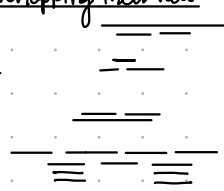
Greedy Algorithms

OPTIMIZATION: pick largest/smallest ___ subject to ___
pick in order by ___.

Min Change Possible:
Modulus repeatedly.

Interval Scheduling:
Q. Find largest set of non-overlapping intervals:
1. Largest Interval First:
2. Smallest Interval First
3. Earliest Start Time
4. Least Conflicts:
5. Earliest End Time: Correct!



Largest Set: Earliest Finish Time

Let S be empty list — $O(1)$

Let F be list of intervals sorted by finish time — $O(n \log n)$

For each $x \in F$: — $O(1)$ by checking against last added
If x is compatible with S, add x to S

Proof: GREEDY STAYS AHEAD

LEMMA: i-th element in S will never have later finish time than any other list sorted by finish time

$[s_1, t_1]$, ... $[s_n, t_n]$ → s output
T → any other input

$\forall\, i$: $S[i].f \leq T[i].f$

inductive proof: INDUCT ON I

Base Case: $i = 1$
S[1] is first interval, we select by earliest finish time.

Assume: Lemma True upto k.

Inductive step: Prove for k+1

$S[k].f \leq T[k].f \leq T[k+1].s$

If $T[k+1].f < S[k+1].f$
we would've picked that instead

LEMMA 2: NO ALTERNATE SOLUTIONS EXIST

Assume we're suboptimal [Contradiction]
$\exists$ T, $|S| < |T|$ if $|S| = m$, $|T| \geq m+1$

Let S, T be sorted by finish times. from Lemma 1

$\forall\, i \leq m$, $s[i].f \leq T[i].f \leq T[m+1].s$

T[m+1] is compatible with S!

Our algorithm would have added it!

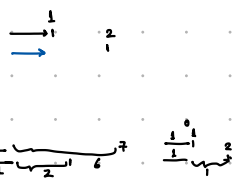
MINIMIZE LATENESS:

Jobs j_i → deadline d_i, duration t_i, finishes f_i

Lateness: $l_i = \max(0, f_i - d_i)$

L of Schedule = max late job ← MINIMIZE THIS "WORST LATENESS"

STRATEGIES:
1. Latest deadline first:
2. Earliest deadline
3. Least time to complete
4. Most "Slack Time"
5. Least "Slack Time"
6. Earliest Deadline First



MIN LATENESS: EARLIEST DEADLINE

return sorted (J) — $O(n \log n)$

PROOF: TWO JOBS

j_1-j_2 case	j_2-j_1 case
$l_1 = t_1 - d_1$	$l_1 = t_1 + t_2 - d_1$ max
$l_2 = t_1 + t_2 - d_2$ max	$l_2 = t_2 - d_2$

→ assume any could be 0 and $d_1 \leq d_2$

$t_1 + t_2 - d_2 \leq t_1 + t_2 - d_1$

LEMMA: given j_1 & j_2, if $d_1 \leq d_2$ then if $t_2 \leq t_1$
l_1 is atleast as bad as any lateness in $j_1 j_2$ schedule

GREEDY EXCHANGE ARGUMENT

Let $j_1, \dots j_n$ be a schedule jobs
if there is a pair $d_i > d_{i+1}$

$l_i = t^* + t_i - d_i$
where t^* is the start time of

$l_i = t^* - t_i - d_i$

$l_{i+1} = t^* + t_i + t_{i+1} - d_{i+1}$

after swap:

$l_i' = t^* + t_{i+1} - d_{i+1}$
$l_{i+1}' = t^* + t_i + t_{i+1} - d_i$

* Lateness of swapping jobs doesn't change rest of the jobs

end result: schedule sorted by deadline!

TREE:

A graph is tree if it has any 2 properties

1. Connected
2. Acyclic
3. If has n nodes, n-1 edges

PROVE: N NODES AND N-1 TREES

1. n edges are too many
2. n-2 edges too few

PROVE: GRAPH WITH N NODES AND M EDGES HAVE ATLEAST N-M-1 COMPONENTS

INDUCTIVE STEP: $G(V,E)$ with n nodes and $k+1$ edges
$E' = E \setminus \{e_{k+1}\}$, $G' = (V, E')$
G' has n nodes & k edges, n-k components atleast.

Inserting an edge into graph reduces components by at most 1: $G' + \{e_{k+1}\} \to G$
G has $(n-k-1)$ components.

SPANNING TREE:

given $G = (V, E)$
ST contains the edges to make ST a tree
$T \subseteq E$ s.t. (V, T) is a tree

MIN WEIGHT SPANNING TREE

G IS CONNECTED edge +ve or -ve

Let T be empty

Let E be edges sorted by weight — $O(m \log m)$

For each $e \in E$ and $e <= n-1$;
If $T \cup \{e\}$ is acyclic, insert e into T

KRUSKAL'S! — $O(m \cdot n)$

PRIMS ALGORITHM:

choose lightest edge weight which joins our component

can do from any vertex!

Let T be empty

Let $C = \{s\}$ for some $s \in V$

Let E be sorted by weight

Until C contains all vertices:
Grow C — $O(m \cdot n)$ unless you do heap

CORRECTNESS: KRUSKAL AND PRIMS

Q. Kruskal produce ST? No cycle if end with ≥ 2 components there is a crossing edge we would've considered it!

Q. Prim's Produce ST?
Suppose T is missing a vertex. Since G is connected, there must be a crossing edge we'd have considered!

KRUSKAL'S OPTIMALITY

LEMMA — KRUSKAL STAYS AHEAD

T_K be kruskal edges on $G_1 = (V, E)$
T_K sorted by weight $(e_1, \dots, e_{n-1})$
For spanning trees $T' = (f_1, \dots, f_{n-1})$ sorted by weight

$\forall\, i$: $w(e_i) \leq w(f_i)$

i-th edge is least weight possible

Let $T = (e_1, \dots, e_{n-1})$ be o/p of kruskal's and $T' = (f_1, \dots, f_{n-1})$
Suppose $\exists\, k$ s.t. $w(e_k) > w(f_k)$
Consider $\{e_1, \dots, e_{k-1}\}$ and $\{f_1, \dots, f_k\}$
$\exists\, f_i \in \{f_1, \dots f_k\} \setminus \{e_1, \dots, e_{k-1}\}$ s.t.
$\{e_1, \dots, e_{k-1}, f_i\}$ is acyclic from

LEMMA 2:
If E_1 and E_2 are both acyclic and $|E_1| > |E_2|$, then...
There is some edge $e \in E_1 \setminus E_2$ s.t. $E_2 \cup \{e\}$ is acyclic

BUT: if $\{e_1, \dots, e_{k-1}, f_i\}$ is acyclic, we would have added before e_k
but $f_i \notin \{e_1, \dots, e_{k-1}\}$

COROLLARY:

Can't have an MST which "falls behind" Kruskal's and then "catches up".
- As soon as the i-th sorted by weight is heavier than ours, it can't be an MST.

COROLLARY OF COROLLARY:

If each edge in G is different, F_1 has a unique MST.

CUT-PROPERTY LEMMA:

Let $G = (V, E)$ be a weighted undirected graph
For any cut $(S, V \setminus S)$ of G
If there is a unique lightest edge e crossing the cut
Then e is in every MST of G

PROOF:
Suppose MST T without e [minimum crossing edge]
T must contain at least one edge crossing the cut [the same one as e], say ci
Then the tree containing the other edge T U {e} has a cycle.
Because n nodes and n edges.
And since T is a tree, it must contain a path from u to v
Then the path must contain some edge ci
We can thus swap e with ci

MORE EFFICIENT KRUSKAL'S: UNION FIND

UNION-FIND: Keeps collection of sets where each belongs to one set.

INITIALIZE/INSERT: $O(n)$ for all n elements

FIND(x): Report label x (the root of x's tree) $O(\log(n))$

MERGE(X,Y): $O(\log(n))$

AMORTIZED TIME UNION-FIND:

$O(\log^*(n))$

Some operations take $\log(n)$ steps

But for $m \geq n$ operations

Total: $O(m \log^*(n))$

$\log(n)$: Number of times needed to divide n to get 1.

$\log^*(n)$: Number of times needed to apply log to reach 1

NEW KRUSKAL'S RUNTIME: WITH UNION FIND

With Union Time: $O(m \log n)$ [to sort edges!]

But if edges sorted by weight:

$O(m \log^* n)$

PRIMS WITH HEAPS:

V	Min Cost
a	∞
b	∞
c	∞
d	∞
e	∞
f	∞
g	∞

1. Build heap w/ ∞ min cost edge value for each v
2. Choose start node
3. Remove from heap
4. Update neighbors of node
5. Choose - min cost vertex

Initialize union-find is $O(n)$

Sorting still $O(m \log n)$

m cycle-checks: $O(\log(n))$ each

n merges: $O(\log(n))$ each

Total: $O(m \log(n))$

Let T be empty

Let E be edges sorted by weight

For each $e \in E$:
If $T \cup \{e\}$ is acyclic, insert e into T

PROVE THAT ADDING A CONSTANT WEIGHT TO ALL EDGES IN MST DOESNT CHANGE MST:

↳ if we add a constant term to this ordering, the ordering itself doesn't change.

$$W(T) = \sum_{e \in T} l_e, \quad W'(T) = \sum_{e \in T} l_e' = W(T) + \underbrace{a}_{(|V|-1)} \underbrace{(|V|-1)}_{(|V^*|+1)}$$

$$W'(T) - W'(S) = W(T) + a - W(S) - a \leq 0$$
$$= W(T) - W(S) \leq 0$$
$$= W(T) \leq W(S) \quad \square$$

1. If G has $|E| > |V|-1$ and unique W_{max} edge wt. MST(G) doesn't have W_{max} edge. False

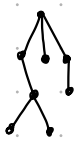
PERFECT-MATCHING IN A TREE:

Set $E' \subseteq E$ in $G(V,E)$ s.t. all $v \in V$ occur once in E'.

$O(m+n)$

1. Find leaf nodes of T.
2. If two leaf nodes have same parent, return False.
3. Otherwise, find parents of all leaf nodes. Add edges E connecting them to our matching
4. Remove parent and leaf vertices.
5. If G is empty, return True. If G has no edges and some vertices left return False. Otherwise recurse.

MAX UNWEIGHTED INDSET



For Leaf: EXCHANGE PROOF

1. If parent is included, swap parent w/ leaf
2. If parent isn't included add leaf.

ALGORITHM:

1. Add leafs to I
2. Remove leafs + parents
3. Repeat until no vertices left

SOLVING GREEDY

- Pick intervals in order of [start, finish, length, weight, etc.]
  - Interval scheduling! Picks in order of finish time.
- Pick edges in order of [weight, value]
  - Kruskal's Minimum Spanning Tree algorithm! Keep adding next smallest edge possible.
- Pick nodes in order of [weight, value, distance to source]
  - Dijkstra's Algorithm! Add next node that has shortest distance to source.
- Pick tasks in order of [weight, deadline/finish, etc.]
  - Minimizing lateness! Do tasks in order of deadline, no matter the time it takes.
- Assign tasks one by one to the least crowded bin each time

PROVING GREEDY:

- Stays-ahead argument
  - Show that at each step, our algorithm does better than any other algorithm.
  - Example: Kruskal's stays-ahead lemma.
- Exchange argument
  - Show that we can transform any solution to the problem to the greedy solution without hurting its quality.
  - Example: proof for lateness minimization (lecture 15).
- Structural argument
  - Show that the problem imposes structural bounds on how optimal the solution can be, and then argue that our argument achieves this bound.
  - Example: scheduling all intervals (practice #1).
- Ad-hoc arguments
- For any type of argument, don't forget to finish the proof by demonstrating that our solution is optimal.

DIFFERENT TYPES OF MST PROBLEMS:

1. MST with leaf node set U on G(V,E):
   1. Find MST on V/U
   2. Find min edges connecting T to U nodes
2. Maximum Spanning Tree:
   1. Negate Edge Weights
3.

DYNAMIC PROGRAMMING

FIBONACCI: Memoization Table

O(N) Bottom-Up: Table-Filling
No Recursive Stack!

Memoization: Top-down

ALGOS: 1. Find a Recurrence
2. Fill a Table

PROOFS OF CORRECTNESS:
USUALLY: RECURRENCE + BASE CASES
TRIVIAL INDUCTIVE ARGUMENT

RUNTIME:
Table has x entries, with y operations: $O(x \cdot y)$

INDEPENDENT SETS:

$I \subseteq V$ is independent if no pairs of vertices have edges between them
$\forall u, v \in I: \{u, v\} \notin E$

WEIGHTED INDEPENDENT SET (IN PATHS)

HOW TO ENUMERATE ALL POSSIBLE STEPS
binary string?
recursive backtracking

MAX WEIGHT TABLE FILLING

Input: $w_1, \dots, w_n$
Initialize table $V[0, \dots, n]$
$V[0] = 0, V[1] = \max(w_1, 0)$

For $i = 2$ to n:
$V[i] = \max(V[i-1], V[i-2] + w_i)$

Return $V[n]$

O(N)

FINDING SET: RECOVER SET ONCE

With table filled out, how to check if v_n is in solution?

Build-Set$(w_1, \dots, w_n, V)$

If $V[n] = V[n-1]$:
Return Build-Set$(w_1, \dots, w_{n-1}, V)$

Else:
Add v_n to set
Return Build-Set$(w_1, \dots, w_{n-2}, V)$

$T(N) \leq O(1) + T(n-1)$

WEIGHTED INTERVAL SCHEDULING

Input: list of weighted intervals
Initialize table $V[0, \dots, n], V[0] = 0$ O(n)

Let F be list of intervals sorted by finish time O(nlogn)
Let w_i be weight of $F[i]$

For $i = 1$ to n: O(logn)
Find latest index k such that $F[k]$ ends before $F[i]$ starts
$V[i] = \max(V[i-1], V[i-k] + w_i)$ binary search

Return $V[n]$

SEQUENCE PROBLEM:

MAX FOLDS FOR A-U, C-G PAIRS WITH NO SHARP TURNS AND NO CROSSING

if (i,j), $j - i \geq 5$
if (i,j) and (k,l) w/ $i < j$ and $k < l$,
$i < k < j < l$ NOT POSSIBLE

KNAPSACK PROBLEM

KNAPSACK TABLE FILLING:

Input: $C, w_1, \dots, w_n, v_1, \dots, v_n$

Initialize table $V[0, \dots, n][0, \dots, C]$ with base cases filled in
//For any negative j, treat $V[i, j] = -\infty$

For $i = 1$ to n:
For $j = 1$ to C:
$V[i, j] \leftarrow \max(V[i-1, j], V[i-1, j-w_i] + v_i)$

Return $V[n, C]$

$O(n \cdot C)$ if C is in binary $C \leq \log(C)$
PSUEDO-POLYNOMIAL

FLOW NETWORKS:

1. Directed Graph $G(V,E)$
2. Source s, Sink t
3. Edge capacity $c(e) \geq 0 \quad \forall e \in E$

Flow: Assignment $f(e)$ $\forall e \in E$ s.t:
1. Non-negative: $f(e) \geq 0$
2. Capacity: $\forall e \in E$: $f(e) \leq c(e)$
3. Conservation: $\forall v \in E / \{s,t\}$, flow in = flow out

Max Flow: $s_{out} = t_{in}$ is greatest volume

WHY? PROVE

$[s_{out}] + [t_{out}] + \sum_{u \in V/\{s,t\}} [u_{out}] = [s_{in}] + [t_{in}] + \sum_{u \in V/\{s,t\}} u_{in}$

($[t_{out}] = 0$, $[s_{in}] = 0$)

$\Rightarrow [s_{out}] = [t_{in}]$ — conservation

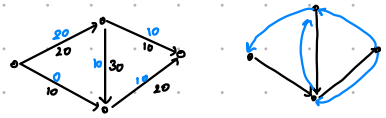
AUGMENTING PATHS:

Path from s to t which may have
- backwards edges
- \+ room to add flow to forward
- \+ room to remove in backward edges

LEMMA: ∃ Augmenting Path iff flow not max

RESIDUAL GRAPH: G, f

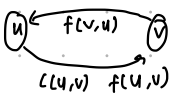
G_{res}: has edge (a,b) if
- flow can be added $a \to b$
- flow can be removed $b \to a$



forward edge: $c(u,v) - f(u,v)$

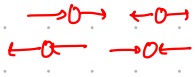
backward edge: $f(v,u)$

both: $c(u,v) - f(u,v) + f(v,u)$



FORD-FULKERSON PROOF:

1. **Feasibility:** Does it give a valid flow?
   - can augmentation reduce < 0 — cases
   - put flow over capacity
   - violate conservation



2. **Optimality and Cuts:**

s-t cut (A,B) where $s \in A$ and $t \in B$

$$c(A,B) = \sum_{u \in A, v \in B} c(u,v)$$

only FORWARD not BACKWARD edges

CUT-FLOW LEMMA:

$|f| = [\text{flow out of } A] - [\text{flow into } A]$

PROOF:

$$= \sum_{e \text{ from } A} f(e) - \sum_{e \text{ into } A} f(e)$$

$$= \sum_{v \in A} \left(\sum_{e \text{ from } v} f(e) - \sum_{e \text{ into } v} f(e) \right)$$

0 for each vertex in A except s

$$= \sum_{e \text{ from } s} f(e) - 0 = |f|$$

THEOREM:

given flow network G and flow f and s-t cut (A,B) in G

$|f| \leq c(A,B)$ BOTTLENECK

$|f| = [\text{flow out of } A] - [\text{flow into } A]$

$|f| \leq [\text{flow out of } A] \leq [\text{capacity out of } A]$

$\Rightarrow |f| \leq c(A,B)$

BIPARTITE GRAPH:

$G = (V, E)$

vertices partitioned into X and Y, s.t.
X and Y are independent vertices

$x, y \in L, R: \{x,y\} \notin E$

MATCHING:

$G \to$ UNDIRECTED

$M \subseteq E$ is a matching if no 2 edges in M share a vertex

OPTIMALITY PROOF: FORD-FULKERSON

$\forall f$ given by FF

$\forall$ s-t cuts (A,B), $|f| \leq c(A,B)$

$\therefore |f| = c(A,B)$ gives us maximum

Cutting the Residual Graph:

When Ford Fulkerson has finished:

G_f has a set of
- vertices A reachable by s, and
- vertices B unreachable by s

$t \in B$ because otherwise we would've found the path
- $\Rightarrow (A, B)$ is a valid s - t cut
- and $|f| = c(A, B)$

Claim 1: [flow out of A] = $c(A, B)$
- This is because all edges leaving A are full. Otherwise, there would be an edge in G_F from some $u \in A$ to some $v \in B$.
- But then, v would be reachable by s

Claim 2: [flow into A] = 0
- All edges from B to A have 0 flow
- Otherwise, there would be edge in G_F from some $u \in A$ to some $v \in B$.
- But then, v would be reachable by s

$|f|$ = [flow out of A] - [flow into A] = $c(A, B)$

Since no flow can do better than $c(A, B)$, f is optimal.

(A, B) is the minimum capacity cut.
- $|f| <= c(A, B)$
- $|f|$ is also a lower bound on the capacity of the cut.

COROLLARY: MAX-FLOW = MIN CUT

Network: $G, s, t, c(\cdot)$

Min capacity over all s-t cuts in G is the value of max-flow of G

$$\min_{\text{cuts } (A,B) \text{ of } G} c(A,B) = \max_{\text{flows on } G} |f|$$

1. Compute max flow f and residual G_f
2. Let A be vertices reachable from s, B everything else

$M(n+m) + O(n+m)$

Min cut is the COST OF DISCONNECTING NETWORK; WHERE TO BUILD NEW ROADS

MAXIMIZE BIPARTITE MATCHINGS: $O(m \cdot F)$

$F = m$

$O(m^2)$

1. Direct all edges from L to R w/ $e_c = 1$
2. Add source to L and sink to R
3. All the source edges are allowed to send 1 unit of flow; and sink can receive 1 unit of flow

PROOF:

1. Show $|f| \geq |M|$ if there is an M:
   → edges tell you where to send flow through
2. Show if f, retrieve M s.t $|M| \geq |f|$:
   ↳ You can't convert from ANY flow into a matching
   ↳ Each edge must have 0 or 1 unit of flow
   choose edges w/

LEMMA: INTEGRAL FLOWS:

If all capacities are integers, FF gives us an integral flow.

FORD-FULKERSON:

Input: Flow network G with source s, sink t, and edge capacities $c(\cdot)$

Initialize f to trivial flow — 0 on each edge — always a valid flow!

Generate residual graph G_f with capacities $c_f(\cdot)$

Loop: do F times worst-case scenario
- Find path $P = e_1, \ldots, e_k$ from s to t in G_f ★ BFS/DFS $O(m+n)$
- If no path exists, end loop
- Let $b \leftarrow \min(c_f(e_1), \ldots, c_f(e_k))$ $O(N)$
- Update f by pushing b units of flow through P $O(N)$

$O((n+m)F)$, $m \geq \Omega(n)$

$\Rightarrow O(m \cdot F)$ Pseudo Polynomial Time

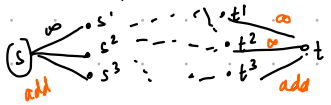
★ HOW TO SEARCH FOR AN AUGMENTING PATH?

$O(m \cdot n)$ improved to $m^{1+o(1)}$

$n \geq \sqrt{m}$, $m^{1.00000\ldots01}$

MODIFICATIONS:

1. Max flow on multiple sources and sinks:



2. Vertex capacities

convert $v \to v_{in} \xrightarrow{v_c} v_{out}$

3. Edge demands: Min flow across edge

New $G' = (V', E')$
- $s' \to v, \forall v \in V$ · $t \to s$
- $v \to t', \forall v \in V$

$c'(e)$:
- $c'(s', v) = \sum_{u \in V} \ell(u,v), \forall v \in V$
- $c'(v, t') = \sum_{w \in V} \ell(v,w), \forall v \in V$
- $c'(e) = u(e) - \ell(e) \quad \forall e \in V$
- $c'(t, s) = \infty$

3 GROUP MATCHING:

construct $Z \subseteq M \times W \times L$ (k, m, n)

s.t. $(u,v) \notin P_1$ where $P_1 \subseteq M \times W$

$(v,w) \notin P_2$ where $P_2 \subseteq W \times L$

$c = 1$

$O(k^2 mn)$

G' = M, W, L and W' copy of W

$m_i \to w_j$ if $(m_i, w_j) \notin P_1$, $w_i \to l_i$ if $(w_i, l_i) \notin P_2$

$w_i \to w_i'$

$s \to m_i$ and $l_i \to t$

Algorithm: Ford-Fulkerson

```
function AUGMENT(f, P)
    b ← bottleneck(P, f)
    for e = (u, v) ∈ P do
        if e is a forward edge then f(e) ← f(e) + b
        else if e is a backward edge then f(e) = f(e) − b
        end if
    end for
    return f
end function
function FORD-FULKERSON(G = (V, E), s, t)
    f(e) ← 0 ∀e ∈ E
    while s − t path in G_f exists do
        P ← s − t path
        f' = augment(f, P)
        f ← f', G_f ← G_f'
    end while
    return f
end function
```

Algorithm: Union-Find-Based Kruskal's Algo

```
function KRUSKAL(G = (V, E))
    T ← ∅, U ← Initialize(V)
    sort edges of E by cost
    for every (v, w) ∈ E do
        if Find(U, v) ≠ Find(U, w) then
            T ← T ∪ {(v, w)}
            Union(U, v, w)
        end if
    end for
    return T
end function
```

P vs NP:

Efficiently SOLVABLE vs. VERIFIABLE time

Length n input

Polynomial time $\exists c > 0$ $O(n^c)$

for Graphs $O(n^c \cdot m^c)$; $O(n^c + m^c)$

How many bits to write graph (n, m)? n^2

$v_1 \ldots v_{n-1} \longrightarrow \log(n)$ bits as binary

$e_1 \ldots e_{n-1} \longrightarrow 2\log(n)$ bits as binary

$O((n+m) \cdot \log n)$

Number of Bits:

Adjacency Matrix M_G $M_G = n^2$

$n^c = N^{c/2}$, $m \leq N$

$n^c \cdot m^c \leq N^{c/2} \cdot N^c \leq N^{2c}$

Adjacency List L_G:

$\Theta((n+m)\log(n))$

Search Problems: given x, find y given conditions

Decision Problems: Yes or No

IS THIS LIST SORTED?

Search: G w/ s & t, find length of shortest path from s to t

Decision: Given G w/ nodes s & t, & integer K is there path from s to t $\leq K$

There is poly-time alg iff poly-time for decision for search

SORTING:

OG PROB: Given list L, Sort (L)

DECISION: Is i^{th} element j^{th} order stat? Run n^2 times to find sorted list.

OG PROB: Given list L, o/p Sort (L) in binary

DECISION: Given L and i, o/p the i^{th}-bit of Sort (L). Is the i^{th} bit 1?

FOR ANY SEARCH PROBLEM

Corresponding Decision Problem.

"Is the i^{th} bit of original output 1?"

CLASS P:

All decision problems solvable in poly time.

VERIFIER FOR COMPOSITE PROBLEM:

Input: Number A in binary

Output: T is A composite else F

No known poly time alg for finding factors

Verifier receives additional i/p certificate "c"

1. If "Yes" answer, $\exists$ c s.t. Verifier says Yes
2. If "No" answer $\forall$ c, verifier c says "No"

DECISION PROBLEMS:

SHORTPATH asks "given G, s, t, k is there a path from s to t of len $\leq k$"

LONGPATH asks "given G, s, t, k is there a path from s to t of len $\geq k$"

INDSET asks "given G, k, is there an independent set in G of size $\geq k$"

PERF-MATCH asks "given bipartite graph G, is there matching which covers every vertex?"

VERIFIER FOR SHORTPATH:

1. Input: G, s, t int k
   c: list of vertices $v_1, \ldots, v_j$
   Check $j \leq k+1$
   Check if $v_1, \ldots, v_j$ is path
   Check $v_1 = s$ and $v_j = t$
   return True else False

INDSET: check if ind set check if atleast k

PERF-MATCH: check all vertices check independent

CLASS NP: Efficiency requirement for verifies

For inputs of length n

- Only allowed to receive certificates c of poly length
- Must run in poly time

NP: Efficiently verifiable problems

IS P = NP?

$P \subseteq NP$ is TRUE

- **If P = NP:** can factor large numbers & break RSA in polynomial time
  - but algos could take n^{10000}
- **If P ≠ NP:**
  - ↳ At least 1 problem in P not in NP.
  - ↳ Doesn't tell us that:
    - Can't break RSA in poly-time
    - Any NP problem require EXPONENTIAL time COULD be $\log(!)$ $2^{\log n^2}$

MANY-ONE REDUCTIONS / KARP:

If A and B are decision problems:
A function f which:
[f does not have to be 1-1 or onto]
If x is "yes" of A, then f(x) is "yes" of B
If x is "no" of A, then f(x) is "no" of B

Clique of Graph: Set $C \subseteq V$ of $G(E, V)$ s.t all pairs of (u, v) in C are adjacent

INDSET ⟶ CLIQUE REDUCTION:

If input is G = (V,E) and k, make
E' = {{u, v} belongs to V | {u, v} doesn't belong to E}
Output G' = (V, E') and k

REDUCTION REQUIREMENTS:

Check that our reduction:

1. Is computed in Polynomial time $O(N^2)$
2. If G has ind set of size $\geq k$, then G' has size $\geq k$
3. If G has no ind set of size $\geq k$, then G' has no clique of size $\geq k$.

$INDSET \leq_p CLIQUE$

reducible

Proof for 2: If G has ind set I of size $\geq k$ Then I is a clique in G' $\geq k$.

Proof for 3: G has no $I \geq k \Rightarrow$ G' has no clique of size $\geq k$

Contrapositive:

G' has a clique $\geq k \Rightarrow$ G has $I \geq k$

$\therefore A \leq_p B \equiv A \leq_K B$

If A CAN'T BE solved in poly, neither can B. If poly-time alg for B, then A can be solved in poly-time.

LEMMA: $A \leq_p B$ and $B \in P \Rightarrow A \in P$

NP-Hardness:

If we could find a problem H so every problem in NP reduces to H

Then if $H \in P$, we can say P = NP

A problem H is NP-Hard if $\forall A \in NP$: $A \leq_p H$

NP-Completeness:

Is there a problem in NP which is NP-hard?

Problem C is NP-Complete if $C \in NP$ and C is NP-hard

INDSET, LONGPATH, CLIQUE, HAMCYCLE, Decision Question of KNAPSACK

How to show that a Problem is NP-Complete?

1. Show $C \in NP$
2. Show $C \in$ NP-hard

But! NP-hard problems are transitive. Just show $C \leq_p H$, $\exists H \in$ NP-hard.

SHOW: If $A \leq_p B$ and $B \leq_p C$ then $A \leq_p C$

3-SAT PROBLEM — 3-CNF Formulas

CONJUCTIVE NORMAL FORM:

φ in CNF if $\varphi = C_1 \wedge C_2 \wedge \ldots C_m$

C_i is OR of one or more literals

k-CNF: k literals per C

3-CNF: $(x_1 \vee x_2 \vee x_3) \wedge (x_4 \vee x_5 \vee x_6)$

SATISFIABILITY:

$\varphi(x_1, \ldots, x_n)$ is boolean formula w/ $x_1 \ldots x_n$

φ is satisfiable if there is an assignment to $x_1, \ldots x_n$ s.t

$\varphi(x_1, \ldots x_n)$ is True

example:

$(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \ldots$

OTHER SAT PROBLEMS: NP-COMPLETES

Is 3-CNF satisfiable?

Circuit-SAT: boolean circuit on $x_1, \ldots x_n$

SAT: φ in CNF, is φ satisfiable?

Formula-SAT: given BF φ, is φ satisfiable

POLY TIME REDUCTIONS:

Suppose A solves X in O(1). Y is reduced to X in poly time $Y \leq_p X$, if any instance of Y can be solved by applying A.

Y reduced to X; $Y \longrightarrow X$; $Y \leq_p X$

if X solvable in poly, Y solvable in poly

3 SAT → INDSET : "REDUCTION FROM"

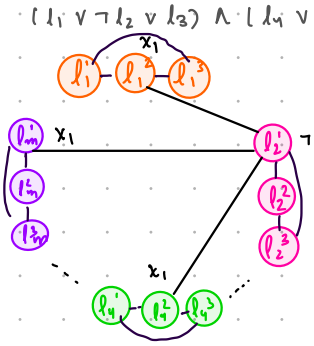
Reduce 3-SAT → INDSET

given φ, make G, k

φ is satisfiable iff G has $|INDSET| \geq k$

SET-UP:

$(l_1 \vee \neg l_2 \vee l_3) \wedge (l_4 \vee \neg l_1 \vee l_5)$



I. Valid Assignment Edges
- Edge between 2 negatives

II. Cover Each Clause:
- So that 1 literal is chosen from each clause s.t. our capacity k is met

PROOF:

1. Can compute in poly time
   $O(m^2)$: Valid Assignment (Each pair)
   ↓ where m is no of clauses
2. If φ is SAT, G has INDSET $\geq m$.
   Atleast 1 literal true in each clause, pick that vertex in G
   - No adjacent vertices since one per clause
   - can't pick x_i and $\neg x_i$ since can't be both T and F
3. If φ is NOT SAT, G doesn't has INDSET $\geq m$
   Contrapositive : If G has INDSET $\geq m$, φ is SAT

   for each vertex l_i^j in INDSET C
   Set l_i^j to be true
   If any variables are unset, set them to anything!

NP - COMPLETE : SET-COVERING PROBLEM

INPUT: List U of elements, list of sets $S_1 \dots S_n$ each subset of U, int k

OUTPUT: Is there $C \subseteq U$ of k elements $x_1 \dots x_k$ s.t. $\forall i : S_i \cap C \neq \emptyset$

IS THERE AN ELEMENT FROM EACH SUBSET IN C

VC → Set Covering Problem

$G = (V,E)$, k → $S_1, \dots, S_n, U, k$

$U = V$

$\{S_1, \dots, S_n\} = E$

NP - COMPLETE : SET PACKING PROBLEM

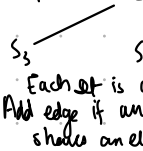
Input: List U of elements, list of sets $S_1 \dots S_n$ each subset of U, int k

DECISION: Is there a collection of k disjoint subsets $S_i \dots S_k$?

INDSET → Set Packing

Replace each vertex with a set and its neighbors

Set Packing → INDSET



Each set is a vertex
Add edge if any sets share an element

NP-COMPLETE : SUBSET SUM PROBLEM

Input: List of integers $A[1, \dots, n]$ and target integer t

Decision: Is there a set of indices $x_1, \dots, x_k$ s.t.

$$\sum_{i=1}^{k} A[x_i] = t$$

Challenge : Show Knapsack is NP-complete using this

NP-COMPLETE: PARTION PROBLEM

INPUT: List of integers $A[1, \dots n]$

Decision: Is there a partition of indices $[1, \dots n]$ into L, R s.t.

$$\sum_{i \in L} A[i] = \sum_{i \in R} A[i]$$

SUBSET SUM → PARTITION

Subset sums with $k = \sum_i A / 2$

GRAPH COLORING : NP-Complete

INPUT: $G(V,E)$ undirected & k

OUTPUT: Is there a proper* k-coloring of G

* Assignment of colors to V s.t. no adjacent vertices have same color

3-COLORING: NP-Complete $k = 3$

Challenge : 3-SAT → 3-colorings

PATH PROBLEMS:

WHICH ONES ARE EFFICIENTLY SOLVABLE ?

1. Connectivity
2. Cycle Detection
3. Longest / Shortest paths in DAGS (+ve / -ve)
4. Shortest path w/ non-negative edge weights
5. Shortest cycle w/ non-negative edge weights
6. DETECTING -ve WT CYCLES
7. Shortest paths w/o -ve WT CYCLES

PROBLEM W/ NON-NEGATIVE EDGE WEIGHTS

- -ve weight cycle → no shortest path
- Djikstra's relies on idea $s \to \dots \to u' \to u$
  $s \to u'$ is shorter than $s \to u$
  but NOT TRUE w/ NEG edge wts

BELLMAN-FORD : -VE EDGE CYCLES

Detects -ve weight cycles in $O(mn)$

Also finds shortest paths as long as no -ve wt cycle exists

FLOYD-WARSHALL : ALL PAIRS SHORTEST PATH

$O(n^3)$, finds shortest path b/w each pair of V

As long as no -ve wt cycles

RECURRENCE : Length of shortest path from $u \to v \leq k$ steps

OTHER QUESTIONS: w/ -ve wt cycles

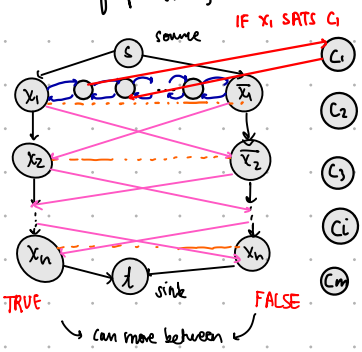
1. Shortest Simple paths in G
2. Least Weight Cycle in G
3. Maximum Wt Cycle } same as Min/shortest
4. Longest Path
5. Hamiltonian Cycle / Cycle (each ✓ once) Path
6. Travelling Salesman [min cost cycle visiting each ✓]

ALL NP-HARD

HAMILTONIAN PATH : NP COMPLETE PROOF

1. 3-SAT → HAMPATH 2^N paths

Take φ to G where G has graph iff φ is SAT

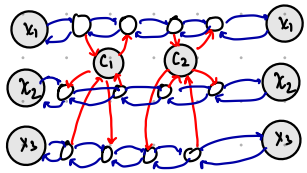


Path using all vertices will be ONE THAT $s \to t$ visiting x_i and $\overline{x_i}$

1. Visit x_i before $x_{i'}$ ≈ set x_i to TRUE
2. " $x_{i'}$ " x_i ≈ set x_i to FALSE

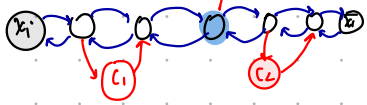
CLAUSE GADGET

If $C_1 = (x_1 \vee \overline{x_2} \vee x_3)$ &
$C_2 = (x_1 \vee x_2 \vee \overline{x_3})$



1. Can we make G in poly-time Y
2. If φ sat, will there a path through each v? Y
3. If there's a path through each v is V SAT?

BUFFER NODE B/W EACH CLAUSE



LINEAR PROGRAMMING:

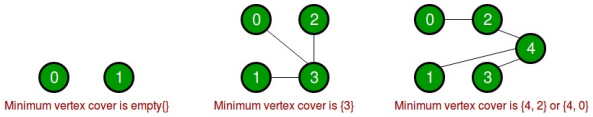
1. Constraints — LINEAR equalities / inequalities
2. Objective — also LINEAR, to MAX / MIN

REDUCE ALMOST 3-SAT TO 4-SAT

Almost 3-SAT $\leq_p$ 4-SA

VERTEX COVER:

Set $V' \subseteq V$ in $G(V, E)$ s.t.
for all $(u, v) \in E$, either u or $v \in V$.



VERTEX COVER → INDSET:

Make complement graph G' such that
$\forall\ v \in V'$ (vertex cover) and $\nexists (u, v) \in E$
with either u or v in $\in V'$,

$G'(V/V_c, E/E_c)$

PROBLEMS:

JUMP GAME : Can reach last index given max jumps from each position?

DP GREEDY

MIN JUMP GAME

DP GREEDY

MAXIMAL SQUARE

DOMINO TILING $n \times 2$ and $n \times 3$

MAX SUBARRAYSUM

MIN PATH SUM

COIN CHANGE

LIS → longest increasing subsequence

DP proof template

We claim that *opt[x]* correctly computes [description of subproblem with input *x*].

Base case: we initialized *opt[0]* to [some number], so our claim holds for $x = 0$.

Induction: assume we've computed *opt[0]*, *opt[1]*, ..., *opt[x-1]* correctly. [Do one of the following:

- Explain why our recurrence formula is correct
- Suppose *opt[x]* is not optimal. Then show a contradiction (e.g., show that this leads to the conclusion that *opt[x-1]* was not optimal, or show something impossible happened like we did on the previous slide)

Problem: Prove $\mathbf{Y} \leq_p \mathbf{X}$. Y Reducable to X

1. Start with an instance of **Y**, denoted y.
2. Construct an instance of **X**, denoted x.
3. Prove that x has a satisfying instance $\iff$ y has a satisfyin instance.
4. Conclusion: **X** is at least as "hard" as **Y**.
   If Y is NPC, X is NPC.