Integer Multiplication

Big Integers: Stored as array of digits, not bits

↳ useful in cryptography

Addition: input:= a[1-n], b[1-n] array of digits

output - C[1-(n+1)] where c = a+b

ex)
$$\begin{array}{rrcccccc} & & \multicolumn{6}{c}{\xleftrightarrow{\hspace{6em}} (n \text{ digits})} \\ & a = & 1 & 2 & 3 & 4 & 1' & 6 \\ + & b = & 2 & 1 & 3 & 4 & 5 & 6 \\ \hline & c = & 3 & 3 & 6 & 8 & 7 & 2 \end{array}$$

Simple Arithmetic: O(1) per digit → O(n) time

Multiplication: input:= a[1-n], b[1-n] array of digits

output - c[1-2n] where c = a×b

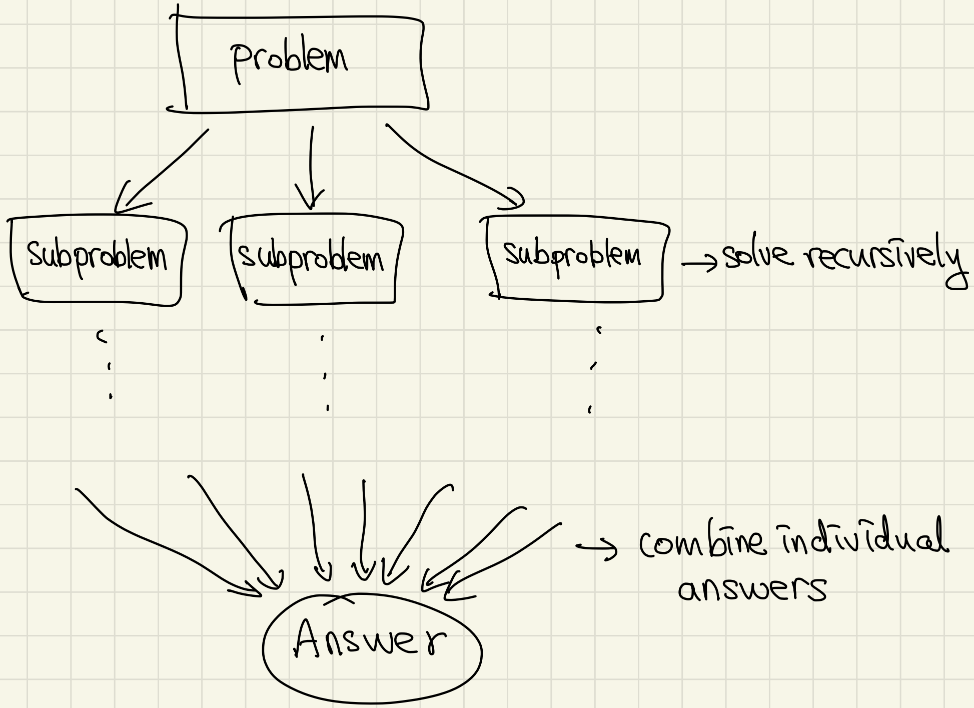
ex) a = 1231, b = 2121

$$\begin{array}{l|ccccccc} n \text{ times} & & & & \multicolumn{4}{c}{\xleftrightarrow{\hspace{5em}} (n \text{ digits})} \\ & & & & 1 & 2 & 3 & 1 \\ & & & 2 & 4 & 6 & 2 & - \\ & & 1 & 2 & 3 & 1 & — & — \\ \downarrow & 2 & 4 & 6 & 2 & — & — & — \\ \hline c= & 2 & 6 & 1 & 0 & 9 & 5 & 1 \end{array}$$

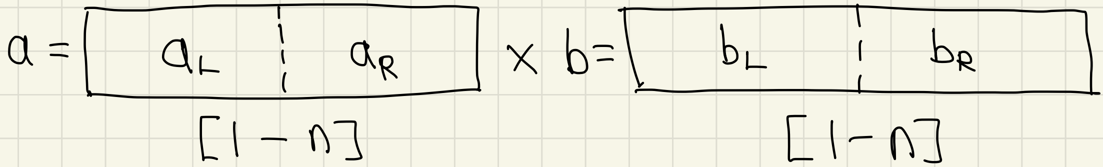
Runtime: adding n digits n times at least → $\geq O(n^2)$

↳ Can we do better?

Divide & Conquer Paradigm: split, solve, combine



→ how to apply this to multiplication?



ex) $a = (123)456 \quad b = (654)321$

$$= 123 \times 10^3 + 456 \qquad = 654 \times 10^3 + 321$$

generally, $X = X_L \cdot 10^{n/2} + X_R$.

$$\rightarrow a \times b = (a_L 10^{n/2} + a_R)(b_L 10^{n/2} + b_R)$$

$$= a_L b_L 10^n + (a_R b_L + a_L b_R) 10^{n/2} + a_R b_R$$

we need to calculate 4 products: $a_L b_L, a_L b_R, a_R b_L, a_R b_R$

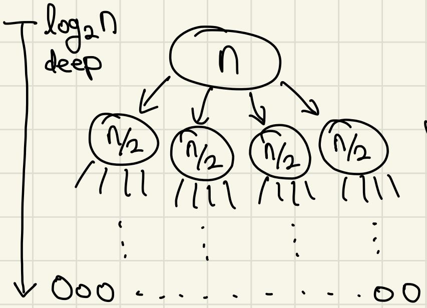
each number is $n/2$ digits → recursive definition!

```
MULT(a[1-n], b[1-n]):
  - If $n < 2$, return $a \times b$.
  - Split $a \rightarrow a_L, a_R$, $b \rightarrow b_L, b_R$
  - $P_1 \leftarrow$ MULT($a_L, b_L$)
  - $P_2 \leftarrow$ MULT($a_L, b_R$)
  - $P_3 \leftarrow$ MULT($a_R, b_L$)
  - $P_4 \leftarrow$ MULT($a_R, b_R$)
  - Return $P_1 \cdot 10^n + (P_2 + P_3) \cdot 10^{n/2} + P_4$
```

appending zeros, not recursive call

Runtime: $T[n]$:= time taken for n digit input

$$T[n] = 4 \cdot T[n/2] + \underbrace{O(n)} \text{ (addition, some } c \cdot n)$$



# of nodes: $1, 4, 16, \ldots 4^k \ldots, 4^{\log_2 n}$

work per node: $c \cdot n, c \cdot (n/2), c \cdot (n/4), \ldots c \cdot (\frac{n}{2^k}) \ldots, c \cdot (\frac{n}{2^{\log_2 n}})$

→ total work: $1 \cdot cn + 4 \cdot c \cdot \frac{n}{2} + \cdots + 4^{\log_2 n} \cdot c \cdot \frac{n}{2^{\log_2 n}}$

$$= \cdots + cn\left(\frac{4^k}{2^k}\right) + \cdots = \underline{O(cn \cdot 2^{\log_2 n})}$$

$$\Rightarrow O(cn \cdot 2^{\log_2 n}) = O(c \cdot n \cdot n^{\log_2 2}) = O(cn^2) = \underline{\underline{O(n^2)}}$$

Idea: Somehow, reduce 4 recursive calls to 3.

↳ If possible, equation becomes $O(c\cdot n\cdot(\frac{3}{2})^{\log_2 n}) = O(n\cdot n^{\log_2 3/2})$

$= O(n^{\log_2 2}\cdot n^{\log_2 3/2}) = O(n^{\log_2(2\cdot 3/2)}) = O(n^{\log_2 3}) \simeq \underline{\underline{O(n^{1.7})}}$

Observation: $a = a_L 10^{n/2} + a_R,\ b = b_L 10^{n/2} + b_R$

$\rightarrow a \times b = (a_L b_L)10^n + (a_L b_R + a_R b_L)10^{n/2} + a_R b_R$

$= \underbrace{(a_L b_L)}_{}10^n + \left[\underbrace{(a_L + a_R)(b_L + b_R)}_{} - \underbrace{a_L b_L}_{} - \underbrace{a_R b_R}_{}\right]10^{n/2} + \underbrace{a_R b_R}_{}$

```
KMULT(a[1-n], b[1-n]):
  - If $n<2$, return $a \times b$.
  - Split $a \rightarrow a_L, a_R$, $b \rightarrow b_L, b_R$
  - $P_1 \leftarrow$ KMULT($a_L, b_L$)
  - $P_2 \leftarrow$ KMULT($a_R, b_R$)
  - $P_3 \leftarrow$ KMULT($(a_L + a_R), (b_L + b_R)$)
  - Return $P_1 \cdot 10^n + (P_3 - P_1 - P_2)10^{n/2} + P_2$
```

Geometric Progression Fact

1) Sum of a n-term geometric progression $\propto O$(last term) when ratio > 1.

Recurrence Relations

ex1) $T[n] = \underbrace{T[n-1]} + \sqrt{n}$

$$= \underbrace{T[n-2] + \sqrt{n-1}} + \sqrt{n}$$

$$= T[n-3] + \sqrt{n-2} + \sqrt{n-1} + \sqrt{n}$$

$$= \underbrace{T[\overset{\nearrow 1}{1}]} + \sqrt{2} + \sqrt{3} + \cdots + \sqrt{n-1} + \sqrt{n}$$

$\sqrt{1} + \sqrt{2} + \cdots + \sqrt{n} \leq n \cdot \sqrt{n}$ $(n \cdot (\text{last term})) = n^{1.5}$

$\sqrt{1} + \sqrt{2} + \underbrace{\cdots + \sqrt{n}}_{\text{second half}} \geq \sqrt{n/2} + \cdots + \sqrt{n} \geq \frac{n}{2}\sqrt{\frac{n}{2}} = \left(\frac{n}{2}\right)^{1.5} = n^{1.5}/2\sqrt{2}$

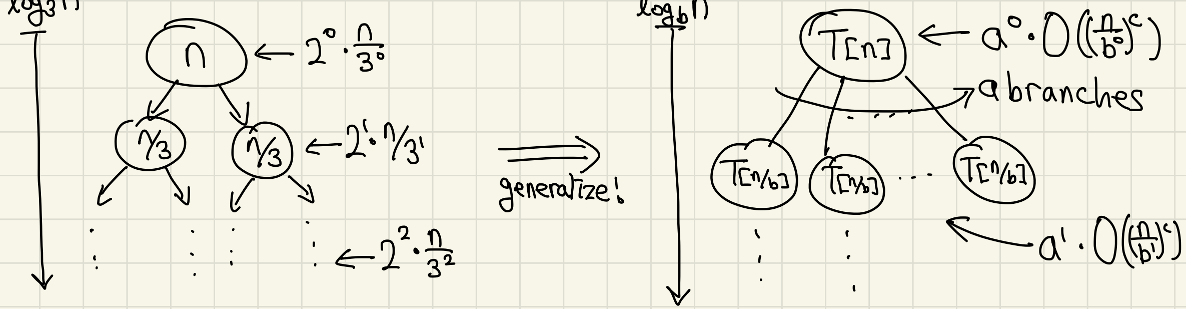
$\rightarrow T[n] = \Theta(n^{1.5})$ (bounded by $\frac{n^{1.5}}{2\sqrt{2}} \leq T[n] \leq n^{1.5}$)

ex2) $T[n] = 2T[n/3] + n$

$$= 2\left[2T[n/9] + n/3\right] + n$$

$$= 2\left[2\left[2T[n/27] + \frac{n}{9}\right] + n/3\right] + n$$

Strategy: draw a tree for visualization



$\log_3 n$
n
$\leftarrow 2^0 \cdot \frac{n}{3^0}$
$n/3$
$n/3$
$\leftarrow 2^1 \cdot n/3^1$
$\leftarrow 2^2 \cdot \frac{n}{3^2}$
generalize!
$\log_b n$
$T[n]$
$\leftarrow a^0 \cdot O((\frac{n}{b^0})^c)$
a branches
$T[n/b]$
$T[n/b]$
$T[n/b]$
$a^1 \cdot O((\frac{n}{b^1})^c)$

Master Theorem

Suppose function $T: \mathbb{N} \to \mathbb{R}^+$ satisfies relation

$$T[n] = aT[n/b] + O(n^c).$$

case 1: $c < \log_b a \to T[n] = O(n^{\log_b a})$.

↳ # of tree nodes dominates the runtime.

case 2: $c = \log_b a \to T[n] = O(n^c \log n)$.

↳ branching and work each layer are balanced.

case 3: $c > \log_b a \to T[n] = O(n^c)$.

↳ work inside the node dominates the runtime.

Matrix Multiplication

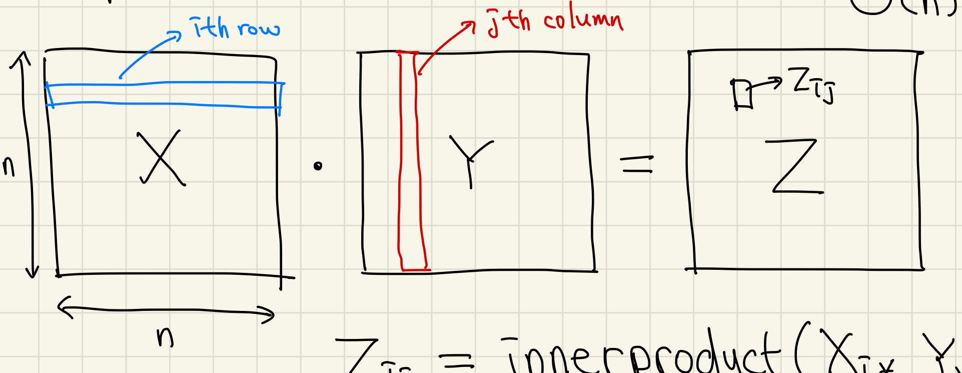
Input: X, Y $n \times n$ matrices

Output: $Z = X \cdot Y$

(Inner product of $\vec{x}, \vec{y}$

$= x_1y_1 + x_2y_2 \cdots + x_ny_n$)

↳ $O(n)$ operations



$$Z_{ij} = \text{innerproduct}(X_{i,*}, Y_{*,j})$$

Naive MatMul: Calculate each entry Z_{ij} seperately.

↳ Each entry takes $O(n)$, and total n^2 entries exists.

$$\Rightarrow O(n)\cdot n^2 = \underline{\underline{O(n^3)}} \text{ time}$$

Use Divide & Conquer: split X and Y into smaller matrices

$$\underset{X}{\begin{bmatrix} A & B \\ C & D \end{bmatrix}} \cdot \underset{Y}{\begin{bmatrix} E & F \\ G & H \end{bmatrix}} = \underset{Z}{\begin{bmatrix} AE+BG & AF+BH \\ CE+DH & CF+DH \end{bmatrix}}$$

→ can treat small matrices like values

A ⋯ H are $(n/2)\times(n/2)$ matrices.

Now, computing $\underbrace{AE, BG, \cdots, CF, DH}_{\text{↳ 8 total}}$ gives Z.

```
MATMUL(X, Y)
  - $X \to \begin{bmatrix} A & B \\ C & D \end{bmatrix}$, $Y \to \begin{bmatrix} E & F \\ G & H \end{bmatrix}$
  - $P_1 \leftarrow$ MATMUL(A, E) ⋯ $P_8 \leftarrow$ ⋯
  - Return $\begin{bmatrix} (P_1+P_2) & (P_3+P_4) \\ (P_5+P_6) & (P_7+P_8) \end{bmatrix}$
```

$$\Rightarrow T[n] = 8T[n/2] + \overset{\text{cost for matrix addition}}{O(n^2)}$$

↳ by Master Theorem, $T[n] = \underline{\underline{\overset{\text{no improvement...}}{O(n^3)}}}$.

Strassen's algorithm actually gives 7 recursive calls!

↳ $T[n] = 7T[n/2] + O(n^2) \to T[n] = \underline{\underline{O(n^{\log_2 7 \approx 2.81})}}$

Finding Triangles

Input: Graph $G = (V, E)$ on n-nodes.

$A[i,j] = 1\{(i,j) \text{ is connected}\}$.

Goal: Find a triangular connection in the graph.

((u,v,w) such that $A[u,v] \cap A[u,w] \cap A[v,w]$)

↳ Naively, checking all triplets takes $O(n^3)$ time.

exercise1) use Strassen's to solve in $O(n^{\log 7})$ time.

exercise2) try without Strassen's.

Finding Median

→ unsorted!

Input: list of n numbers, Output: $\lceil n/2 \rceil$-th smallest number

Naive Algo: sort the list, then output the $\lceil n/2 \rceil$th index.

↳ Sorting takes $\Theta(n \log n)$ time

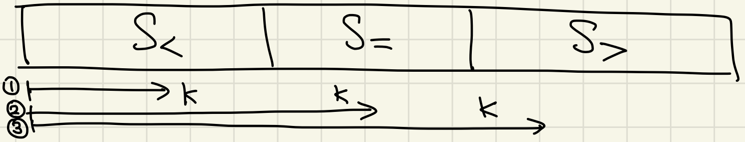
New Idea: Randomized $\Theta(n)$ time algorithm.

First, generalize the question to SELECTing $\underline{\underline{k\text{-th}}}$ smallest.

SELECT(A[1-n], k): outputs k-th smallest element in a.

↳ MEDIAN(A) = SELECT(A, $\frac{|A|}{2}$).

- Pick a random element $v \in A$ as a pivot.
- Split A into $S_< = \{a_i \mid a < v\}$, $S_= = \{a_i \mid a_i = v\}$, and $S_> = \{a_i \mid a_i > v\}$ ($O(n)$ time)



- case1: $k \leq |S_<|$. → Return SELECT($S_<$, k).

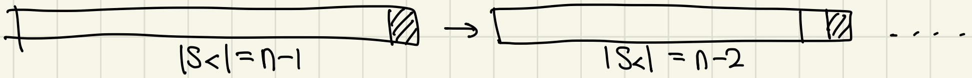
  case 2: $|S_<| < k \leq |S_<| + |S_=|$ → Return v. ($\forall e \in S_=, e = v$).

  case 3: $|S_<| + |S_=| < k$ → Return SELECT($S_>$, $k - |S_<| - |S_=|$).

Runtime Analysis: how to analyze a randomized algorithm?

↳ Best Case: first pivot is the kth element → $\Theta(n)$ (only splitting)

↳ Worst Case: pivot is the largest (or smallest) element <u>every time</u> → $\Theta(n^2)$



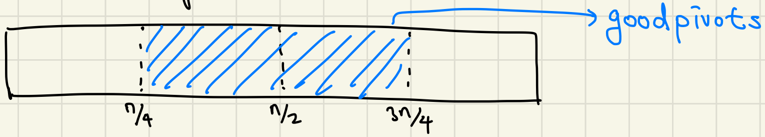
→ Define $T[n]$:= <u>Expected</u> runtime of SELECT

(the runtime is a random variable → $E[x] = \sum_{a \in X} Pr(X=a) \cdot a$)

Intuition: there is a reasonable chance that the random pivot is "good enough" to break into two significantly small lists.

define "good pivot": a pivot between $[n/4, 3n/4]$th smallest

for a sorted list:



Observation 1) Every good pivot splits both lists into lists smaller than $3n/4$ in size. (boundary → $n/4$, $3n/4$)

2) The probability that a random pivot is good is $1/2$.

$$\Rightarrow E[T[n]] = \underbrace{\overset{①}{E[T[n] \text{ before first good pivot}]} + \overset{②}{E[T[n] \text{ after first good pivot}]}}$$

↳ linearity of expectation $E = E_1 + E_2$

let v be the first time we hit a good pivot.

$$① \ (\underbrace{E[\#\text{of pivots before good pivot}]} \times \underbrace{n}) \leq 2n$$

↳ upper bound

(E[# of coin tosses before first heads]) $= 2$

$$② \ \leq E[T[3n/4]] \text{ (list size significantly dropped)}$$

$$\Rightarrow E[T[n]] = E[T[3n/4]] + \Theta(n) \xRightarrow[\text{Masters Theorem}]{} E[T[n]] = \Theta(n)$$

Examples in D&Q

1) Exponentiation: number n ⇒ a^n in decimal (array of digits)

ex) $2^{50} = \underbrace{2 \cdot 2 \cdot \cdots \cdot 2}_{50} = (2^{25})^2 \cdot 2^{25} = (2^{12})^2 \cdot 2 \cdot 2^{12} = (2^6)^2 \cdots$

$\text{EXP}(a, n : \text{integer}) \Rightarrow a^n$

- Base case: if $n=1$, return a.
- $B \leftarrow \text{EXP}(a, \lfloor n/2 \rfloor)$.

- If n is even, return $B \times B$.
- Else, return $B \times B \times a$.

Runtime: $T[n] = T[n/2] + \Theta$(time to multiply numbers)

If $a=2$, $2^n \rightarrow n$ bits long $\Rightarrow T[n] = T[n/2] + \Theta(M(n))$ where $M(n) :=$ time to multiply 2 n-digit numbers.

If $M(n) \gg n^{1.00001}$, $T[n] = \Theta(M(n))$ (by Masters Theorem)

2) Binary to Decimal: $B[1-n]$ bits $\Longrightarrow D[1-m]$ decimal array

Naive ex) $(1011011)_2 = 1 \times 2^6 + 0 \times 2^5 + \cdots + 1 \times 2^0 = 91$.

↳ $\Theta(n)$ additions of $\Omega(n)$-digit numbers $\rightarrow \Omega(n^2)$

D&Q approach ex) $(\underbrace{\overset{B_L}{1011}}\,\underbrace{\overset{B_R}{1100}})_2 = (1011)_2 \times 2^4 + (1100)_2$.

$= 11 \times 16 + 12 = 188$.

B2D ($a[1-n]$) $\Rightarrow$ decimal digit array

- Base case: len$(a) == 1 \rightarrow$ return $a[0]$
- $a_L \leftarrow a[1-n/2]$, $a_R \leftarrow a[(n/2+1)-n]$
- $d_L \leftarrow$ B2D(a_L), $d_R \leftarrow$ B2D(a_R)
- $C \leftarrow$ EXP$(2, n/2)$
- Return $d_L \underbrace{\times} C \underbrace{+} d_R$. → of n-digit numbers!

Runtime: $T[n] = 2T[n/2] + \Theta(EXP(2, n/2)) + \Theta(\text{n-digit mult})$
$+ \Theta(\text{n-digit addition}) \Rightarrow T[n/2] = \Theta(M(n))$

3) Closest Pair: n $\overset{p:=}{(x,y)}$ points in plane $\Rightarrow$ closest pair $\{\overset{P_i=}{(x_i, y_i)}, \overset{P_j=}{(x_j, y_j)}\}$

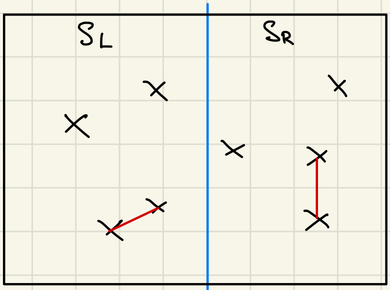
Naive: check all (P_i, P_j) pairs' distance, and find the smallest.

↳ $\Theta(n^2)$ runtime due to pairing

D&Q: $\{P_1, P_2, \ldots, P_n\} \longrightarrow \overset{A:=}{\{P_1, \ldots, P_{n/2}\}}, \overset{B:=}{\{P_{(n/2+1)}, \ldots, P_n\}}$?

a better splitting: split the plane (the geometry)

↳ sort the points in increasing x-coordinate, then split.



Recurse to find closest pair in S_L & S_R.

$d \leftarrow \min(\text{Closest}(S_L), \text{Closest}(S_R))$.

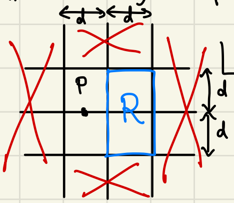
What if the actual closest pair is split?

↳ Naive: $n/2 \times n/2$ pairs $\rightarrow \Theta(n^2)$ runtime... → how to prune?

Idea 1) take strip of width d on each side of the line.

↳ not very helpful for worst-case analysis...

Idea 2) a point P only needs to be tested with points $\geq d$ away.



↳ only points in region R needs to be tested.

however, there are at most 8 points in region R.

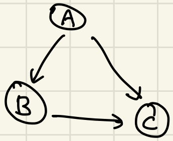
→ For every point P, # of comparisons ≤ 8

↳ $\Theta(n)$ pairs need comparison!

$\Rightarrow T[n] = 2T[n/2] + \Theta(n) \Rightarrow T[n] = \Theta(n \log n)$

Graphs

Graphs: $G=(V,E)$. $(u,v) \in E$ if $u \to v$.



Directed - edges have directions.

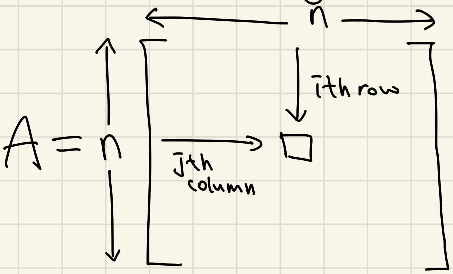
Parameters: $n = |V| =$ # of vertices

$m = |E| =$ # of edges

$\Rightarrow$ for all non-multi-edge graphs, $m < n^2$

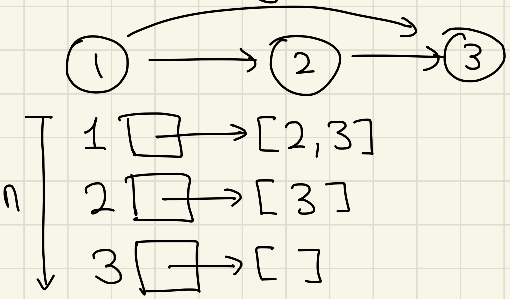
Representation on computers: $V = \{1 \ldots n\}$. $E = ?$

1) Adjacency Matrix



$A[i,j] = 1\{(i,j) \in E\}$

2) Adjacency List (of out-edges)



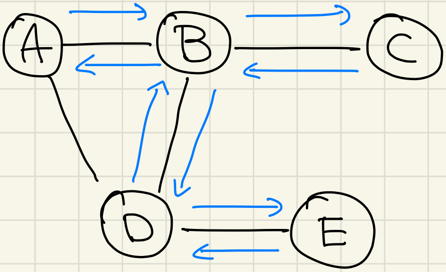
what are trade-offs of each representation?

	Matrix	List
size(memory)	$\Theta(n^2)$	$\Theta(n+m)$
query time ($(u,v)\in E$?)	$\Theta(1)$	$\Theta(\deg(u))$
neighbor enumeration of u	$\Theta(n)$	$\Theta(\deg(u))$

Connectivity: Is there a path from u to v?

↳ Is G connected? What are connected components?

DFS in Undirected Graphs



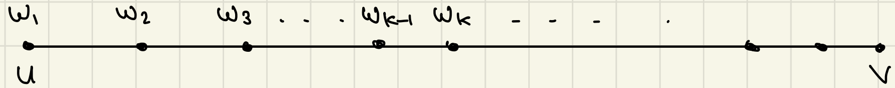
ex) explore(A) → A, B, C, DE

```
explore ( vertex v ):
  - visited[v] = true
  - for each edge v → w:
      - if not visited[w]: explore[w]
DFS(Graph G):   ← generalized to disconnected graphs
  - visited[u] = false  ∀u ∈ V
  - for each vertex v ∈ V:
      - if not visited[v]: explore(v)
```

Property: explore(u) visits exactly the vertices v such that Graph G has a path from u to v.

Proof: 1) Vertex v is reached $\Rightarrow \exists$ path from u to v (trivial)

2) $\exists$ path from u to v $\Rightarrow$ Vertex v is reached by explore(u)



Suppose explore(u) does not reach v, for the sake of contradiction.

Let w_k be the first vertex on the path that is not reached.

$\Rightarrow$ w_{k-1} is reached. $\Rightarrow$ explore(w_{k-1}) is called.

In explore(w_{k-1}), all edges incident to w_{k-1} will be explored, including w_k. → Contradiction, explore(u) reaches v. //

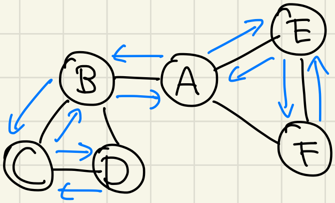
Finding Connected Components: Modify explore and DFS!

```
DFS(Graph G):
  - count = 0
  - ccnum ← int[n]
  - visited[u] = false ∀u ∈ V
  - for each vertex v ∈ V:
      - if not visited[v]: explore(v), count += 1
```

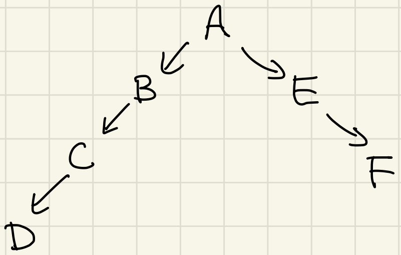
```
explore(vertex v):
  - visited[v] = true
  - ccnum[v] = count
  - for each . . .
```

ensures that only connected components will have same # in ccnum.

DFS Search Tree:



ex) explore(A) calls



Runtime of DFS: 1) explore(V) is called once per DFS.

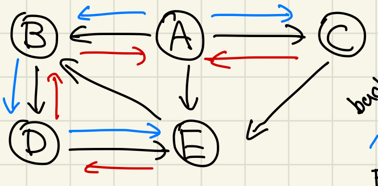
2) Inside explore(V), set visited[V] = true ← $\Theta(1)$ time,

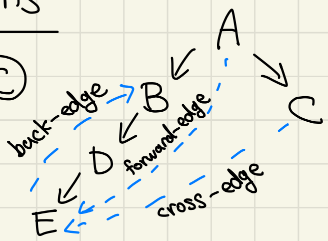
then enumerate alledges $V \to W$ ← $\Theta(\deg(V))$ time

Total time $= \sum_{v \in V}(1+\deg(v)) = \underline{\underline{\Theta(n+m)}}$ $\left(\sum_{v\in V}\deg(v) = \Theta(|E|) = \Theta(m)\right)$

DFS In Directed Graphs

ex)





Recording times: increment a clock everytime we reach or leave a vertex, and set pre[n] and post[n]

→when DFS "enters"

→when DFS "leaves"/ "returns"

In explore(v):

```
    ⋮
pre[v] = clock
 clock += 1
for each · · ·
  · · · ·
post[v] = clock
clock += 1
```

In DFS(G):

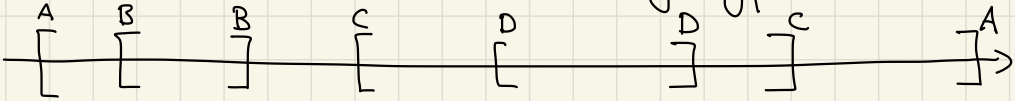
```
clock = 0

pre, post ← int[n]
for all v ∈ V:
    · · · ·
```

in above example:

A = [1, 10]
B = [2, 7]
C = [8, 9]
D = [3, 6]
E = [4, 5]

← [pre[v], post[v]] for all v∈V

Pre&Post numbers can inform the edge types between nodes.

ex) 

for an edge u→v: if $\underset{u}{[}\ \underset{v}{[}\ \underset{v}{]}\ \underset{u}{]}$, tree or forward edge.

if $\underset{v}{[}\ \underset{u}{[}\ \underset{u}{]}\ \underset{v}{]}$, back edge. if $\underset{v}{[}\ \underset{v}{]}\ \underset{u}{[}\ \underset{u}{]}$, cross edge.

$\underset{u}{[}\ \underset{v}{[}\ \underset{u}{]}\ \underset{v}{]}$ is impossible (can't close u before v closes)

⇒ For all edges u→v, post[u] < post[v] iff u→v is a back edge.

↳ no back edges ⟺ no directed cycles ⟺ is DAG

Directed Acyclic Graphs

DAG: Directed graph with no directed cycles.

Applications: 1) Modeling dependencies / prerequisites

↳ u→v if u is a prerequisite for v.

Source code compilation → checking dependency cycles

2) Partially ordered sets (comparisons, but not transitive)

↳ ex) box sizes: box A fits in box B.

source: node with no incoming edges

sink: node with no outgoing edges

↳ every DAG has at least one source & one sink.

Topological Sort (Linearization)

TOPSORT(DAG G) $\Rightarrow$ ordering of all vertices $[v_1, v_2, \cdots, v_n]$

(all edges of the linearized vertices head from left to right)

Algorithm 1: ($\Theta(m+n)$)

- Run DFS to compute pre & post values
- Output vertices in decreasing post values

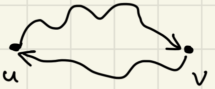
Algorithm 2: ($\Theta(?)$, depends on implementation details)

- Pop a source node, output it
- Repeat with the remaining smaller DAG.

Proof of Correctness of Algo 1: (the concept)

$\forall$ edge $u \to v$, prove $post[u] > post[v]$

Connectivity in Directed Graphs



↳ can have other nodes in between

u is <u>strongly connected</u> to v iff $\exists$path $u \rightsquigarrow v$ & $\exists$path $v \rightsquigarrow u$

↳ every directed graph can be decomposed in to a dag of strongly connected components (<u>DAG of SCCs</u>) *

How to decompose a directed graph into SCCs?

↳ goal: label all vertices with their "component number".

Intuition: Run explore(v) for some vertex v in a "sink SCC".
↳ sink in the DAG of SCCs

This will recover exactly that sink SCC.

Repeatedly recovering sink SCCs will complete the task.

⇒ But how to locate a vertex in a sink SCC?

FACT: In a DFS traversal, a vertex with the highest post value will be in a source SCC. (exits very last in DFS)

↳ how to get sink SCC? ⇒ reverse edges in G!

KOSARAJU's algorithm (DAG G):

- Construct a reverse graph of G, G^R.
- Run DFS on G^R to compute $post_R$ values.
- Run DFS on G by exploring vertices in decreasing order of $post_R$.

↳ every iteration of last step recovers exactly an SCC.

Breath-First Search

- Maintain a queue for edges to explore next

↳ Naturally solves the <u>shortest path</u> question

Dijkstra's Algorithm $(G=(V,E),\ \overset{\text{start node}}{s} \in V) \Rightarrow \text{dist}[v \in V]$

Intuition: Imagine a liquid spill at node s. The liquid moves unit distance in 1 time step. Simulate this liquid's motion.

↳ Simulating every timestep can be inefficient...

⇒ only simulate the "interesting" times when a node is reached!

↳ make a note on ETA of s's neighbors, and fast-forward to closest

once a node is reached, update ETA of its neighbors

Data structure needed → <u>Priority Queue</u> of $(\overset{\text{(key)}}{\text{time}}, \overset{\text{(value)}}{\text{vertex}})$ pairs

Operations required → <u>deleteMin()</u>: pop & return the smallest time

<u>decreaseTime(time', vertex)</u>: if (t, vertex) is a part of the PQ, $t \leftarrow \min(t, \text{time}')$

```
DIJKSTRA'S (G, We, S):
    dist[v] ← ∞, dist[s] ← 0.
    Q ← make queue, Q.insert(dist[v], v) ∀ v∈V.
```

```
While Q is not empty:
    $(t, v) \leftarrow Q.\text{deleteMin}()$
    for $v \to u \in E$:
        $Q.\text{decreaseTime}(\text{dist}[v] + W_{v \to u}, u)$
Return dist
```

Binary Heap: supports deleteMin & insert (also delete)

↳ both operations take $\Theta(\log(|V|))$, deleteMin called $|V|$ times, insert called $|E|$ times $\Rightarrow$ $\underline{\underline{T(\text{Dijkstra's}) = \Theta(\log(m)(m+n))}}$

(Generally, $\Theta(m \cdot T(\text{deleteMin}) + n \cdot T(\text{decreaseKey}) + m \cdot T(\text{insert}))$)

Bellman-Ford Algorithm: an alternative shortest-path

Intuition: All edges are rubber bands of lengths equal to weight.

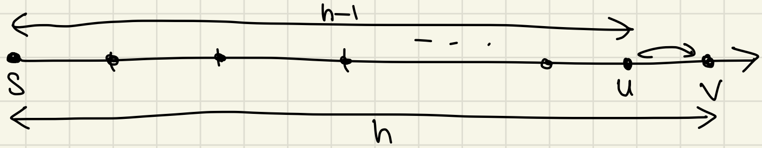
Initially, all bands are stretched upto "infinity".

Every edge is updated by "unstretching" a node to the left.

↳ the order of updates does not matter

Mathematical Analysis: $D[\underset{v}{\text{vertex}}, \underset{h}{\text{hop}}] :=$ length of shortespath $s \rightsquigarrow v$ that only uses at most $\underline{h \text{ hops}}$ (# of edges the path goes through in the path)

↳ $D[v,h] = D[u,h-1] + w_{u\to v}$



```
BELLMAN-FORD(G, w, s):
 $\forall v \in V, D[v,0] \leftarrow \infty$, $D[s,0] \leftarrow 0$
 for all hops h from 1 to $|V|-1$:
     for each edge $u \to v$:        a dynamic programming example!
         $D[v,h] = \min(D[u,h-1] + w_{u\to v}, D[v,h])$
 $\forall v \in V, dist[v] \leftarrow D[v,|V|-1]$
```

↳ for space efficiency, we can replace D with dist and update in-place ($dist[v] = \min(dist[v] + w_{u\to v}, dist[v])$

Connection to intuition: inner loop is unstretching all edges once, outer loop is repeating the inner loop in case some rubber band is unsatisfied

Greedy Algorithm

Goal: Optimize a multi-step decision process

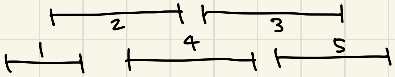
Being "Greedy": Optimize for next step only, works sometimes

↳ If the local <u>optimum can be connected to a global optimal point</u>.

Task Scheduling Problem: n jobs with start and end times

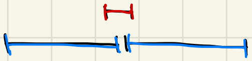
↳ schedule as many <u>number of jobs</u> without overlaps

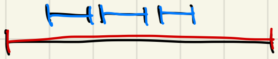
→optimal

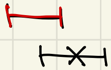
ex) 

→ T2&T3, T1&T4, <u>T1&T4&T5</u>, . . .

Possible strategies: ① shortest first ② begin at first ③ finish first

①  → not optimal (picks one job over two)

②  → not optimal (picks one job over three)

③  → optimal! how to prove local → global connection?

Claim: Greedily picking the first job that finishes without overlapping is the optimal solution

Proof: Greedy Solution $[S_1, e_1] \cdots [S_R, e_R]$

Optimal Solution $[S_1, e_1] \cdots [S_L, e_L]$

Observation: $R \leq L$ since L is optimal.

$$\forall i \in [0,R],\ H_i = \underbrace{\overbrace{[s_1, e_1] \cdots [S_{iR}, e_{iR}]}^{\text{Greedy}}}_{\text{first i jobs from Greedy}} \underbrace{\overbrace{[S_{i+1_L}, S_{i+1_L}] \cdots [S_L, e_L]}^{\text{Optimal}}}_{\text{rest from optimal}}$$

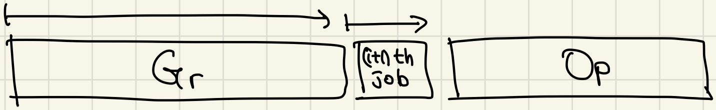
→ H_0 is the optimal solution, and H_R is full greedy + leftover optimal

Now, we argue that all $H_i \in [H_0, H_R]$ are optimal.

Base Case : H_0 is trivially optimal (by definition)

Induction: Given that H_i is optimal, prove that H_{i+1} is optimal

H_i :



When the greedy algorithm picks the (i+1)th job, it picks the job with the earliest finish time $\le e_{i+1_L}$ (by greediness)

$\rightarrow e_i < S_{i+1} < \underbrace{\overbrace{e_{i+1_R} \le e_{i+1_L}}^{\text{greediness}} \overbrace{< S_{i+2_L}}^{\text{by construction of Op}}}_{\text{non-overlapping}}$

⇒ Greedy preserves number of jobs and does not overlap with the start of the next optimal job, S_{i+2_L}. Also, since the procedure will continue until e_L, $R = L$ in all case.

```
SCHEDULE(n jobs with [$S_n$, $e_n$]):
 A ← ∅, $t^*$ ← −∞   → end time of last scheduled job
 for each j in [1...n]:
   if $t^* \le S_j$ : A.add([$S_j$, $e_j$]), $t^*$ ← $e_j$
 return A
```

Runtime: $O(n)$ if sorted by e_n, $O(n \log n)$ if not

Compression (Huffman Encoding): Encoding with least number of bits

In text T with alphabet π and frequency f_π,

minimize $cost(T): \sum_{\pi} f_\pi \cdot$ (# of bits π is encoded to)

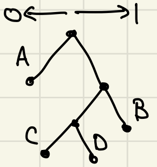
ex) T=100

π	f_π	2 bits	unequal?
A	80	00	0
B	10	01	1
C	5	10	10
D	5	11	11
cost(T)		200	<200

→ unequal bits reduce cost(T), but it introduces ambiguity such as 10 → BA, or C?

→ Prefix Freeness Property needed!

Prefix Freeness: no encoding is a prefix of another



↳ can be represented by leaves in a full binary tree (all nodes have either 0 or 2 children)

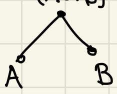
Strategies: ① schedule the most frequent first ② least frequent first

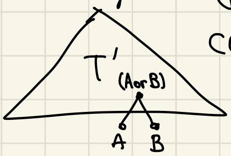
① not optimal, may not be worth adding 1 bit to all others

② build the tree bottom-up → optimal!

Given $\{f_1, \ldots, f_n\}$, pick lowest frequencies f_A & f_B, remove them and add a new frequency $f_{(A or B)} = f_A + f_B$. Iterate.

(AorB) → $f_{(AorB)} = f_A + f_B$





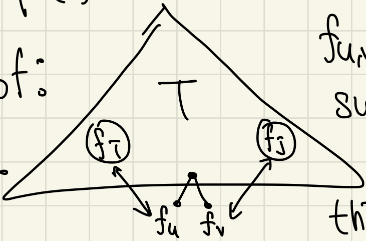
$cost(T) = cost(T') + \underbrace{f_A + f_B}$

both A and B contribute 1 bit if selected

```
HUFFMAN(T, π, f):
  Q ← priority queue of min f value,
  insert all f into Q
  while Q.size() > 1 :
    $f_A, f_B$ ← Q.pop(), Q.pop()
    $f_{(A or B)} \leftarrow f_A + f_B$, construct edge $f_{(A or B)} \rightarrow f_A, f_B$
    Q.insert($f_{(A or B)}$)
  return Q.pop()
```

Optimality Proof:

T is the optimal.

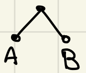


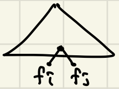
$f_{u,v}$ are deepest leaf nodes. switch f_u&f_v with f_i&f_j with the lowest frequencies. this can only reduce cost(T).

However, T is already optimal. ⇒ f_i & f_j are already in place of f_u&f_v.

⇒ constructing T with lowest frequencies at bottom is consistent with the optimal T.

→ HUFFMAN enforces this at every step

Base Case: $n=2$ →  (only possible configuration)

Induction: f_i, f_j →  for (n+1) frequencies, we can reduce it to n frequencies consistent with the optimal T.

n frequencies is solved by IH $\Rightarrow$ $(n+1)$ frequencies also solved! //

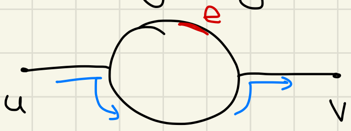
Runtime: n inserts, deletes for max depth $\log n$ $\rightarrow$ $O(n \log n)$

Minimum Spanning Trees

Tree: An undirected graph that is (i) connected and (ii) acyclic.

Property 1: removing a cycle edge does not disconnect a graph.

Proof:



case1) $u \rightsquigarrow v$ path does not use edge e.

↳ trivial, done.

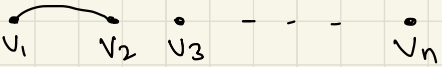
case2) $u \rightsquigarrow v$ involves e $(u \rightsquigarrow e \rightsquigarrow v)$

In case 2, we can always construct another path without e.

↳ take the "other direction" of the cycle. //

Property 2: A tree with n vertices has $(n-1)$ edges.

Proof:



$t=0$ → n components

$t=1$ → $(n-1)$ component

Adding an edge will always reduce # of components by 1

↳ if the new edge connects two vertices in the same component, it will introduce a cycle

$\Rightarrow$ at time $(n-1)$, there will be 1 component left, the tree. //

Property 3: A connected graph with n vertices & (n-1) edges is a tree.

Proof: Assume the graph has acycle. Remove the cycle edge. By property 1, it is still connected. Repeat until all cycles are gone. It should have (n-1) edges by property 2. However, since we started with (n-1) edges, it means that there were no cycles to remove to begin with → original graph is a tree. //

$MST(G=(V,E), w_e) \Rightarrow T=(V,E')$ s.t. $E' \subseteq E$ s.t.

$cost(T) = \sum_{e \in E'} w_e$ is minimized

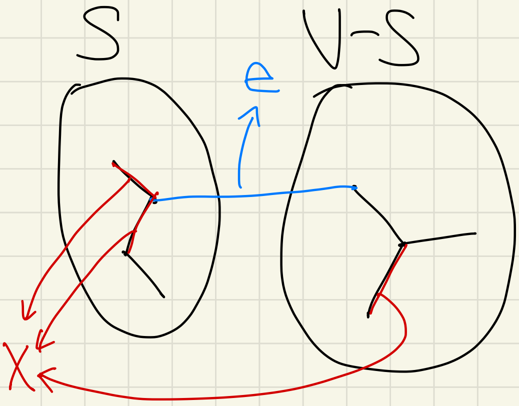
Take a greedy approach: Add the least weighted edge that does not introduce a cycle; and iterate.

Main Theorem: (i) Let $X \subseteq E$ be part of some MST T of G.

(ii) $S \subseteq V$ be a set s.t. there are no edges in X from S to $V-S$.

(iii) Let $e \in E$ be the lightest edge from S to $V-S$.

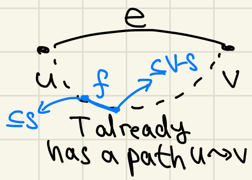
$\Rightarrow$ $X+e$ is apart of some MST of G, not necessarily the MST defined above.



Consider $T+e$:

case 1) $e \in T \Rightarrow X+e \subseteq T+e$

case 2) $e \notin T \Rightarrow T+e$ has a cycle



→ there has to be the first edge that crosses S to V-S in the T path

Claim: $w_f \geq w_e$, since e is the lightest edge from S to V-S.

Now, consider $T' := T+e-f$. ① By property 1, $\underline{T' \text{ is connected}}$.

② T' still has $(n-1)$ edges → By property 3, $\underline{T' \text{ is a tree}}$.

③ $\text{cost}(T') = \text{cost}(T) + W_e - w_f$. By the claim above,

$\text{cost}(T') \leq \text{cost}(T)$. However, since T is an MST,

$\text{cost}(T') = \text{cost}(T)$. $\Rightarrow$ $\underline{T' \text{ is an MST}}$, different from T.

$\overset{\hookrightarrow \text{by (i)}}{\underline{X \subseteq T}}$, $\overset{\hookrightarrow \text{by (ii)}}{\underline{f \notin X}}$, $e \in T' \Rightarrow X+e \subseteq T'$, which is an MST.

$\Rightarrow X+e$ is still a part of some MST, albeit not T but T'. //

Kruskal's Algorithm: go over all edges in increasing weights. Add it if it doesn't introduce a cycle; skip otherwise.

Claim: Kruskal's finds an MST.

Base Case: $X = \varnothing$ → part of every MST

Induction: $X \rightarrow X+e$ still is a part of an MST by the Main Theorem proved above.

Implementation: ① track connected components ② cycle detection

Union Find: makeSet(x): makes singleton set $\{x\}$.

find(x): find the set x belongs to. union(x,y): make a union of the set containing x and the set containing y.

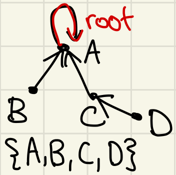
```
KRUSKAL(G, w):
  for all $v \in V$, makeSet(v).
  $X \leftarrow \emptyset$. sort edges E by w.
  $\forall (u,v) \in E$ in sorted order,
    if find(u) $\neq$ find(v):
      $X \leftarrow X \cup \{(u,v)\}$
      union(u,v)
  return X
```

Runtime: $O(\log(|V|)(|V|+|E|))$

↳ $|E|\log(|V|)$ sorting,

$2|E|\ \underbrace{\underline{\text{find}}\text{ calls}, |V|\ \underline{\text{union}}}_{\text{both } O(\log(|V|))}$ calls

The Union Find Data Structure



$\pi(x)$: parent of x. rank(x): height of tree under x

makeset(x): set $\pi(x) = x$, $\text{rank}(x) = 0$.

find(x): if $\pi(x) \neq x$, find($\pi(x)$). else, return x.

for union, connect the root of x to root of y, or vice versa.

How to choose between $x \rightarrow y$ or $x \leftarrow y$?

Observation: minimizing rank optimizes find operations.

→ leads to shallower tree, less ancestors to call

Union(x,y):

```
Union(x,y):
    $r_x, r_y \leftarrow$ find(x), find(y)
    if $\text{rank}(r_x) \leq \text{rank}(r_y)$:
        $\pi(r_x) \leftarrow r_y$   # $r_x$ goes "under" $r_y$
        if $\text{rank}(r_x) == \text{rank}(r_y)$, $\text{rank}(r_y)$ += 1.
    else, $\pi(r_y) \leftarrow r_x$   # $r_y$ goes "under" $r_x$.
```

→ $O(\log n)$, ($O(\log^* n)$ if path compressed)

Runtimes: makeset → $O(1)$, find & union → $O(\text{rank of root(s)})$

Claim: If $\text{rank}(x) = r$, then x has $\geq 2^r$ nodes in tree rooted in r.

Base Case: $r = 0 \rightarrow$ # of nodes $= 1 \geq 2^0$ ✓

Induction: $r \rightarrow r+1$ # of nodes in the first tree + second tree $\geq 2^k + 2^k = 2^{k+1}$

Prim's Algorithm: exploit the Main Theorem like Dijkstra's

$X \leftarrow \emptyset$, Repeat until $|X| = (n-1)$:

Pick $S \subseteq V$ s.t. there are no edges in X crossing S & $V-S$.

Let e be the minimum weighted edge from S to $V-S$.

$X \leftarrow X \cup \{e\}$. → X spans exactly 1 more vertex now.

↳ S is just all vertices that X currently spans.

↳ $|S| = |X| + 1$

⇒ Implement using priority queue like Dijkstra's.

Runtime: $O(\log(|V|)(|V|+|E|))$

Horn's Formula: given boolean variables $(x_1, \dots, x_n)$ and clauses $C_1, \dots, C_m$ s.t. $\forall C_i$, either $\overset{\text{pure negation}}{\underline{(\bar{x}_1 \cup \bar{x}_2 \cup \dots)}}$ or $\overset{\text{implication to } x_\alpha}{\underline{(\bar{x}_1 \cup \dots \cup x_\alpha)}}$, is there an assignment that satisfies $F = C_1 \cap C_2 \cap \dots \cap C_m$?

* $(\bar{x}_1 \cup \bar{x}_2 \cup \dots \cup x) \equiv (x_1 \cap x_2 \cdots) \Rightarrow x$, $(\Rightarrow x)$ is a special case.

ex) $(w \cap y \cap z) \Rightarrow x$ $\qquad$ $\underline{(\bar{w} \cup \bar{x} \cup \bar{y})}$ ↳ not satisfiable

$(x \cap z) \Rightarrow w$ $\qquad$ $(\bar{z})$

$x \Rightarrow y$ → true

$\Rightarrow x$ → true

$(x \cap y) \Rightarrow w$ → true

⇒ this system is unsatisfiable

Greedy Approach: set all variables to False. set a variable to True only if absolutely necessary.

```
HORN(F):
  set all variable $x \in X$ to False
  while $\exists$ an unsatisfied implication clause C:
      set the right hand variable to True
  if any negation clause is unsatisfied, return "unsatisfiable".
  else, return the assignment $x_1, \ldots, x_n$.
```

Runtime: $O(|F| \times n)$, where $|F| \propto$ # of clauses & variables

Correctness: If HORN(F) sets a variable to TRUE, then it is TRUE in any satisfying assignment to F.

Base Case: $k=1 \rightarrow (\Rightarrow x)$ will be trivially $x \leftarrow$ TRUE.

IH: $k \rightarrow (k+1) \rightarrow x_{i_1}, \ldots, x_{i_k}$ are all set to TRUE. $x_{i_{k+1}}$ is the new variable about to be set to TRUE.

$\underbrace{(x_{i_1} \cap x_{i_2} \cap \cdots \cap x_{i_k})}_{\hookrightarrow \text{all or subset of previous TRUE assignment}} \Rightarrow x_{i_{k+1}}$ is the only way, which ensures that $x_{i_{k+1}}$ is always set to TRUE. //

Claim: HORN(F) is correct.

case1) HORN(F) outputs an assignment (true by definition)

case2) HORN(F) outputs "unsatisfiable".

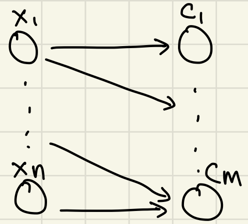
↳ only sets those variables to be TRUE that are TRUE in every satisfying assignment, if F were satisfiable.

↳ then, some pure negative clause is always unsatisfied!

⇒ F is indeed unsatisfiable. //

Can we improve the runtime from $O(|F| \times n)$?

Idea:



add edge (x_i, c_j) if x_i appears on the LHS of C_j.

Observation: 1) if C_i has no incoming edges, RHS is TRUE.

2) Once x_i is set to TRUE, we can remove the vertex since it does not affect the implications anymore.

↳ implement using a queue that contains all TRUE variables.

⇒ only recompute clauses that are affected by assignments!

Runtime: $O(|F| + n)$, where $|F| \propto$ # of edges in graph

* no clauses with no incoming edges ⇒ all variables set to FALSE is valid

Dynamic Programming

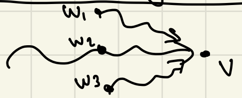
"A versatile and powerful algorithm design tool"

Longest Path in DAG: DAG $G(V,E) \Rightarrow \ell$, the longest path length

Subproblem: $L(v) :=$ length of the longest path ending in v, $\ell = \max_{v \in V} L(v)$

↳ make subproblems such that bigger problems depend on smaller ones!

Connecting Subproblems: Recurrence Relation



$L(v) = 1 + \max_{(w,v)\in E}(L(w))$, 0 if $\forall w \in V, (w,v) \notin E$.

↳ naive recursive implementation recomputes same $L(w)$ many times, leading to exponential time. → start with smallest problem!

Avoid Recomputation: memoization of $L(w)$ values

- topologically sort G s.t. all i-th vertex has edges (i,j) where $j > i$.
- set $L(i) = 0$ for all i.
- For all $i = 1, \dots, n$, set $L(i) \leftarrow 1 + \max_{(j,i)\in E}(L(j))$, 0 if no incoming edges

Runtime: $O(|V| + |E|)$

Longest Increasing Subsequence: $a[1 \dots n] \rightarrow \ell$, length of LIS

↳ Reduces to finding longest path in DAG!

Consider $G(V,E)$ s.t. $V := \{1, \dots, n\}$, $E := \{(i,j) \mid i<j \text{ and } a[i] \leq a[j]\}$

DP Approach: 1) define an appropriate subproblem

2) write a recurrence relation to connect subproblems

3) determine the order of computation (DAG-structure!)

Edit Distance: $x[1,\dots,n]$ & $y[1,\dots,m] \Rightarrow$ minimum edit keystrokes s.t. $x=y$

1 keystroke needed to add, remove, or substitute a character.

ex) CAP → CUP (1 keystroke, replace A → U)

AAPPL → APPLE (2, remove A, add E)

SUNNY → SNNY → SNOY → SNOWY (3)

Visualization:

Delete	Insert	Sub	Keep
N	␣	N	Y
␣	W	O	Y

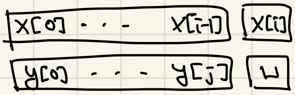
⇒

K	D	K	S	I	K
S	U	N	N	␣	Y
S	␣	N	O	W	Y

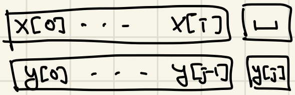
1) Subproblem: $E(i,j) := \text{EDIT}(x[1:i], y[1:j])$

ex) $E(\overset{\text{empty string}}{\emptyset}, S)$, $E(SUN, SNO)$, $E(SUNNY, SNOWY)$

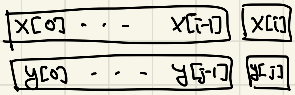
2) Recurrence Relation: edit $x[1 \dots i] \rightarrow y[1 \dots j]$, the last step has to be one of the three keystrokes, del, sub, or add.



del



add



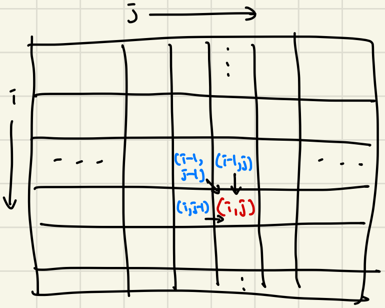
sub/keep

$$\Rightarrow E(i,j) = \min \begin{cases} 1 + E(i-1,j) \\ 1 + E(i,j-1) \\ \text{diff}(x[i],y[j]) + E(i-1,j-1) \end{cases}$$

$\text{diff}(a,b) := \begin{cases} 1 \text{ if } a == b, \\ 0 \text{ if } a \neq b \end{cases}$

Base Case: $E(0,0) = 0$, $\forall j$, $E(0,j) = E(j,0) = j$

3) Order of Computation: $E(i,j)$ depends on $\underline{E(i-1,j)}$, $\underline{E(i,j-1)}$, and $\underline{E(i-1,j-1)}$



computing row-by-row or column-by-column satisfies the dependency requirements.

ex)	∅	S	N	O	W	Y
∅	0	1	2	3	4	5
S	1	0	1	2	3	4
U	2	1				
N	3	2				
N	4	3				
Y	5	4				

```
∀i∈[1...n], E(i,0) ← i
∀j∈[1...m], E(0,j) ← j
for all i∈[1...n],
    forall j∈[i...m],
        $E(i,j) = \min \begin{cases} 1+E(i,j-1) \\ 1+E(i-1,j) \\ \text{diff}(x[i],y[j]) + E(i-1,j-1) \end{cases}$
return E(i,j)
```

Runtime: $O(nm)$

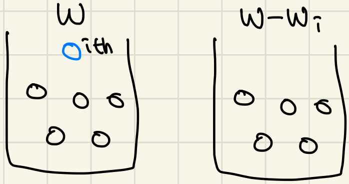
To retreive the edit <u>path</u>, keep a back pointer to keep track of which last step leads to the solution.

Knapsack: total weight capacity W, weight-value pairs (w_i, v_i), $i \in [1...n]$

$\Rightarrow$ maximum total value while total weight $\leq W$

Two variations: with replacement, or without repetition?

With replacement: there should be a "last item" that was added

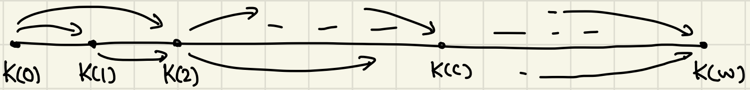


claim: without the i-th item, the remaining items is an optimal solution to knapsack $(W-w_i)$.

1) $K(C)$ = max value when capacity $C = 0 \dots W$

2) $K(C) = \max_{i: C \geq w_i} \{v_i + K(C - w_i)\}$, Base case: $K(0) = 0$.

3)



(nice linear ordering!)

```
KNAPSACK(W, V[1...n], w[1...n]):
    $K(0) \leftarrow 0$
    for $C = 1 \dots W$:
        $K(C) = \max_{i: w_i \leq C} \{v_i + K(C - w_i)\}$
    return $K(W)$
```

Runtime: $O(n\underline{W})$ → exponential w.r.t $\log(W)$ ≃ length of input

no replacement: recurrence needs to "carry" which were picked!

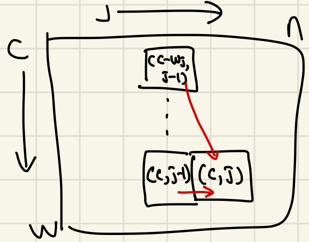
1) $K(C, j)$: max value when capacity $C = 0 \dots W$ using only items $\underline{1 \dots j}$.

2) $K(C, j) \rightarrow K(C, j-1)$ if $C < w_j$. what about $C \geq w_j$?

$\rightarrow \max\{\underbrace{K(C, j-1)}_{\text{not used any way}}, \underbrace{w_j + K(C-w_j, j-1)}_{\text{jth item is used! no more of it}}\}$ if $C \geq w_j$.

Base Case: $\forall j$, $K(0, j) = 0$.

3) a 2-D matrix with dimension C, j.



row-by-row or column-by-column both work

Runtime: $O(nW)$ (each entry takes $O(1)$)

$\longrightarrow m_0 m_1 m_2$ multiplications

Chain Matrix Multiplication: $A[m_0 \times m_1], B[m_1 \times m_2] \Rightarrow C[m_0 \times m_2]$

If we have a series of matmuls, $A \times B \times C \times D \times \cdots$,

what is the best parenthezation for calculation?

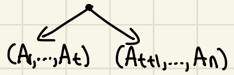
ex) $\underset{50\times20}{A} \times \underset{20\times1}{B} \times \underset{1\times10}{C} \times \underset{10\times100}{D}$

$(A\times(B\times C))\times D \rightarrow$ 60,200 multiplications

$A\times((B\times C)\times D) \rightarrow$ 120,200 muls

$(A\times B)\times(C\times D) \rightarrow$ 7000 muls

Input: $\underset{m_0\times m_1}{A_1}, \underset{m_1\times m_2}{A_2}, \ldots, \underset{m_{n-1}\times m_n}{A_n} \Rightarrow$ minimum # of multiplications needed

1) 

$M(1,\ldots,n) = M(1,\ldots,t) + M(t+1,\ldots,n) + m_0 m_t m_n$

$M(i,j) :=$ minimum # of multiplications needed for matrices $A_i, \ldots, A_j$.

↳ not prefixes any more, can be any consecutive orders!

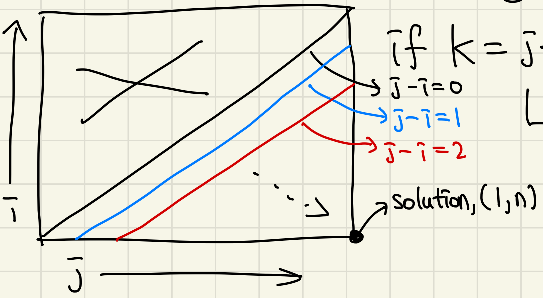
$M(1,n)$ → the final answer we want

2) $M(i,j) = \min\limits_{i\le k\le j} \{M(i,k)+M(k+1,j)+m_{i-1}m_k m_j\}$

↳ $(A_i \times \cdots \times A_k)\times(A_{k+1}\times\cdots\times A_j)$ configuration

Base Case: $\forall i\le n, M(i,i)=0$ (no need to multiply anything)

3) Observation: $M(i,j)$ is only valid when $j \ge i$



if $k = j-i$, $M(i,j)$ is dependent on $i,j<k$

↳ $M(i,j)$ only uses the "lines above"

Runtime: $O(n^2 \times n) = O(n^3)$

Common Subproblem Structures

1) input $x_1 \ldots x_n$ and subproblem is first i, $x_1 \ldots x_i$

2) input $x_1 \ldots x_n$ & $y_1 \ldots x_m$ → $x_1 \ldots x_i$ & $y_1 \ldots y_i$

3) input $x_1 \ldots x_n$ → $x_i \ldots x_j$ (in the middle)

Shortest Path in Graphs: edges with negative weights?

↳ DAG, or without negative cycles → leads to infinitely negative paths

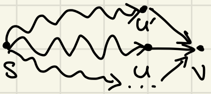
↱ not negative edges

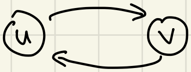
Dijkstra's → $O((n+m)\log n)$, Bellman-Ford → $O(nm)$,

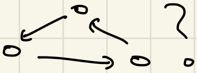
DAG-SSSP → $O(n+m)$ (DP problem)

Single Source Shortest Path (SSSP): $G(V,E)$, w_e, $s \rightarrow dist(v)$

1) $dist(v) :=$ shortest path from s to v for $v \in V$

2) $dist(v) := \min_{(u,v)\in E}\{dist(u) + w_{uv}\}$ 

3) 

?  ? how to resolve dependencies?

⇒ Need to redefine the subproblems

1) $dist(v,k) :=$ Shortest path $s \rightsquigarrow v$ with <u>at most k edges</u>

↳ Base Case: $dist(s,0) = 0$, $dist(v,0) = \infty$ for $v \in V/\{s\}$

2) Case I: Optimal path takes less than k edges

Case II: Optimal path needs exactly k edges → similar to previous trial!

↳ $\overset{\text{case I}}{dist(v,k-1)}$ vs $\overset{\text{case II}}{dist(u,k-1) + w_{uv}}$

$$\Rightarrow dist(v,k) := \min\{\underline{dist(v,k-1)}, \underline{\min_{(u,v)\in E}\{dist(u,k-1) + w_{uv}\}}\}$$

3) Nice ordering to compute $k = 1, 2, \ldots, (n-1)$ → max number of edges without cycles

Runtime: $O(n\cdot(\overset{\rightarrow\text{ setting first min}}{n} + \underbrace{m}_{\rightarrow\text{ setting second min}})) \simeq O(nm)$ (B-F)

→ Very similar to B-F, but B-F can terminate faster if ordering is good

(B-F can update multiple vertices correctly in the same loop)

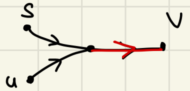
⇒ Insted, SSSP gives <u>all shortest path with at most k edges</u>!

SS Reliable Shortest Path: $G(V,E), W_e, S,$ bound $\underline{\underline{B}} \Rightarrow \min\{s \rightsquigarrow v\}$ with at most B edges

$\Rightarrow$ just refer to $dist(v,B)$ from SSSP!

All Pairs Shortest Paths: $G(V,E), w \rightarrow \forall u,v \in V$, minimum $dist(u,v)$

↳ Running B-F n times to get all paths? → $O(n^2 m)$ time

There are overlapping computation in B-F: 

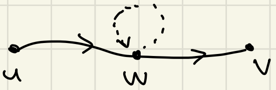
$dist(u,v,k)$:= shortest path $u \rightsquigarrow v$ with at most k edges ...?

↳ still gives $O(n^2 m)$ solution because no information about overlap

1) $dist(u,v,k)$:= shortest path $u \rightsquigarrow v$ that <u>takes vertices in $\{1,\dots,k\}$ only</u>

Base Case: $dist(u,v,0) = W_{uv}$ (no additional vertices visited)

Claim: on the shortest path $u \rightsquigarrow v$, no vertex occurs twice.

Proof:  cycle $w \rightsquigarrow w$ will only increase the path

2) Case I: doesn't need the k-th vertex for $dist(u,v,k)$

Case II: including the k-th vertex is the optimal

→k can be excluded!←

↳ $dist(u,v,k) := \min\{dist(u,v,k-1), dist(u,k,k-1) + dist(k,v,k-1)\}$

3) $d(u,v,k)$ depend on $d(\cdot,\cdot,k-1) \Rightarrow O(n^3)$ time

↳ $\forall i,j \in V$, $d(i,j,n)$ is the shortest path $i \rightsquigarrow j$

Traveling Salesman Problem: n cities, d_{ij} $(i \neq j)$ → minimum $\overset{1 \rightsquigarrow 1}{\text{spanning}}$ cycle

Brute Force: Enumerate all possible paths → $n! \simeq n^n$ paths

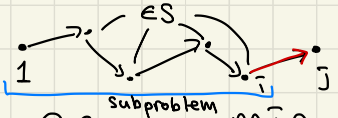
If $C(j) := $ cost of minimum path $1 \rightsquigarrow j$ → no information about path!

Simplification: TS can <u>end in any of the n cities</u>

$\rightarrow C(S,j) := S \subseteq \{1,\dots,n\}$ s.t. $j, 1 \in S$, least cost path that ...

① starts at node 1, ② visits all nodes in S, ③ ends in node j.
(exactly once)

$\hookrightarrow$ roughly $\overbrace{2^n}^{\text{sets}} \times \overbrace{n}^{j}$ subproblems (better than $n!$)



$i \in S \setminus \{1,j\}$ s.t. i is the last node before j.

$$\Rightarrow C(S,j) = \min_{i \in S \setminus \{1,j\}} \{C(S \setminus \{j\}, i) + d_{ij}\}$$

Base Case: $C(\{1\},1) = 0$, $C\{S,1\} = \infty$ for all $|S| \geq 2$,

$\forall j \neq 1$, $C(\{1,j\},j) = d_{ij}$ (most simple path $i \rightsquigarrow j$, just $i \to j$)

$$\Rightarrow C(S,j) = \min_{i \in S \setminus \{j\}} \{C(S \setminus \{j\}, i) + d_{ij}\}, \text{ when } |S| > 2.$$

$$C(S,j) = \cancelto{0}{C(\{1\},1)} + d_{ij} = d_{ij}, \text{ when } |S| = 2. \text{ (equivalent definition)}$$

Solving the actual TSP: $\min_{j \in S \setminus \{1\}} \{C(\{1,\dots,n\},j) + d_{j1}\}$ gives closure $1 \rightsquigarrow 1$.

$\hookrightarrow$ need to test $j = 2,3,\dots,n$ seperately $\to (n-1) \cdot O(2^n \cdot n) = \underline{\underline{O(2^n n^2)}}$ time

When coding, useful to pull out the $|S| = s$ loop to the outermost loop.

Independent Sets: for $G(V,E)$, $I \subseteq V$ s.t. $\forall u,v \in I, (u,v) \notin E$

goal is to find the largest independent set $I := \mathrm{Ind}(G)$.

↳ NP-hard, but tree problem is easier.

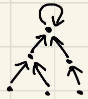
Tree Max Independent Set: Tree $G(V,E) \rightarrow \mathrm{Ind}(G)$.

1) $I(v) :=$ size of maximal independent set of subtree rooted at v.

2) $I(v) = \max\left\{ \sum_{u \in C(v)} I(u),\ 1 + \sum_{u \in G(v)} I(u) \right\}$, where $\begin{cases} C(v) := \text{children of } v \\ G(v) := \text{grand children of } v \end{cases}$

Base Case: $I(v) = 1$ if v is a leaf node ($\equiv$ v has no children)

3) Compute leaves to root. (need to dynamically build $C(v), G(v)$)

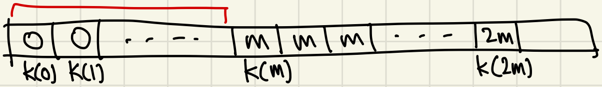


↳ Implementation as union-find like parent structure, then top sort.

↳ enforces DAG!

Runtime: linear w.r.t. vertices for all steps → $O(n)$ time

Knapsack Revisited: what if w_is are multiples of m?



→ ineffecient, bloated by m

↳ there are subproblems that don't need to be considered at all!

⇒ make a hash table for memoization of only relevant values

Coin Denomination Problem: $X_1, \dots, X_n; V$ → min # of coins if possible

ex) $x=(5,10)$, $V=15$ → $(5,10)$. $x=(5,10)$, $V=12$ → impossible

→ similar to knapsack, but enforces exact matching of value

1) $K(v) :=$ minimum # of coins needed to give change v (∞ if impossible)

2) $K(v) := \min_{i: x_i \le v} \{K(v - x_i) + 1\}$ → naturally set to ∞ if no solution exists.

Base Case: $K(0) = 0$ (no coins needed to match change of 0)

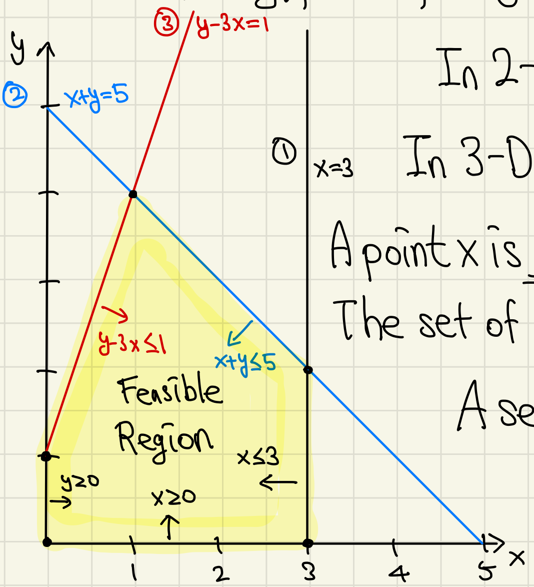
3) Iterate 1 to V. → Implementation can set ∞ if no subset exists.

Runtime: still $O(vn)$ time.

Linear Programming

Real number $\overset{(x,y)}{\text{variables}}$, Linear $\overset{(x+y \le 200, \dots)}{\text{constraints}}$ (degree 1 polynomial), Linear $\overset{\max(x+3y)}{\text{objective}}$

ex) $\max(x+2y)$, ① $x \le 3$, ② $x+y \le 5$, ③ $y - 3x \le 1$, $x, y \ge 0$



In 2-D, every constraint is a line.

In 3-D, every constraint is a plane, and so on.

A point x is feasible if it satisfies all constraints.

The set of all feasible points is a convex set.

A set $S \subseteq \mathbb{R}^n$ is convex if $\forall p, p' \in S$, the line connecting p and $p' \subseteq S$.

The optimum of a linear program can be achieved at a $\overset{\text{(vertex)}}{\text{corner}}$.

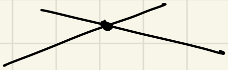
↳ Intuition: move the objective function until it touches only a tip

Simplex Algorithm: A Straightforward way to solve LP

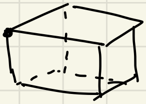
- Start at some vertex
- Keep moving to neighboring vertices to increase the objective

⇒ Why is this even an effective strategy?

In 2-D, a corner is an intersection of two lines.



In 3-D, a corner is an intersection of three planes.



In n-D, a corner is an intersection of n hyperplanes!

↳ Finding a corner from n constraints is just solving system of linear eqs.

⇒ m constraints in n dimensions → $\binom{m}{n}$ total corners ($\simeq \exp(n)$)

↳ not a good idea to perform linear search of all corners

⇒ Iterative improvement with Simplex is expected to be better

(Simplex could take exponential time, but is efficient in practice.)

"Ellipsoid Algorithm" & "Interior Point Methods" are provably linear.

Now, how do we find the "neighboring corners"?

↳ Swap one of the constraints (equation) to another one!

⇒ Also, we can prove the optimality of a corner by linearly manipulating constraints

Edge Cases: No feasible region (infeasible), Unbounded Optimum

Writing LP with matrices: $x_1 \dots x_n \in \mathbb{R}^n$

$$\begin{cases} a_{11}x_1 + \cdots + a_{1n}x_n \le b_1 \\ \quad \vdots \qquad\qquad \vdots \qquad \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n \le b_m \end{cases} \Rightarrow A\overset{[x_1 \dots x_n]^T}{\vec{x}} \le \overset{[b_1 \dots b_m]^T}{\vec{b}}$$

maximize $c_1x_1 + \cdots + c_nx_n \Rightarrow \overset{[c_1 \dots c_n]}{\vec{C}^T}\vec{x}$

LP: Duality

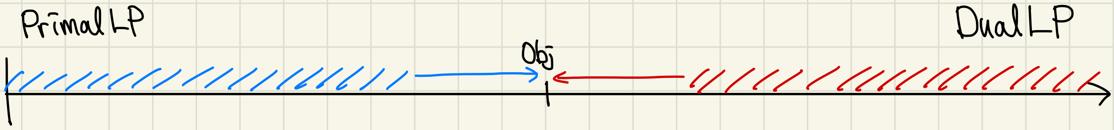
Primal LP (max) $\qquad\qquad$ Dual (min)

$$[A][\vec{x}] \le [\vec{b}] \quad \Longleftrightarrow \quad [A^T][\vec{y}] \ge [\vec{c}]$$

$$\text{MAX}([\vec{c}^T][\vec{x}]) \qquad\qquad \text{MIN}([\vec{b}^T][\vec{y}])$$

$$[\vec{x}] \ge \emptyset \qquad\qquad [\vec{y}] \ge \emptyset$$

→ Trivially, a dual of the dual is the primal.



Weak Duality: $\left(\text{Any solution to Primal LP}\right) \le \left(\text{Any solution to Dual LP}\right)$ (by definition)

Strong Duality: If Primal LP is bounded, OPT(Primal) = OPT(Dual)

↳ If Primal LP is unbounded, Dual LP is infeasible, & vice versa.

Zero-Sum Games: One player wins, then the other loses.

ex)

$$A = \begin{array}{c|c|c|c} & R & P & S \\ \hline R & 0 & -1 & 1 \\ \hline P & 1 & 0 & -1 \\ \hline S & -1 & 1 & 0 \end{array}$$

→ The row player gets A[r][c] & column player loses A[r][c] after each choosing r and c.

Value of the game := Payoff of row player assuming optimal strategy.

There are actually two versions of the game: who goes first?

↳ row player goes first: $\max_r(\overbrace{\min_c(A[r][c])})$

↳ column player goes first: $\min_c(\overbrace{\max_r(A[r][c])})$

considers the opponent's behaviour

first move

second move

⇒ The second player is always at an advantage ($\max_r \min_c A[r][c] \leq \min_c \max_r A[r][c]$)

Pure Strategy: Player deterministically picks a row or column

Mixed Strategy: Player picks a probability distribution over their choices

ex)

$$\begin{array}{c|c|c} & 1 & 2 \\ \hline 1 & 20 & -30 \\ \hline 2 & 10 & 40 \end{array}$$

Row: $Pr[r=1] = 1/4$, $P_r[r=2] = 3/4 \rightarrow (p_1, p_2)$

Column: $Pr[c=1] = 2/3$, $Pr[c=2] = 1/3 \rightarrow (q_1, q_2)$

(expected)

Payoff $= E[p,q] := \sum_{p_i \in p} \sum_{q_i \in q} p_i q_i A[r][c]$ where $p_i := Pr[r = r_i]$, $q_i := Pr[c = c_i]$

Value of game: $\text{Max}_p(\text{Min}_q(E[p,q]))$ or $\text{Min}_q(\text{Max}_p(E[p,q]))$

LP_A (row goes first)

(column goes first) LP_B

↳ We can write LP for each game, LP_A & LP_B.

↳ LP_A and LP_B will be duals of each other ⇒ Same optimum

→ Order of the game doesn't matter anymore!

LP_A) $\underset{\{p_1,p_2\}}{\mathrm{Max}}\left(\underset{\{q_1,q_2\}}{\mathrm{Min}}\left[20p_1q_1-30p_1q_2+10p_2q_1+40p_2q_2\right]\right]$

where $p_1+p_2=1$ and $q_1+q_2=1$.

Observation: The second player actually doesn't need to use a mixed strategy! (q_1 and q_2 are binary complements)

ex) $p_1=0.5,\ p_2=0.5 \rightarrow E[p,q]=\alpha q_1+\beta q_2$ where $\begin{cases}\alpha=15\\ \beta=-5\end{cases}$

↳ $E[q]$ becomes a linear combination of q → just maximize one!

↳ In other words, there will always be one best strategy given p

$$\rightarrow \underset{\{p_1,p_2\}}{\mathrm{Max}}\left[\mathrm{Min}\begin{Bmatrix}(q_1=0,q_2=1)\rightarrow -30p_1+40p_2\\ (q_1=1,q_2=0)\rightarrow 20p_1+10p_2\end{Bmatrix}\right]$$

⇒ Formulate into an $LP_A := \max(z)$ where

$$\begin{cases}z\le -30p_1+40p_2, & p_1+p_2=1\\ z\le 20p_1+10p_2, & p_1,p_2\ge 0.\end{cases}$$

↳ optimal (p_1,p_2) will give the optimal strategy.

LP_B) $\underset{\{q_1,q_2\}}{\mathrm{Min}}\left(\underset{\{p_1,p_2\}}{\mathrm{Max}}\left[20p_1q_1-30p_1q_2+10p_2q_1+40p_2q_2\right]\right]$

where $p_1+p_2=1$ and $q_1+q_2=1$.

$$\rightarrow \min_{\{q_1,q_2\}} \left[\max \left\{ \begin{array}{l} (p_1=0,p_2=1) \rightarrow 10q_1+40q_2 \\ (p_1=1,p_2=0) \rightarrow 20q_1-30q_2 \end{array} \right\} \right]$$

$\Rightarrow LP_B := \min(z)$ where

$$\begin{cases} z \geq 10q_1+40q_2 & q_1+q_2=1 \\ z \geq 20q_1-30q_2 & q_1,q_2 \geq 0. \end{cases}$$

↳ optimal (q_1,q_2) will give the optimal strategy.

Observation: LP_A and LP_B are duals of each other!

⇒ By strong duality, $OPT(LP_A) = OPT(LP_B)$.

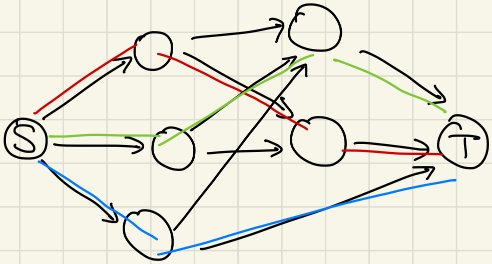
⇒ For zero-sum games, the order of play is interchangable.

Maximum Flow

Setup: 1) Directed Graph $G(V,E)$

2) Capacities $C_e \; \forall e \in E$

3) Source S & Sink T



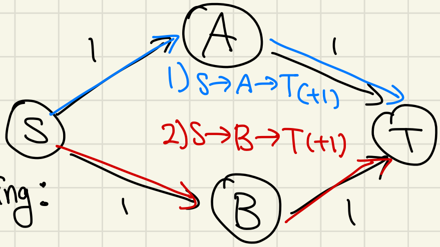
⇒ What is the maximum rate of flow from S to T?

S-t-flow := assignment $f = E \rightarrow \mathbb{R}^+$ such that:

1) For each edge e, flow on $f_e \leq C_e$. (capacity constraint)

2) For all vertices v, $\sum_{u\to v} f_{u\to v} = \sum_{v\to w} f_{v\to w}$ (conservation constraint)

⇒ Max s-t-flow := $\text{Max}\left(\sum_{s\to u} f_{s\to u}\right)$

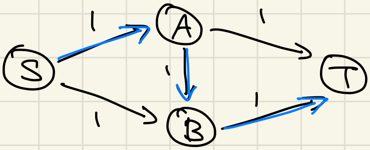


Algorithm Formulation. Repeat the following:

1) Find an s-t path P that has leftover capacity

2) Add the flow along P to the current flow

→ This algorithm fails. Consider the following graph:



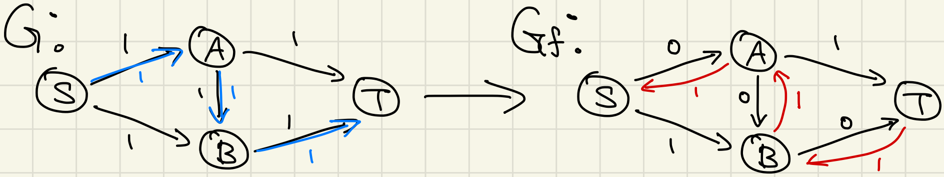
1) S→A→B→T (+1)

2) S→B→T doesn't work because B→T is already saturated by the first step.
→ Terminate, flow=1.

↳ We could have chosen $\underbrace{\text{1) } S\to A\to T\ (+1) \text{ and } \text{2) } S\to B\to T\ (+1)}_{\hookrightarrow \text{flow}=2}$!

⇒ We need a way to "backtrack" our mistakes

Residual Graph G_f: measures what capacities are left in graph



The new edge B→A is "reversing" the flow of A→B

(We have unlocked "the ability" to send one unit back from B to A)

⇒ ∀e∈E, $u \xrightarrow{C_e} v$, G_f will have $u \underset{f_e}{\overset{C_e - f_e}{\rightleftarrows}} v$. ($C_{uv} + C_{vu} = C_e$)

Execution: Find P on G, compute G_{fP}. $G \leftarrow G_{fP}$. Repeat.

Optimality Argument: $\exists$ cut (L,R) s.t. $S \in L, T \in R$, where the flow $L \to R$ is at most the optimal flow! an s-t cut

The capacity of cut: $\text{Capacity}(L,R) = \sum_{u \to v} \{C_{uv} \mid u \in L, v \in R\}$

"weak duality"

Claim: In any graph, every s-t flow $\leq$ capacity of every s-t cut

"strong duality"

Theorem: In any graph, maximum s-t flow = capacity of s-t min-cut

Proof: 1) Execute the algorithm. At termination, there is no more s-t path in the residual graph G_f.

2) Consider $L = \{$set of vertices reachable by a path from s in $G_f\}$. Then, $R = V \setminus L$. This (L,R) is a cut.

3) $\nexists$ no edge from L to R in G_f (if reachable, it would be in L.)

$\Leftrightarrow$ Every edge from L to R in G is saturated ($C_{uv} = C_e$).

$\Leftrightarrow \forall$ edge e from L to R, $f_e = C_e$. $\Rightarrow$ Total Flow $= \sum_e C_e$.

Conclusions: 1) At termination, $\exists$ cut with value = flow assigned since all flows $\leq$ all cut capacity. they are only equal when min-maxxed!

$\Rightarrow$ At termination, current flow = max flow.

2) (Corollary) In a network $G(V,E)$, if all capacities are integers, $\exists$ a max flow assignment which is also integral!

* In general, LP solutions need not be integral!

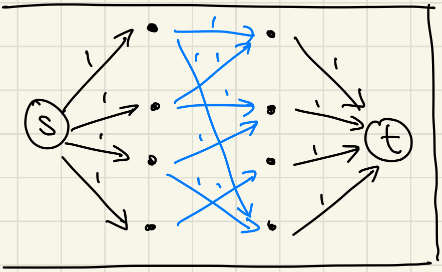
Perfect Matching: $G(U \cup V, E)$ where $|U| = |V| = n$.

Is there a perfect matching between U and V (1-to-1 matching)?

Matching := A set of disjoint pairs, perfect matching: all vertices are matched.

* Perfect Matching reduces to finding max flow!

↳ add source & sink to only flow to U and V, respectively.



Now, compute the max s-t flow where all edge capacities are 1. Then, Max Flow $= n$ iff $\exists$ a perfect matching!

Assignment Problems: 1) n schools with capacity $c_1 \dots c_n$.

2) m children with set of schools they can be assigned to

↳ $G(U \cup V, E) := (i,j) \in E$ if child i can go to school j ($i \in U$, $j \in V$)

⇒ Turn it into a max-flow s.t. ⓢ → {kids} → {schools} → ⓣ and each school has capacity c_i for the edge to t.

(Out of Scope) Solving LP via Gradient Descent

Optimization vs Feasibility

↳ Maximize $c^T x$ subject to $Ax \leq b$

↳ Find x satisfying $Px \leq q$ (no objective function)

Theorem: An algorithm for Feasibility of LPs

⇒ An algorithm for Optimization of LPs

Proof: Given an optimization problem (A, b, c), convert the objective function to an additional constraint $c^T x \geq n$.

The value of n can be bounded tightly via binary search, given an algorithm to solve for its feasibility!

⇒ We can focus on solving feasibility of LPs.

ε-separating line: any line ℓ s.t. p^* is on one side and p is on the other side and is at least ε-away from ℓ.

Point Pursuit Game: Alice is at point p^*, Bob is at point $p^{(0)}$. Alice is giving directions to Bob to reach her.

At round t: Bob is at point $p^{(t)}$. Alice tells Bob her seperating line between p^* and $p^{(t)}$.

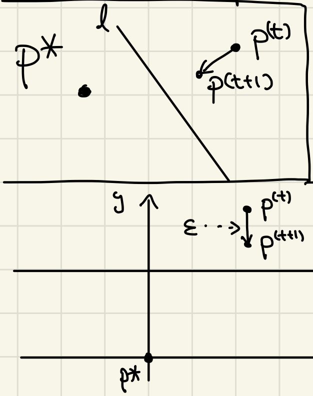
Bob updates his location $p^{(t)} \to p^{(t+1)}$.

Bob's strategy: Move ε-distance directly towards the seperating line. Repeat with the new line.

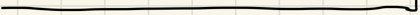
↳ Formally, if line given is $ax+by=c$, Bob moves ε along the direction $\perp$ to the line $\Rightarrow p^{(t+1)} = p^{(t)} + \varepsilon \cdot \vec{v}$ where $\vec{v} = (-b, a)$.

Claim: In each iteration, the square distance between Alice and Bob decreases by at least ε^2.

↳ $\text{dist}(p^{(t+1)}, p^*) \leq \text{dist}(p^{(t)}, p^*) - \varepsilon^2$.



By rotation and translation, move p^* to origin and make l perpendicular to the x-axis.



Let $p^{(t)} := (x,y)$. Then, $p^{(t+1)} = (x, y-\varepsilon)$.

$\text{dist}(p^{(t)}, p^*) = x^2+y^2$. $\text{dist}(p^{(t+1)}, p^*) = x^2+y^2-2y\varepsilon+\varepsilon^2$.

difference $= 2y\varepsilon - \varepsilon^2$. Observe that $y > \varepsilon$.

Then, $\underline{\text{difference} \geq 2\varepsilon^2 - \varepsilon^2 = \varepsilon^2}$. Thus, $p^{(t+1)}$ will be at least ε^2 closer to p^* in squared distance.

Outcome: If distance $\text{dist}(p^*, p^{(0)}) \le D$, then the game terminates in $O(D^2/\varepsilon^2)$ steps. This is irrespective of Alice's strategy in choosing lines.

LP Feasibility: Set of linear constraints $Ax \le b \Rightarrow$ Find x satisfying all conditions OR report failure.

A weaker goal: Find x that is ε-close to satisfying all constraints

Main Point: Violated constraint $\Leftrightarrow$ seperating line!

↳ point p vidates some constraint $\ell \Leftrightarrow \ell$ is a seperating line between p and some feasible point p^*.

Algorithm for LP feasibility:

```
-set $p^{(0)} \leftarrow (0,0)$.
- for $t = 0 \ldots T$:
                                              the weaker ε-close constraints
    - check if $p^t$ satisfies all constraints. If yes, return $p^{(t)}$.
    - Let $\ell$ be a violated constraint. Move $p^{(t)}$ directly towards $\ell$
      to produce $p^{(t+1)}$.
                                                          implied from result
                                                           of Alice-Bob game
- after T iterations, return "no feasible solution within distance $\varepsilon\sqrt{T}$."
```

ε-seperation oracle: a subroutine for LP that returns one violated ε-constraint for any point, if it exists. If not, returns "satisfied".

↳ The first step of the feasibility algorithm can be replaced with this.

Fair Work Allocation: n workers, $\forall$ worker i, $\begin{cases} l_i := \text{minimum work} \\ u_i := \text{maximum work} \end{cases}$, total work W, then assign work to workers satisfying constraints.

LP: $x_i :=$ work assigned to ith worker, $\sum x_i \geq W$, $l_i \leq x_i \leq u_i$.

Fairness: No set of $n/4$ workers do more than $W/2$ work.

↳ $\forall S \subseteq [n] \mid |S| = n/4$, $\sum_{i \in S} x_i \leq W/2$. $\rightarrow \binom{n}{n/4} \propto \exp(n)$ constraints!

Seperation oracle: sort $x_1 \ldots x_n$. pick $S \leftarrow \{\text{largest } n/4 \text{ values of } [n]\}$.

check if $\sum_{i \in S} x_i > W/2$. $\Rightarrow$ ε-LP solver is implementable!

ε-seperation is powerful enough to solve <u>infinitely many constraints</u> given an efficient ε-seperation oracle!

ex) find a point on an overlapping region of circles $C_1 \ldots C_n$.

↳ if $p \notin C_i$, a tangent to C_i gives a seperating line.

Sets defined by (in)finitely many linear constraints $\Leftrightarrow$ Convex sets!

Search Problems, P & NP

"Can we always find efficient algorithm for any optimization task?"

SAT: formula $\phi(x_1,\dots,x_n) \Rightarrow$ satisfying assignment or report None.

↳ Brute force (trying all assignments) takes $O(2^n)$ time

↳ still has an efficient VERIFICATION algorithm for a solution!

$\Rightarrow$ Verify$(\phi, (x_1,\dots,x_n)) \rightarrow$ output $\phi(x_1,\dots,x_n)$.

Search Problem: A problem that has an algorithm VERIFY such that a proposed solution S can be checked in poly. time w.r.t. the instance I. → VERIFY(I,S) := True / False

Class P: search problems we can find a solution in poly. time.

Class NP: all search problems (we can verify a solution in poly. time.)

↳ $P \subseteq NP$!

Lemma) Graph 3-Coloring $\in$ NP.

Proof: VERIFY$(G(V,E), c: V \rightarrow \{R,G,B\}) :=$ output 1 if $\forall (u,v) \in E$, $c(u) \neq c(v)$ and $c(v) \in \{R,G,B\}$. Else, output 0.

Vertex Cover: $G(V,E)$, bound $b \rightarrow A \subseteq V$ s.t. $|A| \leq b$ s.t. $\forall (u,v) \in E$, $u \in A$ OR $v \in A$, or report None.

Lemma) $VC \in NP$.

Proof: VERIFY$((G(V,E), b), A) :=$ output $\emptyset$ if $|A| > b$ or $\exists (u,v) \in E$ s.t. $u \notin A$ AND $v \notin A$. Else, output 1.

Factoring: $N = pq$ (p, q are large primes) $\Rightarrow p, q$

Lemma) Factoring $\in NP$.

Proof) VERIFY$(N, (p,q)) :=$ output 1 if $N = p \cdot q$, $\emptyset$ otherwise.

Lemma) TSP with bound $b \in NP$.

Proof: VERIFY$((n, d_{ij}, b), \tau : \{1 \ldots n\} \rightarrow \{1 \ldots n\}) :=$ output 1 if $d_{\tau(1)\tau(2)} + \cdots + d_{\tau(n)\tau(1)} \leq b$ AND $\forall i,j \in \{1, \ldots, n\}, \tau(i) \neq \tau(j) \mid i \neq j$.

Rudrata/Hamiltonian Cycle: $G(V,E) \Rightarrow \tau : \{1, \ldots, n\} \rightarrow V$ s.t. $(\tau(1), \tau(2)), \ldots, (\tau(n), \tau(1)) \in E$.

Lemma) RC/HC $\in NP$.

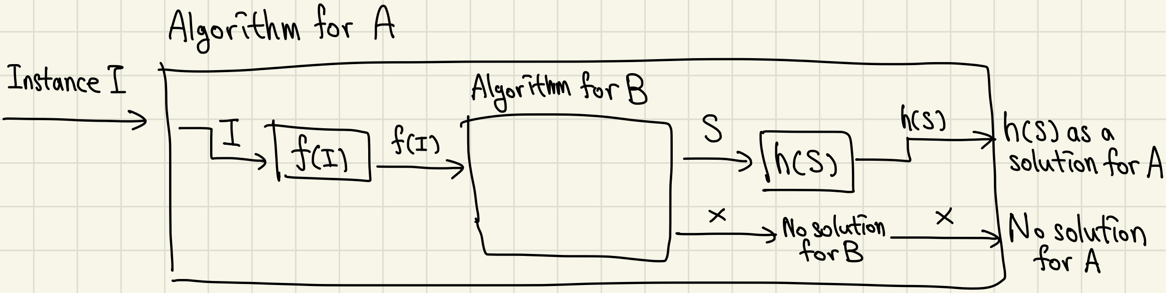
Proof: VERIFY$(G(V,E), \tau : \{1 \ldots n\} \rightarrow V\} :=$ output 1 if $\forall i,j, \tau(i) \neq \tau(j)$ AND $(\tau(1), \tau(2)), \ldots, (\tau(n), \tau(1)) \in E$. output $\emptyset$ otherwise.

Reductions

A "reduces to" B (⟶), if A can be implemented in B in poly. time.

↳ an algorithm for B yields an algorithm for A!

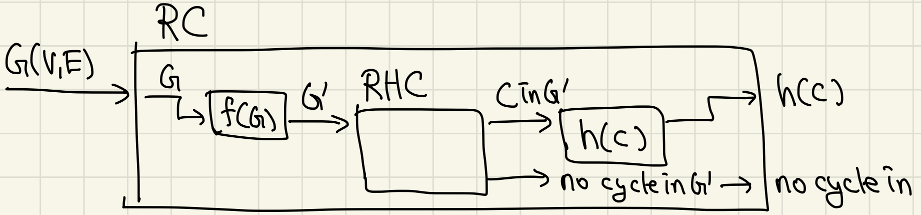
⇒ B is <u>at least as hard as A!</u> ($A \leq B$).

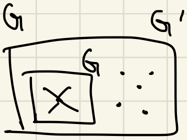


Reduction needs to specify functions (f, h) where $f, h \in P$, and if B outputs S as a solution to $f(I)$, then $h(s)$ is a solution to I. Also, if B outputs None, then no solution exists for I as well.

⮤ if I has a solution, then $f(I)$ also has a solution. (easier to prove!)

ex) Rudrata Cycle → Rudrata Half Cycle (need to visit $|V|/2$ vertices)





$f: G \rightarrow G' := E' = E,\ V' = V \cup (n+1, \dots, 2n)\ (n = |V|)$

↳ adds n extra vertices not connected to any other vertices.

Lemma1) f & $h \in P$. Proof: Trivial. //

Lemma2) If C is a RHC in G', then $h(C)=C$ is also RC in G.

Proof: C does not contain vertices $(n+1), \ldots, 2n$. Also, $|C|=n$ since $|V'|=2n$. $\Rightarrow$ C contains all vertices $1, \ldots, n$ and is a RC. //

Lemma3) If G has a RC, then G' has a RHC.

Proof: Let C be the RC in G. Then, C is also the RHC in G'. //

$\Rightarrow$ RC $\rightarrow$ RHC. //

ex) SAT $\rightarrow$ 3-SAT (each clause has at most 3 variables).

Reduction argument: If a clause in SAT has more than 3 variables, $(a_1 \vee a_2 \vee \ldots \vee a_k)$, introduce variables $y_1, \ldots, y_{k-3}$. Then, split up the clause to $(a_1 \vee a_2 \vee \overbrace{y_1) \wedge (\overline{y}_1}{} \vee a_3 \vee \overbrace{y_2) \wedge}{} \ldots \overbrace{\wedge (\overline{y}_{k-3}}{} \vee a_{k-1} \vee a_k)$.

Call this procedure for any ϕ, f. We also need $h(S)$ to recover a solution to ϕ from S. $h(S)$ just drops all y variables.

Lemma1) $f, h \in P$. Proof: Trivial.

Lemma2) If $\omega := f(\phi)$ has a satisfying assignment, then $h(S)$ satisfies ϕ.

Proof: $\exists i$ s.t. $a_i = T$. then, $(a_1 \vee \ldots \vee a_n) = $ True.

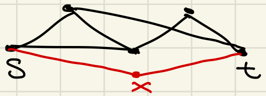
Lemma3) If ϕ has a satisfying assignment, ω also has one.

Proof: Let some $a_i = T$. construct $y_1, \dots, y_{i-2}$ tobe True and the rest of y variables to False.//

Composition of Reduction: If $A \to B$ & $B \to C$, then $A \to C$.

Proof: $f_{AC}(I) = f_{BC}(f_{AB}(I))$, $h_{CA}(S) = h_{BA}(h_{CB}(S))$.

ex) (s,t)-Rudrata Path → Rudrata Cycle



$f(G,s,t) \to G'(V',E')$. $V' := V \cup \{x\}$, $E' = E \cup \{(x,s),(x,t)\}$.

$h(\overset{\text{RC in } G'}{C}) = C \setminus \{(x,s),(x,t)\}$.

1) Runtime of f and h are polynomial. Trivial.//

2) If S is a RC in G', then h(S) is an (s,t)-RP in G.

3) If G has an (s,t)-RP in G, then G' has a RC.

by construction

Circuit SAT : A Boolean Circuit C (DAG with 5 kinds of gates)

1) AND & OR gates w/ indegree 2 2) NOT gate w/ indegree 1

3) known input gates 4) unknown input gates

→ assignment to unknown input gates s.t. output gate evaluates to TRUE

Core Argument: Circuit SAT → SAT

$f(C) \to \forall$ gate in circuit C, we will introduce a variable g.

true gate $\to (g)$. false gate $\to (\bar{g})$. or gate $\to \begin{Bmatrix} h_1 \Rightarrow g_1 & g_1 \Rightarrow h_1 \vee h_2 \\ h_2 \Rightarrow g_1 & \end{Bmatrix} = \begin{pmatrix} g \vee \overline{h_1} & h_1 \vee h_2 \vee \bar{g} \\ g \vee \overline{h_2} & \end{pmatrix}$

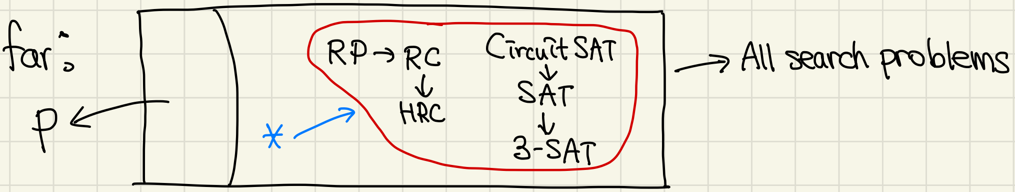
and gate $\to \begin{Bmatrix} g \Rightarrow h_1 & h_1 \wedge h_2 \Rightarrow g \\ g \Rightarrow h_2 & \end{Bmatrix} = \begin{pmatrix} h_1 \vee \bar{g} & g \vee \bar{h_1} \vee \bar{h_2} \\ h_2 \vee \bar{g} & \end{pmatrix}$. Output gate $\to (g)$.

1) poly time (trivial)

2) $h(S) = S|_{\text{unknown input gates}}$

3) given a solution for C, we can satisfy the SAT clauses.

So far:



NP-Completeness: All other search problem reduces to it.

Lemma: $\forall A \in NP$, $A \to$ Circuit SAT

Proof: $\text{VERIFY}_A(I_A, S_A) \to \{0,1\}$. (poly time in $|I_A|$).

$\hookrightarrow C_{\text{VERIFY}_A, I_A}(w) = \text{VERIFY}_A(I_A, w)$. $\Rightarrow$ $f(I_A) = C_{\text{VERIFY}_A, I_A}$.

1) $f \& h \in P$ (unrolling VERIFY_A & I_A is poly time, h is identity)

2) S is a solution to Circuit SAT, then S is asolution to A

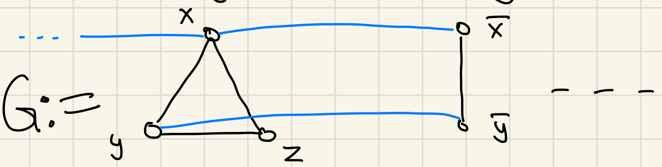
3) If S has a solution, then so does C_{VERIFY_A, I_A}.

ex) 3-SAT ⟶ Independent Set $(G(V,E), g \Rightarrow S \subseteq V \text{ s.t. } |S| = g, \overset{\forall u,v \in S,}{} (u,v) \notin E)$

WLOG, each clause in $\emptyset$ has more than one variable. $\underset{\hookrightarrow \text{True}}{(x)}$ $\underset{\hookrightarrow \text{False}}{(\bar{y})}$

$\emptyset := (x \vee y \vee z) \wedge (\bar{x} \vee \bar{y})$... For each variable, introduce a node.

$G :=$



connect all variables with its negation.

Let $g = \#$ of clauses in $\emptyset$, $G(V,E) =$ the graph induced by $\emptyset$.

1) Transformation is bounded by # of clauses & variables. //

2) IS in G of size g, then we can construct a satisfying assignment for $\emptyset$.

↳ Picks exactly one literal in each clause to be TRUE.

3) If $\emptyset$ has a satisfying assignment ⇒ an IS in G of size g

⇒ Independent Set is also NP-Complete!

ex) Independent Set ⟶ Vertex Cover $(G(V,E), b \Rightarrow S \subseteq V, |S| = b \text{ s.t. } \forall (u,v) \in E, (u \in S) \vee (v \in S))$

$f(G,g) = G, \overset{b=}{|V|} - g$ (the complementary vertices of IS is a vertex cover!)

↳ S is an IS, then $\forall u,v \in S, (u,v) \in E$. Then, $\forall e \in E$, $u \in V \setminus S$ or $v \in V \setminus S$.

$h(S) = V \setminus S$. ⇒ Vertex Cover is also NP-Complete!

ex) Independent Set → $\overset{\text{finding a complete graph of size } g}{\text{Clique}}$ $(G(V,E), g \Rightarrow S \subseteq V, |S| = g \text{ s.t. } \forall u,v \in S, u \neq v, (u,v) \in E.)$

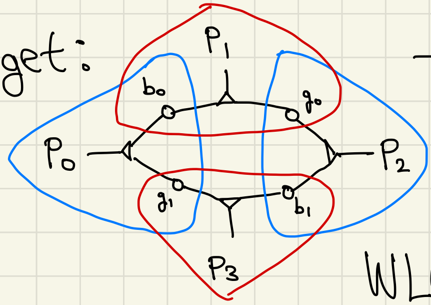
$f(G(V,E), g) = (G'(V,E'), g)$ s.t. $E' = (V \times V) \setminus E$ (the "not friends" edges)
complement set of edges

3D Matching: n boys, girls, and pets, preference triplets $\{(b,g,p)\}$

→ n-disjoint triplets (NP-Complete)

ex) 3SAT ⟶ 3D Matching (need to introduce a gadget)

Gadget:



→ $\overset{\text{(False)}}{p_0 \,\&\, p_2}$ free, or $\overset{\text{(True)}}{p_1 \,\&\, p_3}$ free

$(b_0,g_0,p_1),(b_1,g_1,p_3)$ $(b_0,g_1,p_0),(b_1,g_0,p_2)$

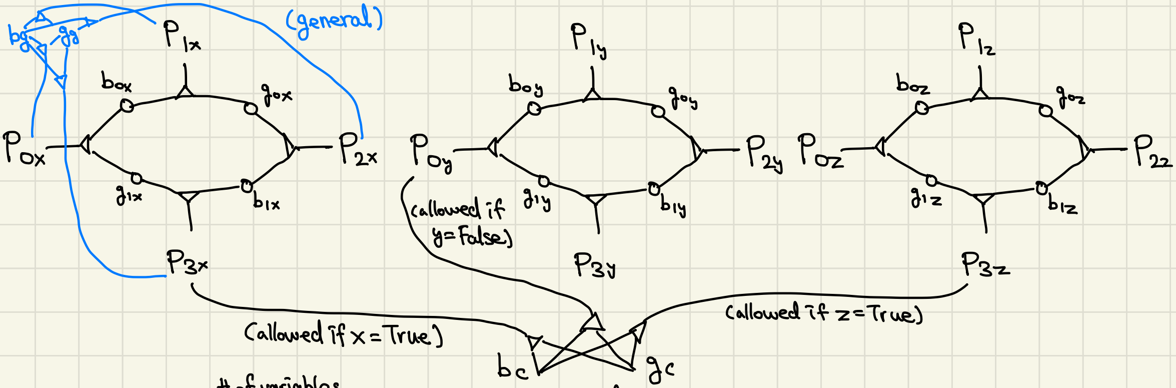
↳ this can act like an on/off switch!

WLOG, $\phi = (x \cdots)(\cdots x \cdots)(\cdots\cdots \bar{x}) \cdots\cdots$.

→ we want to restrict each x and $\bar{x}$ to appear at most 2 times.

↳ change all x to x_i, and add clause $(\bar{x}_1 \vee x_2)(\bar{x}_2 \vee x_3) \cdots (\bar{x}_k \vee x_1)$

to ensure all x_i are of the same assignment. If $c = (x \vee \bar{y} \vee z)$,

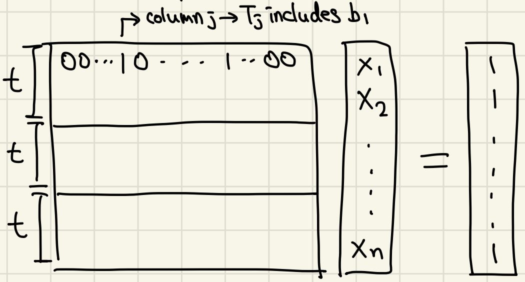


→ currently $4\overset{\substack{\text{\# of variables} \\ \uparrow}}{n}$ pets and $(2n+\overset{\substack{\text{\# of clauses} \\ \uparrow}}{m})$ girls & boys → introduce $(2n-m)$ "general" boys & girls that can be paired with any pet in a gadget.

Zero-One Equations (ZOE): $A\in\{0,1\}^{m\times n}\rightarrow\vec{x}\in\{0,1\}^{n}$ s.t. $A\vec{x}=1$.

ex) $\underset{n \text{ preferences} \rightarrow t \text{ triplets}}{\text{3DMatching}} \longrightarrow$ ZOE

$T_1, T_2, \ldots, T_n$ assigned to $x_1, x_2, \ldots, x_n$ where $x_i=\emptyset$ if T_i is not a part of the solution, and $x_i=1$ if it is.



→ in the first row, set it to 1 only for the column for T_i including b_i.

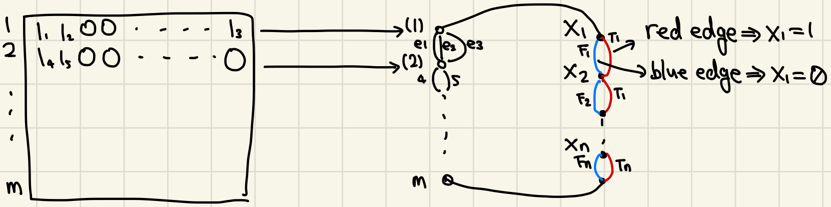
→ continue for all boys, then girls & pets.

⇒ This enforces that all boys, girls, and pets must be selected once!

ex) ZOE ⟶ RC ((1) ZOE → RC w/ paired edges (2) RC w/ paired edges → RC)

RC w/ paired edges: $G(V,E), C\subseteq(E\times E)\rightarrow \underset{S\subseteq E}{RC}$ s.t. $\forall(u,v)\in C$, $XOR\left(\substack{u\in S,\\ v\in S}\right)$

(1) ZOE → RC w/ paired edges

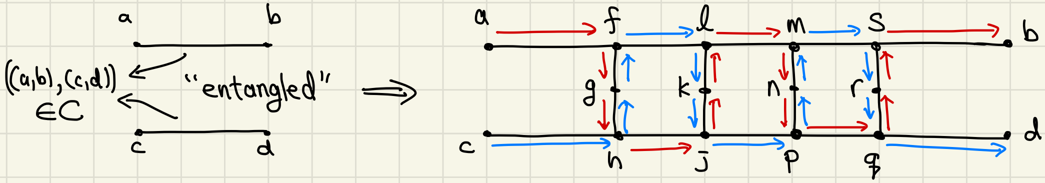


$1\cdot x_1+1\cdot x_2+1\cdot x_n=1 \rightarrow (e_1,f_1),(e_2,f_2),(e_3,f_n)\in C$, and so forth.

→ RC also constrains to choose between (t_i,f_i) and (e_1,e_2,e_3)....

⇒ each row multiplied by $\vec{x}$ will have to add up to 1 iff ∃RC!

(2) RC w/ paired edges → RC (idea: reduce size of C by 1)



⇒ this gadget implies the entanglement without an explicit constraint!

(trying to exit to the wrong side (a→c), (b→d) will not work)

→ do this for all constraints in C ⇒ RC without paired edges constraints!

ex) RC ⟶ TSP (d_{ij} & $B \to T: \{1 \dots n\} \to \{1 \dots n\}$ s.t. $d_{T(1)T(2)} \cdots \leq B$)

$G \longrightarrow d_{ij} = 1$ if $(i,j) \in E$, 2 if $(i,j) \notin E$. $B = |V|$.

⇒ the TSP will find exactly a RC of G!

ex) ZOE ⟶ Subset Sum ($[a_1 \dots a_n], w \to S \subseteq [n]$ s.t. $\sum_{i \in S} a_i = w$)

$$A \begin{bmatrix} \underset{a_1}{\boxed{}} & \cdots & \underset{a_n}{\boxed{}} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \underset{w}{\begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}}$$

$a_i := \sum_j A_{ij}(n+1)^j$, $w = \sum (n+1)^j$

(base is $(n+1)$ because of carry-over)

Coping with NP

1) "Intelligent" Exponential Search → usually efficient

2) Approximation Algorithm → poly time, suboptimal but bounded w.r.t. optimal

3) Heuristics → no guarantees on runtime nor optimality

Intelligent Exponential Search

Backtracking: consider SAT with instance $\emptyset = (w \vee x \vee y \vee z) \wedge (w \vee \bar{z}) \ldots$

By setting $w = \emptyset$ or 1, we can reduce the formula to a smaller one or realize that it is unsatisfiable. Whenever some subtree is unsatisfiable, it will keep being unsatisfiable, so stop searching there.

Branch & Bound: Generalization of backtracking to optimization

Consider TSP with instance d_{ij}, $\min\{d_{T(1)T(2)} + \cdots + d_{T(n)T(1)}\}$.

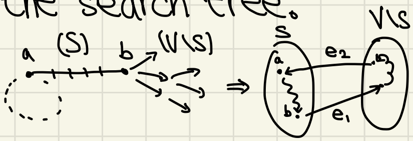
A naive tree expansion has $O(n!)$ nodes. Now, whenever we try to expand a partial solution (node), compare to the best solution so far. If every results from the partial solution is worse than the best solution so far, prune that subtree.

Claim: $W_{TSP} \geq W_{MST}$.

Proof: If W_{TSP} is an optimal solution, removing one edge T results in a spanning tree W_T. Also, $W_{TSP} \geq W_T$ and $W_T \geq W_{MST}$, so $W_{TSP} \geq W_{MST}$. // ⇒ generalize to all possible states!

$[\overset{\overset{\text{start}}{\uparrow}}{a}, \overset{\overset{\overset{\in}{\text{path}}}{\uparrow}}{S}, \overset{\overset{\text{end}}{\uparrow}}{b}]$ can represent any state of the search tree.

→ The starting state is $[a, \{a\}, a]$.



→ $W_{b \to a} \geq W_{e_1} + W_{e_2} + W_{MST}^{(V \setminus S)}$. If this bound is worse than W_{best}, discard.

Approximation Algorithms

For an instance I of a minimization problem, an algorithm A is an α-factor approximate algorithm if $\alpha = \max_I \frac{OPT(I)}{A(I)}$. For maximization problems, $\alpha = \max_I \frac{A(I)}{OPT(I)}$.

Set Cover: Set of elements B, subsets $S_1, S_2, \ldots, S_n \subseteq B$

→ smallest subset of S_i s.t. their union is B.

A greedy algorithm that picks the set S_i with the most uncovered elements at any iteration.

Claim: Let $|B| = n$, $OPT(I) = k$. Then, the greedy algorithm uses at most $k \ln(n)$ sets.

Proof: Let n_t be the # of uncovered elements left after t iterations.

$\overset{(n)}{n_0} \longrightarrow n_1 \longrightarrow n_2 \cdots \longrightarrow n_t$. The optimal solution will have exactly k iteration. We claim that at least one of the sets not selected by the optimal solution has $\frac{n_t}{k}$ uncovered elements. If that is not the case, $< \frac{n_t}{k} \times k = n_t$, a contradiction. Then, the greedy algorithm will have to pick a set of at least $\frac{n_t}{k}$ uncovered elements. $\Rightarrow n_{t+1} \leq n_t - \frac{n_t}{k} = n_t(1-\frac{1}{k})$

$\Rightarrow n_t \leq n(1-\frac{1}{k})^t < ne^{-t/k}$. If $ne^{-t/k} < 1$, $t < k\ln(n)$.

Vertex Cover: $G(V,E) \rightarrow S \subseteq V$ s.t. $|S|$ is minimized & S touches all edges.

$\rightarrow B = \{e_1, \dots, e_m\}$, $S_u = \{e \mid$ one of vertices in e is u & $e \in E\}$.

Proposed Solution: Find a maximal matching $M \subseteq E$, then return all endpoints of edges in M.

(i) Size of any VC $\geq |M|$ (at least one vertex per edge)

(ii) $|S| = 2|M|$ (two vertices per edge)

(iii) S is a VC (if not, $\exists$ edge e_{uv} s.t. $(u \notin S) \cap (v \notin S)$, which means that M is not fully constructed yet.)

$\Rightarrow |S| = 2|M| \leq 2(VC) \Rightarrow 2(OPT\ VC) \geq |S|$, and S is a VC.

Clustering: Points $\{x_1,\dots,x_n\}$, dist$(\cdot,\cdot)$, integer k

Assumptions about dist function: ① $d(x,y)\ge 0$, ② $d(x,y)=0$ iff $x=y$

③ $d(x,y)=d(y,x)$, ④ $d(x,i)+d(i,y)\ge d(x,y)$ (Triangle inequality)

→ $\overset{(<n)}{k}$ clusters $C_1,\dots,C_k$ s.t. $\max_j\left\{\overbrace{\max_{x,y\in C_j}\{dist(x,y)\}}^{\text{"diameter" of } C_j}\right\}$ is minimized.

The Algorithm: pick $\mu_1\in X$ as the first cluster center.

for $i=2\dots k$: Let $\mu_i\in X$ be the point $\underbrace{\text{farthest}}$ from $\mu_1\dots\mu_{i-1}$.
→ minimum is largest

create k clusters: $C_i=\{$all $x\in X$ closest to $\mu_i\}$

↳ Let μ_{k+1} be the next point about to be picked if we were to continue,

and let r be the distance from $\{\mu_1\dots\mu_k\}$ to μ_{k+1}, i.e. $\min_j\{dist(\mu_i,\mu_{k+1})\}$.

1) $\forall x\in C_i$, $d(x,\mu_i)\le r$, since μ_{k+1} is the farthest point from all μ_i.

2) $\forall i,j\in[k+1]$, $d(\mu_i,\mu_j)\ge r$, since μ_i is always greedily selected.

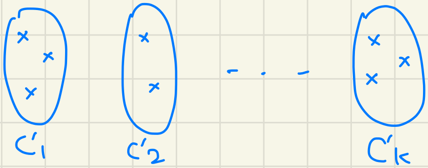
↳ in fact, each iteration will pick a point closer to the cluster than prev.

Lemma: $\forall i, \forall x,y\in C_i$, $d_A\le \underline{d(x,y)\le 2r}$. ---- (i)

Proof: $d(x,y)\le d(x,\mu_i)+d(\mu_i,y)$ (by Triangle Inequality)

since $d(x,\mu_i)\le r$ and $\overbrace{d(\mu_i,y)\le r}^{\text{(by obs.1)}}$, $d(x,y)\le r+r=2r$. //

OPT:



$X=\{x_1,\dots,x_n\}$

↳ $\{\mu_1,\dots,\mu_{k+1}\}$ where?

Claim: $\exists t \in [k], i,j \in [k+1], \mu_i \in C'_t$ and $\mu_j \in C'_t$ (by Pigeonhole Principle).

→ the diameter of $C'_t \geq \overbrace{d(\mu_i, \mu_j) \geq r}^{\text{(by obs.2)}}$. ⇒ $\underline{\underline{d_{OPT} \geq r}}$ - - - (ii)

⇒ Putting (i) and (ii) together, $\underline{\underline{d_A \leq 2 d_{OPT}}}$. //

Recall the reduction RC → TSP, where $d_{ij} = 1$ if $(i,j) \in E$, else $1 + C$.

→ If G has a RC ⇒ G' has a TSP solution of cost $n = |V|$.

If G doesn't have a RC ⇒ G' has no TSP solution of cost $\leq n + C$.

There is also a reduction RC → α-TSP, where α-TSP gives the solution T s.t. $d_{T(1)T(2)} + \cdots + d_{T(n)T(1)} \leq \alpha\, d^{OPT}_{TSP}$.

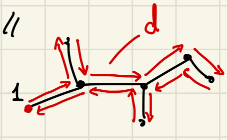
⇒ TSP has no efficient approximation algorithm!

Proof: set $C = \alpha n$. Then, if G has a RC, G' has a TSP solution of cost n, and otherwise, G' has no TSP solution of cost $n + n\alpha$ $= (n+1)\alpha$. → Are we doomed? ⇒ make some assumptions!

2-TSP with Triangle Inequality: d_{ij} s.t. $\forall i,j,k$, $d_{ij} + d_{jk} \geq d_{ik}$.

Lemma: $d_{MST} \leq d^{OPT}_{TSP}$. (proved last time). //

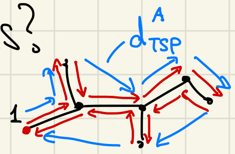
ⓐ MST can be a good starting point.



ⓑ d, a naive traversal of MST, will be less than $2 \cdot d_{MST}$.

↳ This is already a result, $d \leq 2d_{MST} \leq 2d_{TSP}^{OPT}$!

© what if we just "skip" the already visited vertices?



↳ $d_{TSP}^{A} \leq d$ (by Triangle Inequality) $\Rightarrow \underline{\underline{d_{TSP}^{A} \leq 2d_{TSP}^{OPT}}}$. //

Knapsack w/o repetition: $(w_1, \dots, w_n), (v_1, \dots, v_n) \Rightarrow \max(\Sigma v_i)$ where $\Sigma w_i \leq W$.

↳ for $0 < \varepsilon < 1$, we will give an approximation algorithm s.t. $\underbrace{K}_{\downarrow} \geq (1-\varepsilon)\underbrace{K^*}_{OPT}$

↳ runtime will be polynomial w.r.t. n and $\frac{1}{\varepsilon}$ (precision) little worse guarantee than K^*

Main Idea: The reason why we had $O(nW)$ or $O(nV)$ of exp. time is due to large numbers → what if we sacrificed precision?

Algorithm: Discard any items $w_i > W$. Let $V_{max} = \max_i V_i$. Then, rescale $\hat{V}_i = \lfloor V_i \cdot \frac{n}{\varepsilon \cdot V_{max}} \rfloor$. Run DP knapsack with $\{\hat{V}_i\}$. Output solution.

Runtime: $n \times \frac{n}{\varepsilon} \times n = O(n^3/\varepsilon)$.

Precision: $(V_1, \dots, V_n) = S \rightarrow (\hat{V}_1, \dots, \hat{V}_n) = \hat{S}$. Let $\hat{K}$ be lossy sum of S.

1) $\sum_{i \in S} \hat{V}_i = \sum_{i \in S} \lfloor V_i \frac{n}{\varepsilon \cdot V_{max}} \rfloor \geq \sum_{i \in S} \left(\frac{V_i n}{\varepsilon V_{max}} - 1\right) \geq \frac{\overset{\rightarrow K^*}{(\sum_{i \in S} V_i)} n}{\varepsilon V_{max}} - |\overset{\overset{n}{\uparrow}}{\hat{S}}| \geq \left(\frac{K^*}{\varepsilon V_{max}} - 1\right) n$.

↳ $\hat{K} \geq \left(\frac{K^*}{\varepsilon V_{max}} - 1\right) n$.

2) $\sum_{i \in \hat{S}} V_i \geq \sum_{i \in \hat{S}} V_i \frac{\varepsilon V_{max}}{n} \geq \left(\sum_{i \in \hat{S}} \hat{V}_i\right) \frac{\varepsilon V_{max}}{n} = \left(\frac{K^*}{\varepsilon V_{max}} - 1\right) \cancel{n} \cdot \frac{\varepsilon V_{max}}{\cancel{n}} = K^* - \varepsilon V_{max}$

$\geq K^*(1-\varepsilon) \Rightarrow$ can approximate to arbitrary precision!

Heuristics

Local Search Heuristics: Let s be any candidate solution. While there is some solution s' in the neighborhood of s for which $cost(s') < cost(s)$, replace $s \leftarrow s'$. Return s.

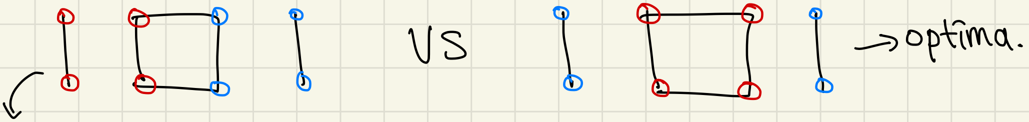
ex) For TSP, perturb two edges to find best neighbor in $O(n^2)$ time.

↳ If we find three edges to permute, $O(n^3)$ time.

A problem - the algorithm might encounter a "local optima", but this can be overcome by emperical hyperparameter tuning.

Graph Partition: $G(V,E)$ of $\mathbb{R}^+$ edge weights $\rightarrow A,B \subseteq V$ s.t. $|A|=|B|$ and the capacity of the cut (A,B) is minimized.

ex)



this state is now "stuck" if our neighborhood is swapping pairs.

1) Randomization & Restarts: hope multiple trials give better solutions

2) Simulated Annealing: Sometimes act suboptimally, with temperature T

Annealing Formula: if $cost(s') > cost(s)$, accept with $P_r = e^{-\frac{(cost(s')-cost(s))}{T}}$

(Out of Scope) Interactive Proofs

"Thinking of NP as a proof"

$$\emptyset$$

$$\underset{\text{prover}}{P} \xrightarrow[\text{satisfying assignment}]{w} \underset{\text{verifier}}{V} \Rightarrow \text{is } \emptyset(w)=1?$$

Two properties are needed:

1) Completeness: If "$\emptyset$ is true", in $P(\emptyset,w) \longleftrightarrow V(\emptyset)$, V outputs 1.

2) Soundness: If "$\emptyset$ is false", in $P(\emptyset,w) \longleftrightarrow V(\emptyset)$ outputs 1 with a very small probability (e.g. 2^{-n} where n is a parameter)

Some changes: P and V can interact, i.e. can give messages back&forth.

Also, we allow V to give a false negative answer with an arbitrarily small probability.

ex) MatMul: $A \times B = C$ is $> O(n^2)$. However,

$$\underset{(A,B,A\times B=C)}{P} \xrightarrow[C]{\qquad\qquad} \underset{(A,B)}{V} \to \text{can check } C \text{ in } O(n^2)!$$

how?

① $r \leftarrow \{1,\ldots,\overset{\text{large prime}}{q}\}$ ② $C \times \vec{r} = (A \times B) \times \overset{[1,r,\ldots,r^{n-1}]^T}{\vec{r}} = A \times (B \times \vec{r})$

If P gives $D \neq C$, $\exists i$ s.t. $c_i \neq d_i$ & $c_i \cdot \vec{r} = d_i \cdot \vec{r}$ $((c_i - d_i)\cdot \vec{r} = 0)$ with a small enough probability so that V is sound.

Of course, there are nontrivial vectors (c_i-d_i) s.t. $(c_i-d_i)\cdot\vec{r}=0$,

(Finite field, mod q)

specifically $p_0+p_1 r+\cdots+p_{n-1}r_{n-1}=0$, but that probability is $O(\frac{n-1}{q})$.

(degree of the polynomial is $(n-1)$, so there are $(n-1)$ roots, and we can choose over the space of q, which is much larger than it)

Graph Isomorphism: $(G_0(V,E_0), G_1(V,E_1)) \rightarrow \pi: V\rightarrow V$ s.t. $\forall e=(u,v)\in E_0$ iff $(\pi(u),\pi(v))\in E_1$.

↳ basically, is there a permutation s.t. edges are conserved.

$G_0\simeq G_1$ if $\exists\pi$ as a valid isomorphism, $G_0\not\simeq G_1$ if not (non-isomorphism).

↳ interestingly, there is no efficient proof for non-isomorphism.

$$(G_0\not\simeq G_1)$$

$$\begin{tikzcd} P \arrow[r, shift right, "b'"'] & V \arrow[l, shift right, "H=\sigma(G_b)"'] \end{tikzcd}$$

① picks random $\sigma: V\rightarrow V$. ② picks random bit $b\leftarrow\{0,1\}$.

③ sends $H=\sigma(G_b)$ to P. ④ P runs, and sends b', the match, to V.

⑤ V outputs 1 if $b=b'$, $\emptyset$ otherwise.

Completeness: If $G_0\not\simeq G_1$, $P(H)$ will deterministically return $b'=b$.

Soundness: If $G_0\simeq G_1$, $P(H)$ will return $b=\emptyset$ or $b=1$ with $\frac{1}{2}$ chance!

↳ generate $\sigma(\cdot)$ n times, run the protocol, then accepts false negative with $Pr=\frac{1}{2^n}$. $\Rightarrow$ arbitrarily small error bound

What if P wants to share V that $G_0 \simeq G_1$, but not the solution π?

↳ This is zero-knowledge property. If $G_0 \simeq G_1$, then V learns nothing more than the fact that $G_0 \simeq G_1$.

P (G_0, G_1, π) V (G_0, G_1)

① P picks a random permutation $\sigma : V \to V$. ② P sends $H = \sigma(G_1)$.

③ V sends $b \leftarrow \{0,1\}$. ④ if $b=1$, $\phi = \sigma$. else, $\phi = \sigma \circ \pi$.

⑤ P sends $\phi(G_b)$. ⑥ If $\phi(G_b) = H$, V outputs 1, else $\emptyset$.

Completeness: $G_0 \simeq G_1 \simeq H$

Soundness: $G_0 \not\simeq G_1$, then P has no way to consistently give $\phi(G_b) = H$.

Zero Knowledge: $b \leftarrow \{0,1\}$, $\sigma : V \to V$, $H = \sigma(G_0) = \sigma'(G_1)$ (WLOG)

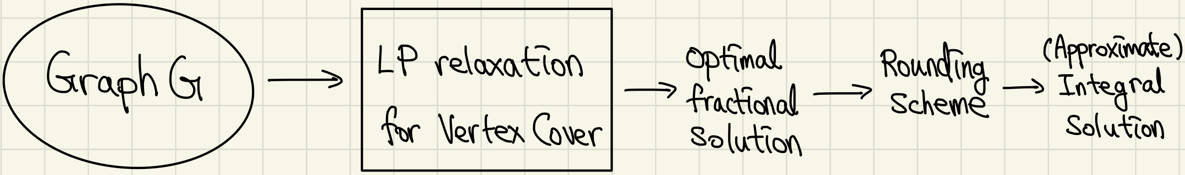
(More) Approximation Algorithms

1) LP based Approx. Algo.: (a) Vertex cover (b) 3-way cut

2) SDP based Approx. Algo

Minimum Vertex Cover: $G(V,E) \rightarrow S \subseteq V$ s.t. $\forall (u,v) \in E$, $u \in S \vee v \in S$.

↳ find vertex cover S of minimum size. (NP-Hard, factor 2 approx.)


Graph G
LP relaxation for Vertex Cover
optimal fractional solution
Rounding Scheme
(Approximate) Integral Solution

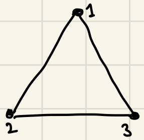
Variables: $\forall i \in V$, x_i. $x_i = 1$ if $i \in S$, $\emptyset$ if not (ideal intention)

Objective: minimize$\left(\sum_{i=1}^{n} x_i\right)$, which is the total size of S.

Constraints: $\forall i \in V$, $0 \leq x_i \leq 1$. (vertex constraint)

$\forall (i,j) \in E$, $x_i + x_j \geq 1$. (edge covering constraint)

ex) Fractional LP solution:



optimal VC = 2

$$\min(x_1 + x_2 + x_3) \Rightarrow \text{LP-OPT} = 1.5 \; (x_1 = x_2 = x_3 = \tfrac{1}{2})$$

$$\begin{cases} x_1 + x_2 \geq 1 \\ x_2 + x_3 \geq 1 \\ x_3 + x_1 \geq 1 \\ 0 \leq x_1, x_2, x_3 \leq 1 \end{cases}$$

however, OPT is actually 2, which is strictly larger than the fractional solution.

Observation: $\forall G$, LP-OPT$(G) \leq$ OPT(G).

$\because$ OPT(G) is the best solution among all integer solutions, while LP-OPT is the best among ALL integer and fractional solutions.

Rounding Scheme: Let x^* be the LP-OPT. $x_i^* \in [0,1]$ $\forall i$.

$S \leftarrow \{i \mid x_i^* \geq 0.5\}$. (set all i at least $\frac{1}{2}$ to 1, others to $\emptyset$.)

Lemma 1: S is a valid VC.

Proof: $\forall (i,j) \in E$, $x_i^* + x_j^* \geq 1. \Rightarrow x_i^* \geq \frac{1}{2} \vee x_j^* \geq \frac{1}{2} \Rightarrow i \in S \vee j \in S$. //

Claim: $|S| \leq 2 \cdot$ LP-OPT. ($|S| \leq 2 \cdot \sum_{i=1}^{n} x_i^*$)

Proof: Consider any vertex $i \in S$. For LHS, it contributes 1 size. For RHS, $2x_i^* \geq 1$ because $x_i^* \geq \frac{1}{2}$ for all $i \in S$. More formally, $|S| = \sum_{i \in X^*} \mathbb{1}\{i \in S\}$. For each i, $\mathbb{1}\{i \in S\} \leq 2 \cdot x_i^*$ because $i \in S \Leftrightarrow x_i^* \geq \frac{1}{2}$. //

Minimum 3-Way Cut: $G(V,E)$, $a,b,c \in V \rightarrow$ Partition a,b,c by cutting the fewest number of edges.

Remark: Minimum 2-Way Cut is Max-Cut problem, which is in P.

However, Minimum 3-Way Cut is NP-Hard.

Variables: $\forall v \in V$, decide whether v resides in component 1, 2, or 3.

↳ $\forall v \in V$, $v \to (v_1, v_2, v_3)$ is a one-hot encoding of inclusion.

($v \to 1 \Leftrightarrow (v_1, v_2, v_3) = (1, 0, 0)$, and so on.)

⇒ $\forall v \in V$, v_1, v_2, v_3 where $v_i = 1$ if $v \in$ Component i, 0 otherwise.

Constraints: $\forall v \in V$, $0 \leq v_1, v_2, v_3 \leq 1$, $v_1 + v_2 + v_3 = 1$. (vector constraints)

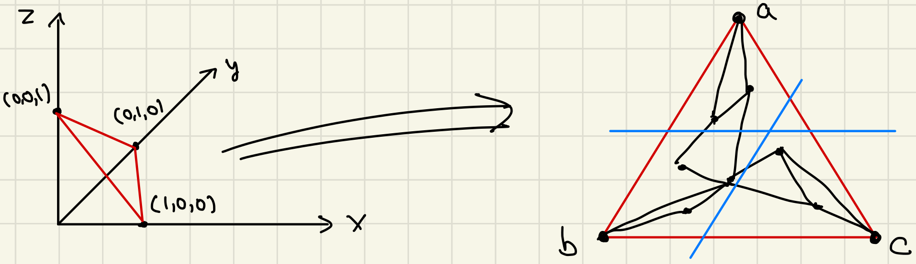
$a = (1, 0, 0)$, $b = (0, 1, 0)$, $c = (0, 0, 1)$ (partition constraint)

Objective: # of edges cut $= \sum_{(u,v) \in E} \mathbb{1}\{(u,v) \text{ is a cut}\}$. Basically, we want to check if u and v are not in the same component.

↳ $\mathbb{1}\{(u,v) \text{ is a cut}\} = (|u_1 - v_1| + |u_2 - v_2| + |u_3 - v_3|) \cdot \frac{1}{2}$

⇒ $\min\left(\frac{1}{2} \sum_{(u,v) \in E} \{|u_1 - v_1| + |u_2 - v_2| + |u_3 - v_3|\}\right)$. ← can be made into an LP with slack variables

Observation: with the constraint $\forall u$, $u_1 + u_2 + u_3 = 1 \wedge 0 \leq u_1, u_2, u_3 \leq 1$, $\vec{u}$ lives on the equilateral triangle of $((1,0,0), (0,1,0), (0,0,1))$.

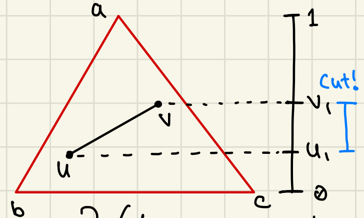


Rounding Scheme: 1) Pick 2 out of 3 sides. 2) make two cuts parallel to the picked sides, with random heights.

Claim: $\Pr[\text{edge }(u,v)\text{ is cut}] \leq \frac{2}{3}\|\vec{u}-\vec{v}\| = \frac{4}{3}\cdot\frac{1}{2}(|u_1-v_1|+|u_2-v_2|+(u_3+v_3|)$.

↳ $E[\#\text{ of edges cut}] = \sum_{(u,v)\in E} \Pr[(u,v)\text{ is cut}] = \frac{4}{3}\text{LP-OPT}$.

Subclaim: For random cut || (b,C), $\Pr[(u,v)\text{ is cut}] = |u_1-v_1|$.



↳ $\Pr[(u,v)\text{ is cut}] = \Pr[\text{cut lands between } u \& v]$

$\Rightarrow \Pr[(u,v)\text{ is cut in the rounding scheme}] \leq$

$\frac{2}{3}(|u_1-v_1|+|u_2-v_2|+|u_3-v_3|)$ (we try two cuts out of three)

$= \frac{4}{3}\left[\frac{1}{2}(|u_1-v_1|+|u_2-v_2|+|u_3-v_3|)\right] \Rightarrow \frac{4}{3}$ approx. factor //

↳ $= \frac{1}{3}(|u_1-v_1|+|u_2-v_2|)+\frac{1}{3}(|u_2-v_2|+|u_3-v_3|)+\frac{1}{3}(|u_3-v_3|-|u_1-v_1|)$

# of edges crossing $S \& \bar{S}$

Maximum Cut: $G(V,E) \rightarrow S \subseteq V$ s.t. $\text{cut}(S,\bar{S})$ is maximized (NP-Hard)

Naive Randomized Algo: randomly assign all vertices into S or $\bar{S}$.

↳ every edge is cut with probability ½. $\Rightarrow \mathbb{E}[\text{cut}(S,\bar{S})] = \frac{1}{2}\cdot|E|$.

Strategy: Use semidefinite programming instead of LP.

→ quadratic program, in this case

Variables: $\forall i \in V$, $x_i = \begin{cases} +1 & \text{if } i \in S \\ -1 & \text{if } i \in \bar{S} \end{cases}$.

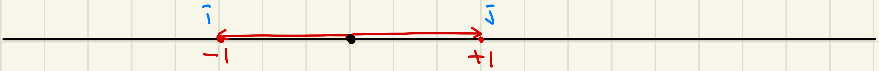
Constraints: $x_i^2 = 1$ ($\Leftrightarrow x_i = \pm 1$).

Objective: $\sum_{(i,j)\in E} \mathbb{1}\{\text{edge }(i,j)\text{ is cut}\} = \sum_{(i,j)\in E} \frac{(x_i-x_j)^2}{4}$. → 4 if $x_i \neq x_j$ (cut), 0 if $x_i = x_j$ (no cut)

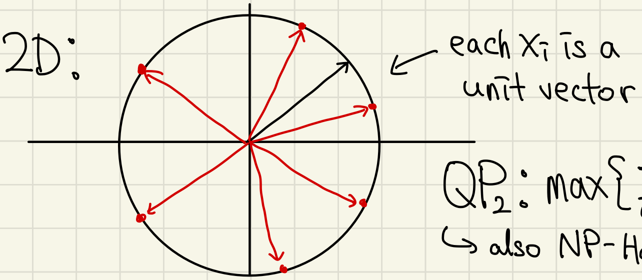
$\Rightarrow$ QP exactly captures Max Cut, but solving QP is NP-Hard.

Insted, look at a relaxation of the program.

1D:



2D:



QP_2: $\max\{\sum_{(i,j)} \|x_i - x_j\|^2\}$ subject to $\|x_i\|^2 = 1$.

↳ also NP-Hard, unfortunately.

⇒ However, QP_n can be solved in polytime! (Semidefinite Program)

QP_n: $\forall i \in [n], \|v_i\|^2 = 1$ where $v_i \in \mathbb{R}^n \rightarrow v_i = (v_i^{(1)}, v_i^{(2)}, \ldots, v_i^{(n)})$.

↳ $\|v_i\|^2 = \sum_{j\in[n]} (v_i^{(j)})^2$. $\max\{\sum_{(i,j)} \|v_i - v_j\|^2\}$. ⇒ SDP for Max Cut

What is an SDP?

Variables: n vectors in n-dimensions ($\mathbb{R}^n$)

Constraints: linear constraints on dot products ($v_i \cdot v_j = \sum_a v_i^{(a)} \cdot v_j^{(a)}$)

Objective: min / max a linear function of dot products

⇒ QP_n is an SDP since $\|v_i\|^2 = 1 = v_i \cdot v_i$, and all equations can be expressed as a linear combination of dot products

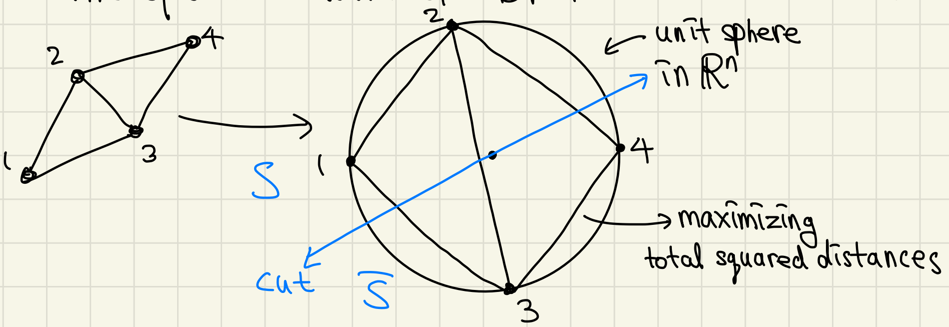
Why is SDP efficient? (can be black boxed)

$K = \{$Set of $\overset{(n\times n)}{\text{matrices}}$ M where all eigenvalues$(M) \geq 0\}$

↳ positive semidefinite matrices ⇒ this is a convex set

M is a positive semidefinite matrices ⇔ $M_{ij} = v_i \cdot v_j$ (all entries are dot products)

⇒ The optimal solution of SDP is n vectors in $\mathbb{R}^n$.


2
4
1
3
S
2
1
4
3
cut
$\bar{S}$
unit sphere in $\mathbb{R}^n$
maximizing total squared distances

Randomized Rounding: 1) Pick a random hyperplane passing origin.

2) Put vertices on one side to S, others to $\bar{S}$.

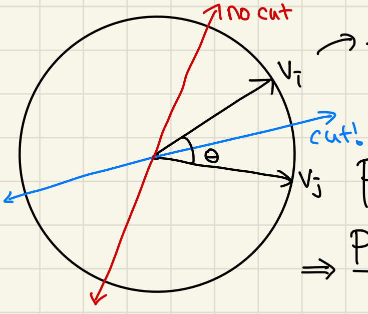
→ SDP is less constrained

Analysis: SDP-OPT $= \sum_{(i,j)} \|v_i - v_j\|^2 \geq$ Integer Max Cut.

Claim: $Pr[(i,j) \text{ is cut}] \geq (0.878) \cdot \|v_i - v_j\|^2$.

≥ 0.878 OPT.

↳ This implies that $E[\text{size of cut}] \geq 0.878 \cdot$ SDP-OPT.



→ the plane (2D) containing v_i & v_j is sufficient for analysis

① $Pr[(i,j) \text{ is separated}] = \frac{\theta}{\pi}$. ② $\|v_i - v_j\|^2 = 2 - 2\cos\theta$.

⇒ $\frac{Pr[(i,j) \text{ is cut}]}{\|v_i - v_j\|^2} = \frac{2 - 2\cos\theta}{\theta/\pi} \geq 0.878$ for all θ (empirical)

⇒ $Pr[(i,j) \text{ is cut}] \geq 0.878 \|v_i - v_j\|^2$. // (In fact, this is the best known ratio.)