GRAPHS

A PATH in an undirected graph $G=(V,E)$ is a sequence of nodes $v_1, v_2, \dots, v_k$ with property that each consecutive path v_{i-1}, v_i is joined by an edge in E

A path is simple if all node are distincts,

A CYCLE is a path in which $v_1 : v_k$ $k>2$ and first $k-1$ nodes are all distincts

Def. If an undirected graph doesn't contain a cycle then it's a tree

G is connected $\iff$ doesn't contain a cycle $\iff$
$\iff$ G has $n-1$ edges

Connectivity & shortest path (a greedy approach)

→ BFS : breadth-first search
→ DFS is depht-first search (finds cycles)

Given two nodes $s \rightsquigarrow t$, they are connected if there exist a path from s to t.

↳ If there are mentiple paths then there is the problem of finding SHORTEST PATH (DIJKSTRA)

In a DIRECTED GRAPH we have STRONG CONNECTIVITY if, for every $u, v \in V$, they are mutually reachable. In general, it means that every node is reachable by every other node.

$$BFS(G) \quad + \quad BFS(G_{REVERSE})$$

Directed Acyclic Graphs

A DAG is a directed graph that contains no directed cycles.

↳ A topological order of a directed graph $G = (V, E)$ is an ordering of its nodes as $v_1, v_2, \dots, v_n$ so that every edge (v_i, v_j) we have $i < j$

if G has a topological order, then G is DAG

[pf. by contr.]

if G is a DAG, then G has a topological ordering

pf by induction

→ Base case: true for $n = 1$

- given DAG on $n > 1$ nodes, find node v with no entering edges (if it doesn't exist then not a dag)
- $G - \{v\}$ is a DAG

→ by inductive hypothesis $G - \{v\}$ has topological ordering

Place v first in topological ordering (since no edge enters v)

→ a topological order!

GREEDY ALGORITHM

DIJKSTRA

Given digraph $G=(V,E)$, edge lenghts $l_e \geq 0$ source $s \in V$ destination $t \in V$, find shortest path $s \rightsquigarrow t$

Greedy approach:

Mantain set of explored nodes S for which algorithm has determined the shortest path distance $d(u)$ from s to u

→ Initialize $S = \{s\}$ $d(s) = 0$

repeatedly choose next node from unexplored which minimizes [pf by $|S|$ induction]

$$\pi(v) = d(u) + l_{(u,v)}$$

with every iterations we'll always find an optimized

we can do efficient implementation based on remembering $\pi(v)$ and using a priority queue to choose unexplored nodes that minimize $\pi(v)$

CYCLES AND CUTS

- A path is sequence of edges which connects sequence of nodes
- A cycle is a path with no repeated nodes or edges other than the starting and ending nodes

A CUT is a partition of nodes into two non empty sets S, $V-S$

a CUTSET of a cut S is the set of edges with exactly one end point in S, basically contains edges that get out of set S

Prop. A cycle and a CUT interset in an even number of edges, since every edge "outgoing" from s eventually returns in S

MINIMUM SPANNING TREE

Let $T=(V,F)$ be subgraph of $G=(V,E)$ TFAE:

- T is SPANNING TREE of G
- T is acyclic and connected
- T has $n-1$ edges
- T is minimally connected, since removal of any edge disconnects it
- T is maximally acyclic, since addition of any edge creates cycle

Given connected graph $G=(V,E)$ with edge cost c_e, an MST is a subset of edges $F \subseteq E$ such that $T(V,F)$ is a spanning tree whose sum of edge cost is minimized

properties of SPANNING TREE

FUNDAMENTAL CYCLE

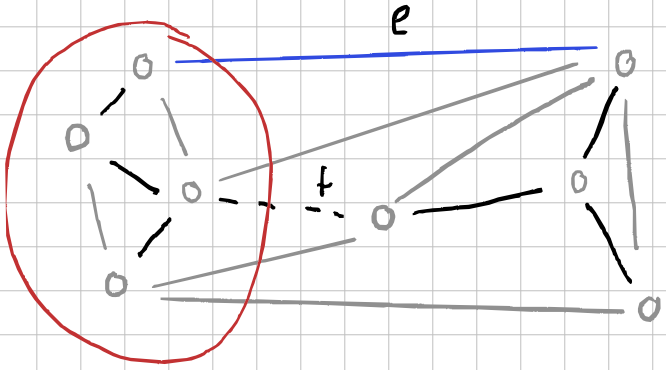
- adding any edge $e \in E \setminus F$ to a spanning tree forms a unique cycle C, and deleting any edge from the cycle creates another SPANNING TREE.

  if, given $e \in E \setminus F$ and $f \in F$, $c_e < c_f$, then the subgraph T is not a MST

FUNDAMENTAL CUTSET

deleting any tree edge f from a spanning tree T divides nodes into two connected components. Let D be cutset (set edges that depart from a cut)

↳ Adding any edge $e \in D$ to $T - \{f\}$ results in new spanning tree



if $c_e < c_f$ then T is not MST

RED Rule

- let C be a cycle with no red edges
- select uncolored edge of C of max weight and color it red

BLUE rule

- Let D be a cutset with no blue edges
- select an uncolored edge of D of min weight and color it blue

Greedy algorithm: Apply red and blue rules (non-deterministically!) until all edges are colored. Then blue edges form an MST

PRIM'S ALGORITHM

Initialize S to any node
Repeat $n-1$ times:
- Add to tree min weight edge with one endpoint in S (kind of applying blue rule)
- Add node to S

(we use just the blue rule)

It's very similar to Dijkstra algorithm, but we don't use $\pi[v]$

KRUSKAL ALGORITHM

Consider edges in ascending order of weight, we add them to tree unless they make a cycle

pf: we may have two cases:
- CASE 1: both endpoint in e in the same blue tree, then we color it red since it would create cycle
- CASE 2 both endpoints of e are in different blue tree we use cutset th. and we add them together

At the start every node forms a cut with single node, then we choose edges based on weight

↳ UNION-FIND data structure ↵

DYNAMIC PROGRAMMING

Break problem into subsets of overlapping subproblems and build up solution to larger subproblem.

We'll use table to cache results!

Weighted interval scheduling

- job j starts at s_j, finishes at f_j and has weight of value v_j
- two jobs are compatible if they don't overlap
- find maximum weight subset of MUTUALLY COMPATIBLE jobs

Greedy algorithms won't work anymore!

We'll need to specify subproblem first!

$OPT(j)$ = value of optimal solution consisting of jobs $1, 2, \ldots, j$

base case $OPT(j) = 0$ if $j = 0$
else we need to find next compatible job $p(j)$

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

we choose between...

- the optimal job adding this to subset and running algs on next compatible jobs.
- seeing the optimal job for the case before j

observation in this case we have binary choice, which is computationally intensive! We do redundant subproblems n time

↳ $O(2^n)$ complexity

we store already calculated values of optimal solution on table.

$M[j] = OPT(j)$

if $M[j]$ not calculated

$M[j] = \max\{v_j + \text{compute\_opt}(M[p(j)]), \text{compute\_opt}(M[$

else

return $M[j]$

in $M[j]$ we have the VALUE of the optimal solution, now what?

we do TOP-down strategy!

```
solution = ∅
M
i = N
while i > 0
    if  v_i + M[p[i]] > M[i-1]
        solution = solution ∪ {i}
        i = p(i)
        continue
    else
        i = i - 1
```

$O(n)$ + $O(n)$ = $O(n)$

↓ Compute_opt

↓ compute_solution

Least squares

linear regression problem: given n points (x_1, y_1), $(x_2, y_2) \dots (x_n, y_n)$ find line $y = ax + b$ such that

$$SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2 \quad \text{is minimized.}$$

there is a solution

$$a = \frac{n \sum_i x_i y_i - \sum_i x_i \sum_i y_i}{n \sum_i x_i^2 - (\sum x_i)^2} \qquad b = \frac{\sum y_i - a \sum x_i}{n}$$

segmented least squares

in the case of multiple points, we may see that having multiple lines can better represent them.

↳ we want to balance accuracy and precision

$$f(x) = E + cL$$

E ↓ sums of SSE

L ↓ number of lines.

we in this case too define $OPT(j)$

$OPT(j)$ = minimum cost for points $p_1 \dots p_j$

$e(i,j)$ we'll have minimum sums of squares from $p_i \dots p_j$

$$OPT(j) = \begin{cases} 0 & j = 0 \\ \min_{1 \le i \le j} \{ e(i,j) + c\, OPT(i-1) \} & \text{else} \end{cases}$$

In questo caso ho un mini ciclo for per calcolare tutto e selezionare il minimo risultato tra tutti

dinamic programm algorithm takes $O(n^3)$ time
$O(n^2)$ space

computationally intensive to calculate $e(i,j)$

KNAPSACK

- given n objects and a knapsack
- every item i has weight $w_i > 0$ and value $v_i > 0$
- knapsack has max weight W

fill knapsack so that it maximises value

No greedy will help with this

we may try to do dynamic programming with

$OPT(i)$ = the max profit subset of items $1 \dots i$

but that wouldn't work because selecting i doesn't imply rejecting another item previously inserted.

we need

$OPT(w, i)$ = the max profit subset of items $1 \dots i$ for max weight w

this means

$$OPT(w,i) = \begin{cases} 0 & i = 0 \\ OPT(w, i-1) & w_i > w \\ \max\left[v_i + OPT(w-w_i, i),\ OPT(w, i-1)\right] & \end{cases}$$

- $v_i + OPT(w-w_i, i)$: choose it & update max weight
- $OPT(w, i-1)$: try next item $i-1$

Running Time

- Takes $O(1)$ per table entry
- $\Theta(nW)$ table entries
- To get solution we get top down strategy: take item i iff ~~OPT~~$(i, w) > t[i+1, w]$

SUMMARY

Outline

- polynomial number of subproblems
- solution to problem can be computed by subproblems
- Natural ordering from smallest to largest

Techniques

- BINARY CHOICE
- MULTIWAY CHOICE
- VARIABLE ADDING
- DYNAMIC PROGRAMMING OVER INTERVALS

Top down - vs bottom up are valid strategies

SEQUENCE ALIGNMENT

Design algorithm that, given two strings, tells us back the "similarity" of two strings, finding the min cost alignment.

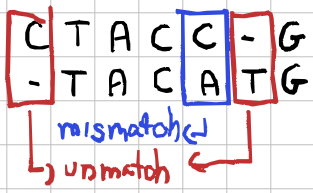
AN ALIGNMENT M is set of ordered pair $x_i - y_j$ such that each item occurs in at most one pair and no crossing

↳ $x_i - y_j$ and $x_{i'} - y_{j'}$ cross if $i < i'$ but $j > j'$

the cost of alignment is

$$\text{COST}(M) = \sum_{(x_i, y_j) \in M} \alpha_{x_i y_j} + \sum_{i : x_i \text{ unmatch}} \delta_i + \sum_{j : y_j \text{ unmatched}} \delta_j$$

Ex



we can define subproblem

OPT(i,j) : min cost of alignment $x_1, \dots, x_i$ and $y_1, \dots, y_j$

$$\text{OPT}(i,j) = \begin{cases} j\delta & i = 0 \\ i\delta & j = 0 \\ \min \begin{cases} \alpha_{x_i y_j} + \text{OPT}(i-1, j-1) & \text{alignment cost} \\ \delta + \text{OPT}(i-1, j) & \text{gaps cost} \\ \delta + \text{OPT}(i, j-1) & \end{cases} \end{cases}$$

this has $O(nm)$ time cost and $O(n \cdot m)$ space ∴ Space cost can be optimized!

Hirschberg algorithm

It's based on a mix of divide-and-conquer and dynamic programming.
Uses a graph to visualize the problem, and modifies it to use $f(i,j)$ the shortest path from $(0,0)$ to (i,j).

$$OPT(i,j) = f(i,j) \quad \forall\, i,j \quad i \le n,\ j \le m$$

pf. by induction

- base case $OPT(0,0) = f(0,0)$ □
- let's assume this true for all $i'+j' < i+j$
- Last edge on shortest path may have come from $[i-1, j-1]$, $[i-1, j]$, $[i, j-1]$

- $f(i,j) = \min\left\{a_{x_i y_j} + f(i-1, j-1),\ \delta + f(i-1, j),\ \delta + f(i, j-1)\right\}$

$= \min\left\{a_{x_i y_j} + OPT(i-1, j-1),\ \delta + OPT(i-1, j) + \delta + OPT(i, j-1)\right\}$

$= OPT(i,j)$ □

Let $f(i,j)$ shortest path $= OPT(i,j)$,

- we can compute $f(\cdot,j)$ for any j in $O(mn)$ time and $O(n+m)$ space
- Let $g(i,j)$ the shortest path from (i,j) to (m,n) we can compute that by reversing the graph, and inverting roles of $(0,0)$ and (m,n)

cost of shortest path is $f(i,j) + g(i,j)$

Let q be an index that minimizes $f(q, n/2) + g(q, n/2)$
then, there exist path from $(0,0)$ to (i,j) uses $(q, n/2)$

This becomes a divide and conquer algorithm
DIVIDE : find q that minimizes $f(q,n/2) + g(q,n/2)$, align x_q, $y_{n/2}$
CONQUER recursively compute alignment optimally

Let $T(m,n)$ max running time of Hirschberg's algorithm on strings of lenght at most n, m

$$T(m,n) \leq 2T(m, n/2) + O(mn)$$
$$= O(mn \log n)$$

Bellman - Ford

Used for finding shortest path in the case in which we remove assumption of edges with non negative weights

Dijkstra can fail, even if we add constant to make all numbers positive

Def: Negative cycle is a directed cycle in which the sum of its edge weights is negative

if it exist a negative cycle in a path $s \rightsquigarrow t$, then there can not be a cheapest path, since following the cycle will always be more convenient

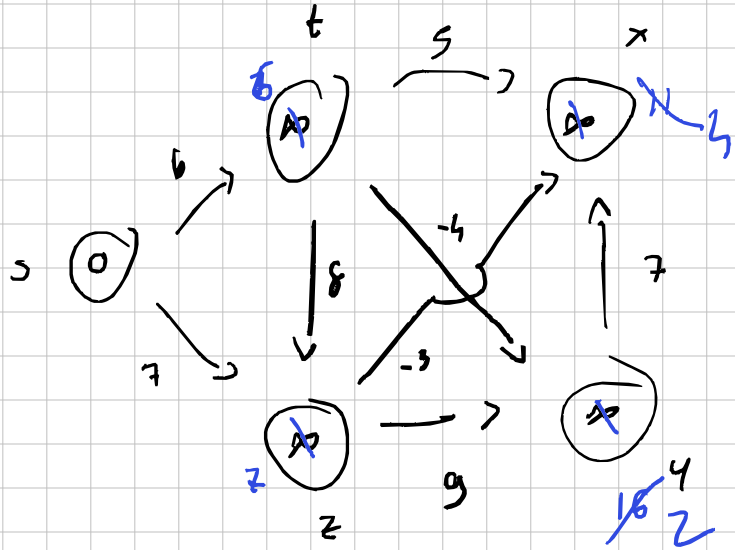
Lemma: If g has no negative cycles then there exists a cheapest path $s \rightsquigarrow t$ with at most $n-1$ edges.

$OPT(i,v)$ = cheapest path from v to t that uses $\leq i$ edges.

2 cases

- OPT uses $i-1$ edges $\quad OPT(i-1, v)$
- opt uses exactly i edges, then we select other best edges with minimal cost

$$OPT(i,v) = \begin{cases} \infty & i = 0 \\ \min\left\{ OPT(i-1,v),\ \min_{(v,w)\in E}\left\{ OPT(i-1,w) + c_{vw} \right\} \right\} \end{cases}$$



the algorith iterates N-1 times, since at worst a path contains N-1 edges, so it may have N-1 relaxations possible.

↳ we can detect a negative cycle if, after N-1 th relaxation, we have another shortest path

Computational cost

Since we have two parameters, the space cost of the table is $O(n^2)$, and time cost is $O(nm)$

↳ HIGH

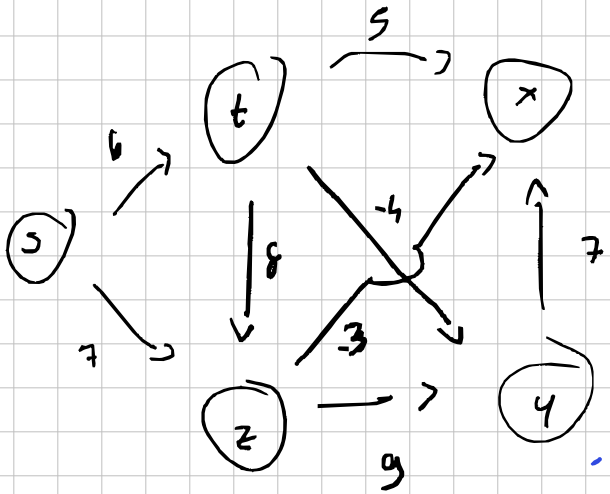
optimization: maintain a successor (i,v) that points to next node on cheapest path, and compute optimal costs, considering only edge with

$M[i,v] = M[i-1,w] + c_{vw}$

⇒ maintain 2 1d-array instead of 2d array:

$d(v)$ = cost of path $v \to t$ (cheapest so far)

successor(v) = next node on $v \to t$ path



	s	t	x	y	z
d(v) =	0	∞	∞	∞	∞
=	0	6	∞	∞	7
=	0	6	4	2	7

successor(v) =	/	s	z	t	s

Bellman-Ford $(V, E, s \in V)$

$\forall v \in V \mid v \neq s$
  $d[v] = inf$
$d(s) = 0$

for $|V| - 1$ times do
  for each edge (u, v) with w in edges do
    if $d[v] > d[u] + c(u, v)$
      $d[v] = d[u] + c(u, v)$
      $successor[u] = v$

with these two tables it's easy to do tree of shortest path

Distance Vector Protocol

- Node ≈ router
- Edge ≈ link
- cost of edge ≈ link delay

delay always ≥ 0, but we use Bellman-Ford because it uses information from only neighbouring node

Each router maintains a vector of shortest path lenghts to every other node, and hop of each path.

↳ Gets regularly updated, since cost may change or full-on disconnect

Link State Routing

- Each router stores entire path
  → based on Dijkstra's algorithm

Used to connect a small part of neighbouring routers.

↳ Border routers use BGP (distance vector) to connect to other neighbouring nets, while we use OSPF for net.

BGP	OSPF
Border Gateway Protocol	Open Shortest Path First

CHECKING FOR NEGATIVE CYCLES

if $OPT(n, v) = OPT(n-1, v)$ $\forall v$, then no negative cycles can reach t, else, if

$OPT(n-1, v) > OPT(n, v)$ for some node v, then any cheapest path v to w contains a cycle W, which is a negative cycle

we can prove it by contradiction
- since $OPT(n, v) < OPT(n-1, v)$, we know that shortest path P has n edges
- pigeonhole principle, P contains a cycle W
- deleting W yields a $v \to t$ path with $\leq n$ edges =) W has a negative cost.

How to find cycle?

- add node t with 0-weight to all nodes
- G has negative cycle iff G' has negative cycle that reaches t
- if $OPT(n, v) = OPT(n-1, v)$ $\forall v$ then no negative cycle
- if NOT extract directed cycle from path to v to t (doesn't contain t since t doesn't have leaving edge)

(this takes $O(n^2)$ space! inefficient :) we use 2 -1d arrays)

we iterate for n passes (instead of $n-1$), if we have update at n pass (ex $d(s)$) we define $pass(v)$ last pass updated.

Observe $pass(s) = n$ and $pass(successor(v)) \geq pass(v) - 1$ following successor pointers will repeat node.

That is a cycle!

NETWORK FLOW I

Max-flow & Min-cut problem

A network may be abstracted by a graph $G = (V, E)$, and with material flowing between edges.
Each edge will have a maximum capacity inherent to it
$c(e) > 0 \quad \forall e \in E$

↳ We'll have a source $s \in V$ and a sink $t \in V$

an **st-cut** is a partition (A, B) of the vertices with $s \in A, t \in B$
the **capacity** is the sum of the capacities of the edges from A to B

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$

min-cut problem find the cut with minimum capacity

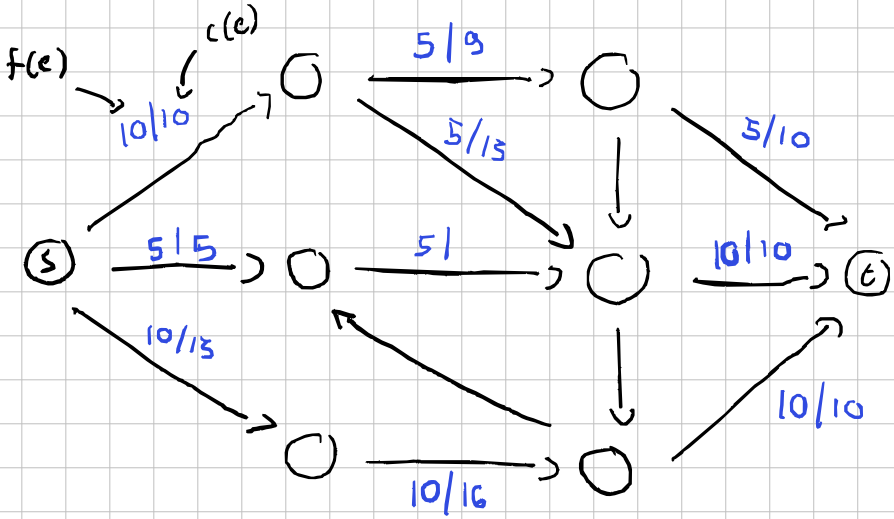
we'll define a **st-flow** as a function f that satisfies:

- $\forall e \in E \quad 0 \leq f(e) \leq c(e)$ [capacity]
- $\forall v \in V - \{s, t\} \quad \sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

The **value** of a **flow** is

$$val(f) = \sum_{e \text{ out of } s} f(e)$$

max flow problem find the maximum flow value such that f follows rule



$$\text{val}(f) = \sum_{e \text{ out } s} f(e) = \sum_{e \text{ on } t} f(e) = 25$$

greedy algorithm

- Start with $f(e)$ $\forall e \in E$
- find an s-t path where $f(e) < c(e)$ $\forall e \in P (s \rightsquigarrow t)$
- augment flow along P
- repeat until we get stuck

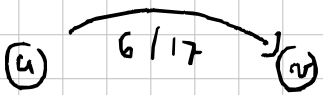
works, although bit slow

residual graph

we eliminate the need of storing two values for each edge

ORIGINAL EDGE

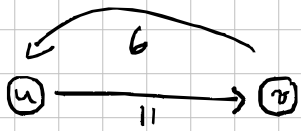
$e = (u, v) \in E$
- flow $f(e)$
- capacity $c(e)$



RESIDUAL EDGE

we "undo" the flow sent
$e = (u, v)$, $e^R = (v, u)$
residual capacity

$$c_f(e) = \begin{cases} c(e) - f(e) & e \in E \\ f(e) & e^R \in E \end{cases}$$



RESIDUAL GRAPH : $G_f : (V, E_f)$
- we have residual edges with positive residual capacity
- $E_f = \{ e : f(e) < c(e) \} \cup \{ e^R : f(e) > 0 \}$

as a key property, we have that f' is a flow in G_f iff $f + f'$ is a flow in G. [we verify flow property]

This creates the possibility of, given a flow, is to find the bottleneck capacity of augmenting path P (path in residual graph G_f), which is the minimum residual capacity of any edge in P

prop: let f be a flow and let P be an augmenting path in G_f, then f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$

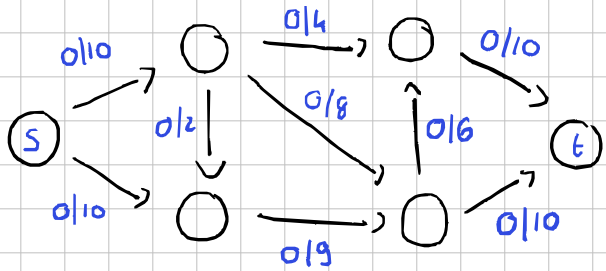
```
Augment (f, c, P)
    b ← find_bottleneck (P)
    for each edge e ∈ P
      if (e ∈ E) : f(e) ← f(e) + b
      else         f(e^R) ← f(e^R) - b
    return f
```

an augmenting path is a $s \rightsquigarrow t$ path in the residual edges E^R

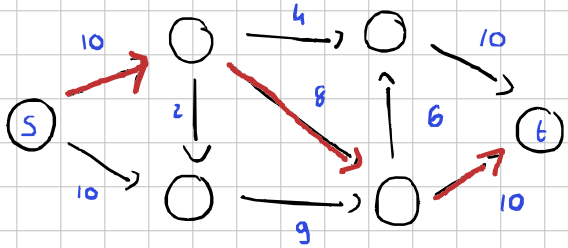
the goal here is to "saturate" a possible path!

Ex

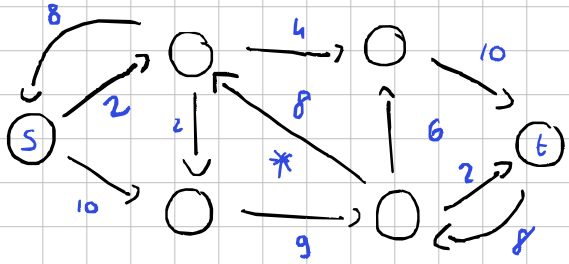
graph:



residual graph



augment $(p, b=8, G_f, c)$



* Notice in here we don't have anymore the residual edge, since $c(e) = f(e)$

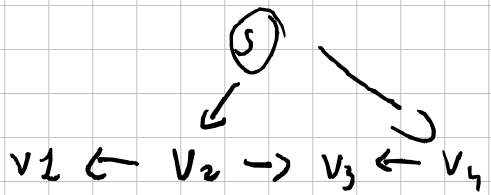
FORD-FULKERSON ALGORITHM

depends heavily on residual graph and augmenting paths

start with $f(e)$ $\forall e \in E$
find augmentation path P in G_f
augment flow among P
repeat until stuck

```
Ford-Fulkerson (G, s, t, c)
  FOREACH edge e ∈ E: f(e) = 0
    Gf = residual graph
    while exists an augmenting path P in Gf ←
      f ← augment(f, c, P)
      update Gf
  return f
```

EXAMPLE on 07NetworkFlow I - 20 / 27



let f be any flow and let (A,B) be any cut. The net flow across (A,B) equals the value of f

$$\sum_{e \text{ in } A} f(e) - \sum_{e \text{ in } B} f(e) = val(f)$$

this is true because of the flow conservation lemma

weak duality

let f be any flow, and let (A,B) be any partition, then $v(f) \leq cap(A,B)$

pf. $$v(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in } A} f(e)$$

$$\leq \sum_{e \text{ into } A} f(e)$$

$$\leq \sum_{e \text{ out of } A} f(e)$$

$$= cap(A,B)$$

the value of the flow can't be greater than what the capacity of $A = [s]$ tells us.

At the optimal value, $v(f) = cap(A,B)$

Augmenting path a flow is max flow iff no augmenting paths

max-flow min-cut theorem value of the max flow = capacity of min-cut

Proof: 3 conditions are equivalent for any flow f

I there exist a cut (A,B) such that $cap(A,B) = val(f)$
II f is a max flow
III there is no augmenting path with respect to f

a $(I \to II)$ Suppose (A,B) cut that $cap(A,B) = val(f)$
for any flow f' $val(f') \leq cap(A,B) = val(f)$
f is max flow! ↳ weak duality

b $(II \to III)$ we want to prove contrary $= !III = !II$.
suppose there is augmenting path in f, then we can augment and have a flow better.
So II not true

c $(III \to I)$
let f be a flow with no augmenting paths. Let A be set of nodes reachable from s in residual graph G_f
- $s \in A$ definition of A
- $t \notin A$ by definition of flow

$$val(f) = \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ into } A} f(e)$$

max-flow min-cut theorem ↳

$$= \sum_{e \text{ out of } A} c(e)$$

$$= cap(A,B)$$

CAPACITY SCALING ALGORITHM

we assume that capacities are integers between 1 and C, and flow values $f(e)$ and $c_f(e)$ remain integers.

The algorithm terminates in at most $val(f^*) \leq nC$ iterations, since every augmentation increases value by at least 1

Ford Fulkerson Time = $O(nmC)$

The efficiency of the algorithm relies heavily on choosing a good augmenting path! Some choices lead to exponential times.

Intuition we'll choose augmenting path with highest bottleneck capacity. We'll maintain a scaling parameter Δ, with which we'll create $G_f(\Delta)$ with edges with capacity $\geq \Delta$

Capacity scaling (G, s, t, c)

for each edge $e \in E$ $f(e) = 0$
Δ = largest power of $2 \leq C$

while $(\Delta \geq 1)$
  $G_f(\Delta)$ residual Δ-graph
  while (exist augmenting path P in $G_f(\Delta)$)
    $f \leftarrow$ augment (f, c, P)
    update (G_f)
  $\Delta = \Delta/2$

return f

if capacity-scaling algorithm terminates, then f is max flow

pf:
if $\Delta = 1$ $G_f(\Delta) = G_f$
when we augment $\Delta = 1$ then we have found max flow for ford fulkerson

algorithm runs in $O(m^2 \log C)$

Shortest Augmenting Path

We want to choose an augmenting path such that we have the fewest number of edges. → BFS

if we use this, throughout the algorithm the length of the shortest path never decreases, it actually strictly increases!

th. Shortest augmenting path algorithm runs in $O(m^2 n)$ time:
- → $O(m+n)$ for BFS
- → $O(m)$ augmentations for length k
- → If there is an augmenting path, there is a simple one
  $O(mn)$ augmentations

Shortest-augmenting path (G, s, t, c)

```
for each edge e∈E : f(e) = 0
  G_f residual graph
  while ( Exist path augmenting in G_f)
      P ← BFS(G_f, s, t)
      f ← augment(f, c, P)
return f
```

Not bad!

UNIT CAPACITY

In the specific case in which $C = 1$ we can do optimizations
A network is a unit-capacity simple network if.

- Every edge capacity is 1
- Every node has at most 1 entering or 1 leaving edge

if G is unit-capacity network and f a 0-1 flow,
then G_f is unit capacity simple network

↳ shortest augmenting path computes alg. in
$O(mn^{1/2})$

Phases of normal augmentations:

- explicitly maintain Level graph L_G
- start at s, advance along L_G until t or stuck
- if reaching t, augment and update L_G (delete all edges a path chosen)
- if gets stuck, delete node from L_G and goto previous node

We can associate problems to unit-capacity simple network.
Bipartite matching

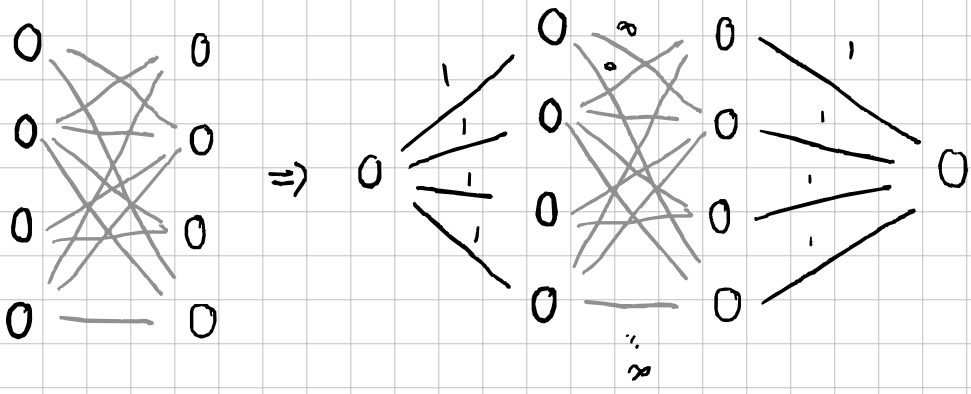
BIPARTITE MATCHING

Simplified version of a NP problem, that says given an undirected graph $G=(V,E)$ a subset of edges $M \subseteq E$ is a matching if each node appears in AT most one edge in M

↳ we want to find a MAX CARDINALITY MATCHING

given difficulty of problem, we can simplify it by considering a BIPARTITE GRAPH (which is a graph that can be partitioned into subsets L and R such that every edge connects a node in L to one in R

We can model the problem, reducing it to a max flow problem



We created a directed graph in which
- → edges in bipartite graph go from L to R with weight ∞
- → we create a node s and t
- → we connect s to every $l \in L$ with unit capacity
- → we connect every $r \in R$ to t with unit capacity

if problem in diagraph is solvable, and $val(f) = |L|$ then it's solvable for original problem

max cardinality of matching G = value of max flow in G'

pf: (assume max matching M: k in G)
- → we consider flow on every edge departing from s
- → we send 1 unit along every path k
- → f max flow with value k

pf. assume val max flow = G
- → let f be max flow in G'
- → integrality theorem → k integral and can be {0-1}, so edges are selected (matching) or not.
- → if we choose edge (u, v), then ∄ (u, v') since flow is 0.

Edge-disjoint Paths

two paths are edge-disjoint if they have no edge in common.
we may want to find the max number of edge-disjoint paths that respect requirements.

↳ Possible by assigning u.t. capacity to edges and resolve max flow problem

PF:

max number edge disjoint path $\geq$ value of max flow

pf ($\leq$)
- suppose max flow value is k
  for integrality theorem there exist 0,1 flow of value k
- consider edge (s,u) with $f=1$
  → conservation th. exist edge (u,v) $f=1$
  → continue until finding t
- Produces k edge-disjoint path.

pf ($\geq$)
- suppose that there is k Edge disjoint path.

⋮

You can create an edge-disjoint path from bipartite matching by applying sort

In case of undirected graph we can replace edge with equivalent antiparallel edges

CIRCULATION

It's a natural extension of max flow problems.

Def: Given digraph $G=(V,E)$ with nonnegative edge capacities $c(e)$ and node supply with demands $d(v)$, a **CIRCULATION** : function that satisfies:

- for each edge $e \in E$ $\quad 0 \le f(e) \le c(e)$ [capacity]
- for each vertex $v \in V$ $\quad \sum_{e \text{ in } v} f(e) - \sum_{e \text{ out } v} f(e) = d(v)$ [conservation]

we'll call vertex with $d(v) \neq 0$ with:

- **SUPPLY** if $d(v) < 0$
- **DEMAND** if $d(v) > 0$

if $\forall e \in E$ & $\forall v \in V$ $\quad c(e) \in \mathbb{Z}$, $d(v) \in \mathbb{Z}$, then if exists a circulation that will be integer.

given (V,E,c,d) there doesn't exist a circulation iff there exist a node partition (A,B) in which $\sum_{v \in B} d(v) > cap(A,B)$

we can model circulation as a max-flow problem:

we can model lower bounds as circulation with demands.
→ send $\ell(e)$ units of flow along edge e)
update demands of both endpoints

Th. exists circulation in G iff there exists circulation in G'

Extension to Max Flow

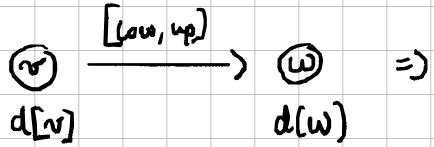
We may have multiple demand and supply nodes!

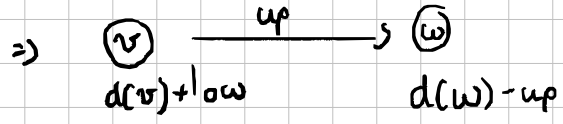
demand has $d(v) > 0$
supply has $d(v) < 0$
transshipment node has $d(v) = 0$
} $\forall v \in V$

We may find optimal flow by adding one source and one sink, and connect

$\forall v \in$ demand $\rightarrow$ t
$\forall r \in$ supply $\rightarrow$ s

We may have a lower / upper bound on edges flow.
We will change the $(v, w) \in$
$\rightarrow$ $d(v)$ + Lower
$d(w)$ - Lower





survey design

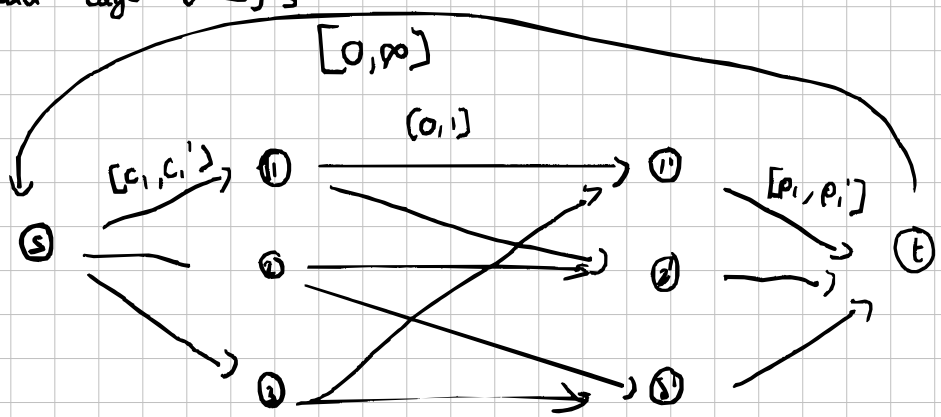
- design survey asking n_1 consumers about n_2 products
- can ask consumer i about product j if they own it
- can ask consumer i, between c_i and c_i' questions
- ask between p_j and p_j' about product j

→ design survey that meets these specs

Bipartite perfect Matching

- add edge (i,j) if c_j owns prod i
- add edge from s to consumer j
- add edge from product i to t

add edge $t \to s$



Airline Scheduling

Provide scheduling that is efficient in terms of
- equipment, crew, satisfaction
- weather, breakdown

Assigning crew to flights
→ input set of n flights for given day
flight i: leaves origin o_i at time s_i
arrives at destination d_i at time f_i

minimize number of flight crews

Circulation formation

- for each flight i, add nodes v_i and v_i'
- add source s with demand $-c$ and edges (s, u_i) with cap 1
- add sink t with demand c and edges (v_i', c) with cap = 1
- $\forall i$ add edge (u_i, v_i) with lower bound and cap = 1
- if flight j reachable from i, add edge (v_i', u_j) with cap = 1

we need to guess the amount of crews c, and see if its possible to find a value that satisfies circulations.

$O(n)$ nodes $O(n^2)$ edges
at most n crews needed.
can be found in $O(n^3 \log n)$ time
binary search

value of flow between 0 and n (at most n augmentations)

IMAGE SEGMENTATION

given an image, separate them from background.

Will label them with their probability of being a background!

- V = set of pixels
- E = pairs of neighbouring pixels
- $a_i \geq 0$ probability of foreground
- $b_i \geq 0$ probability of background
- $p_{ij} \geq 0$ separation penalty for labeling i as fg and j as bg or viceversa

A pixels in foreground
B pixels in background

we need to find partition (A, B) that maximising

$$\sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}$$

Maximising that its equal to MINIMIZING

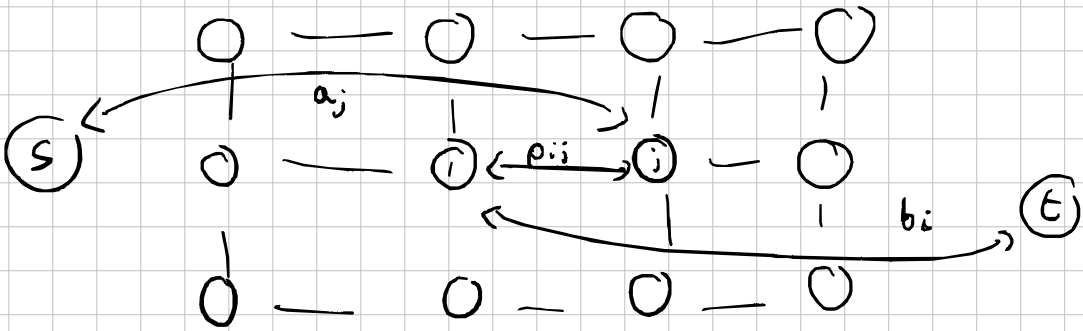
$$\left(\sum_{i \in V} a_i + \sum_{j \in V} b_j \right) - \left(\sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij} \right)$$

$$= \sum_{j \in B} a_j + \sum_{i \in A} b_i + \sum_{\substack{(i,j) \in E \\ |A \cap \{i,j\}| = 1}} p_{ij}$$

- { indirected graph + min cut }

can be formulated as a min-cut problem

- Include node for each pixel
- antiparallel edges
- add source s to correspond to foreground
- add sink t to correspond to background



Project Selection

positive or negative

- possible projects P: project v has associated profit p_v
  → Prerequisites: if $(v,w) \in E$, cannot do project v unless doing also project w

A subset of projects $A \subseteq P$ feasable if every prerequisite ~~that~~ of every project in A is also in A

[Circulation]

Assign capacity ∞ to every prerequisite edge
- Add edge (s,v) with capacity p_v if $p_v > 0$
- Add edge (v,t) with capacity $-p_v$ if $p_v < 0$

$p_s = p_v = 0$

Find min-cut → minimum capacity equals to max profit
- Infinite capacities ensures that $A - \{s\}$ feasable
- max revenue is

$$\text{cap}(A,B) = \sum_{v \in A : p_v > 0} p_v + \sum_{v \in B : p_v < 0} -p_v$$

$$= \underbrace{\sum_{v : p_v > 0} p_v}_{\text{constant}} - \sum_{v \in A} p_v$$

Class of Problems

We are dealing with two classes of problems even though there are many more:

$P = \{$problems solvable in polynomial time$\}$

$NP = \{$decisional problem solvable in nondeterministic polynomial time, or also that there exist an efficient certifier for it$\}$

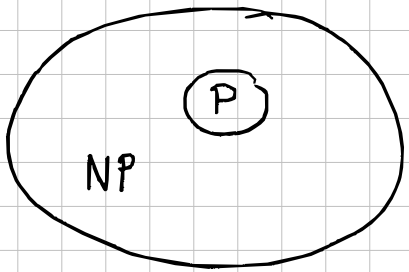
An efficient certifier $B(s,t)$ is an algorithm that
- it runs in polytime on $|s|, |t|$
- There exist a polytime function $p(\cdot)$ s.t. for we have $s \in X$ iff: $\exists$ a string t (certificate) s.t. $|t| \leq p(|s|)$ and $B(s,t) =$ "yes"
  - ↳ t should be polynomial on the input problem.

s is the INSTANCE of our problem X and t is a solution.

Th: $P \subseteq NP$

Pf: there exist a polytime algorithm A that answers the problem. A certificate possible could be

$$B(s,t) = A(s)$$



What problems are in $NP \setminus P$

↳ We don't know!

$P \overset{??}{=} NP$

NP-completeness

A NP-complete problem is a problem that "it's as hard" as all the problems in NP.
X is NP-complete if

- $X \in NP$ (certificate + verifier)
- $\forall Y \in NP \quad Y \leq_P X$ (NP-hard)

this can mean that if $X \in P$, then $P = NP$

if X is NP-complete, then we can say that:

- Z is NP-complete iff $X \leq_P Z$

(hard part is finding the first NP-complete problem)

Reductions

We saw that we can reduce a problem A to a problem B if there exist a poly time algorithm that converts ANY instance of problem A to a problem B.

We can say that A is "as hard" as B

The NP-completeness is useful, because if we find a NP-complete problem that is in class P, then P = NP

3-SAT

It's one of the first NP-complete problems found. Difficult part is proving that every NP problem is reducible to 3 SAT!

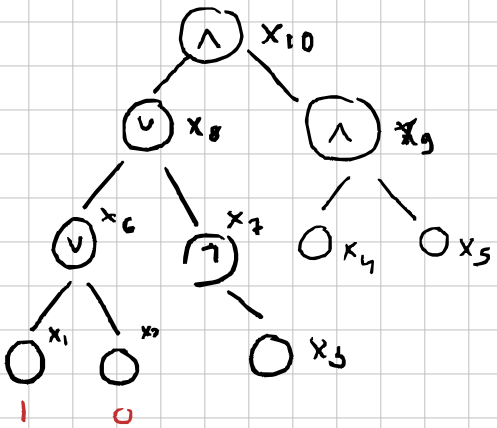
Given $X \in NP$, there exist verifier algorithm. An algorithm is just a sequence of logic gates AND, OR, NOT.

$\forall X \in NP \quad X \leq_P$ Circuit satisfiability

We'll prove that 3-SAT is NP-complete

Pf.
1) 3-SAT in NP
2) Circuit satisfyability $\in_P$ 3-SAT

I can reduce 3-sat to circuit satisfyability.



$1 = \{x_1\}$
$0 = \{x_2\}$

we need to create a formula:

I create a variable for every node. Insert clauses that force the variables to implement corresponding operations.

NOT: $(x_7 \vee x_3), (\bar{x}_7 \vee \bar{x}_3)$
$x_3 = 1 \Rightarrow x_7 = 0$

AND: 3 clauses
$(\bar{x}_9 \vee x_4)\ (\bar{x}_9 \vee x_5)\ (x_9 \vee \bar{x}_4 \vee \bar{x}_5$

OR: 3 clauses
$(x_6 \vee x_8)\ (x_7 \vee \bar{x}_8)\ (\bar{x}_6 \vee \bar{x}_7 \vee x_8)$

we need to find values for x_3, x_4 and x_5 that satisfy the assignment such that $x_{10} = 1$

We need to make every clause of lenght 3: will add variables z_i and clauses such $z_i = 0 \ \forall i$

3-DIMENTIONAL MATCHING

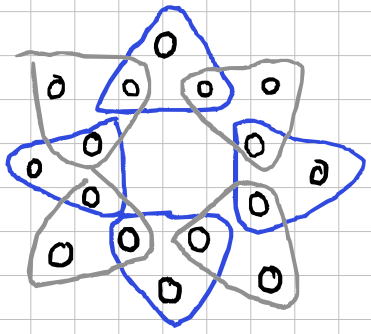
(3D-Matching)

Given disjoint sets X, Y, Z, each size n. Given triples $T \subseteq X \times Y \times Z$.
The goal is to choose subset of triples S such that EVERY element $e \in X \cup Y \cup Z$ appears in only one triple $s \in S$.

Th: 3D-matching is NP-complete
Pf: 1) 3D-matching $\in$ NP
(easy)

2) We'll reduce 3-SAT to this $3\text{-SAT} \leq_p$ 3D-matching
given an instance $x \in$ 3-SAT with n variables and k clauses we'll add $2nk$ elements $\forall x_i$

- we can encode all variables in GADGET



we'll say, for every x_i, that:
- $x_i = 1$ iff the even tips are free
- $x_i = 0$ elsewhere

Each CORE element will be COVERED once

we'll have a tip for every clause

- we'll create CLAUSE GADGETS. For every variable x_j in our clause C_i we'll create 2 elements p_i, p_i' an we will connect them to every tip b_j, z_i (according if x_j or $\bar{x}_j$)

we'll need all p_i elements to be covered.

- we'll create CLEANUP GADGETS that will cover all of tips that aren't covered by clauses

Most NP reductions are done with 3-SAT!

SUBSET-SUM: given n integers $A = \{a_1, a_2, \ldots a_n\}$ and given target T, is there subset $S \in A$ such that $\sum_{s \in S} s = T$

Th: subset-sum is NP-complete

PF: 1. s.s. $\in$ NP

2. 3D-matching $\leq_p$ subset sum

try to map 3d matching to subset sum. We'll assume, for simplicity, that $n = 3$

$X = \{x_1, x_2, \ldots, x_n\}$
$Y = \{y_1, y_2, \ldots, y_n\}$
$Z = \{z_1, z_2, \ldots, z_n\}$

$S \subseteq X \times Y \times Z$
$(x_i, y_j, z_k) \in S$
$|S| = n$

I will use decimal numbers with $3n$ digits.

Ex $n = 4$

sets =

	x_1	x_2	x_3	x_4	y_1	y_2	y_3	y_4	z_1	z_2	z_3	z_4	
S_1													$\to 10^{11} + 10^5 + 10^0 = w_1$
S_2													$\to 10^{10} + 10^7 + x^2 = w_2$
S_3													
S_4													
S_5													
S_6													
S_7													
S_8													
S_9													$\to 10^9 + 10^6 + 10^1 = w_9$
W													$\to 10^{12} - 1 = W$

I want to select n sets such that, cover all elements

TRAVELING SALESMAN PROBLEM

Input: n sites $v_1, \dots, v_n$

$\forall (v_i, v_j)$: here nonnegative distance $d(v_i, v_j)$

$d(v_i, v_j) \neq d(v_j, v_i)$

Given instance of the problem, and a budget D is there a tour that starts and finishes on v visits all the sites and has lenght $\leq D$?

HAMILTONIAN CYCLES

Given a directed graph, does it have a simple cycle that visits all the nodes?

3-SAT $\leq_p$ hamiltonian cycles

we'll create directed st graph with n. levels, one for each variable and 2nk nodes for each level.

clause gadgets. if $x_i = 1$ we'll add node v and 2 edges $(v_{i,2j}, u)$ $(u, v_{i,2j+1})$ and $x_i = 0$ contrary left to right

Exist hamiltonian cycle iff $\forall C_j$ at least one of the edges that pass u_j are passed!

We can also do

HAMILTONIAN CYCLES $\leq_p$ TRAVELING SALESMAN

Scheduling with Release Time and Deadlining

Input n jobs that must be scheduled in a single machine. Each job j has:
- Release time r_j, where it means that job j is available after r_j
- Duration t_j, requiring contiguous t_j time steps
- Deadline d_j, must finish before d_j

Does there exist a schedule that allows to process all n jobs.

th: SRTD is NP-complete
1. " is NP ✓
2. subset-sum $\leq_p$ SRTD

pf: We have instance of subset sum $w_1, w_2 \dots, w_n, W$.
We create instance of SRTD with n+1 jobs.
for $j = 1 \dots n$
job $j :=$ $r_j = 0$ $t_j = w_j$ $d_j = 1 + \sum w_i$

for job $n+1 :=$ $r_{n+1} = W$ $t_{n+1} = 1$ $d_{n+1} = W+1$
we claim that there exist numbers w_i that sum to W iff. i can schedule all jobs.
Job n+1 splits the time available into two partitions: one of lenght W and one of lenght $\sum w_i - W$.
All jobs can be scheduled iff i can find some jobs with total duration W. Then corresponding numbers have sum of w_i's equals to W.

Co-NP Class

Regarding the NP-class we can do an observation; to say that a problem is in NP we present a certificate, verifying that it's solvable in Polytime.

String s is a "yes" iff there exists short t such that $B(s,t) = "yes"$. Negating it is not easy: s is "no" iff FOR ALL SHORT t we have $B(s,t) = no$.

↳ **Co-NP class is the complementary class to NP !**

We can also say that if $X \in P$ we can also say that $\bar{X} \in P$ since once calculated X with A we can produce $\bar{A}$ that flips result over.

This cannot be said about $X \in NP$, since for all $s \in \bar{X}$ iff for all t of lenght at most $(|p|)$, $B(s,t) = no$.
Cannot just invert, since definition changed from "exist t" to "for all ts"

Pr: Is Co-NP = NP ? Is Co-NP ≠ NP ?

answering to this question would also mean to solve the problem P = NP, since : (cl. P=NP)

pf:

$$X \in NP \Rightarrow X \in P \Rightarrow \bar{X} \in P \Rightarrow \bar{X} \in NP \Rightarrow X \in Co\text{-}NP$$

$$X \in Co\text{-}NP \Rightarrow \bar{X} \in NP \Rightarrow \bar{X} \in P \Rightarrow X \in P \Rightarrow X \in NP$$

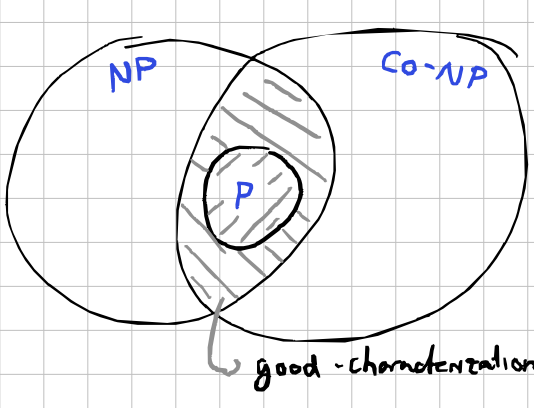
this would mean that $NP \subseteq Co\text{-}NP$ and $Co\,NP \subseteq NP$
$\Rightarrow Co\text{-}NP = NP$

Good-characterization

if problem $X \in NP$ and also $X \in Co\text{-}NP$ we say that it has good characterization, since it's easy to demonstrate $X \in NP \cap Co\text{-}NP$.
Ex: Max-flow min-cut theorem : we can prove network has flow v by presenting a flow, or also demonstrate inverse by presenting a cut with cap < v

if $X \in P \Rightarrow X \in P \cap NP \cap Co\text{-}NP$

Is $P = NP \cap Co\text{-}NP$? Open question



Approximation algorithm

Idea is to create algorithm that goes "close" to solution, in case where a poly-time algorithm probably doesn't exist.

↳ POLYNOMIAL APPROXIMATION ALGORITHM ↲

We'll have four main design techniques:
- Greedy
- primal-dual
- linear programming and routing
- dynamic programming

GREEDY PROGRAMMING

Load balancing problem

Problem: Set of $m = \{M_1, M_2, \dots, M_m\}$ machines, and set of $j = \{j_1, j_2, \dots j_n\}$ jobs. Each job j has processing time t_j.

Assign jobs to machine to keep as balanced as possible. We'll denote $A(i)$ set of jobs assigned to M_i, so it'll run for a time of

$$T_i = \sum_{j \in A(i)} t_j$$

This will be our load.
We seek to minimize the makespan, which is $T = \max(T_i)$.
Scheduling problem is NP-hard.

Designing a greedy algorithm that assigns a job j to the smallest load so far is not optimal. Ordering counts!

Greedy - balance algorithm

Start with no jobs assigned
Set $T = 0$ and $A(i) = \emptyset$ for all M_i
for $j = 1, \dots, n$
  let M_i such that achieves $\min_k T_k$
  Assign job j to machine M_i
  $A(i) \leftarrow A(i) \cup \{j\}$
  $T_i \leftarrow T_i + t_j$
return $\max_i T_i$

Ex

$|m| = 3$

$j = \{2,3,4,6,2,2\} \rightarrow 8$ $\quad$ $j = \{6,4,3,2,2,2\} \rightarrow 7$

Greedy Algorithm analysis

Let T denote makespan of resulting assignment. We would like to say that $T + \Delta < T^*$, with Δ small, but we don't know optimal T^* value.
We'll be going to find a lower bound of the optimum. Since total processing time is known, we may have at best that all processes equally share it.

$$T^* \geq \frac{1}{m} \sum_j t_j$$

It may be too weak. Suppose that we have a set of jobs, in which one of them is sufficiently longer than the sum of them. In that case optimal solution would be to assign that job to a machine by itself, but this lower bound doesn't catch that possibility.
We'll add an additional lower bound

$$T^* \geq \max_j t_j$$

th: Algorithm Greedy-Balance produces an assignment of jobs : machines with makespan $T \leq 2T^*$ 1

pf: It'll consist of switching between the two lower bounds previously found. Right before assigning job j to M_i, the machine had the least load of any machine. Since M_i was selected, then every other machine had a load of at least $T_i - t_j$, then after adding we'll have $\sum_k T_k \geq m(T_i - t_j)$. equivalently:

$$T_i - t_j \leq \frac{1}{m} \sum_k T_k$$

with the total load being the sum of all t_j. So quantity on the right it's exactly our 1st lower bound on optimal value:

$$T_i - t_j \leq T^*$$

We need now to account for remaining part of load M_i, which is the final job j. We use the 2nd lower bound where $t_j \leq T^*$ to find our final solution:

$$T_i = (T_i - t_j) + t_j \leq 2T^*$$

Since our makespan is exactly T_i we have solution

It's easy to see how ordering the set of jobs by descending running time would greatly improve our result. See figure 11.3 of book, which keeps good balance of algorithm until last one arrives.
Creating a dynamic programming algorithm would probably be more efficient, but we're watching for greedy approx. algorithms.

Sorted-balance algorithm

Start with no jobs assigned
Set $T = 0$ and $A(i) = \emptyset$ for all M_i
Sort jobs by decreasing order on t_j
assume $t_1 \geq t_2 \geq \dots \geq t_n$
for $j = 1, \dots, n$
  let M_i such that achieves $\min_k T_k$
  Assign job j to machine M_i
  $A(i) \leftarrow A(i) \cup \{j\}$
  $T_i \leftarrow T_i + t_j$
return $\max_i T_i$

Improvement comes from observation. with $|j| \leq m$, solution will be optimal, since every one will have one process. We can use the lower bound to say that if we have more than m jobs, then $T^* \geq 2t_{m+1}$
We'll consider only first $m+1$ jobs. They each take at least t_{m+1} time. Since we have m machines and $m+1$ jobs, at least one of them will have assigned two of the jobs. This machine will have at least $2t_{m+1}$ processing time.

th: Algorithm sorted-balance produces an assignment of jobs to machines with makespan $T \leq \frac{3}{2} T^*$

pf: follow proof similar to the last one. We'll consider machine M_i that has max load. if $|A(i)| = 1$, then solution is optimal.
We'll assume that M_i has two or more jobs assigned. Let t_j be last job assigned. $j \geq m+1$, thus $t_j \leq t_{m+1} \leq \frac{1}{2} T^*$.
Now we'll continue with other proof, but with slight change. At end, we had $T_i - t_j \leq T^*$ and $t_j \leq T^*$.
We now that second inequality is a bit more precise than before, so adding two inequalities gives us the bound $T_i - t_j + t_j \leq T^* + \frac{1}{2} T^*$ which leads us to

$$T_i \leq \frac{3}{2} T^*$$

CENTER SELECTION PROBLEM

- Given a set $S = \{s_1, \dots, s_n\}$ of sites $|S| = n$
- Given $k \in \mathbb{N}$
- $dist : S \times S \to \mathbb{R}^+$

$\forall s,t,r \in S$

$dist(s,t) \geq 0$ and $dist(s,t) = 0$ iff $s = t$

$dist(s,t) = dist(t,s)$

$dist(s,r) \leq dist(s,t) + dist(t,r)$

I want to open C centers $C = \{c_1, c_2, \dots, c_k\}$

1. define $dist(s,C) = \min_{c \in C} dist(s,c)$

We'll say that C forms an r-cover if $dist(s,C) \leq r$ the minimum r for what C forms r-cover $s \in S$ will be called covering radius of E and is called $r(C)$

$$r(C) = \max_{s \in S} dist(s,C)$$

we want to find set C s.t. covering radius is minimal

Center selection problem is NP-hard.

We set-up the problem assuming we know the covering radius r.

If $s_i, s_j \in S$, covered by same c_i with covering radius r, then

$$dist(s_i, s_j) \leq 2r$$

Algorithm:

S' sites to be covered $:= S$

let $C = \emptyset$

while $S' \neq \emptyset$

  select any site $s \in S'$ and add to C

  remove all sites $s_i \in S'$ s.t. $dist(s, s_i) \leq 2r$

if $|C| \leq k$

  then r covering radius for center C

else

  claim no k centers with covering radius $\leq r(C)$

2-approximation algorithm

if alg selects more than k centers, then for any $|C^*| = k$ covering radius $r(C) > r$

Pf (by contr): assume opposite, so $\exists\, C^* : |C^*| = k$ with $r(C^*) \leq r$.
Each center $c \in C$ selected by alg $c \in S$, and $\exists\, c^* \in C^*$ s.t. $dist(c, c^*) \leq r$.
Each center c^* cannot be mapped to more than one c!
$c, c' \in C$, $c^* \in C^*$. $dist(c, c') > 2r$, since otherwise it couldn't have been selected, but also for triangular disequality

$$dist(c, c') \leq dist(c, c^*) + dist(c^*, c') \leq 2r \quad \lightning$$
$$dist(c, c') > 2r$$

eliminate prerequisite of knowing optimal radius
we may need to try to find a better solution by iterating smartly with different values of r INFINITELY!
We can work around by iteratively selecting k sites such to maximise $dist(s, C)$.

assume $k \leq |S|$ otherwise $C = S$
select any site $s \in S$ and add to C : $\{s\}$
while $|C| < k$
  select s s.t. max $dist(s, C)$
  add site s to C
return C as selected set of sites.

this returns set C of k points s.t. $r(C) \leq 2r(C^*)$ with C^* being optimal.

Pf: let $r = r(C^*)$ minimum radius cover for k centers. We assume that we obtain set of k centers with $r(C) > 2r$ and contradict.
let s be a site which $dist(s, C) > 2r$. We consider set intermediate C', we'll add c' to the C'. $dist(c', C') \geq 2r$.
Since we always choose max dist, we can say that

$$dist(c', C') \geq dist(s, C') \geq dist(s, C) > 2r.$$

this is a specific case of previous algorithm!

(WEIGHTED) SET COVER

given set U of n elements and list $S = \{S_1, \dots, S_m\}$ of subsets of U.

A set cover is collection of subsets such that $\bigcup_{i \text{ chosen}} S_i = U$

In weighted version we have an associated weight $w_i \geq 0$ for every subset. Goal is to choose subsets such that

$$\sum_{S \in C} w_i$$

is minimized. if $w = 1$ we have set cover issue.

We want to choose a subset S to add based on minimizing the cost-to-size ratio.

Algorithm:

Greedy-Set-Cover

start with $R = U$ and no sets selected

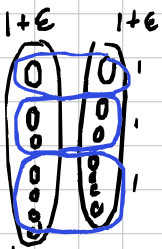
while $R \neq 0$

select set S_i that minimizes $\frac{w_i}{|S_i \cap R|}$

delete set S_i from R

end

return selected sets. Not optimum!

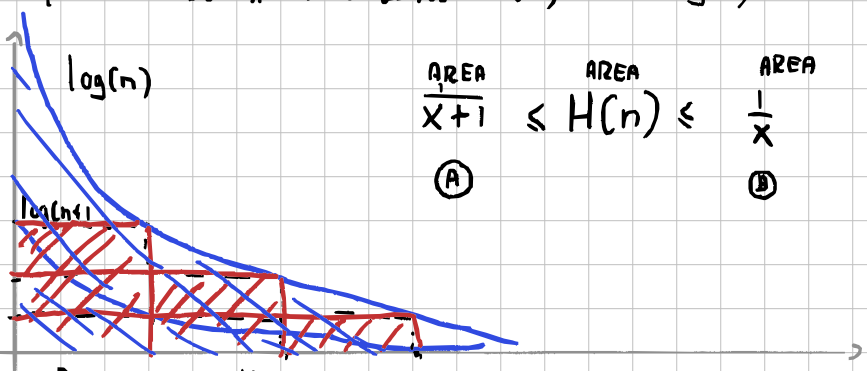


11.6

Fig. 11.6 can be extended indefinitely, with difference between greedy solution and optimum being $\simeq \log(|U|)$

C1: Greedy-Set-cover is $O(\log n)$-approximation.

(to present pf. we use harmonic series $H(n) = \Theta(\log n)$)



$$\frac{\text{AREA}}{x+1} \leq \text{AREA} \leq \frac{\text{AREA}}{x}$$

(A) (B)

(A): $\int_0^n \frac{1}{x+1} = \int_1^{n+1} \frac{1}{y}\,dy = \ln y \Big|_1^{n+1} = \ln(n+1)$

(B): $1 + \int_1^n \frac{1}{x}\,dx = 1 + \ln x \Big|_1^n = 1 + \ln(n)$

$\ln(n+1) \leq H(n) \leq 1 + \ln(n)$

$\Rightarrow H(n) = \Theta(\ln(n))$

C_s are the costs of covering the elements with set S_i

$$C_s = \frac{\omega_{S_i}}{|S_i \cap R|}$$

Pf: Let C be greedy solution

$$\omega(C) = \sum_{S \in C} \omega(S) = \sum_{s \in U} C_s \quad (1)$$

we'll now try to lower bound the cost of the optimal solution using C_s. As each set S_i is selected its weight is distributed to costs C_s of newly covered element.

Lemma: for every set S_k, we have $\omega(S_k) \geq \frac{1}{H(S_k)} \cdot \sum_{s \in S_k} C_s$

Let $S_k = \{S_1, \dots S_d\}$ where $|S_k| = d$, renaming elements as their order of cover in greedy algorithm.

Consider $j \in S_k$. When S_j got covered using some set S_i there were at least $|S_k \cap R| \geq d - j + 1$ uncovered elements

$$C_j = \frac{\omega(S_i)}{|S_k \cap R|} \leq \frac{\omega(S_k)}{|S_k \cap R|} \leq \frac{\omega(S_k)}{d - j + 1} \quad (2)$$

↓ iteration before Greedy.

$$\sum_{s \in S_k} C_j \leq \sum_{j=1}^{d} \frac{\omega(S_k)}{d-j+1} = \omega(S_k) \cdot \sum_{j=1}^{d} \frac{1}{d-j+1} = \omega(S_k) \sum_{j=1}^{d} \frac{1}{j}$$

$$C_{S_j} \leq \omega(S_k) H(|S_k|)$$

let $d^* = \max_{k \in [m]} |S_k|$, and C^* be opt.

$$\omega(C^*) = \sum_{S_k \in C^*} \omega(S_k) \geq \frac{1}{H(S_k)} \sum_{S_k \in C^*} \sum_{j \in S_k} C_j \geq \frac{1}{H(d^*)} \sum_{s \in U} C_s =$$

$$\overset{(1)}{=} \frac{\omega(C)}{H(d^*)} \Rightarrow$$

$$\Rightarrow \omega(C) \leq H(d^*) \cdot \omega(C^*) \leq H(n) \cdot \omega(C) = \Theta(\log(n)) \cdot \omega(C^*)$$

(WEIGHTED) VERTEX COVER

Given:

a graph $G=(V,E)$ and weight function $w: V \to R^+$
choose subset $V' \subseteq V$ s.t. $\forall e=(i,j)$ $i \in V'$ or $j \in V'$ that minimizes $\sum_{i \in V'} w(i)$

It can be done by considering it a special case of vertex cover. We can use greedy algorithm and have $O(\log n)$-approximation. In general we cannot create approx. algorithms based on reduction properties!

Pricing method algorithm

It's also called primal-dual method. Weights of nodes are costs, with every edge as prices that incident edges have to pay to have node.
I'll say that a set of prices $p(e)$ on edge is FAIR if $\forall i \in V$ I have

$$\sum_{e \in (i,j)} p(e) \leq w(i)$$

for any solution S^* of weighted vertex cover problem and any set of prices $p(e)$ that are fair we have that

$$w(S^*) \geq \sum_{e \in E} p(e)$$

since

$$w(S^*) = \sum_{i \in S^*} w(i) \geq \sum_{i \in S^*} \sum_{e=(i,j) \in E} p(e) \geq \sum_{e \in E} p(e)$$

- $\geq$: $p(e)$ fair
- $\geq$: vertex cover ⇓ every edge must be covered at least ONCE

I'll say that a node i is tight, or paid for if:

$$\sum_{e=(i,j)} p(e) = w(i)$$

Alg: set $p_e = 0 \quad \forall e \in E$
while $\exists e \in E = (i,j)$ s.t. neither i or j is tight
  select that edge
  increase $p(e)$ without violating fairness
set S as set of all tight nodes
return.

Lemma: Set S returned by vertex-cover-approx alg and prices $p(e)$ satisfy $w(S) \leq 2 \sum_{e \in E} p(e)$

Pf: $\forall i \in S$ we have $\sum p(e) = w(i)$, so we'll have

$$w(S) = \sum_{i \in S} w(i) = \sum_{i \in S} \sum_{e=(i,j)} p(e) \overset{①}{\leq} 2 \sum p(e)$$

↳ Legitimate that $\exists e=(i,j)$ in which both are tight; e will be counted twice

②: $w(S^*) \geq \sum_{e \in E} p(e)$ □

$$\Rightarrow \quad w(S) \leq 2 \sum_{e \in E} p(e) \leq 2\, w(S^*)$$ □

2-approximation algorithm!

LINEAR PROGRAMMING

In LP I have variables $x_1, x_2, \dots, x_n$
I have a matrix constraints $A \in \mathbb{R}^{m \times n}$ (with m constraints)
I have cost vector $\vec{c} \in R^n$ $[c_1, c_2, \dots, c_n]^T$
I have constraint vector $\vec{b} \in \mathbb{R}^m$ $[b_1, \dots, b_m]^T$
Prob:

given $\min c^T x$
s.t. $Ax \geq B \qquad x \geq \vec{0}$

Ex

$$A = \begin{bmatrix} 0 & -1 \\ 2 & 1 \\ 1 & 3 \end{bmatrix} \qquad b = \begin{bmatrix} -6 \\ 4 \\ 3 \end{bmatrix}$$

we can solve LP in polytime in m and n

Now we will model vertex cover as an Integer Program (IP), which is:
- each variable $x_1, \dots, x_n$ corresponds to node in graph

An IP that covers vertex cover is $\vec{w}^T = [w_1, \dots, w_n]$, $w_i = w(i)\ \forall i \in V$

$$\min \quad \vec{w}^T \vec{x} = \sum_{i=1}^{n} w_i x_i$$
s.t.
$$\forall (i,j) \in E \quad x_i + x_j \geq 1$$
$$\forall i : x_i \in \{0,1\}$$

$x_i = 1 \Leftrightarrow$ node i is in vertex cover.

we will use VC.IP to design an algorithm for VC
Let's look at the linear relaxation of VC.IP

$$\min \quad \vec{w}^T \vec{x} = \sum_{i=1}^{n} w_i x_i$$
s.t.
$$\forall (i,j) \in E \quad x_i + x_j \geq 1$$
$$\forall i : x_i \geq 0$$
$$x_i \leq 0$$

// linear relaxation

} VC.LP

VC.LP can be solved in poly-time (simplex method).
let S^* be the optimal solution VC.IP, and let ω_{LP} be the cost of solution of VC.LP

Lemma: Cost of optimal $\omega(S^*) \geq \omega_{LP}$ (3)

P.f: Every solution of VC.IP is also a solution of VC.LP

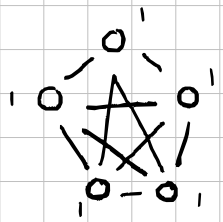
We call the ratio $\frac{\omega(S^*)}{\omega_{LP}}$ the INTEGRALITY GAP.
Let's solve in polytime the linear problem. We'll get values for $\vec{x}$ with $x_i \in \vec{x}$ $0 \leq x_i \leq 1$.
I want to use solution to create S for Vertex Cover:

If for some node i we have $x_i = 1 \Rightarrow x_i \in S$
if for some node i we have $x_i = 0 \Rightarrow x_i \notin S$
for every other node i $0 < x_i < 1$

As an example:

$\omega(S^*) = 2$
$\omega_{LP} = 3/2$ (i set $x_i = 1/2$ $\forall i$)

Even worse: Fully connected graph of n nodes



$\omega(S^*) = n-1 = 4$
$\omega_{LP} = x_i = 1/2 \Rightarrow \omega_{LP} = \frac{n}{2}$

$$\frac{\omega(S^*)}{\omega_{LP}} = 2\frac{n-1}{n} \xrightarrow{n\to\infty} 2$$

Alg: Solve VC-LP and obtain solution $\vec{x}^* = [x_1^*, \ldots, x_n^*]^T$

Let $S = \{ i \in [n] : x_i^* \geq 1/2 \}$

Claim 1: S is a vertex cover

Consider an edge $(i,j) \in E$

x^* is solution to VC.LP, which satisfies all constraints!

$x_i^* + x_j^* \geq 1$. if $x_i \in [0,1]$ then $\max\{x_i, x_j\} \geq 1/2$,

so at least one is selected.

Claim 2: $w(S) \leq 2\, w_{LP}$ ⑥ $(w_{LP} : \sum_i^n w_i x_i^*)$

$w(S) = \sum_{i \in S} w(i)$

$w_{LP} \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{2} \sum_{i \in S} w_i$ $\Big\}$ $w(S) \leq 2\, w_{LP}$

$i \in S \quad x_i^* \geq 1/2$

from ③ and ④...

Th: $w(S) \leq 2\, w_{LP} \leq 2 w(S^*)$

$2\, w_{LP} \leq 2 w(S^*)$

1

Randomized Algorithms

weighted set cover

n elems : $\{e_1, \dots, e_n\}$
m sets : $\{S_1, \dots, S_m\}$
w_j weight of set S_j

define $x_j = \begin{cases} 1 & \text{if i select } S_j \\ 0 & \text{otherwise} \end{cases}$

$$\min \sum_j^m w_j x_j$$

$$\text{s.t.} \sum_{j : e_i \in S_j} x_j \geq 1 \qquad \forall i \in [n] \qquad // \; e_i \text{ covered by at least one set}$$

$$x_j \in \{0, 1\}$$

↳ $\frac{1}{m}$ approximation, since every value on worst case should have $1/m$

On IP we select
$x_j = 1$ iff linear relaxation of $x_j \geq 1/m$

Randomized Rounding for weighted set cover

Let w_{LP} be the optimal solution (SCLP) and let C^* be the optimal solution to set cover (so also for SCIP). Then

$$w_{LP} \leq w(C^*) = \sum_{S_j \in C^*} w_j \qquad (1)$$

We'll create a probabilistic algorithm that, with high probability, will be correct.

$w_{LP} = \sum w_j x_j^*$

Alg:

Let $\vec{x}^* = [x_1^*, x_2^*, \dots, x_m^*]$ be the opt. solution (SCLP)

for $l = 1 \dots T = c \cdot \ln(n)$

  $C_l = \{\}$

  for $j = 1 \dots m$

    insert S_j into C_l with probability x_j^*

$C = \bigcup_{l=1}^{c \cdot \ln(n)} C_l$

return C

this algorithm is doing RANDOMIZED ROUNDING. using x_i^* as a probability.

Claim 1 and Claim 2 will show that the randomized set cover algorithm has an expected approximation ratio $2 \ln(n)$ and guarantees w.p. $\geq 1 - 1/n$ that each element is covered.

Cl$_1$: The expected cost of C ($E[\omega(C)]$) $\leq c \ln(n) \cdot \omega(C^*)$

Pf: Define the (indicator) random variable $\forall \ell \in [c \cdot \ln(n)]$

$$X_j^{\ell} = \begin{cases} 1 & \text{if } S_j \in C_\ell \\ 0 & \text{otherwise} \end{cases} \qquad \forall j \in [m]$$

$$\Pr(X_j^{\ell} = 1) = X_j^*$$

$$E[X_j^{\ell}] = 1 \cdot \Pr(X_j^{\ell} = 1) + 0 \cdot \Pr(X_j^{\ell} = 0) = X_j^*$$

Consider set C_ℓ

$$E[\omega(C_L)] = \sum_{j=1}^{m} \omega_j E[X_j^{\ell}] =$$

$$= \sum_{j=1}^{m} \omega_j X_j^* = \omega_{LP} \qquad (2)$$

[Pr. linearity of expectations
- for two rand var X, Y
$E[X+Y] = E[X] + E[Y]$
- for RV X and const c
$E[cX] = cE[X]$
True also if X Y are not independent!]

for the final set $C = \bigcup_{\ell=1}^{c\ln(n)} C_\ell$

$$E[\omega(C)] \leq E\left[\sum_{\ell=1}^{c\ln(n)} \omega(C_\ell)\right] \overset{(2)}{\leq} c \cdot \ln(n) \cdot \omega_{LP} \overset{(1)}{\leq}$$

$$\leq c \cdot \ln(n)\, \omega(C^*) \qquad \blacksquare$$

C12: Each element is covered with probability of at least $\geq 1 - 1/n$.

Pf: Consider element e_i $(i \in [n])$

$$\Pr(e_i \notin C) = \Pr\left(\bigcap_{\ell=1}^{c\ln(n)} e \notin C_\ell\right) = \left[C_\ell \text{ are independent}\right] =$$

$$= \prod_{\ell=1}^{c\ln(n)} \Pr(e_i \notin C_\ell) \quad (3)$$

$$\Pr(e_i \notin C_\ell) = \Pr(\forall j : e_i \in S_j : X_j^\ell = 0) = \left[\text{Independence of the set selection}\right]$$

$$= \prod_{j:e_i \in S_j} \Pr(X_j^\ell = 0) = \prod_{j:e_i \in S_j} 1 - X_j^* \leq$$

[for any x we have that $1+x \leq e^x$]

$$\leq \prod_{j:e_i \in S_j} e^{-X_j^*} = e^{-\sum_{j:e_i \in S_j} X_j^*} \leq e^{-1}$$

↳ satisfiability of $\sum X_j \geq 1$ (SC LP)

$$\Pr(e_i \notin C) \overset{(3)}{=} \prod_{\ell=1}^{c\ln(n)} \leq \prod_{\ell=1}^{c\ln(n)} e^{-1}$$

$$= e^{-c\ln(n)} = n^{-c} = \frac{1}{n^c}$$

$$\Pr(\text{"I don't cover some element"}) = \Pr\left(\bigcup_{i=1}^{n} \{e_i \notin C\}\right)$$

$$\leq \sum_{i=1}^{n} \Pr(e_i \notin C) = \quad \rightarrow \text{union property.}$$

$$= n \cdot \frac{1}{n^c} = \frac{1}{n^{c-1}}$$

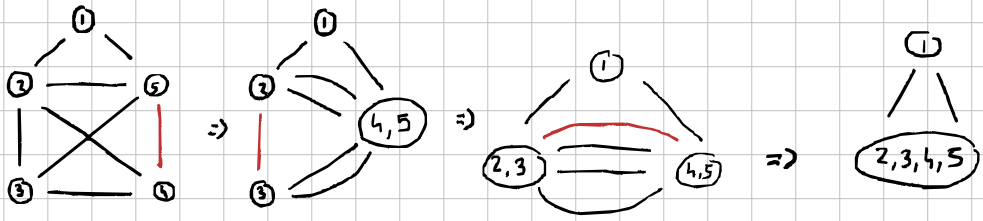
by choosing $c = 2$ we see that $\Pr(\text{"-"}) \geq 1 - \frac{1}{n}$

[what about for $c = 1$?] ■

Randomized Global Mincut

We'll recursively apply contraction to multigraph $G=(V,E)$
for each node v we'll have $S(v)$ all nodes contracted to v

If G has two nodes return $S(v_1), S(v_2)$
else
  choose edge $e=(u,v) \in E$ randomly
  let G' be multigraph resulting from contraction of e,
  new node $z_{u,v}$ replacing u & v.
  Define $S(z_{uv}) = S(u) \cup S(v)$
  apply contraction algorithm recursively to G'
endif



it won't always be correct.
Let's pick a minimum cut, and assume size is k.
If i never contract one of those k edges then
the algorithm may be optimal!
what's the probability of this happening?

Let E_i be the event: "i don't contract an edge
in the cut in the i-th round for $(i \in [n-2])$

to compute probability of success i will: ④

$$Pr(E_1) \cdot Pr(E_2|E_1) \cdot Pr(E_3|E,E_2) \cdot \ldots \cdot Pr(E_{n-2}|E,\ldots E_{n-3})$$

with $Pr(E_1) = 1 - \frac{k}{m}$

cl: $m \geq \frac{n \cdot k}{2}$
pf degree of each node is $\geq k$ and $m = \frac{1}{2}\sum_{i=1}^{n} \text{degree}(i) \geq \frac{nk}{2}$

more generally, at the $j+i$-th contraction.

$$\Pr(E_{j+1} \mid E_1, \dots, E_j) = ?$$

- $n-j$ nodes
- $m \geq \frac{(n-j)k}{2}$ edges

$$= \geq 1 - \frac{k}{(n-j)k/2} = 1 - \frac{2}{n-j}$$

④ $\Rightarrow$

$$\geq \left(1-\frac{2}{n}\right)\left(1-\frac{2}{n-1}\right)\left(1-\frac{2}{n-2}\right)\cdots\left(1-\frac{2}{4}\right)\left(1-\frac{2}{3}\right) =$$

$$= \text{serie telescopica} = \frac{2}{n(n-1)} = \binom{n}{2}^{-1}$$

the probability to succeed is generally pretty low $\left[\geq \binom{n}{2}^{-1}\right]$

This is always an upper bound on min cut, so i can repeat T times and register lower min-cut.

Probability of failing all T times is

$$\leq \left(1 - \frac{2}{n(n-1)}\right)^T \leq \left(e^{-2/n(n-1)}\right)^T = e^{\frac{-2T}{n(n-1)}}$$

$1+x \leq e^x$

we want $e^{\frac{2T}{n(n-1)}} = n$

$$\cdot \quad \frac{2T}{n(n-1)} \geq \log(n)$$

$$= T = O(n^2 \log(n))$$