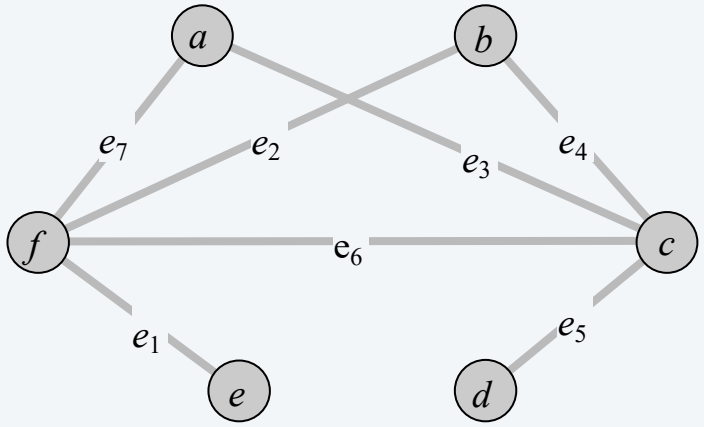
Vertex cover reduces to set cover

Theorem. VERTEX-COVER $\leq_P$ SET-COVER.

Pf. Given a VERTEX-COVER instance $G = (V, E)$ and k, we construct a SET-COVER instance (U, S, k) that has a set cover of size k iff G has a vertex cover of size k.

Construction.

- Universe $U = E$.
- Include one subset for each node $v \in V$: $S_v = \{e \in E : e \text{ incident to } v\}$.



vertex cover instance
(k = 2)

$U = \{1, 2, 3, 4, 5, 6, 7\}$

$S_a = \{3, 7\}$ $S_b = \{2, 4\}$

$S_c = \{3, 4, 5, 6\}$ $S_d = \{5\}$

$S_e = \{1\}$ $S_f = \{1, 2, 6, 7\}$

set cover instance
(k = 2)

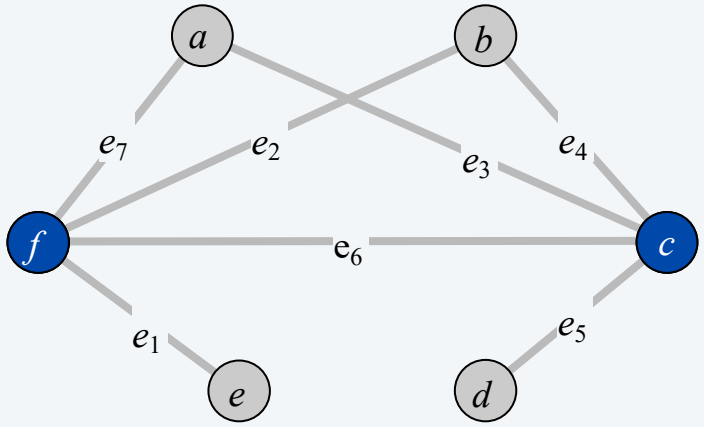
Vertex cover reduces to set cover

Lemma. $G = (V, E)$ contains a vertex cover of size k iff (U, S, k) contains a set cover of size k.

Pf. $\Rightarrow$ Let $X \subseteq V$ be a vertex cover of size k in G.

- Then $Y = \{ S_v : v \in X \}$ is a set cover of size k. ▪

"yes" instances of VERTEX-COVER are solved correctly



vertex cover instance
(k = 2)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

set cover instance
(k = 2)

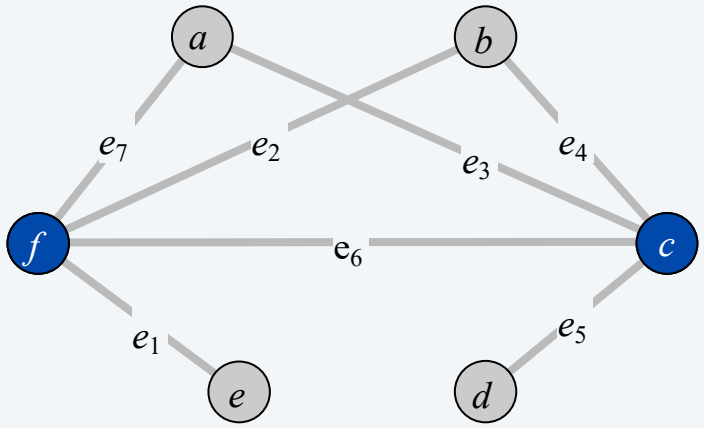
Vertex cover reduces to set cover

Lemma. $G = (V, E)$ contains a vertex cover of size k iff (U, S, k) contains a set cover of size k.

Pf. $\Leftarrow$ Let $Y \subseteq S$ be a set cover of size k in (U, S, k).

- Then $X = \{ v : S_v \in Y \}$ is a vertex cover of size k in G. ▪

"no" instances of VERTEX-COVER are solved correctly



vertex cover instance
(k = 2)

$U = \{ 1, 2, 3, 4, 5, 6, 7 \}$

$S_a = \{ 3, 7 \}$ $S_b = \{ 2, 4 \}$

$S_c = \{ 3, 4, 5, 6 \}$ $S_d = \{ 5 \}$

$S_e = \{ 1 \}$ $S_f = \{ 1, 2, 6, 7 \}$

set cover instance
(k = 2)

历年真题

8. Given a weighted, directed graph with nodes numbered 1,2,……,n, finding the length of a shortest path from vertex 1 to vertex n or determining that no such path exists can be done in polynomial time. （ ）

给定一个带权有向图，图的节点为 1,2,……,n，要计算节点 1 和 n 之间的最短路径或判断这样的路径不存在，这个问题可以在多项式时间内解决。（ ）

9. Given a graph, deciding whether it has a clique of 100 nodes is NP-complete, assuming P≠NP. （ ）

给定一个图，确定图中是否存在一个正好包含 100 个节点的团是一个 NP-complete 的问题，假设 P≠NP。（ ）

10. Given a graph, deciding whether it is possible to remove half the nodes in the graph so that the remaining graph is 3-colorable is solvable in polynomial time, assuming P≠NP. （ ）

给定一个图，确定是否可以去除图中一半的节点后使得该图可以三染色，这可以在多项式时间内完成，假设 P≠NP。（ ）

Problem 4: **(20 points)** For any k, the language k-COLOR is defined to be the set of (undirected) graphs whose vertices can be colored with at most k distinct colors, in such a way that no two adjacent vertices are colored the same color. In class, we learned that 2-COLOR $\in$ P and 3-COLOR is NP-complete.

1. Prove that 4-COLOR is NP-complete.

Part 1. 4-COLOR is in NP. The coloring is the certificate (i.e., a list of nodes and colors).
The following is a verifier V for 4-COLOR.
V="On input $\langle G, c \rangle$,

1. Check that c includes ≤ 4 colors.
2. Color each node of G as specified by c.
3. For each node, check that it has a unique color from each of its neighbors.
4. If all checks pass, *accept*; otherwise, *reject*."

Part 2. 4-COLOR is NP-hard. We give a polynomial-time reduction from 3-COLOR to 4-COLOR. The reduction maps a graph G into a new graph G' such that $G \in$ 3-COLOR if and only if G' 4-COLOR. We do so by setting G' to G, and then adding a new node y and connecting y to each node in G'.

If G is 3-colorable, then G' can be 4-colored exactly as G with y being the only node colored with the additional color. Similarly, if G' is 4-colorable, then we know that node y must be the only node of its color – this is because it is connected to every other node in G'. Thus, we know that G must be 3-colorable.

This reduction takes linear time to add a single node and G edges.

Since 4-COLOR is in NP and NP-hard, we know it is NP-complete.

2. If a problem is in NP, it cannot be solved by a deterministic algorithm in polynomial time.
（ ）
如果一个问题属于 NP，它将不能被一个时间复杂度为多项式的确定性算法解决。
（ ）

10) 所有的 NP 问题都可以约化到 NPC 问题。（ ）

7. Prim's algorithm is one algorithm for finding a minimum spanning tree in a graph and the Floyd-Warshall algorithm can be used to solve the all-pairs shortest-paths problem on a directed graph. Which of the following must be true? （ ）

i. Prim's algorithm is a greedy algorithm.

ii. Prim's algorithm is a dynamic programming algorithm.

iii. Floyd-Warshall algorithm is a greedy algorithm.

iv. Floyd-Warshall algorithm is a dynamic programming algorithm.

Prim 算法是一种计算图的最小生成树算法，Floyd-Warshall 算法可用于计算有向图中所有节点对之间的最短路径。以下哪些表述肯定是正确的？（ ）

i. Prim 算法是一种贪心算法。

ii. Prim 算法是一种动态规划算法。

iii. Floyd-Warshall 算法是一种贪心算法。

iv. Floyd-Warshall 算法是一种动态规划算法。

A. i 和 iii

B. i 和 iv

C. ii 和 iii

D. ii 和 iv

5. Write $X \leq_p Y$ to mean that a problem X is polynomial-time reducible to problem Y. Suppose that a problem X is in P, a problem Y is in NP, and $X \leq_p Y$. Which of the following ***must*** be true? （ ）

A. X is in NP
B. X is NP-complete
C. Y is NP-complete
D. If a problem Z is in P, then $Z \leq_p X$

用 $X \leq_p Y$ 表示问题 X 可以多项式时间规约到问题 Y。假设问题 X 属于 P，问题 Y 属于 NP，同时 $X \leq_p Y$。下列哪些项一定为真？（ ）

A. X 属于 NP
B. X 是 NP-complete
C. Y 是 NP-complete
D. 假定 Z 属于 P，那么 $Z \leq_p X$

Q&A