ALGORITHM DESIGN

Edoardo Puglisi

ALGORITHM ANALYSIS

We need to be able to track algorithms computational cost, asymptotic order of growth and so on.

Desirable scaling property: when input size doubles, the algorithm should slow by a constant c

Algorithm is poly-time if has the previous property $\longrightarrow$ efficient

Worst case: running time guarantee for any input of size n. Some exponential algorithms are still used cause worst case is very rare.

Analysis types:

- worst case
- probabilistic: expected running time of randomized algorithm
- Amortized: worst case running time for any sequence of n operations
- Average-case: expected running time for any input of n size

} + other more complex analysis

Big-Oh notation: $T(n)$ is $O(f(n))$ if there is a constant $c>0$ and $n_0 \geq 0$ such that $T(n) < c \cdot f(n)$ for all $n > n_0$ (upper bound)

Big-Omega notation: $T(n)$ is $\Omega(f(n))$ if there is a constant $c>0$ and $n_0 \geq 0$ such that $T(n) > c \cdot f(n)$ for all $n > n_0$ (lower bound)

Big-Theta notation: $T(n)$ is $\Theta(f(n))$ if there are $c_1>0$, $c_2>0$ and $n_0 \geq 0$ such that $c_1 \cdot f(n) < T(n) < c_2 \cdot f(n)$ for all $n > n_0$

STABLE MATCHING PROBLEM

Given a set of preferences among employers and applicants, can we assign applicants to employers so that for every employer E, and every applicant A who is not scheduled to work for E, at least one of the following two things is the case?

- E prefers every one of its accepted applicants to A; or
- A prefers her current situation over working for employer E

Unstable pair: E prefers A to one of its accepted applicants, A prefers E to its assigned employer.

A stable assignment is an assignment without unstable pairs.

eg. H = set of hospitals S = med students. A matching M is a set of ordered pairs h-s with $h \in H$ and $s \in S$: each h and each s appears in at most one pair of M. M is perfect if $|M| = |S| = |H| = n$

GALE-SHAPLEY Algorithm for stable matching:

```
INITIALIZE M to empty matching
WHILE (some hospital h is unmatched and hasn't proposed to every student)
    s ← first student on h's list to whom h has not yet proposed
    IF (s is unmatched)
        ADD h-s to matching M
    ELSE IF (s prefers h to current partner h')
        REPLACE h'-s with h-s in matching M
    ELSE
        s rejects h
RETURN stable matching M
```

Always ends with perfect matching

Terminates after at most n^2 iterations of while loop

Efficient implementation: $O(n^2)$, represent hospitals and student using index $(1, \dots, n)$.

- Maintain a list of free hospitals (in stack/queue)
- Maintain two arrays student[h] and hospital[s]: if h matched to s then hospital[s] = h and student[h] = s, 0 otherwise.
- For each hospital maintain a list of students ordered by preference and a pointer to next student for proposal.

- For each student create inverse of preference list of hospitals (constant time access after $O(n)$ preprocessing)

ie

	1	2	3	4	5	6	7	8
index (preference classification)								
hosp[ ] =	8	3	7	1	4	5	6	2

hospital "none"

	1	2	3	4	5	6	7	8
Inverse[ ] =	4	8	2	5	6	7	3	1

DIRECTED GRAPHS

Strong connectivity: every node pair is mutually reachable

↳ if s is strongly connected with u and also with v then u is strongly connected with v too.

Algorithm $O(n+m)$ with n = # nodes and m = # edges :

- Pick any node s
- Bfs from s
- Bfs from s in reversed graph
- True if all nodes visited in both Bfs

A strongly connected **component** is a subset of nodes strongly connected

DAG: directed acyclic graph

Topological Order: ordering every node $v_1 \dots v_n$ so that for every edge (v_i, v_j) $i < j$

- If G has a topological order then G is a DAG.
- If G is a DAG then G has a node without entering edges → starting node
- If G is a DAG then G has a topological ordering.

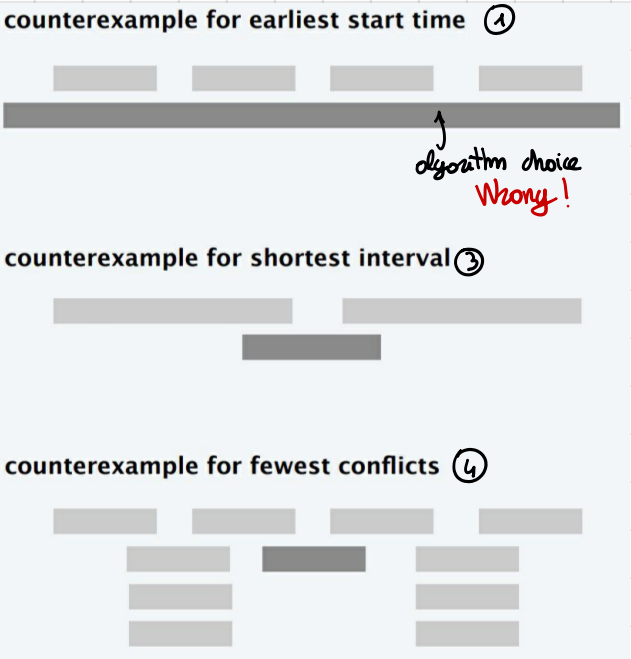
Algorithm $O(m+n)$:

- Find node without incoming edges and order it first (v)
- Delete it from graph G
- Repeat in $G - \{v\}$

GREEDY ALGORITHMS

Interval scheduling: Job j start at s_j and finish at f_j, two jobs are compatible if not overlapping → find max subset of mutually compatible jobs

Strategies: ① consider jobs in ascending order of s_j ② ascending order of f_j ③ ascending order of $f_j - s_j$ ④ ascending order of conflicts c_j



The right strategy is to order using finishing time which is an optimal solution

EARLIEST-FINISH-TIME-FIRST $(n, s_1, s_2, \dots, s_n, f_1, f_2, \dots, f_n)$

SORT jobs by finish time so that $f_1 \le f_2 \le \dots \le f_n$

$A \leftarrow \phi$ ← set of jobs selected

FOR $j = 1$ TO n

  IF job j is compatible with A

    $A \leftarrow A \cup \{j\}$

RETURN A

- Keep track of job j^* that was added last to A
- Job j is compatible with A if $s_j \ge f_{j^*}$
- $O(n \log n)$ for sorting

Interval Partitioning: Lecture j start at s_j and finish at f_j → find min number of classrooms to schedule all lecture
Some strategies as before. This time the optimal solution is the first one.

counterexample for earliest finish time ②

counterexample for shortest interval ③

counterexample for fewest conflicts ④

EARLIEST-START-TIME-FIRST $(n, s_1, s_2, \ldots, s_n, f_1, f_2, \ldots, f_n)$

SORT lectures by start time so that $s_1 \le s_2 \le \ldots \le s_n$.
$d \leftarrow 0$ ← number of allocated classrooms
FOR $j = 1$ TO n
  IF lecture j is compatible with some classroom
    Schedule lecture j in any such classroom k.
  ELSE
    Allocate a new classroom $d + 1$.
    Schedule lecture j in classroom $d + 1$.
    $d \leftarrow d + 1$
RETURN schedule.

- Store classrooms in priority queue (key = finish time of its last lecture)
- Compare s_j with key of min in priority queue to determinate if lecture j is compatible
- Increase key of classroom k to f_j to add lecture j to classroom k
- Priority queue operations $O(n)$
- Sorting in $O(n \log n)$

classrooms needed ≥ depth = the maximum number that contain any given time of a set of open intervals.

Scheduling to minimize lateness: Job j require t_j unit of processing time and must be ended due to time d_j, $f_j = s_j + t_j$, lateness $\ell = \max\{0, f_j - d_j\}$
→ schedule all jobs to minimize maximum lateness $L = \max_j \ell_j$
Strategies: ① ascending order of t_j ② ascending order of d_j ③ ascending order of slack $d_j - t_j$

- [Shortest processing time first] Schedule jobs in ascending order of processing time t_j.

	1	2
t_j	1	10
d_j	100	10

counterexample

- [Smallest slack] Schedule jobs in ascending order of slack $d_j - t_j$.

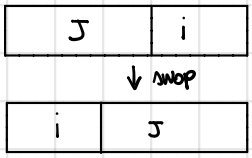
	1	2
t_j	1	10
d_j	2	10

counterexample

EARLIEST-DEADLINE-FIRST $(n, t_1, t_2, \ldots, t_n, d_1, d_2, \ldots, d_n)$

SORT n jobs so that $d_1 \le d_2 \le \ldots \le d_n$.
$t \leftarrow 0$
FOR $j = 1$ TO n
  Assign job j to interval $[t, t + t_j]$.
  $s_j \leftarrow t$; $f_j \leftarrow t + t_j$
  $t \leftarrow t + t_j$
RETURN intervals $[s_1, f_1]$, $[s_2, f_2]$, …, $[s_n, f_n]$.

Inversion: a pair of jobs i and j such that $i < j$ but j scheduled before i
→ Swapping an inversion reduce number of inversions by one and doesn't increase max lateness.



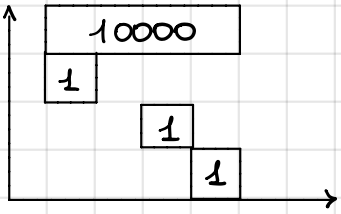
DYNAMIC PROGRAMMING

Greedy: 1 step at time to build optimal solution

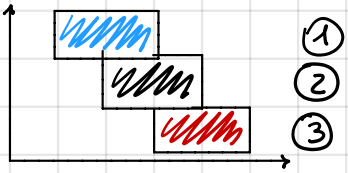
Divide&Conquer: solution of independent subproblem eg. merge-sort.

In dynamic progzomming the two subproblems ARE NOT INDEPENDENT !

WEIGHTED interval scheduling: Job i has an other value in addition to s_i and f_i



Greedy algorithm output will be the 3 minus jobs while the best solution is to chose only the longest but most valued job. $Job_i = (s_i, f_i, v_i)$



$p(1) = 0$
$p(2) = 0$
$p(3) = 1$ cause ① is compatible with ③

$p(j)$ = longest index $i < j$ such that job j is compatible to job i

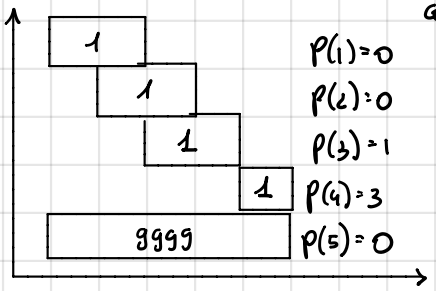
Optimal solution = finite solution

$Opt(j)$ - opt solution on $\{1 \dots j\}$ → Goal = find $Opt(n)$

$Opt(n)$

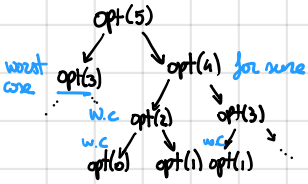
↳ $n \in S^*$ with S^* = optimal set ⇒ $v_n + Opt(p(n))$

↳ $n \notin S^*$ ⇒ $Opt(n-1)$



eg. $opt(0) = 0$, $opt(1) = v_1$

Recursively find best solution. Lot of recursion step!



⇒ memorizing results of each subproblems → $O(n \log n)$

↳ until n-1 !

i	v_i	w_i
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7
…	…	…

budget $W = 11$ W = cost for that job ⇒ $\max \sum v_i$ maximize value anyway with $\sum w_i < W$

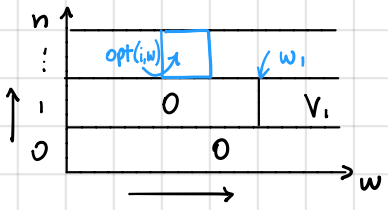
$Opt(j, w)$

↳ $j \in S^* \rightarrow v_j + opt(j-1, w - w_j)$

↳ $j \notin S^* \rightarrow opt(j-1, w)$

$opt(i, w)$ (Bellman equation)

$$\begin{cases} 0 & \text{if } i = 0 \\ opt(i-1, w) & \text{if } w < w_i \\ \max [v_i + opt(i-1, w-w_i),\ opt(i-1, w)] \end{cases}$$

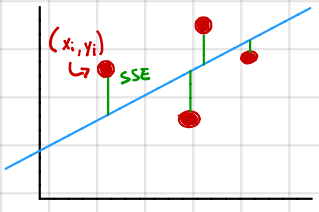


Build this matrix.

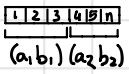
SEGMENTED LEAST SQUARES

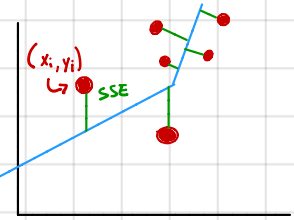
n point in $\mathbb{R}^2$ find $y = ax + b$ that minimizes the sum of squared error: $SSE = \sum_{i=1}^{n} (y_i - ax_i - b)^2$

⇒ find best (a,b) couple



What if points possition needs more lines (segments)? Given the points we can find partition $P = \{[1, f_1], [f_1, f_2] \dots [f_n, n]\}$ eg





L = # of segments

$cost = \sum_{k=1}^{L} SSE(P_k) + cL$ → more line longest cost = regularity term

↳ less line more errors

$Opt(j) = \{P_1 \dots P_j\}$ solution

$e(i,j) = \{P_i \dots P_j\}$ solution (fragment only)

last fragment $i-n \rightarrow e(i,n) + opt(i-1) + c$

but a priori i dont know i so i try to maximize this function

$$Opt(j) = \begin{cases} \max_{i \in [1,j]} [e(i,j) + opt(i-1) + c] \\ 0 \text{ if } j = 0 \end{cases}$$

SEQUENCE ALIGNMENT

- OCURRANCE / OCCURRENCE — 6 mismatches, 1 gap
- OCURRANCE / OCCURRENCE — 1 mismatch, 1 gap

EDIT DISTANCE: function of mismatch and gap cost. δ = gap cost, α_{pq} = mismatch cost

Goal: given 2 strings $x_1 x_2 \dots x_n$ and $y_1 y_2 \dots y_n$ find min-cost solution.

$OPT(i,j)$ = min cost of aligning prefix strings $x_1 \dots x_i$ and $y_1 \dots y_j$

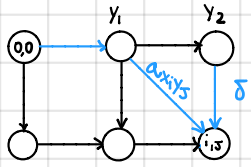
eg PALETTE / PALATE — gap penalty = 2, mismatch penalty = 1
cost = 1·gap + 2·mis = 4

PALETTE / PALA-TE — cost = 1·gap + 1·mis = 3

Bellmon Eq.

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

DP Algorithm needs $O(nm)$ time / space to compute EDIT distance



Hirschberg's Algorithm

Edit distance graph: $f(i,j)$ = length of shortest path from (0,0) to (i,j). $f(i,j) = OPT(i,j)$ $\forall i,j$

Can compute $f(\cdot,j)$ $\forall j$ in $O(nm)$ time and $O(n+m)$ space

$g(i,j)$ = shortest path from (i,j) to (m,n) (using inverted edges) → $f(i,j)+g(i,j)$ = length of shortest path that pass trough (i,j)

Divide & Conquer! Find q that minimizes $f(q, n/2) + g(q, n/2)$ and use (i,j) in final solution.

SHORTEST PATH

For graph with negative weights Dijkstra may not produce the shortest path even adding a constant to every edge

Negative Cycle: directed cycle which sum (of edges) is negative

If G has no negative cycles then there exists a shortest path $v \to t$ that is simple (and has $\le n-1$ edges)

$OPT(i,v)$ = length of shortest $v \to t$ path that uses at most i edges

Case 1: $OPT(i,v) = OPT(i-1,v)$

Case 2: shortest $v \to t$ uses exactly i edges: if (v,w) first in shortest $v \to t$ ⇒ add cost l_{vw} then select best $w \to t$ path using $\le i-1$ edges

Bellmon Eq.

$$OPT(i,v) = \begin{cases} 0 & \text{if } i=0 \text{ and } v=t \\ \infty & \text{if } i=0 \text{ and } v \neq t \\ \min \left\{ OPT(i-1,v), \min_{(v,w) \in E} \{ OPT(i-1,w) + l_{vw} \} \right\} & \text{if } i>0 \end{cases}$$

Given $G=(V,E)$ without negative cycles, DP algorithm compute length of shortest path in $\Theta(nm)$ time and $\Theta(n^2)$ space

Optimization: use 2 vectors instead of a matrix to store nodes during computation.

$d[v]$ = length of shortest $v \to t$ path so far

$successor[v]$ = next node in $v \to t$ path

If $d[w]$ not updated in iteration $i-1$ → no reason to use it in iteration i

Bellmon-Ford Algorithm: $d(v)$ = cost of some $v \to t$ path; after i^{th} pass, $d(v) \le$ cost of cheapest $v \to t$ path using $\le i$ edges

Cheapest $v \to t$ path in $O(mn)$ time / $\Theta(n)$ space if no negative cycles

```
BELLMAN-FORD (V, E, c, t)

FOREACH node v ∈ V
    d(v) ← ∞.
    successor(v) ← null.
d(t) ← 0.
FOR i = 1 TO n − 1
    FOREACH node w ∈ V
        IF (d(w) was updated in previous iteration)
            FOREACH edge (v, w) ∈ E
                IF ( d(v) > d(w) + c_vw)
                    d(v) ← d(w) + c_vw.
                    successor(v) ← w.
    IF no d(w) value changed in iteration i, STOP.
```

1 pass

NETWORK FLOW

Flow Network: tuple $G=(V,E,s,t,c)$, source s destination t and capacity $c(e)>0 \ \forall e \in E$. Think of it as a transportation chain of materials generated in s and delivered to t.

Time assumption depends on the problem.

st-cut: cut, partition (A,B) of nodes with $s \in A$ and $t \in B$

capacity: sum of capacity of edges from A to B $\rightarrow cap(A,B) = \sum_{\substack{e \text{ out} \\ \text{of } A}} c(e)$ ← important!

} **min-cut problem**: find cut of min capacity

st-flow: flow, function f that satisfies:

- $\forall e \in E \quad 0 \le f(e) \le c(e)$ → 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

} **max-flow problem**

Value of f: $val(f) = \sum_{e \text{ out of } s} f(e) - \sum_{e \text{ in to } s} f(e)$

Ofcourse flow out of s is equal to flow into t

FORD-FULKERSON ALGORITHM

Greedy Algorithm: Not Optimal!

- $f(e)=0 \ \forall e \in E$
- Find $s \to t$ path P where each e has $f(e) < c(e)$
- Augment flow along path P: augment $f(e)$ to min $c(e)$ along the path
- Repeat until get stuck

Need a way to undo bad decisions. → **residual network**: when passing through an edge add a reverse edge with capacity equal to capacity of original edge. When passing on reversed edge we simply undo the flow sent on original edge.

An augmenting path is a simple $s \to t$ path on residual network G_f

A **bottle neck capacity** of an augmenting path P is the min residual capacity of any edge in P. → maybe find augmenting path with highest bottleneck

Ford-Fulkerson Algorithm:

- $f(e)=0 \ \forall e \in E$
- find $s \to t$ path in residual network G_f
- augment flow along path P
- Repeat until get stuck.

After algorithm run we find also a cut with capacity equal to flow

Let f be any flow and (A,B) any cut → flow value = net flow across the cut (A,B) $\left[val(f) = \sum_{\substack{e \text{ out of} \\ A}} f(e) - \sum_{\substack{e \text{ in to} \\ A}} f(e)\right]$

Weak duality: given any flow f and any cut (A,B) → $val(f) \le cap(A,B)$

Value of max flow = capacity of min cut **Max flow Min cut Theorem**.

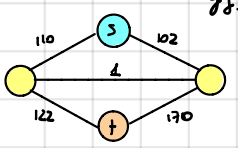
A flow is a max flow iff no augmentings path **Augmenting path Theorem**.

F.F algorithm terminates after at most $val(f^*) \le nC$ augmenting paths where f^* is a max flow in time $O(mnC)$. f^* exists and is integral. [C = max capacity]

If max capacity is C algorithm can take $\ge C$ iterations

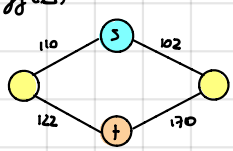
↳ **Capacity-scaling Algorithm**: choose augmenting path with "large" (not largest only) bottleneck capacity. Mantains a scaling factor Δ: augmenting path with bottleneck capacity $\ge \Delta$

G_f.



→ $\Delta = 100$

$G_f(\Delta)$



At each new step Δ is halfed (scaling phase)

Δ is a power of 2.

- $1 + \lfloor \log_2 C \rfloor$ scaling phases
- f = flow at end of Δ scaling phase → max flow value $\le val(f) + m\Delta$
- There are $\le 2m$ augmentations per scaling phase
- Time $O(m^2 \log C)$

SHORTEST AUGMENTING PATH: pick augmenting path with fewest edges → BFS. The shortest-augm.-path algorithm takes $O(m^2n)$ time

The length of a s.a.p. never decreases, can only increase. After at most m augmentations the length strictly increases.

Given a digraph $G=(V,E)$ with source s its level graph is defined by:

- $\ell(v)$ = # edges in $s \to v$ shortest path
- $L_G(V, E_G)$ is the subgraph of G that contains only edges (v,w) where $\ell(v)=\ell(w)+1$

Note: P is a shortest $s \to v$ path in G iff P is a $s \to t$ path in L_G

BIPARTITE MATCHING

Given an undirected graph, subset of edges $M \subseteq E$ is a matching if each node appears in at most one edge in M.

Max Matching: find a max-cardinality matching. Perfect matching if all nodes are involved in the matching.

A graph is bipartite if nodes can be divided in 2 subset L and R such that every edge connect a node L with a node R

Max Flow Formulation:

- Create digraph $G' = \{L \cup R \cup \{s,t\}, E'\}$
- Direct all edges from L to R and assign infinite (or unit) capacity
- Add unit-capacity edges from s to each node in L
- Add unit-capacity edges from each node in R to t

1-to-1 correspondence between matchings of cardinality k in G and integral flows of value k in G'

If exists integral flow (# edges of flow) of value k then exists in G' too

↳ can solve bipartite matching with max-flow formulation

FF algorithm need time $O(mn)$ to find max-cardinality matching in a bipartite graph with $|L|=|R|=n$

A flow network is a simple unit-capacity network if every edge has capacity 1 and every node ($\neq s \wedge \neq t$) has exactly 1 entering and/or 1 leaving edges. In this kind of network Dinitz's algorithm compute maximum flow in $O(m\sqrt{n})$ time

Structure of perfect matching: of course $|L|=|R|$ + iff:

- given S subset of nodes and $N(S)$ set of nodes adjacent to S → $|N(S)| \geq |S|$ $\forall$ subset $S \subseteq L$ $(N(S) \subseteq R)$ Hall's marriage theorem

A sufficient condition to have a perfect matching is that the graph is a k-regular bipartite graph

If not bipartite, finding a max-cardinality matching is a bit more complicated but best known algorithm computes it in time $O(m\sqrt{n})$ Blossom algorithm in $O(n^4)$.

DISJOINT PATHS

Two paths are edge-disjoint if they dont have edge in common. Edge-disjoint path problem: find max number of edge-disjoint $s \to t$ path.

Max flow formulation: assign unit capacity to every edge. There is 1to1 correspondence between k edge-disjoint $s \to t$ paths in G and integral flow of value k in G'. Th.

$F \subseteq E$ disconnects t from s if every $s \to t$ paths uses at least one edge in F. → Network Connectivity find min number of edges whose removal disconnects s from t.

Th. Max number of edge-disjoint $s \to t$ paths = min number of edges whose removal disconnects s from t.

Some things for undirected graphs. Two paths P_1 and P_2 may be edge-disjoint in the digraph but not in the (some) undirected graph. Note: bidirectional edge is described as a pair of two antiparallel edges both with unit capacity

In any flow network exists a max flow f in which for each pair of antiparallel edges e and e', either $f(e)=0$ or $f(e')=0$ or both,

Th. Max number edge-disjoint $s \to t$ paths = max flow value

MENGER'S THEOREMS

1. Given an undirected graph with nodes s and t, the max number of edge-disjoint $s \to t$ paths equals the min number of edges whose removal disconnects s and t
2. Given an undirected graph with nodes s and t nonadjacent, the max number of internally node-disjoint $s \to t$ paths equals the min number of internal nodes whose removal disconnects t from s
3. Given a directed graph with two nonadjacent nodes s and t, the max number of internally node-disjoint $s \to t$ paths equals the min number of internal nodes whose removal disconnects t from s

CIRCULATION

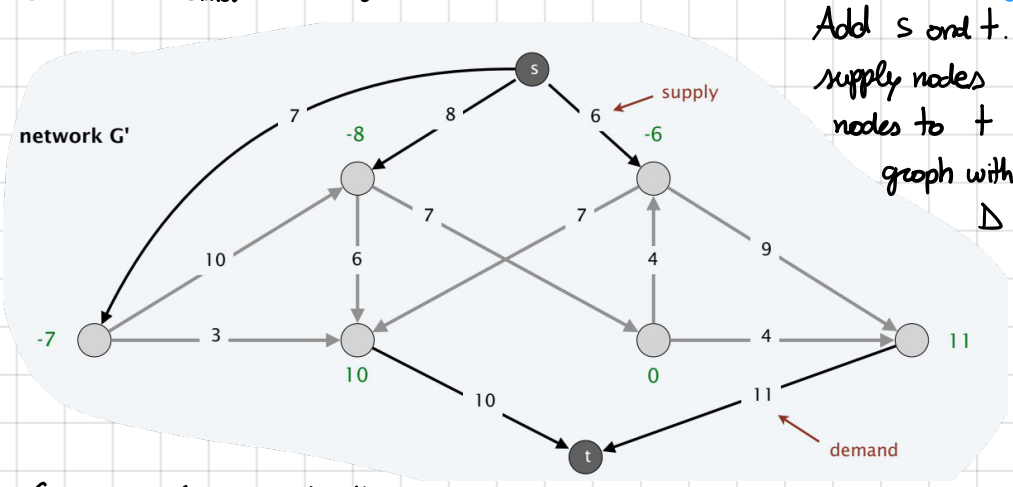
Given a digraph with nonnegative edge capacities $c(e)$ and node supply and demands $d(v)$, circulation is a function that satisfies:

① $\forall e \in E: 0 \le f(e) \le c(e)$ capacity

② $\forall v \in V: \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$ conservation

Nodes with negative $d(v)$ are supply nodes, demand otherwise. Max flow formulation:

Add s and t. Add edge with capacity $-d(v)$ from s to supply nodes and edge with capacity $d(v)$ from demand nodes to t → G has circulation iff. G' (the graph with s and t added) has max flow $\Delta = \sum_{\text{demand}} d(v) = \sum_{\text{supply}} -d(v)$



Given $G = (V, E, c, d)$ there doesn't exist a circulation iff there exists a node partition (A, B) such that $\sum_{v \in B} d(v) > cap(A, B)$

CIRCULATION WITH DEMANDS AND LOWER BOUNDS

In addition we have a lower bound for each edge $e \to \ell(e)$

In this case circulation satisfies:

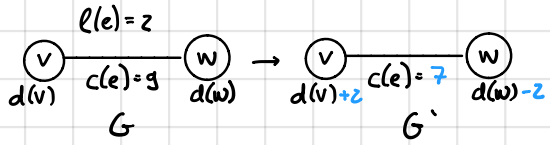
① $\forall e \in E: \ell(e) \le f(e) \le c(e)$

② $\forall v \in V: \sum_{e \text{ into } v} f(e) - \sum_{e \text{ out of } v} f(e) = d(v)$

Max flow formulation:

Send $\ell(e)$ units of flow along e and update demands of both endpoints:

There exists a circulation in G iff there exists in G'



SURVEY DESIGN

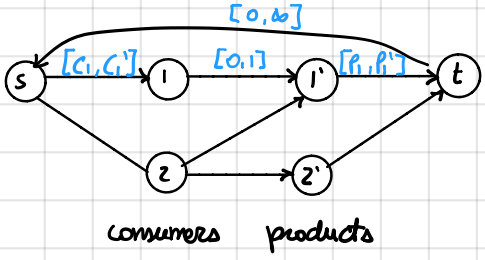
Design survey asking n_1 consumers about n_2 products. Can survey consumer i about product j if i own j.

Ask consumer i between c_i and c_i' questions. Ask between p_j and p_j' consumers about product j.

How? Max flow formulation as circulation problem with lower bounds.

Add edge (i, j) if i owns j. Add edge from s to consumers i. Add edge from products j to t

Add edge from t to s $\Longrightarrow$ Circulation $\Longleftrightarrow$ feasible survey design

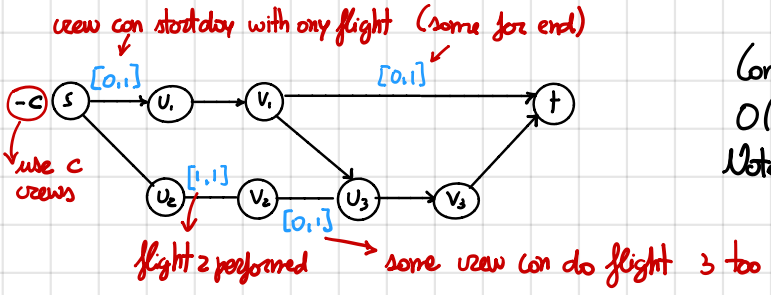


AIRLINE SCHEDULING

Produces schedules that are efficient in term of equipment usage, crew allocation, consumer satisfaction in presence of unpredictable issues like weather, breakdowns etc.

TOY PROBLEM: manage crews by reusing them over multiple flights. Input = k flights for a given day. Flight i leaves origin o_i at time s_i and arrives to d_i at time f_i $\Rightarrow$ minimize number of flight crews.

Circulation formulation: $\forall$ flight i, include two nodes u_i and v_i. Add source s with demand $-c$ (use c crews) and edges (s, u_i) with capacity 1. Add sink t with demand c and edges (v_i, t) with capacity 1. $\forall i$ add edge (u_i, v_i) with lower bound and capacity 1. If flight j reachable from i add edge (v_i, u_j) with capacity 1



Can be solved in $O(k^3 \log k)$ time

$O(k^3)$ time by formulating as minimum flow problem

Note: Real world problems are much more complex!

PROJECT SELECTION

Set of projects P: project v has associated revenue p_v (negative too). Set of prerequisites E: if $(v,w) \in E$ cannot do v unless doing project w too. A subset of projects $A \subseteq P$ is feasible if the prerequisite of every project in A also belong to A $\Rightarrow$ maximize revenue.

Min Cut formulation: assign capacity ∞ to all prerequisite edge. Add edge (s,v) with capacity p_v if $p_v > 0$. Add edge (v,t) with capacity $-p_v$ if $p_v < 0$. $p_s = p_t = 0$

$\Rightarrow (A,B)$ is a min cut iff $A - \{s\}$ is an optimal set of projects

INTRACTABILITY

Algorithm design **patterns**: greedy, divide & conquer, dynamic programming, duality (max flow / min cut), **reductions** = reduce problem in easier ones (but that could cost)

Algorithm design **antipatterns**: NP-completeness ($O(n^k)$)

We will classify algorithms according to computational requirements: some problems can't be solved in practice in polynomial time.

REDUCTION: given X that can be solved in polytime, we say X **polynomial reduces to** problem Y if arbitrary instance of X can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves Y

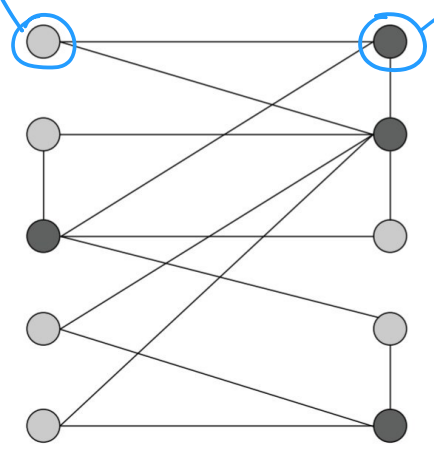
In this case we say $X \leq_P Y$:

- Y solved in polytime $\Rightarrow$ X solved in polytime
- X not solved in polytime $\Rightarrow$ Y not solved in polytime (Establish Intractability)

If $X \leq_P Y$ and $Y \leq_P X$ we say $X \equiv_P Y$ (Establish Equivalence)

REDUCTION by EQUIVALENCE

INDEPENDENT SET: given $G = (V, E)$ and integer k, find a subset of vertices $S \subseteq V$ s.t. $|S| \geq k$ and $\forall e \in E$ at most one of its endpoints is in S.



VERTEX COVER: Given $G = (V, E)$ and integer k, find subset of vertices $S \subseteq V$ s.t. $|S| \leq k$ and $\forall e \in E$ at least one of its endpoints is in S.

Th. Independent Set $\equiv_P$ Vertex Cover

Proof: we show S is independent set iff. $V-S$ is a vertex cover

$\Rightarrow$

Let S be independent set. Consider arbitrary edge (u,v).

S independent set $\Rightarrow u \notin S$ or $v \notin S \Rightarrow u \in V-S$ or $v \in V-S$

Thus $V-S$ covers (u,v)

$\Leftarrow$

Let $V-S$ be a vertex cover. Consider two nodes $u \in S$ and $v \in S$. $(u,v) \in E$ since $V-S$ is a vertex cover. Thus no two nodes in S are joined by an edge $\Rightarrow$ S independent set.

REDUCTION from SPECIAL CASE to GENERAL CASE

Prove that a problem is a special case of a general one.

SET COVER: given set U of elements, a collection $S_1, S_2 \dots S_m$ of subset of U, an integer k, find a collection of $\leq k$ of these sets whose union is equal to U

Eg. m available pieces of software. Set U of n capabilities that we want in our system. I^{TH} piece of software provides the set $S_i \subseteq U$ of capabilities $\rightarrow$ achieve all n capabilities using fewest pieces of software.

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

$k = 2$

$S_1 = \{3, 7\}$

$S_2 = \{3, 4, 5, 6\}$

$S_3 = \{1\}$

$S_4 = \{2, 4\}$

$S_5 = \{5\}$

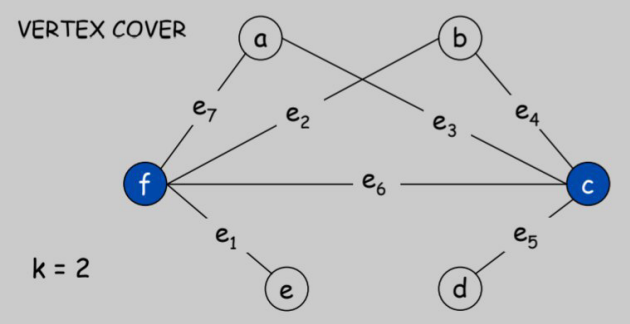
$S_6 = \{1, 2, 6, 7\}$

Th. VertexCover $\leq_p$ SetCover

Proof: given a vertex cover instance $G=(V,E)$, k we construct a set cover instance whose size equals the size of the vertex cover.

Create set cover instance: $-k=k$, $U=E$, $S_v=\{e \in E : \text{incident to } v\}$

Set cover of size $\leq k$ iff. vertex cover of size $\leq k$



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

$k = 2$

$S_1 = \{3, 7\}$

$S_2 = \{3, 4, 5, 6\}$

$S_3 = \{1\}$

$S_4 = \{2, 4\}$

$S_5 = \{5\}$

$S_6 = \{1, 2, 6, 7\}$

SATISFABILITY (REDUCTION with GADGETS)

Conjunctive normal form = formula $\phi = C_1 \wedge C_2 \wedge C_2 \wedge C_4 \ldots$ where C_i = clause = $x_1 \vee \bar{x}_2 \vee x_3$ (literals)

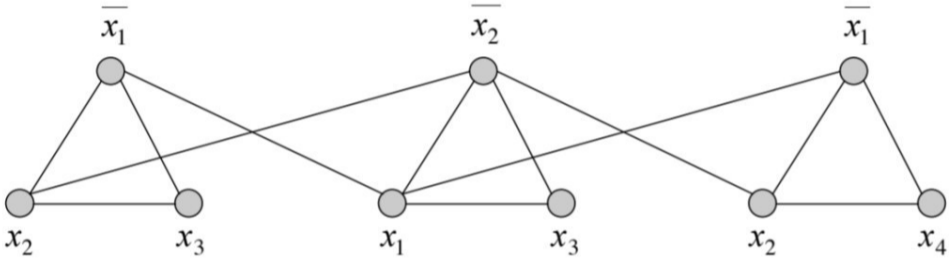
SAT: given ϕ does it have a satisfying truth assignment? 3-SAT: SAT where each clause has 3 literals

Eg. $\Phi = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$ $x_1 = x_2 = \text{true}$ $x_3 = x_4 = \text{false}$

Th. 3-SAT $\leq_p$ Independent Set

Proof: Given an instance Φ of 3-sat we construct an instance (G,k) of independent set that has an independent set of size k iff. Φ is satisfiable

G contains 3 vertices for each clause, one for each literal. Connect 3 literals in a clause in a triangle. Connect literals to each of its negation.



$k = 3$

$$\Phi = (\bar{x}_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$$

G contains independent set of size $K = |\Phi|$ iff. Φ is satisfiable

Proof:

$\Rightarrow$ Let S be independent set of size k. S must contain exactly one vertex in each triangle. Set these literals to true (and any other variable in a consistent way). Truth assignment is consistent and all clauses are satisfied.

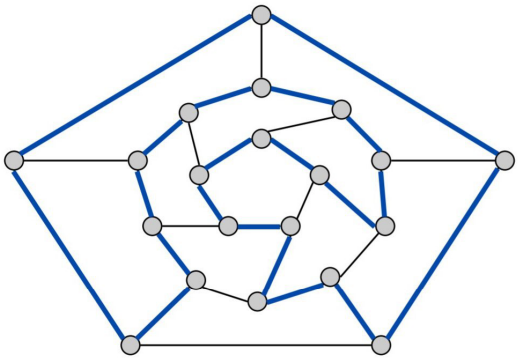
$\Leftarrow$ Given satisfying assignment, select one true literal from each triangle. This is an independent set of size k

Transitivity: if $X \leq_p Y$ and $Y \leq_p Z \rightarrow X \leq_p Z$

$\Rightarrow$ 3Sat $\leq_p$ IndipendantSet $\leq_p$ VertexCover $\leq_p$ SetCover

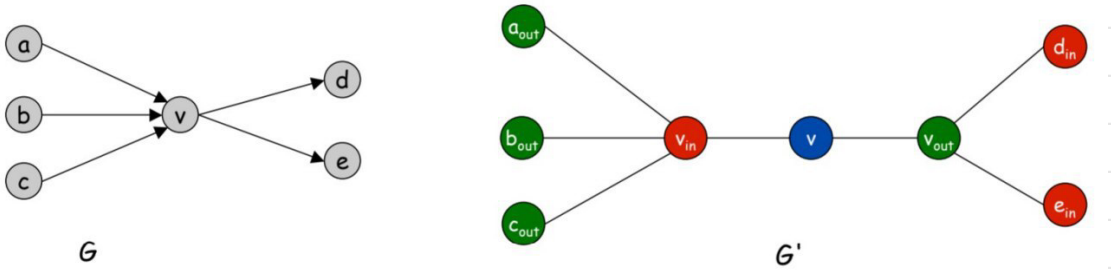
HAMILTONIAN CYCLE

Given an undirected graph find a symple cycle Γ that contains every node (visited once).



We can find Hamiltonian cycle in digraph too Th. DIR-HC $\leq_p$ HC

Proof Given a directed graph $G=(V,E)$ construct an undirected graph G' with $3n$ nodes



$\Rightarrow$ Suppose G has a directed Hamiltonian cycle Γ. Then G' has an undirected HC (some order)

$\Leftarrow$ Suppose G' has an undirected HC Γ'. Γ' must visit nodes in G' using one of following two order:

... B, G, R, B, G, R, B, G, R ... or ... B, R, G, B, R, G, B, R, G ...

Blue nodes in Γ' make up directed Hamiltonian cycle Γ in G, or reverse of one.

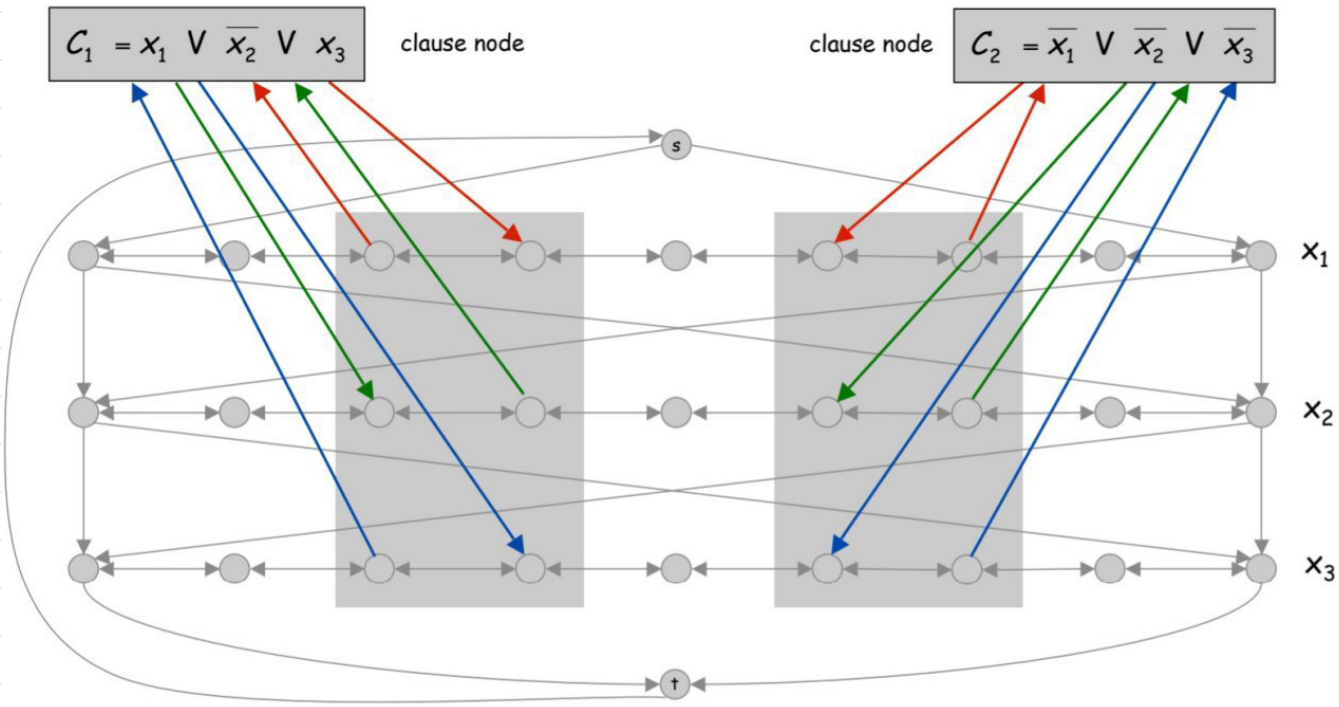
Th. 3Sat $\leq_p$ DIR-HC.

Proof: Given an instance Φ of 3-sat we construct an instance of DIR-HC that has a HC iff. Φ is satisfiable.

First we construct graph that has 2^n HCs which correspond in a natural way to 2^n possible truth assignments.

Given 3-sat instance Φ with n variables x_i and k clauses construct G to have 2^n HCs. Intuition: traverse path i from left to right $\Leftrightarrow$ set variables $x_i = 1$.

Given 3-sat instance Φ with n variables x_i and k clauses. For each clause add a node and 6 edges.



Φ is satisfiability iff. G has HC.

$\Rightarrow$ Suppose 3-sat instance has satisfying assignment x^*. Then define HC in G as follows:

- if $x_i^* = 1$ traverse row i from left to right
- if $x_i^* = 0$ traverse row i from right to left
- for each clause C_j there will be at least one row i in which we are going in "correct" direction to splice node C_j into tour

$\Leftarrow$ Suppose G has HC Γ. If Γ enters clause node C_j it must depart on note edge. Thus nodes immediately before and after C_j are connected by an edge e in G. Removing C_j from cycle and replacing it with edge e yields HC on $G - \{C_j\}$

Continuing in this way we are left with HC Γ' in $G - \{C_1, C_2 \dots C_k\}$

Set $x_i^* = 1$ iff Γ' traverses row i left to right

Since Γ visits each clause node C_j, at least one of the paths is traversed in "correct" and each clause is satisfied.

LONGEST PATH

Find a path of length at least k

Th. 3SAT $\leq_p$ LONGEST PATH

Proof: Redo proof for DIR-HC ignoring back-edge from t to s. Show HC $\leq_p$ Longest Path

TRAVELING SALESMAN PROBLEM (TSP): given n cities and pairwise distance function d(u,v) find a tour of length $\leq D$ (every pairs)

Hamiltonian cycle contains every nodes $\rightarrow$ HAM-CYCLE $\leq_p$ TSP. Proof:

Given G instance of HAM-CYCLE create n cities with $d(u,v) = 1$ if $(u,v) \in E$, 2 otherwise $\rightarrow$ TSP has a tour of length $\leq n$ iff. G is Hamiltonian

3-COLOR

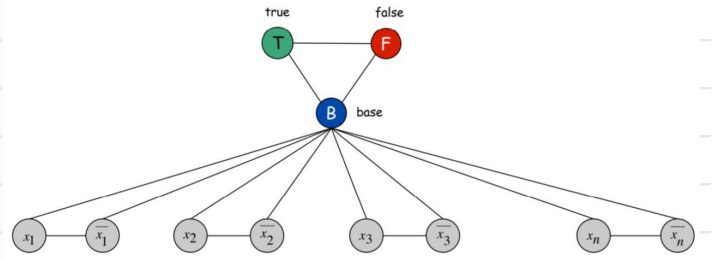
Given undirected graph, color nodes of blue, green or red such that no adjacent nodes have same color. An example of program is assigning variables to machine register so that no more than k registers are used and no two variables needed at same time are assigned to same register.

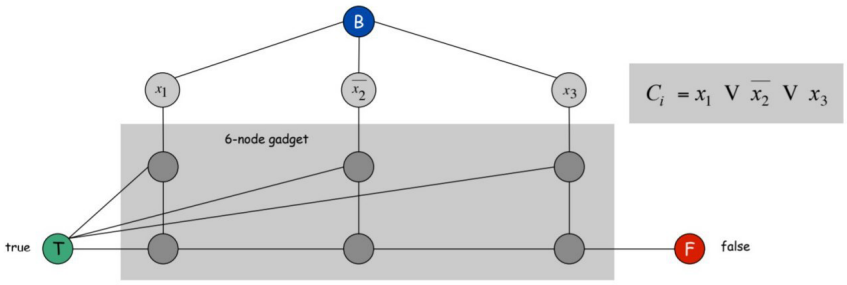
3-COLOR $\leq_p$ K-REGISTER-ALLOCATION $(k \geq 3)$

Th. 3-SAT $\leq_p$ 3-COLOR

Proof: Given 3-sat instance Φ we construct an instance of 3-color that is 3 colorable iff. Φ is satisfiable.

For each literal create a node, create 3 nodes T, F, B, connect them in triangle and each literal to B. Connect each literal to its negation. For each clause add gadget of 6 nodes and 13 edges.

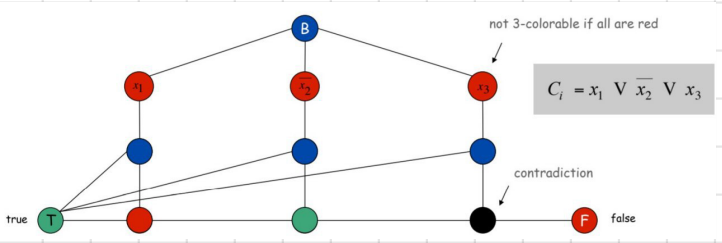


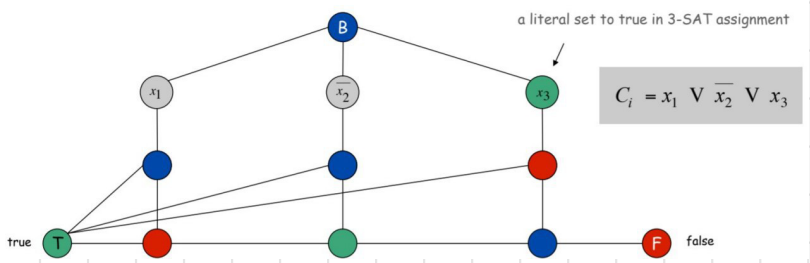


Graph is 3-colorable iff. Φ is satisfiable.

$\Rightarrow$ Suppose graph is 3-colorable. Consider assignment that sets all T literals to true. Ensures each literals is T or F. Ensures a literal and its negation are opposites. Ensures at least one literal in each clause is T.

$\Leftarrow$ Suppose 3-sat formula Φ is satisfiable. Color all literals T. Color node below green node F, and node below that B. Color remaining middle row nodes B. Color remaining bottom nodes T or F as forced





A particular case of 3-color is planar-3-color, 3-coloring a map such that adjacent regions have different colors. A graph is planar if it can be embedded in the plane in such a way that no two edges cross

Th. PLANAR-3-COLOR $\leq_p$ PLANAR-GRAPH-3-COLOR

Th. 3-COLOR $\leq_p$ PLANAR-GRAPH-3-COLOR

NUMERICAL PROBLEMS

Subset Sum: given natural numbers $w_1 \dots w_n$ and a integer W find a subset that adds up to W.

3Sat $\leq_p$ Subset Sum

KNAPSACK: given a set of items X, weights $u_i \geq 0$, values $v_i \geq 0$, a weight limit U and a target value V find a subset $S \subseteq X$ s.t. $\sum_{i \in S} u_i \leq U$ $\sum_{i \in S} v_i \geq V$

SubSet $\leq_p$ Knapsack

DECISION PROBLEMS

X is a set of strings (instances = string s). Algorithm A solves problem x: A(s) = yes iff $s \in X$

Polynomial time: Algorithm A runs in polytime if ∀ s A(s) terminates in $p(|s|)$ "steps" with $|s|$ = length of s and p = polynomial.

Many decision problems can be solved in polytime such as find if a number is prime or two numbers are relatively prime.

Algorithm C(s,t) is a **certifier** for problem X if ∀s, $s \in X$ iff there exists a string t (**certificate**) such that C(s,t) = yes

NP = decision problems for which exists a polytime certifier = C(s,t) polytime algorithm and $|t| \leq p(|s|)$.

E.g.

SAT: given Φ, exists a satisfying assignment?

CERTIFICATE: an assignment of truth values to the n boolean variables

CERTIFIER: check that each clause of Φ has at least one true literal.

} SAT is in NP

E.g.

HAM-CYCLE: exists a simple cycle that visits every nodes?

CERTIFICATE: a permutation of the n nodes

CERTIFIER: check that the permutation contains all nodes and an edge between each pair of adjacent nodes

HAM-CYCLE is in NP

P decision problems with polytime algorithm

EXP " with exponential time algorithm

NP " with polytime certifier

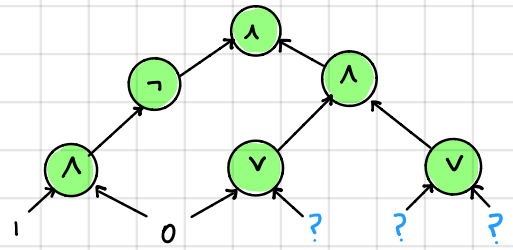
→ $P \subseteq NP$, $NP \subseteq EXP$

Problem X **polynomial transform** (Karp) to problem Y if given any input x to X we can construct an input y such that x is a yes instance of X iff. y is a yes instance of Y.

Note: polynomial transformation is polynomial reduces with just one call to oracle for Y at the end of algorithm for X.

NP-COMPLETE: problem Y in NP such that ∀ problem X in NP, $X \leq_p Y \rightarrow$ Y solvable in polytime iff. P = NP → Possible?

CIRCUIT-SATisfobility: given a combinational circuit built with AND, OR, NOT gates can we set inputs in a way to have output 1?



⇒ 101

Circuit-Sat is NP-complete.

Any algorithm that takes fixed n bit and produces yes/no can be represented by such a circuit. Algorithm in polytime means circuit of poly-size.

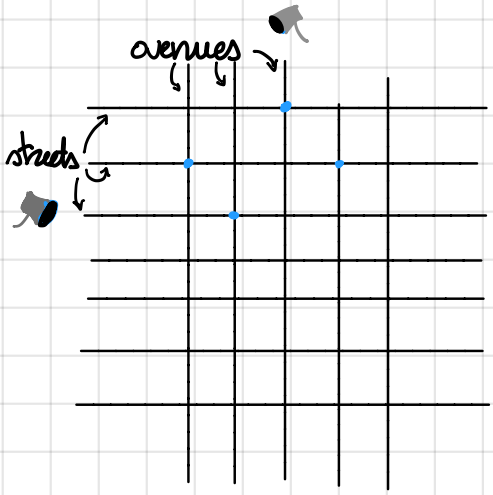
3-SAT is NP-complete given by the fact that CIRCUIT-SAT $\leq_p$ 3-SAT since 3-SAT in NP

In NP, given a decision problem X, its **complement** $\bar{X}$ is the same problem with yes and no answers reverse

→ co-NP = complements of decision problem in NP

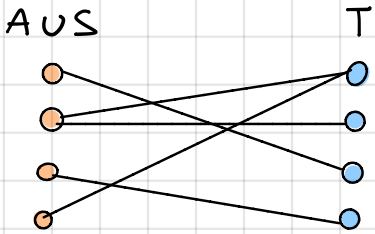
If NP ≠ co-NP then $N \neq NP$

MANHATTAN VERTEX COVER



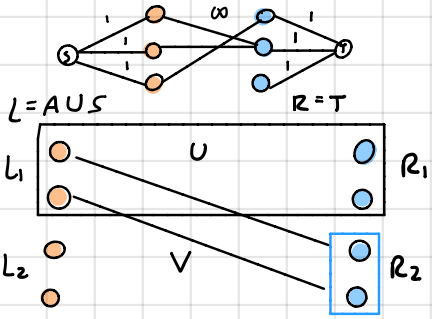
$T \subseteq A \times S$

- Place min number of cameras in order to cover all target points.
- Discuss the complexity of the algorithm.

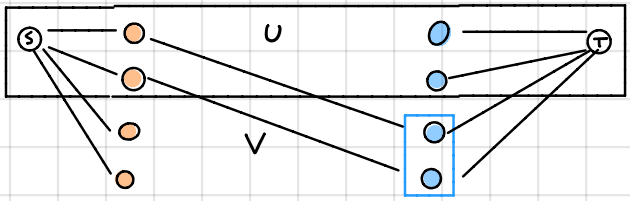


$\Rightarrow$ Vertex cover in bipartite graph.

Cameras on a street/avenue sees all targetpoints in that street/avenue.
We wont place cameras only on the left part of the bipartite graph.
Starting from min cut we compute max matching:



$L_1 = L \cap U$
$R_1 = R \cap U$
$L_2 = L \setminus L_1$
B = vertices of R_2 with at least one edge to L_1



$C = L_2 \cup R_1 \cup B$

lemma 1: C is a vertice cover

lemma 2: $|c|$ is at most equal to the size of min cut

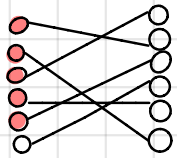
mincut $K = |L_2| + |R_2| + |\text{egdes}(L_1, R_2)|$

$\geq |L_2| + |R_1| + |B| = |c|$

lemma 3: $|\text{max matching}| \leq |\text{Vertex Cover}|$

$\leq |C| \leq |\text{mincut}|$

$\Rightarrow$



$A[1 \dots n]$ Find indices i, j that maximize $\prod_{k=i}^{j} A[k]$ in linear time.

Find maximum value $\prod_{k=i}^{j} A[k]$ $\forall i, j \in [1 \dots n]$

There are value $\leq 1 \Rightarrow$ can't multiply all numbers! UNORDERED!

e.g. 0.001 | 100 | 0.33 | 1000 | 0.0001

mox

Idea: multiply untill > 1. If < 1 stop and change sequence.

$$vol[k] = \begin{cases} A[k] & \text{if } vol[k-1] < 1 \\ A[k] \cdot vol[k-1] & \text{otherwise} \\ A[i] & \text{if } k = 1 \end{cases}$$

Output mox $vol[k]$ $k \in [1 \dots n]$

```
Im , Jm = 0
mox = 0
i, j = 0
vol = 0
for k=1 to n do
   if vol < 1 then
        vol = A[k]
        i, j = k
   else
      vol = A[k] · vol
      j = j+1
   end if
   if vol > mox then
      mox = vol
      Im = i , Jm = j
    end if
end for
```

APPROXIMATION ALGORITHMS

If we need to solve an NP-hard problem we sacrifice the property "solve problem to optimality"
→ ρ-approximation: run in polytime, solve arbitrary instance of problem, find solution within ratio ρ of true optimum.
→ Problem: need to prove that solution is close to opt without knowing opt.

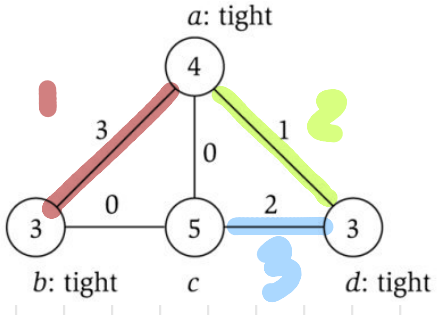
LOAD BALANCING

m machines, n jobs with processing time t_j. Must run contiguously on one machine. a machine can have one job at time.
LOAD: $L_i = \sum_{j \in i} t_j$ = sum of processing times of job running on machine i
MAKESPAN: $L = \max_i L_i$ = max load on any machine.
→ Goal: assign jobs minimizing makespan. → Given n jobs, assign them one by one to the machine whose load is smallest so far.
Lemma 1: optimal makespan $L^* \geq \max_j t_j$ because some machine must process the most time-consuming job.
Lemma 2: $L^* \geq \frac{1}{m} \sum_j t_j$ because total processing time = $\sum_j t_j$ and one of m machine must do at least 1/m of it.
Th: Greedy algorithm is 2-approximation
Pf: Suppose machine i has highest load, when job j is assigned to it, i had the smallest load = $L_i - t_j$
→ $L_i - t_j \leq L_k \quad \forall k \quad 1 \leq k \leq m$. $\boxed{L_i - t_j} \leq \frac{1}{m} \sum_k L_k \rightarrow = \frac{1}{m} \sum_k t_k \rightarrow \boxed{\leq L^*}$ by Lemma 2
Now $L = L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$

What if? before running the algorithm, sort jobs in decreasing order of processing times.
Lemma 3: if there are more than m jobs, $L^* \geq 2t_{m+1}$ because each jobs before j_{m+1} has $t_j > t_{m+1}$ ($t_1 \geq t_2 \geq t_3 \ldots \geq t_{m+1}$ for the ordering) and since there are m+1 jobs and m machines by pidgeonhole principle at least one machine get two jobs.
Th. Longest Processing time (LPT) is 3/2-approximation
Pf: As before, $L = L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq \frac{1}{2} L^*}$ for lemma 3 because $t_j \leq t_{m+1}$

PRICING METHOD

Weighted Vertex Cover: find a vertex cover with min weight
Pricing Method: each edge must be covered by some vertex. Edge $e = (i,j)$ pays $p_e \geq 0$ to use both i and j
Edges incident to i should pay $\leq w_i$ in total → $\forall i \sum_{e=(i,j)} p_e \leq w_i$
$\sum_e p_e \leq w(S)$ (S = vertex cover) because $\sum_{e \in E} p_e \leq \sum_{i \in S} \sum_{e=(i,j)} p_e \leq \sum_{i \in S} w_i = w(S)$ Fairness lemma
↑ every edge covered by at least one node in S
Vertex i is tight if the sum of its incident edges pay is equal to w_i
- Pick edges one by one and increase pay until one of its two node is tight (eg $e_1 = (a,b)$ increased until 3)
- Put that edge in cover and go next (e.g. $e_2 (a,d)$ $p_{e_2} = 1$ cause a needs only one more to reach its weight of 4



Th. Pricing method is 2-approx. of weighted-Vertex-Cover

Pf. algorithm terminates since at least one new node become tight after each iteration of the loop

S = set of tight nodes at the end of it = vertex cover. S^* = optimal vertex cover : $w(S) \le 2w(S^*)$

$$\rightarrow w(S) = \sum_{i \in S} w_i = \sum_{i \in S} \sum_{e=(i,j)} p_e \le \sum_{i \in V} \sum_{e=(i,j)} p_e = 2 \sum_{e \in E} p_e \le 2 w(S^*)$$

all nodes in S tight — $S \subseteq V$, prices ≥ 0 — each edge counted twice — Fairness lemma

KNAPSACK

Bag of size B, n items, each item has size s_i and value v_i.

Goal: select $A \subseteq N$ s.t. $\sum s_i \le B$ $\wedge$ $\max \sum v_i$

$A(i,p)$ = minimum size that has value p at item i

$$A(i+1,p) = \min \begin{cases} \text{if we skip } i+1 \rightarrow A(i,p) \\ \text{if we keep } i+1 \rightarrow s_{i+1} + A(i, p - v_{i+1}) \end{cases}$$

$\rightarrow O(n^2 \cdot v_{max})$ $\longrightarrow$ Seems polynomial but we can increase time just by set v_{max} very large

$\rightarrow$ State $n \times (n v_{max})$

If v_{max} is reasonably small:

fix $\varepsilon > 0$ let $k = \frac{\varepsilon \cdot v_{max}}{n}$ $\rightarrow \tilde{v}_i = \lfloor v_i / k \rfloor$

↳ Run algorithm on $(B, s, \tilde{v}) \rightarrow (1-\varepsilon)$-approximation

For each item i:

$|k\tilde{v}_i - v_i| \le k \longrightarrow V_{TOTAL} - k\tilde{V}_{TOTAL} \le nk = \varepsilon v_{max}$

$\Rightarrow k\tilde{V}_{TOTAL} \ge V_{TOTAL} - \varepsilon v_{max} \ge (1-\varepsilon) V_{TOTAL}$

$\forall \varepsilon > 0 \rightarrow \exists (1-\varepsilon)$-approximation in polytime.

PRIMAL-DUAL METHOD

$\min \sum_{j=1}^{n} c_j x_j \quad s.t. \quad \sum_{j=1}^{n} a_{ij} x_j \geq b_i \quad i = 1 \dots m \qquad x_j \geq 0 \quad j = 1 \dots n$

$\max \sum_{i=1}^{m} b_i y_i \quad s.t. \quad \sum_{i=1}^{m} a_{ij} y_i \leq c_j \quad j = 1 \dots n \qquad y_i \geq 0 \quad i = 1 \dots m$

DUALITY THEOREM: If the primal has finite optimum the the dual has finite optimum. Let x^* and y^* be the primal and the dual optimum solutions. Then

$$\sum_{j=1}^{n} c_j x_j^* = \sum_{i=1}^{m} b_i y_i^*$$

WEAK DUALITY THEOREM: if x is feasible for the primal and y for the Dual, then

$$\sum_{j=1}^{n} c_j x_j \geq \sum_{i=1}^{m} b_i y_i \rightarrow \text{Proof} \rightarrow \sum_{j=1}^{n} c_j x_j \geq \sum_{j=1}^{n} \left(\sum_{i=1}^{m} a_{ij} y_i \right) x_j = \sum_{j=1}^{n} \left(\sum_{j=1}^{m} a_{ij} x_j \right) y_i \geq \sum_{i=1}^{m} b_i y_i$$

x and y are optimal solution iff.

① Primal Complementary Slackness Condition: $\forall 1 \leq j \leq n$ either $x_j = 0$ or $\sum_{i=1}^{m} a_{ij} y_i = c_j$

② Dual Complementary Slackness Condition: $\forall 1 \leq i \leq m$ either $y_i = 0$ or $\sum_{j=1}^{n} a_{ij} x_j = b_i$

Construct algorithm ensuring PCSC and relax DCSC: $\forall 1 \leq i \leq m$ either $y_i = 0$ or $\sum_{j=1}^{n} a_{ij} x_j \leq r \times b_i$ ↳ approximation

Th. If x and y satisfy primal-dual scheme then $\sum_{j=1}^{n} c_j x_j \leq r \times \sum_{i=1}^{m} b_i y_i$

Primal is a relaxation of a problem P, x is integral feasible for P $\rightarrow \sum_{j=1}^{n} c_j x_j \leq r \times \sum_{i=1}^{m} b_i y_i \leq r \times \sum_{i=1}^{m} b_i y_i^* = r \times \sum_{j=1}^{n} c_j x_j^* \leq r \times OPT$

$\rightarrow$ r-approx algorithm

RANDOMIZED ALGORITHMS

Randomization: many times is the only way to solve problems eg. hashing, cryptography etc.

CONTENTION RESOLUTION

Avoid that multiple process access shared resource at same time. (eg database)

Protocol: process request access to database at time t with probability $p = 1/n$

$\rightarrow S[i,t]$ = i access database at time t $\rightarrow \frac{1}{e \cdot n} \leq P[S[i,t]] = \underbrace{p}_{i \text{ access}} \cdot \underbrace{(1-p)^{n-1}}_{\text{others not}} \leq 1/2n$

The prob. that i fails to access in each round is at most $1/e$. After $e \cdot n(c \ln n)$ rounds the prob. is $\leq n^{-c}$ amplification fact.

Fails can be not indipendent eg. more processes access at same time: all fail)

$P[F[t]]$ = prob of at least one of n process fails to access in any round from 1 to t

$= P[\bigcup^{n} F[i,t]] \leq \sum^{n} P[F[i,t]] \leq n(1-1/en)^t$

↑ union bound

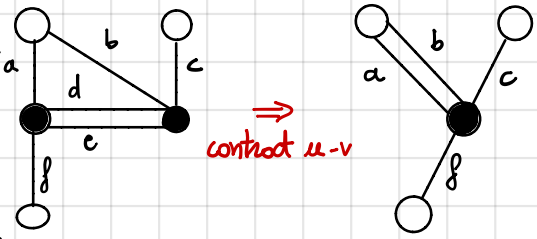
if $t = 2n \lceil en \rceil \lceil c \ln n \rceil \rightarrow P[F[t]] \leq n \cdot n^{-2} = 1/n$

Probability that all process succed within $2e \cdot n \ln n$ round is $\geq 1 - 1/n$

GLOBAL MIN CUT

Given connected, undirected graph find a cut of min cardinality.

Contraction algorithm: pick edge $e = (u,v)$ at random and contract it:



1. replace u and v with single new node w
2. preserve edges updating endpoints of u and v to w
3. keep parallel edges but not selfloops

Repeat until you have only 2 nodes u_1 and v_1 then return the cut

It returns a min cut with probability $\geq 2/n^2$ not always $\rightarrow$ repeat multiple times and take the minimum.

If we repeat $n^2 \ln n$ times: $\underbrace{(1-2/n^2)}_{1 - \text{Prob to fail}}^{n^2 \ln n} = \left[(1-2/n^2)^{n^2/2}\right]^{2 \ln n} \leq \underbrace{(e^{-1})^{2 \ln n}}_{(1-1/x)^x \leq 1/e} = 1/n^2 = \text{prob to fail.}$

EXPECTATION

Given discrete random variable X its expectation $E[x]$ is $\sum^{\infty} j \, P[X=j]$

If $X = 0 \vee 1 \rightarrow E[x] = P[X=1]$

Given two random variable X and Y over same probability space: $E[X+Y] = E[X] + E[Y]$

$\Rightarrow$ Reduce complex calculation into simpler ones. Linearity of expectation

CHERNOFF BOUNDS

Suppose $X_1 \dots X_n$ are indipendent 0-1 random variables and $X = X_1 \dots + X_n$. Then for any $\mu \geq E[X]$ and for any $\delta > 0$ we have: $P[X > (1+\delta)\mu] < \left[\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right]^{\mu}$

↑ sum of indipendent 0-1 random variables is tightly centered on the mean.

For any $\mu \leq E[X]$ and $0 < \delta < 1$ we have: $P[X < (1-\delta)\mu] < e^{-\delta^2 \mu/2}$

GAME THEORY

Games in strategic normal form:
- N players
- $\forall i \in N$ we have a set of strategies S_i
- $S = S_1 \times S_2 \times \dots \times S_n$ = state of the game
- $\forall i \in N$ $u_i : S \to \mathbb{R}$ = utility or payoff function

e.g. rock, paper, scissor game

P1, P2	rock	paper	scissors
rock	0 draw 0	-1 P2 wins +1	+1 P1 wins -1
paper	+1 P1 wins -1	0 draw 0	-1 P2 wins +1
scissors	-1 P2 wins +1	+1 P1 wins -1	0 draw 0

From the number in blue we can construct 2 matrix one for P1 and one for P2

0-sum game: sum of strategies of players is zero.
symmetric game: simmetric matrix, same strategy same utility.

e.g prisoner's dilemma

c_1, c_2	confess	silent
confess	4, 4	1, 5
silent	5, 1	2, 2

Both confess → both 4 years of jail
One confess → 1 years and other 5
both silent → both 2 years.

This is not a 0-sum game.
without knowing the other's strategy the best choice is to confess → DOMINANT STRATEGY SOLUTION
State s is a dominant strategy solution if $\forall i \in N$ and $\forall s' \in S$ $u_i(s_i, s'_{-i}) \geq u_i(s'_i, s'_{-i})$
$S_{-i} = (S_1 \dots S_{i-1}, S_{i+1} \dots S_n)$ = S without S_i
time → $\sum_{i \in N} |s_i| \cdot |s'_{-i}| \cdot |s_i| = \sum_{i \in N} |s_i| \cdot |S|$
⇒ S_i always better than S'_i
Problem: not all game have one dominant strategy solution

e.g. chicken game

u_1, u_2	deviate	straight
deviate	0, 0	-1, 5
straight	5, -1	-100, -100

PNE

two cars, one in front of other. If the both go straight they crash.
Very stupid game!
No dominant strategy solution.

PURE NASH EQUILIBRIUM

time → $\sum_{i \in N} |S| \cdot |S_i|$

A state is a PNE if $\forall i \in N$ and $\forall s'_i \in S_i$: $u_i(s_i, s_{-i}) \geq u_i(s'_i, s_{-i})$
Assuming that other players are not changing their action, the player is selecting his "best" action.
DSS ⇒ PNE

MIXED NASH EQUILIBRIUM

Players choose a strategy with a certain probability → more realistic.
A mixed strategy for $i \in N$ is a probability distribution on set S_i. $P_i : S_i \to [0,1]$ $\sum_{s_i \in S_i} P_i(s_i) = 1$
A mixed state is a collection of mixed strategies $(s_i)_{i \in N}$
$P(s_i) = P_1(s_1) \times P_2(s_2) \times \dots \times P_N(s)$
$E[u_i(s_j)_{j \in N}]$ = expected utility = $\sum_{s \in S} P(s) \cdot u_i(s)$
A mixed state is a MNE if no player has any incentive to deviate from mixed strategy s_i

Take rock-paper-scissor game as example: $(a+b+c=1 \rightarrow$ second player strategies$)$

first player ↓

$a \cdot \frac{1}{3} \cdot (0) + a \cdot \frac{1}{3} \cdot (1) + a \cdot \frac{1}{3} \cdot (-1) +$ (rock)

$b \cdot \frac{1}{3} \cdot (-1) + b \cdot \frac{1}{3} \cdot (0) + b \cdot \frac{1}{3} \cdot (1) +$ (paper)

$c \cdot \frac{1}{3}(1) + c \cdot \frac{1}{3} \cdot (-1) + c \cdot \frac{1}{3} \cdot (0) = 0$ (paper)

The expected utility is a constant 0, always → no reason to change a, b, or c. → the two players "lock" each other in the MNE

BEST RESPONSE strategy: a mixed strategy P_i is a best response to mixed strategies $P_1 \dots P_{i-1}, P_{i+1} \dots P_N$ if $\forall$ mixed strategies P_i' $\sum_{s \in S} P_1(s_1) \times \dots \times P_i(s_i) \times \dots \times P_N(s_N)\, U_i(s) \geq \sum_{s \in S} P_1(s_1) \times \dots \times P_i'(s_i) \times \dots \times P_N(s_N)\, U_i(s)$

SUPPORT OF MIXED STRATEGY P_i : $supp(P_i) = \{ s \in S_i \,,\, P_i(s) > 0 \}$

⇒ A mixed strategy P_i is best response iff all pure strategies in the support are best response strategies.

$I \subseteq S_1 \quad J \subseteq S_2 \qquad a_{is} \quad b_{is}$

$x_1 \dots x_{|S_1|} \quad y_1 \dots y_{|S_2|} \qquad \forall i \in I \; \forall k \in S_1$

$\sum_{s \in J} y_J\, a_{rs} \leq \sum_{j \in J} y_j\, a_{ij} \qquad \sum_{i \in I} x_i\, b_{ik} \leq \sum_{i \in I} x_i\, b_{ij} \quad \forall J \in J \quad \forall k \in S_2$

$\sum_{i \in I} x_i = 1 \qquad \sum_{j \in J} y_j = 1$

$x_i \geq 0 \qquad y_j \geq 0$