Lecture

Greedy algorithms cont.
NP Completeness

Steps:

When might we use a greedy strategy

1. we have optimal substructure
2. we can determine recursively solution
3. we can show that if we make a greedy choice, only one problem remains
4. we can prove greedy choice property holds
5. we can devise algorithm

Streamlined steps:

1. Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve.

2. Prove that there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe.

3. Demonstrate optimal substructure by showing that, having made the greedy choice, combining an optimal solution to the remaining subproblem with the greedy choice gives an optimal solution to the original problem.

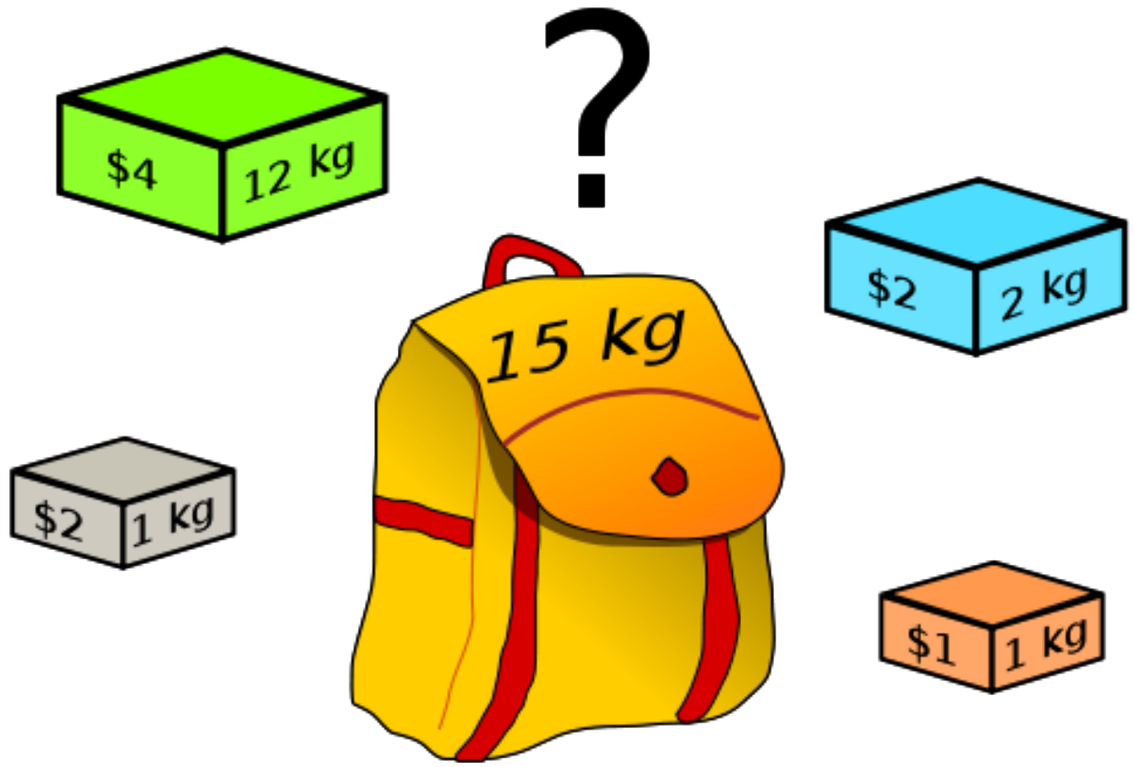
Dynamic Programming

1. optimal substructure
2. overlapping subproblems
3. recursively solve problem - choice depends on *knowing* to solution to all subproblems
4. use memomization or bottom-up

Greedy algorithm requirements

1. optimal substructure
2. greedy choice property - can make the right choice *without knowing* the solution to subproblems

Knapsack Problem



0-1 Knapsack Problem

n items

item i is worth $\$v_i$
weighs w_i

find “best” subset that weighs ≤ W

Fractional Knapsack Problem

same as 0-1 knapsack problem,
but can take a fraction of an object

https://en.wikipedia.org/wiki/File:Knapsack.svg

Rank itms by value/weight: v_i/w_i

$$v_1 / w_1 \geq v_2 / w_2 \geq \cdots \geq v_n / w_n$$

FRACTIONAL-KNAPSACK(v, w, W)

load = 0

i=1

while *load* < *W* and i <= n

  if wi <= *W* - *load*

    take all of item i

  else

    take (*W* - *load)* = w_i of item i

    add what was taken to *load*

    i = i +1

Running Time? $O(n) + O(n \log n)$

If W=50

i	1	2	3
v_i	60	100	120
w_i	10	20	30
v_i/w_i	6	5	4

0-1 knapsack problem if W=50

Greedy solution: take items 1 & 2; value = 160 weight = 30

Optimal solution: take items 2 & 3; value = 220, weight = 50

From https://upload.wikimedia.org/wikipedia/commons/transcoded/9/9e/A_through_Z_in_Morse_code.ogg/A_through_Z_in_Morse_code.ogg.mp3

DATA COMPRESSION

Saving of 20% to 90% are typical

Fixed-length or variable length codewords?

https://en.wikipedia.org/wiki/File:Book_Collage.png

Huge-character data file we wish to store compactly

We have many ways to represent a file of information

We focus on on a ***binary character code***

Each character represented by a ***codeword*** — binary string

Decimal	Hex	Char	Decimal	Hex	Char
64	40	@	96	60	`
65	41	A	97	61	a
66	42	B	98	62	b
67	43	C	99	63	c
68	44	D	100	64	d
69	45	E	101	65	e
70	46	F	102	66	f
71	47	G	103	67	g
72	48	H	104	68	h
73	49	I	105	69	i
74	4A	J	106	6A	j
75	4B	K	107	6B	k
76	4C	L	108	6C	l
77	4D	M	109	6D	m
78	4E	N	110	6E	n
79	4F	O	111	6F	o
80	50	P	112	70	p
81	51	Q	113	71	q
82	52	R	114	72	r
83	53	S	115	73	s
84	54	T	116	74	t
85	55	U	117	75	u
86	56	V	118	76	v
87	57	W	119	77	w
88	58	X	120	78	x
89	59	Y	121	79	y
90	5A	Z	122	7A	z
91	5B	[	123	7B	{
92	5C	\	124	7C	|
93	5D	]	125	7D	}
94	5E	^	126	7E	~
95	5F	_	127	7F	*[DEL]*

From https://simple.wikipedia.org/wiki/ASCII

Computers can store 0 and 1, how can you efficiently store/transmit characters in a computer?

Could we store it more efficiently?

Using UTF-8, this text will take 400 bits to store

01010101 01110011 01101001 01101110 01100111 00100000 01000001 01010011 01000011 01001001
01001001 00101100 00100000 01110100 01101000 01101001 01110011 00100000 01110100 01100101
01111000 01110100 00100000 01110111 01101001 01101100 01101100 00100000 01110100 01100001
01101011 01100101 00100000 00110100 00110000 00110000 00100000 01100010 01101001 01110100
01110011 00100000 01110100 01101111 00100000 01110011 01110100 01101111 01110010 01100101

By looking at the file can we find a way to decrease the number of bits needed to store the file?

- **JPEG**, **MPEG** compress files - they summarize the data and discard minute details (lossy). We want a lossless method

- **zip**, **compress** are tools that compress any sort of file. They look at the file to determine the best way to compress the file in a lossless method.

- If you new your file contained 50 **A**'s, 25 **B**'s and 25 **C**'s could you find a way to store the character so it took less than 800 bits? (Or 700 bits using ASCII)

Storing Data

Storing $AABCAED$ takes 7*7 bits if using ASCII

Space used: $n*l$ where n is the number of characters and l is the length per character

Is it possible to store this more compactly?

$AABCAEDF$

Symbol	A	B	C	D	E	F
Code	000	001	010	011	100	101

concatenate the codewords

Can we do better?

Allow the items to not have the same length - use a prefix code

prefix code: no codeword is prefix of another codeword

Data Compression

input: C and $c.freq$ for each $c \in C$

output: for each $c \in C$, an **optimal codeword**

binary string

Suppose we wanted to store a 100,000 character data file that only contains the characters A, B, C, D, E, F

Symbol	A	B	C	D	E	F
Frequency	45%	13%	12%	6%	9%	5%

Symbol	A	B	C	D	E	F
Code	0	101	100	111	1101	1100

(45·1 + 13·3 + 12·3 + 6·3 +9·4 + 5·4)=194,000 bits

Which is better?

Symbol	A	B	C	D	E	F
Code	000	001	010	011	100	101

300,000 bits

$$\sum_{c \in C} c.freq \cdot d_T(c) = \#\,\text{bits needed to encode file}$$

$d_T(c) =$ length of the code word for c

encoding

Given a code and a message, replace the characters by the codewords. Encode **cab**

Symbol	A	B	C	D	E	F
Code	0	101	100	111	1101	1100

decoding

Given a code, a message is *uniquely decodable* if it can only be decoded in one way. Decode **1000101**

prefix code

A code is a prefix code if no codeword is a prefix of another code

Every message encoded by a prefix code is uniquely decodable

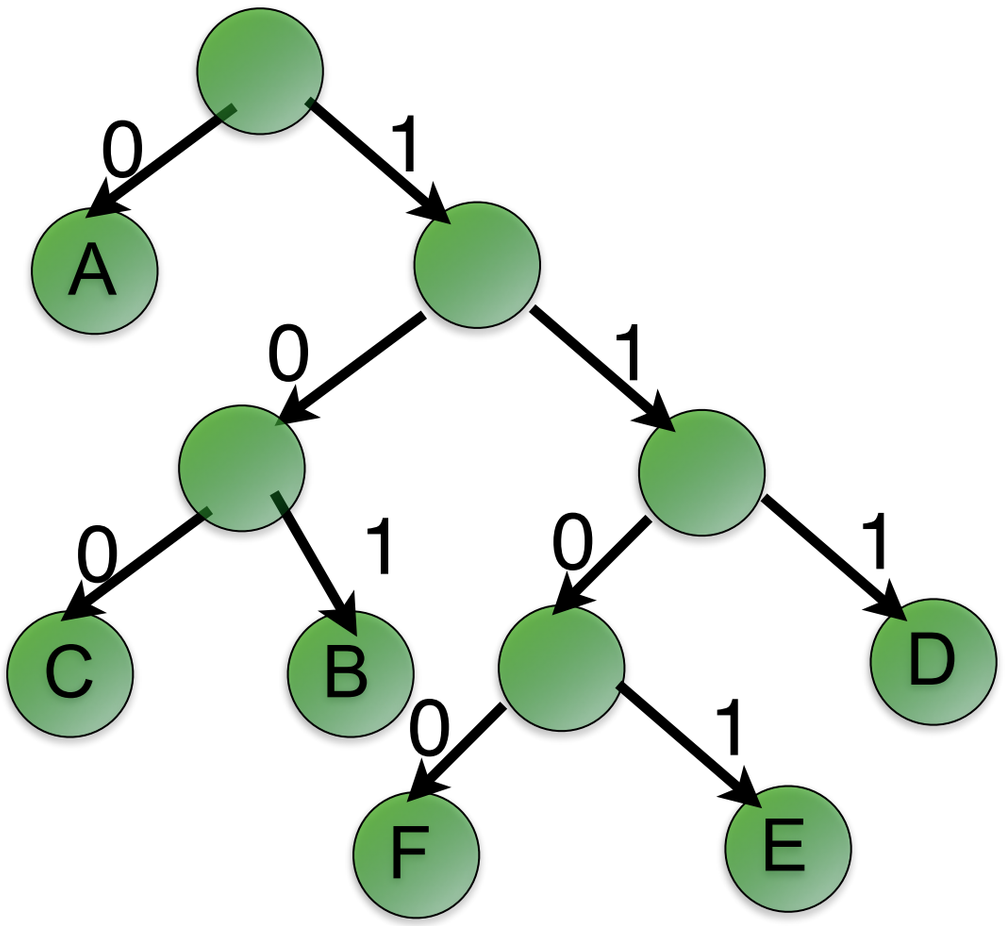
Optimal code is always a full binary tree, every non leaf node has two children

"(equivalently, a tree with minimum weighted path length from the root)."

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

$$d_T(c) = \text{length of the code word for c}$$

$$B(T) = 45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4$$



Symbol	A	B	C	D	E	F
Frequency	45	13	12	16	9	5
Code	0	101	100	111	1101	1100

How can we find the optimal prefix free encoding?

Greedy Algorithm!

Constructing a Huffman Code

Symbol	A	B	C	D	E	F
Frequency	45	13	12	16	9	5

```
HUFFMAN(C)
    n = |C|
    Q = C
    for i=1 to n-1
        allocate a new node z
        z.left = x = EXTRCT-MIN(Q)
        z.right = y = EXTRCT-MIN(Q)
        z.freq = x.freq + y.freq
        INSERT(Q, z)
    return EXTRACT-MIN(Q)
```

Running Time? $O(n) + O(n \log n)$

Correctness Part 1: Greedy Choice Property

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

Lemma

Let C be an alphabet, where $c \in C$ occurs with $c.freq$

Let $x, y \in C$ have the lowest frequencies. WLOG $x.freq \leq y.freq$

There exists a optimal prefix code for C in which the codewords for x and y differ only in their last bit

Proof Cut and Paste

Let T be a tree with the optimal prefix code, that doesn't have the lowest freq items as siblings at the maximum depth

Let a, b be these nodes WLOG $a.freq \leq b.freq$

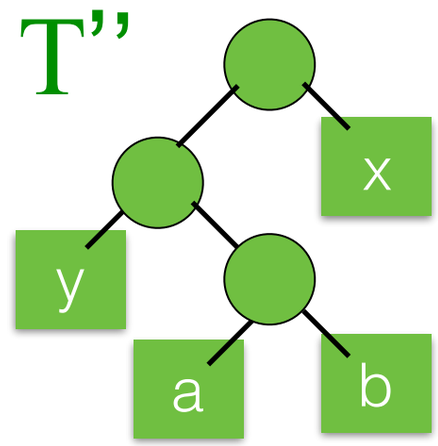
By assumption $x.freq \leq a.freq$ and $y.freq \leq b.freq$

By assumption $x.freq \neq b.freq$ Otherwise $x.freq \leq a.freq \leq b.freq = x.freq$

Create T' by exchanging x and a.

Create T'' by exchanging y and b in T'

Now compute B(T'), B(T'')



$$B(T) - B(T') = \sum_{c \in C} c.freq \cdot d_T(c) - \sum_{c \in C} c.freq \cdot d_{T'}(c)$$

$$= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a)$$

$$= x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x)$$

$$= (a.freq - x.freq) \cdot (d_T(a) - d_T(x)) \geq 0$$ similarly for T' and T'' QED

Correctness Part 2: Optimal Substructure

Lemma

$$B(T) = \sum_{c \in C} c.freq \cdot d_T(c)$$

$$d_T(c) = \text{length of the code}$$

Let x, y in C have minimum frequency

Create C' = C - {x,y} ∪ {z}

Define, the new character, z to have z.freq = x.freq + y.freq

Let T' be any tree representing an optimal prefix code for C'

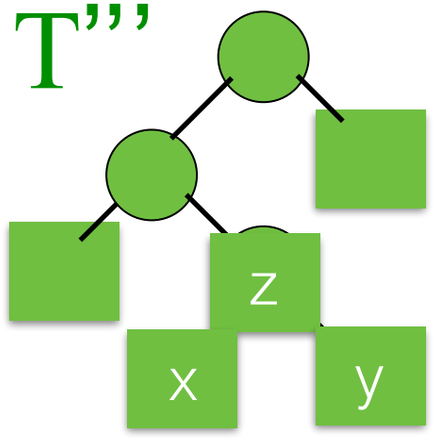
Create T from T' by replacing z with a new internal nodes who's left child is x and whose right child is y. ***T represents an optimal prefix code for C***

Proof

By construction B(T) = B(T') + x.freq + y.freq

Suppose not, suppose T is not optimal.

Let T'' be an optimal tree. B(T'') < B(T)



By previous lemma, we can create T'' such that x and y are siblings.

Create T''' from the tree T'' where the common parent of x and y has been replaced by a leaf z, z.freq = x.freq + y.freq.

B(T''') = B(T'') -x.freq - y.freq < B(T') ⋈

NP-Complete Problems

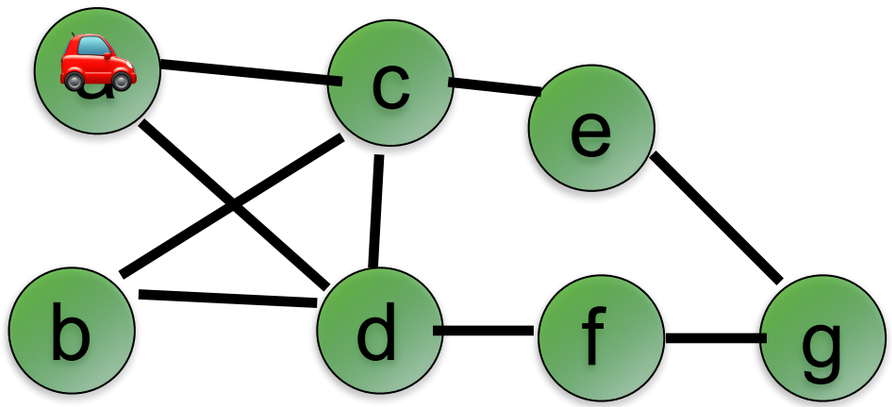
Or how to earn $1,000,000

Millennium Prize

We found polynomial time solutions to many problems where there were exponential number of choices to make!

Can we always do this????

Drive on each road exactly one time



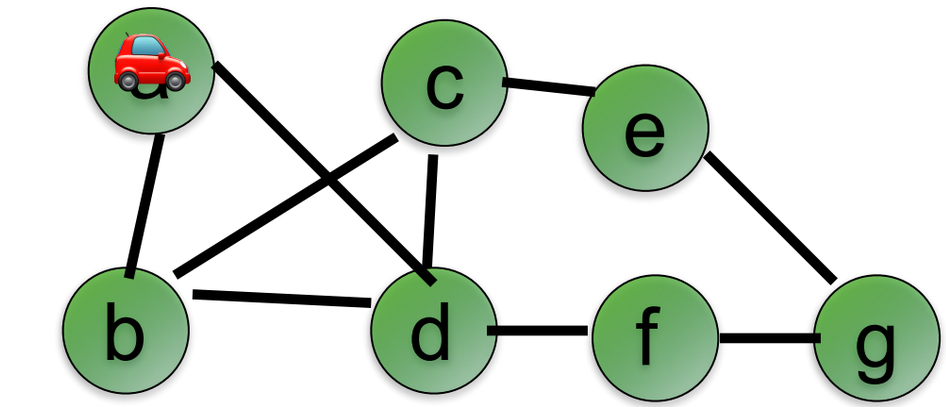
Inspect a set of roads.
Save cost by driving each road exactly one time
End up back at the start.

In Königsberg in Prussia “The problem was to devise a walk through the city that would cross each of those bridges once and only once.”

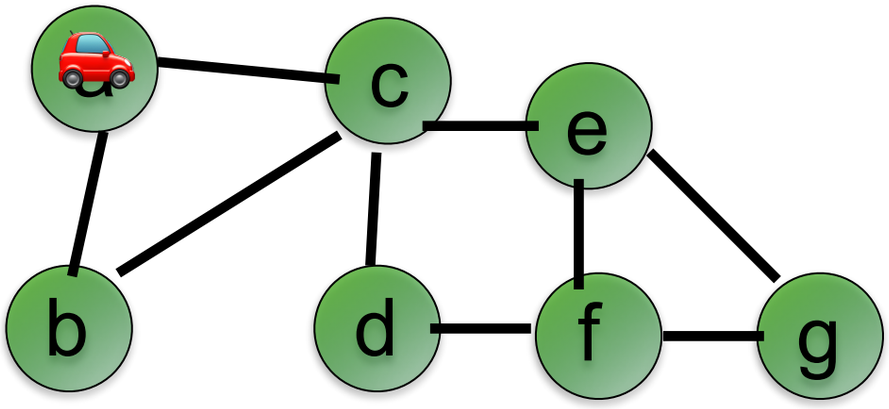
Eulerian Circuit

https://commons.wikimedia.org/wiki/File%3AKonigsberg_bridges.png

Drive to visit each city exactly one time (except first=last city)



You are asked to visit a set of cities. It is very expensive to visit a city, so you want to visit each city exactly one time. Design a tour to do so.



Hamiltonian Circuit

unsolvable

decidable

...

inefficient algorithms n^{n^n}

n^n

intractable

$n!$

2^n

? NP complete

...

efficient algorithms n^3

tractable

n^2

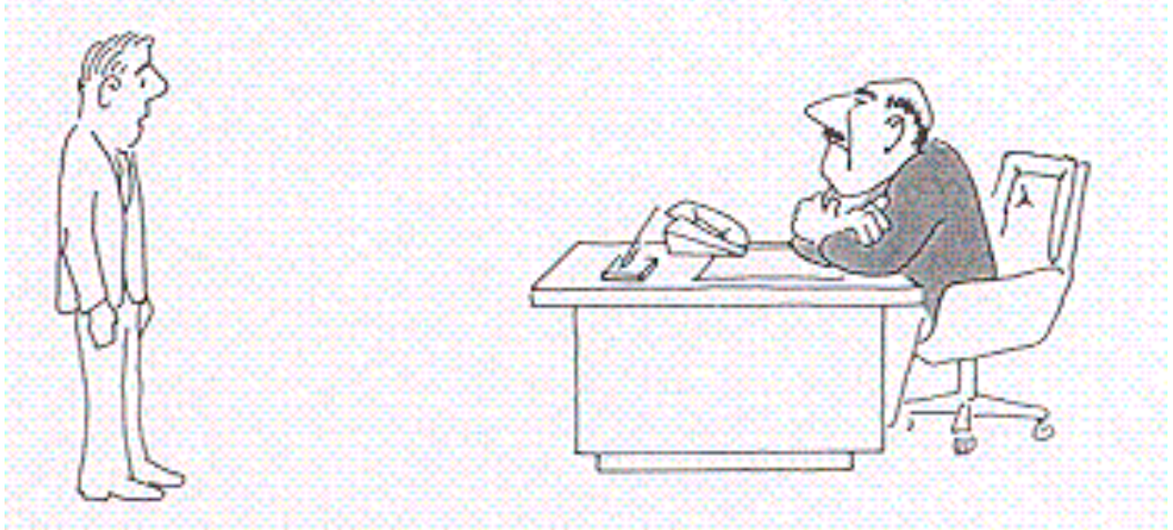
$n \log(n)$

n

$\sqrt{(n)}$

$\log(n)$

1



“I can’t find an efficient algorithm. I guess I’m too dumb.”

“I can’t find an efficient algorithm because no such algorithm is possible!”

“I can’t find an efficient algorithm, but neither can all these famous people.”

Cartoon from “Computers and Intractability: A Guide to the Theory of NP-completeness” by M. Garey and D. Johnson

People have observed that many of the problems* we don't know how to solve efficiently are really similar to each other!

*the problems that are the same, are problems where if you give me a solution I can check your answer efficiently

decision problems vs optimization problems:

decision problems: answer is **1** or **0** (i.e."**yes**" or "**no**")

Does the graph have a cycle?

Is n a prime number?

Are two graphs isomorphic?

Does the graph have a path where each vertex is visited exactly once?

Usually can turn an optimization problem into a decision problem by imposing a bound on the value to be optimized

SHORTEST-PATH -> PATH

given a graph, G, does G have a path from u to v of length less than k

Decision Problems ⬌ Optimization Problems

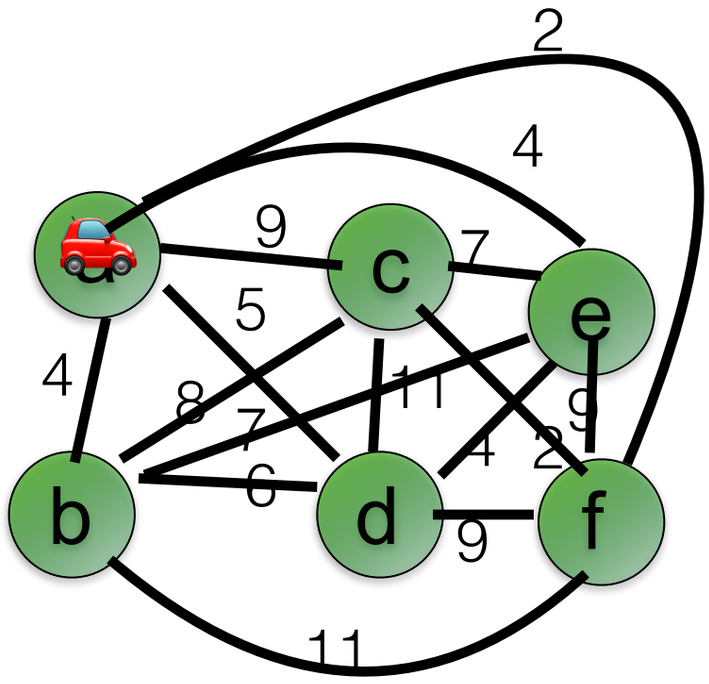
⬅If solve an optimization problem (in polynomial time) then the decision problem can be solved (in polynomial time)

➡If it is possible to solve a decision problem (in polynomial time) then using binary search, can solve an optimization problem (in polynomial time)

If we can provide evidence that a decision problem is hard we have also provided evidence that an optimization problem is hard

Decision problems are easier to state

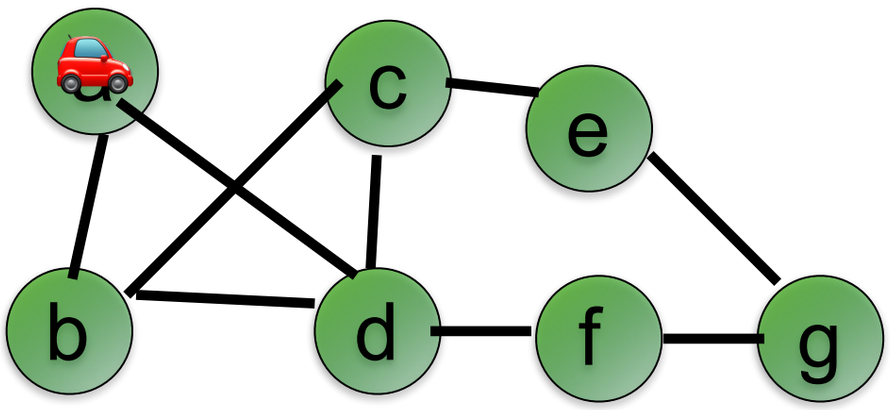
Drive to visit each city exactly one time (except first=last city) where your cost is at most C



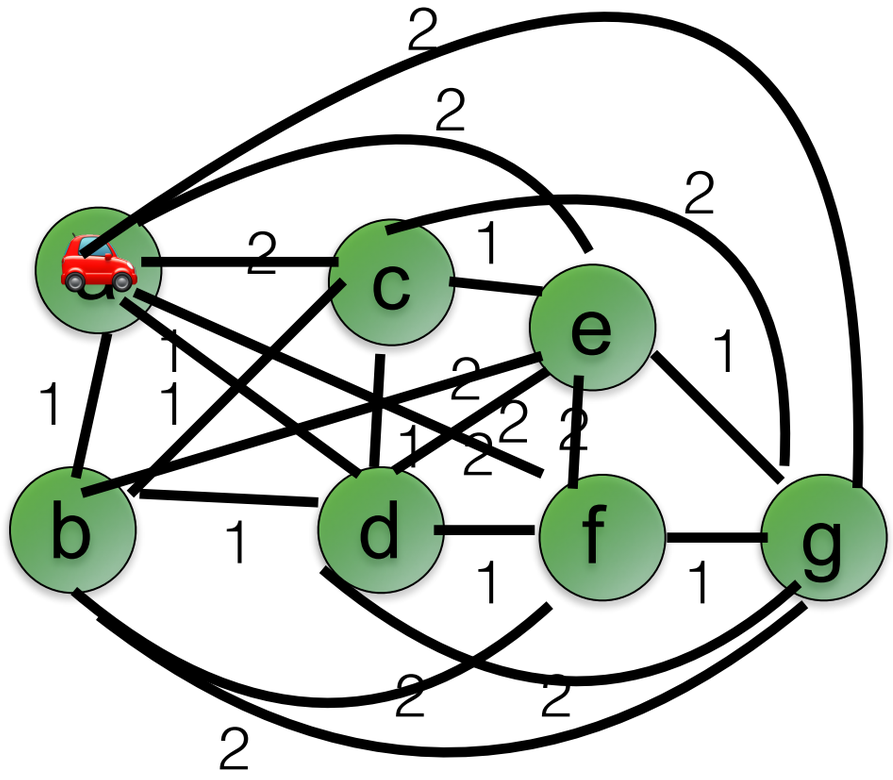
You are asked to visit a set of cities. It is very expensive to drive. Can you , so you want to visit each city exactly one time. Design a tour to do so.

Traveling Salesman Decision Problem

complete graph
cycle at most cost c



Transformation!

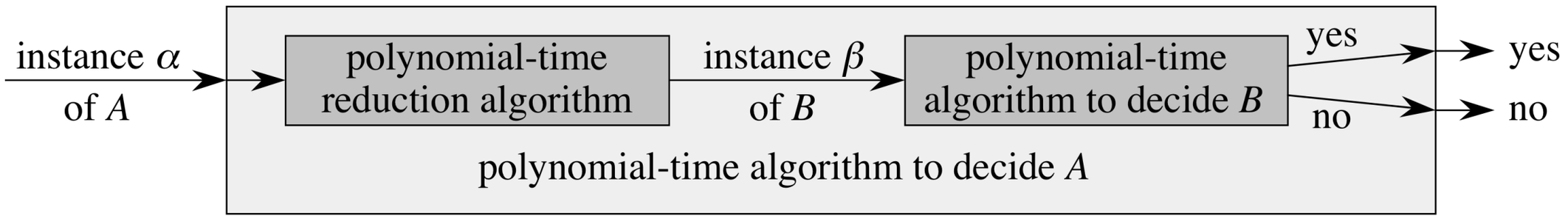


Given G=(V,E)
- assign weight 1 to each edge
- augment the graph so it is complete
- assign weight 2 to the new edges
- Find TSP where C=|V|

If we can solve TSP decision problem we can solve Hamiltonian Circuit

Reductions between problems, $\leq_p$

Examples: 2: A problem A is **Polynomial time reducible** to a problem B if we can solve problem A in polynomial time given an O(1) algorithm for problem B.

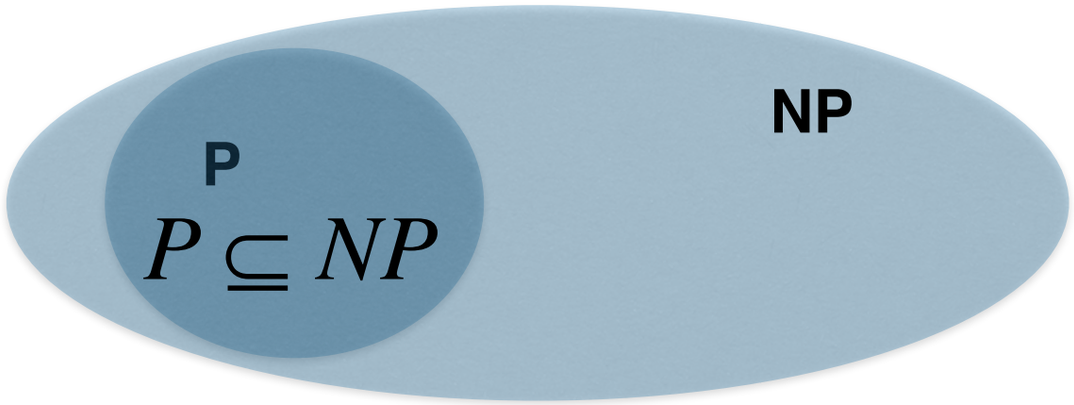


- We showed a polynomial time reduction from Hamiltonian cycle problem to the TSP decision problem
- The cost of the reduction was $O(V^2)$
- Thus TSP decision is at least as hard as Hamiltonian circuit.

picture from CLRS

What about the other decision problems we don’t know how to solve?

Classifying Problems: NP-hard, NP-easy, NP-complete



Def 1: P is the class of all decision problems that can be solved in **polynomial time. i.e.** there exists a constant c such that the running time of the algorithm is $O(n^c)$ where n is the size of the input

Shortest path of cost less than B, Euler circuit, minimum spanning tree of cost less than B

Def 2: An algorithm, A(X, Y), is a **verifier** if given X, a (reasonable) encoding of a problem, and Y a "certificate" for a solution (i.e. enough information to check if a solution is correct (typically the actual solution is given)

Def 3: An algorithm, A(X, Y), is a **polynomial time verifier** if it is a problem verifier and it runs in polynomial time.

Def 4: NP (nondeterministic polynomial time) is the class of all decision problems that can be verified in polynomial time (i.e. there exists a short "certificate" for "1-instances")

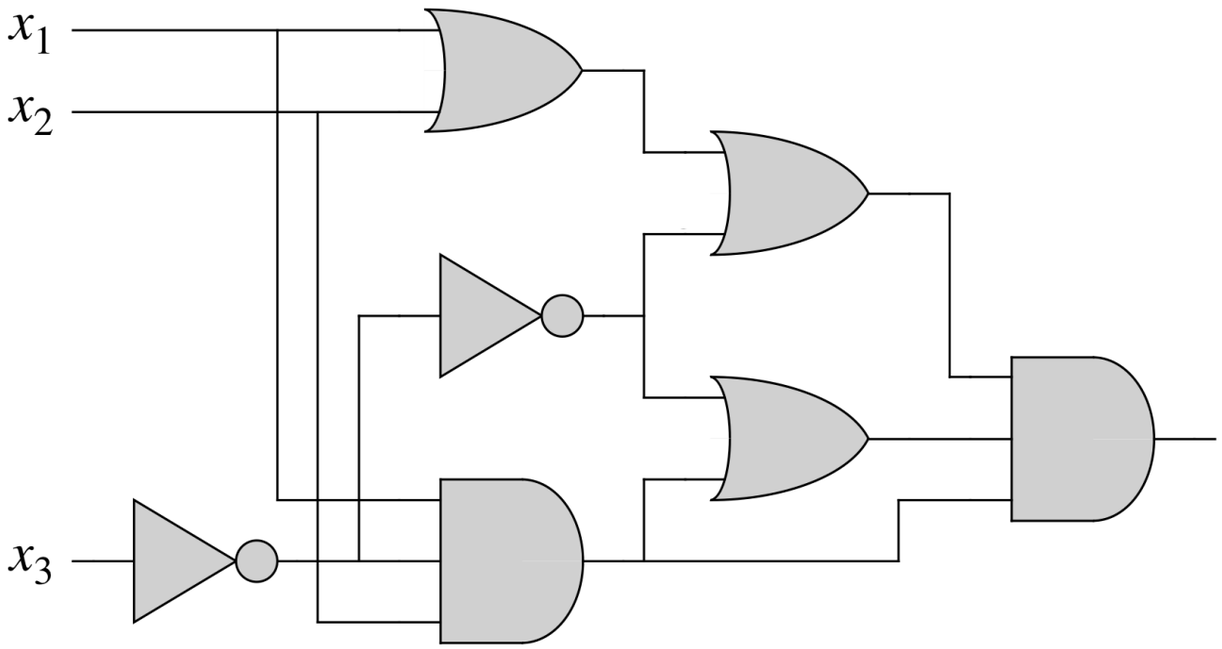
Shortest path of cost less than B, Euler tour, minimum spanning tree of cost less than B, Hamiltonian circuit, Traveling Salesman Decision Problem

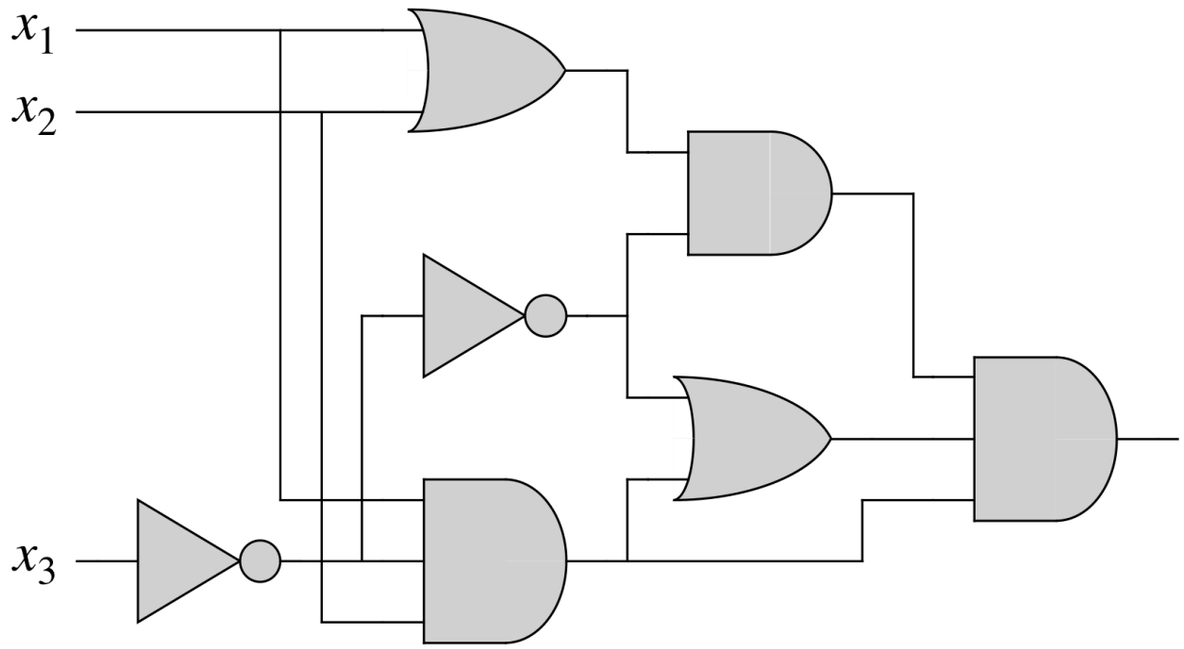
Millennium Prize

https://en.wikipedia.org/wiki/Millennium_Prize_Problems

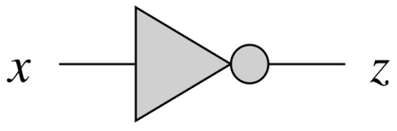
Circuit Satisfiability

Every NP problem can be reduced to circuit satisfiability



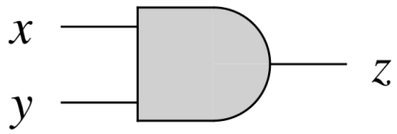


not



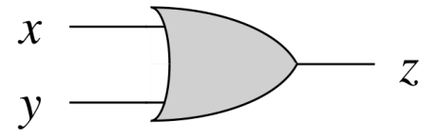
x	$\neg x$
0	1
1	0

and



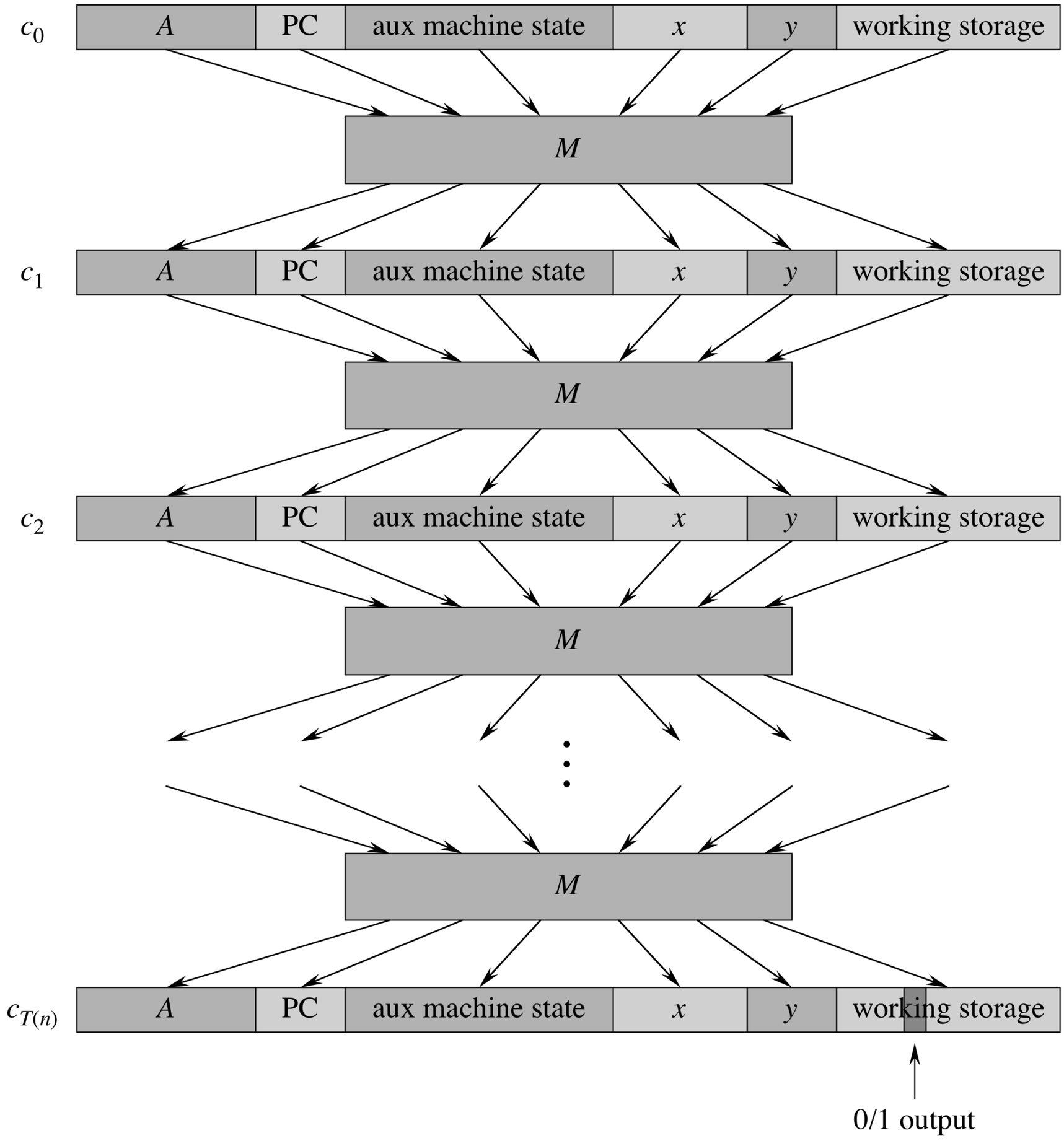
x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

or



x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

from CLRS


c_0
A
PC
aux machine state
x
y
working storage
M
c_1
A
PC
aux machine state
x
y
working storage
M
c_2
A
PC
aux machine state
x
y
working storage
M
M
$c_{T(n)}$
A
PC
aux machine state
x
y
working storage
0/1 output

from CLRS

Circuit Satisfiability

- Every problem in NP can be transformed to circuit satisfiability
- Thus circuit satisfiability is at least as hard as all the other problems in NP (i.e. it is an NP hard problem)
- **NP hard** is defined to be the problems which are at least as hard as all the problems in NP. We just gave the intuition that circuit satisfiability is NP hard.
- Circuit satisfiability is in NP. Proof?
- **NP complete** are the problems that are in NP and are NP hard
- *Circuit satisfiability* is NP-Complete

What to do with a difficult problem…

- Your problem could be NP complete. See if you can reduce another NP complete problem into your problem in polynomial time

- If it is NP complete -

  - determine if an approximate solution will suffice

  - determine if the instances you care about are easy to solve even if finding the solution for the general case might be very very hard to solve (i.e you can only solve it if P = NP)

How to show a problem in NP-complete

- Show your problem is in NP

- Find an NP-complete problem which you can reduce to your problem

$$(x \vee \overline{y} \vee z)(\overline{x} \vee y \vee \overline{z})(\overline{x} \vee \overline{y} \vee \overline{z})$$

$$x = true,\ y = true,\ z = false$$

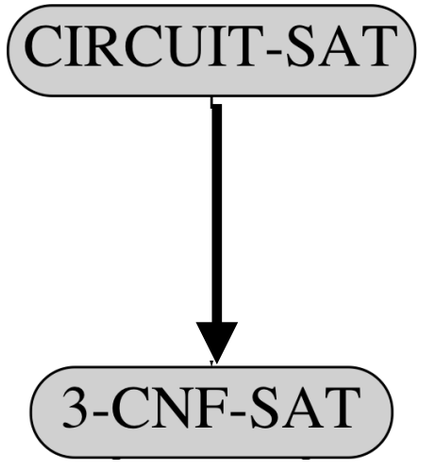
3-CNF

Conjunctive normal form (CNF) is a collection of clauses
Each clause is the disjunction (or, denoted by V) of three literals
Each variable is assigned true or false
A satisfying truth assignment is an assignment such that each clause contains a literal that is true

$$(x \vee \overline{y} \vee \overline{z})(x \vee y \vee z)(\overline{x} \vee \overline{y} \vee z)(\overline{x} \vee \overline{y} \vee \overline{z})(\overline{x} \vee y \vee \overline{z})(\overline{x} \vee y \vee z)(x \vee \overline{y} \vee z)(x \vee y \vee \overline{z})$$

We could check ever assignment of the variables…
only 2^n choices

Can we show 3-CNF is NP Complete?



CIRUIT-SAT ➡ 3 CNF SAT

Create a 3-CNF variable x_i for each circuit input i

Make sure every AND, OR gate has only 2 inputs

If a gate has k > 2 inputs, replace it by k-1 two-input gates

For each gate assign a variable to be the result and create a CNF formula

$a = b \wedge c$ goes to $(a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$

$a = b \vee c$ goes to $(\overline{a} \vee b \vee c) \wedge (a \vee \overline{b}) \wedge (a \vee \overline{c})$

$a = \overline{b}$ goes to $(a \vee b) \wedge (\overline{a} \vee \overline{b})$

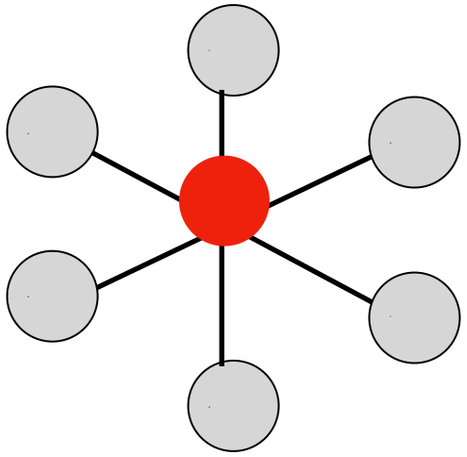
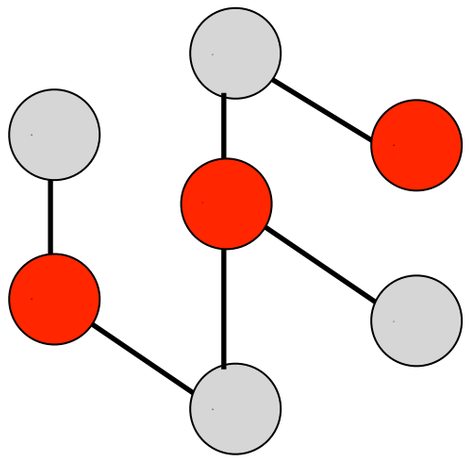
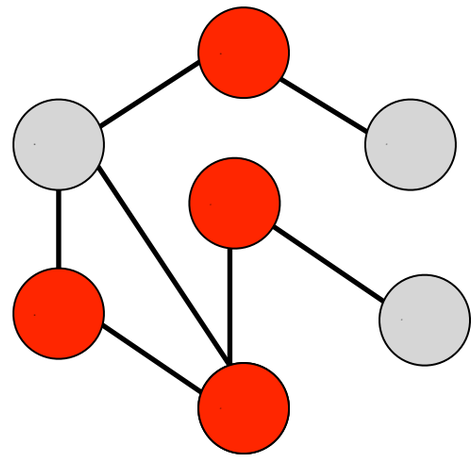
If a =1 then b=1 and c=1
If a=0 then b=0 or c=0

If a =1 then b=1 or c=1
If a=0 then b=0 and c=0

Make every clause have 3 literals

a goes to $(a \vee x \vee y) \wedge (a \vee \overline{x} \vee y) \wedge (a \vee x \vee \overline{y}) \wedge (a \vee \overline{x} \vee \overline{y})$

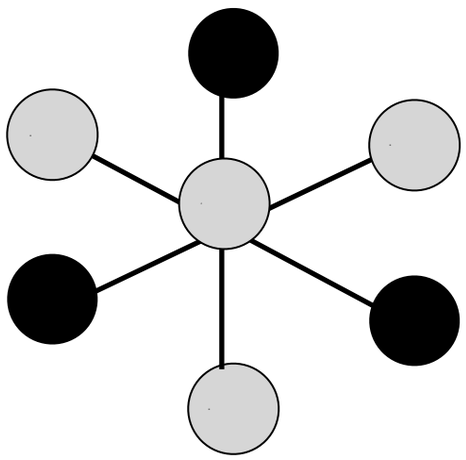
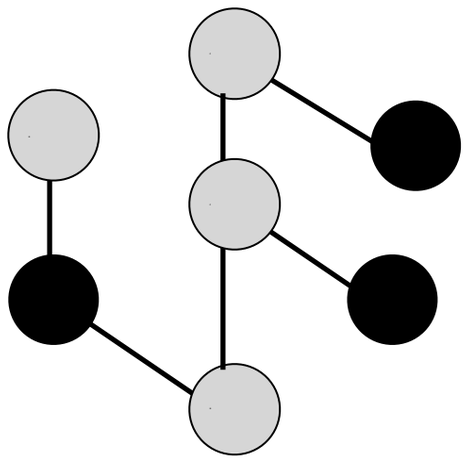
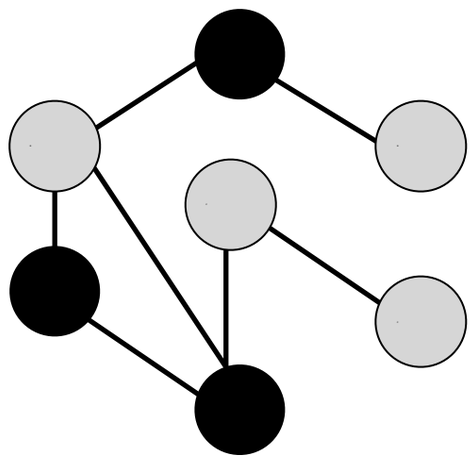
$a \vee b$ goes to $(a \vee b \vee x) \wedge (a \vee b \vee \overline{x})$

Vertex Cover

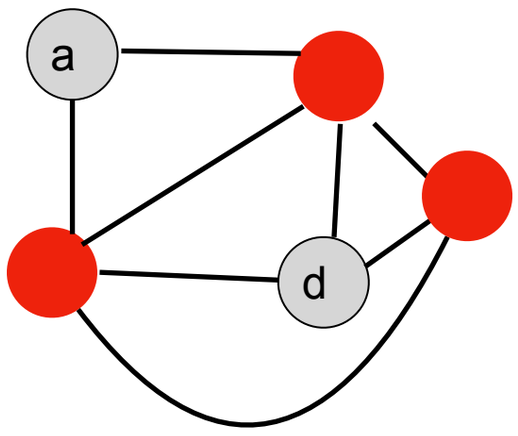
Given a graph G=(V,E), a vertex cover is a set S⊆V such that for every (u,v) ∈ E then either u or v ∈ S (or both)

Vertex cover problem is to find a set S that covers the vertices in G where |S| ≤ k

We could check ever subset of V… only 2^V choices

Next we prove
Vertex Cover S is NP Complete

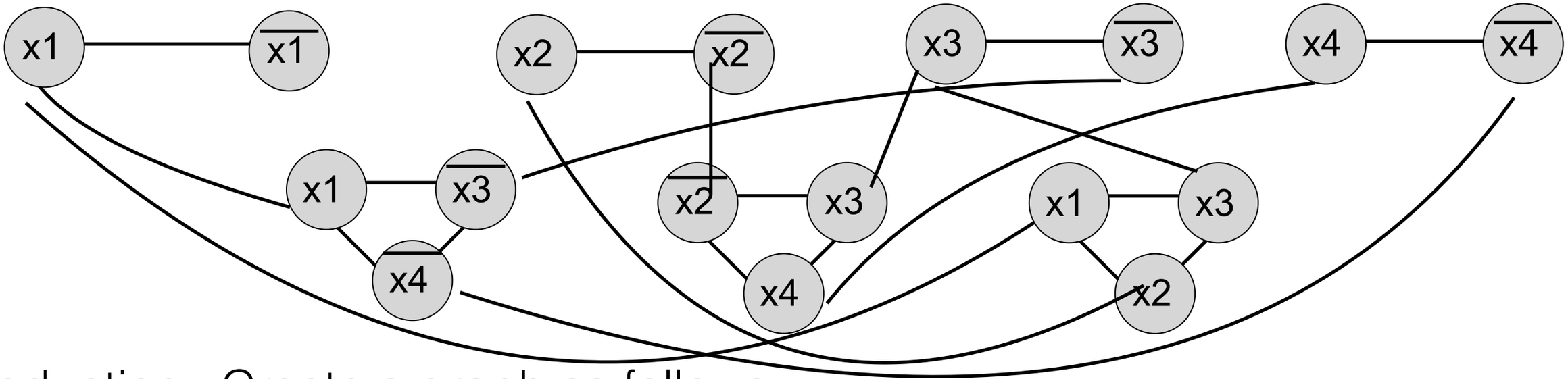


g = 2 X

g = 3 ✔

3 CNF SAT ➡ Vertex Cover

$$(x_1 \vee x_3 \vee x_4) \wedge (x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee x_2 \vee x_3)$$



Reduction: Create a graph as follows:

- construct two graph gadgets:
  1. a “variable gadget”: for every variable x, create two nodes and label one node x and the other x. Put an edge between the nodes
  2. a “clause gadget”: for every clause create three nodes, each labeled with a literal from the clause. Connect the nodes
- connect the “gadgets”: connect every literal in the clause gadget to its corresponding literal in the vertex gadget

There is a vertex cover of size m + 2v (m clauses and v variables) iff there is a satisfying assignment. To show it works we need to prove that:

- If F has a satisfying assignment then G has an vertex cover of size 2m + v
- If G has an vertex cover of size k=m+2v then F has a satisfying assignment.

If we can solve Vertex Cover then we can solve 3 CNF Satisfiability.

If we can solve 3 CNF Satisfiability we can solve Circuit Satisfiability.

If we can solve Circuit Satisfiability we can solve any problem in NP.

If we can solve Vertex Cover we can solve any problem in NP!

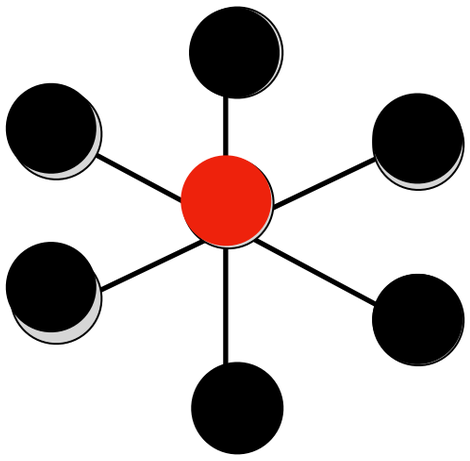
There are hundreds of problems known to be NP-Complete

Assuming P ≠ NP

The best we can do is to solve a special subclass of the problem or find a good approximation

The next few slides were not presented in lecture and will NOT be tested on the final

Vertex Cover Approximation



Example where OPT(I) =1 and solution takes n-1 vertices

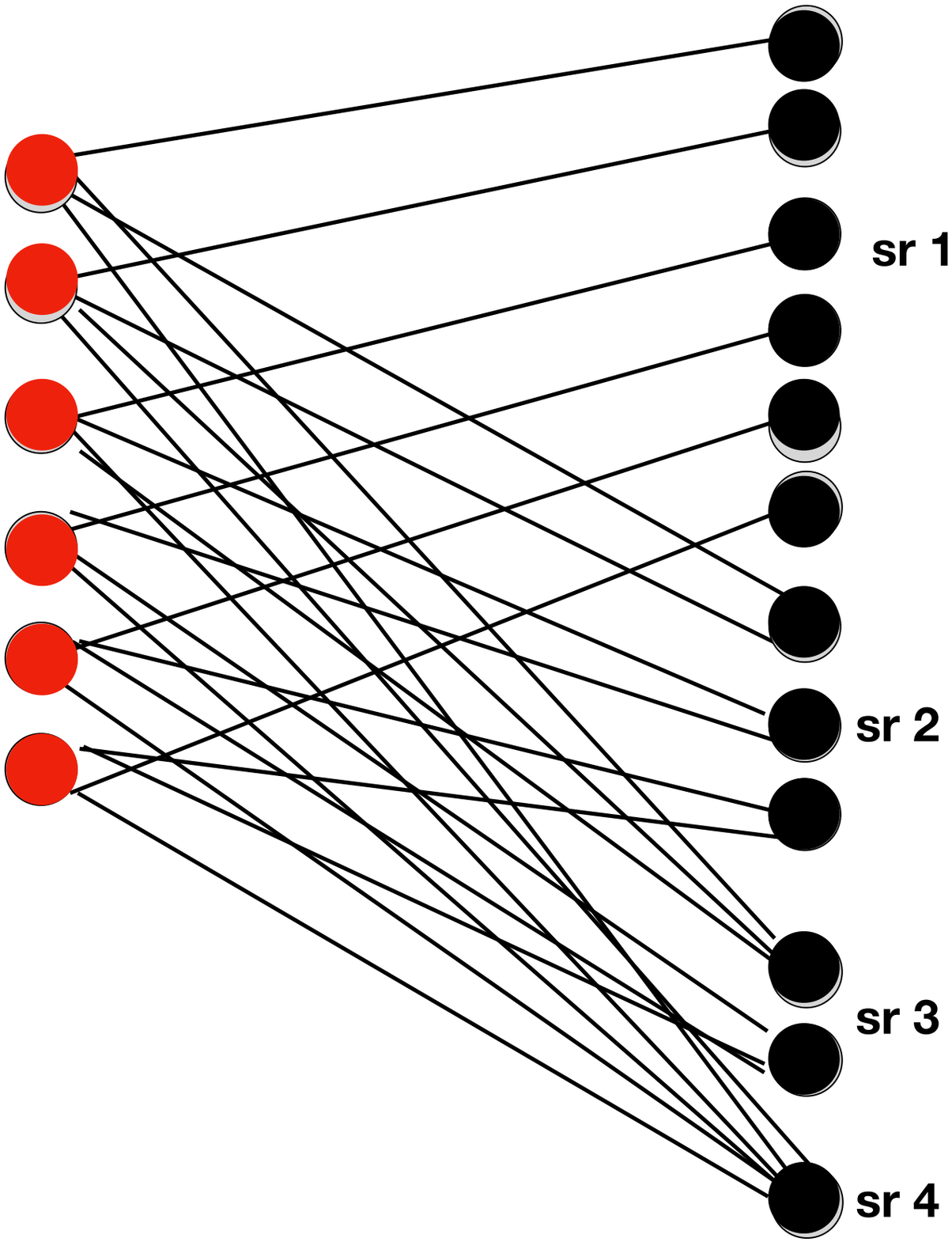
First idea:

C = ∅

Repeat till no uncovered edge:

Choose an vertex, u, incident to an uncovered edge {u,v}, put u into C

Vertex Cover Approximation



Second idea:

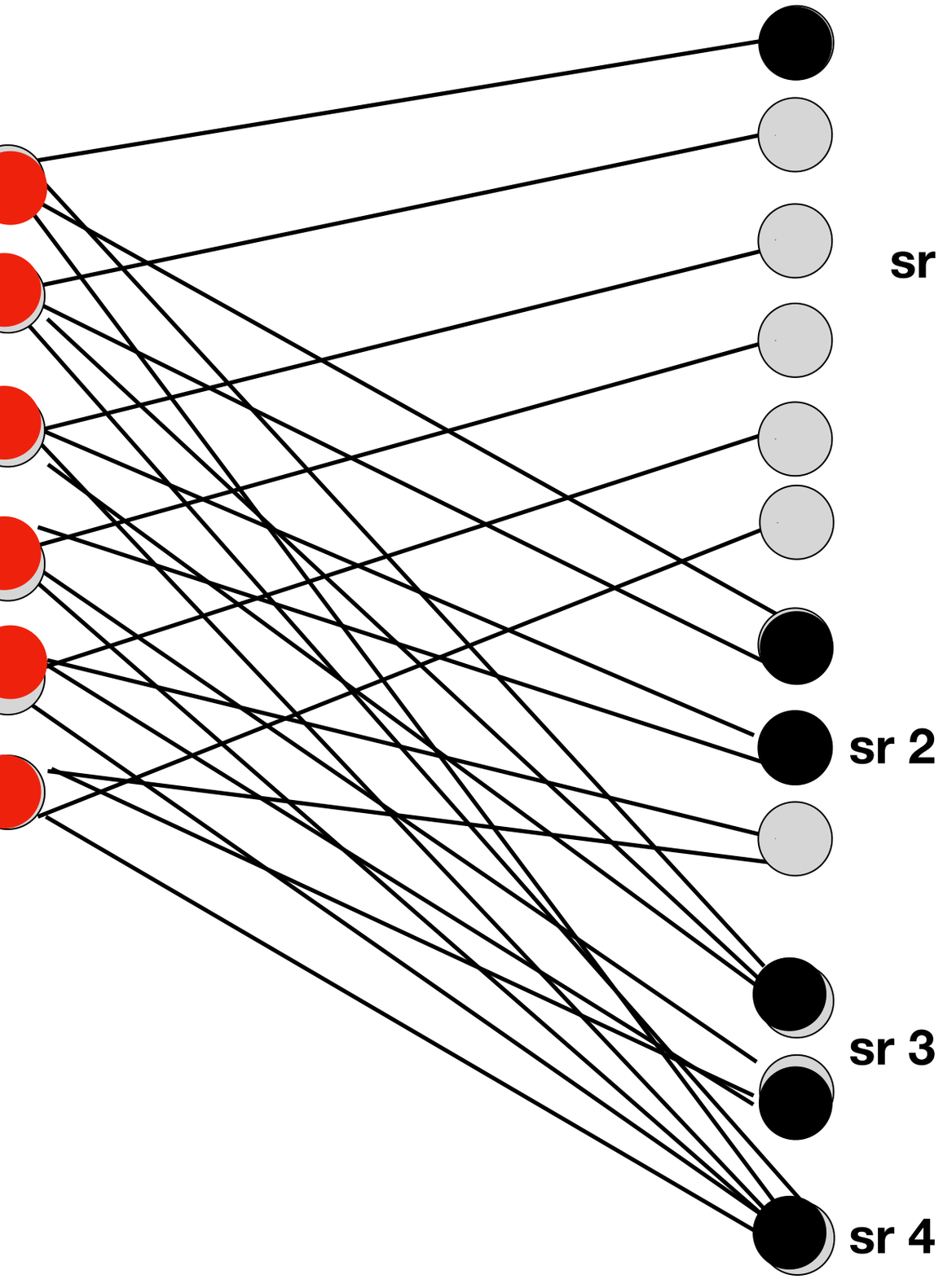
C = ∅

Repeat till no uncovered edge:
Choose an vertex, u, covering most uncovered edges

Example where OPT(I) =t and solution takes n-t vertices

It is possible to prove that this approach can take at least a log(n) factor more than it should

Vertex Cover Approximation



Third idea:

C = ∅

repeat till no uncovered edge:
choose an uncovered edge,{u,v}
add u and v to C

2-approximation