Greedy Algorithms

Optimization Problems: problems based/ involving max or min some functions.

* Using DP to determine best choice is sometimes a overkill.
* Greedy algo.: make the choice that looks best at the moment. Don't considers all subprobs.
* GA does not always yield optimal sol^n.

One possible approach:

Considers DP → Show: can make greedy choice to get optimal sol^n

Topics to cover:

1) Activity Selection Problem
2) ~~Huffman codes~~
3) Matroid theory (combinatorial structures) for greedy algo

* For matroids, greedy algos always produces an optimal sol^n.

Greedy Algorithms in Graphs (to be covered later).

1) MST (Minimum Spanning Trees)
2) Dijkstra's algo for shortest path
3) Chvātal's greedy set-covering heuristics.

An Activity Selection Problem

$S = \{a_1, a_2, \ldots, a_n\}$, a set of n proposed activities that wish to use a resource, say, lecture hall, which can serve only one activity at a time.

For activity a_i

start time: s_i
finish time: f_i, $0 \le s_i \le f_i < \infty$. f_i not included!

* a_i takes place during $[s_i, f_i)$ ← half open set
* a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap, i.e., $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.

In other words, a_i & a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$.

Activity Selection Problem: select a maximum-size subset of mutually compatible activities.

Activities are sorted as:

$f_1 \le f_2 \le f_3 \le \ldots \le f_{n-1} \le f_n$.

Greedy Algorithms

Example

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Q What are mutually compatible activities?

One possibility : $\{a_3, a_9, a_{11}\}$.

a_3 happens in time interval $[0, 6)$

a_9 " " " $[8, 12)$

a_{11} " " " $[12, 16)$

~~Hence~~ since a_3, a_9, a_{11}

Since the time for a_3, a_9, and a_{11} do not overlap, a_3, a_9, and a_{11} are called compatible activities.

second possibility : $\{a_1, a_4, a_8, a_{11}\}$

$[1,4)$ $[5,7)$ $[8,11)$ $[12,16)$

a_1 happens in time interval : $[1, 4)$

a_4 " " " : $[5, 7)$

a_8 " " " : $[8, 11)$

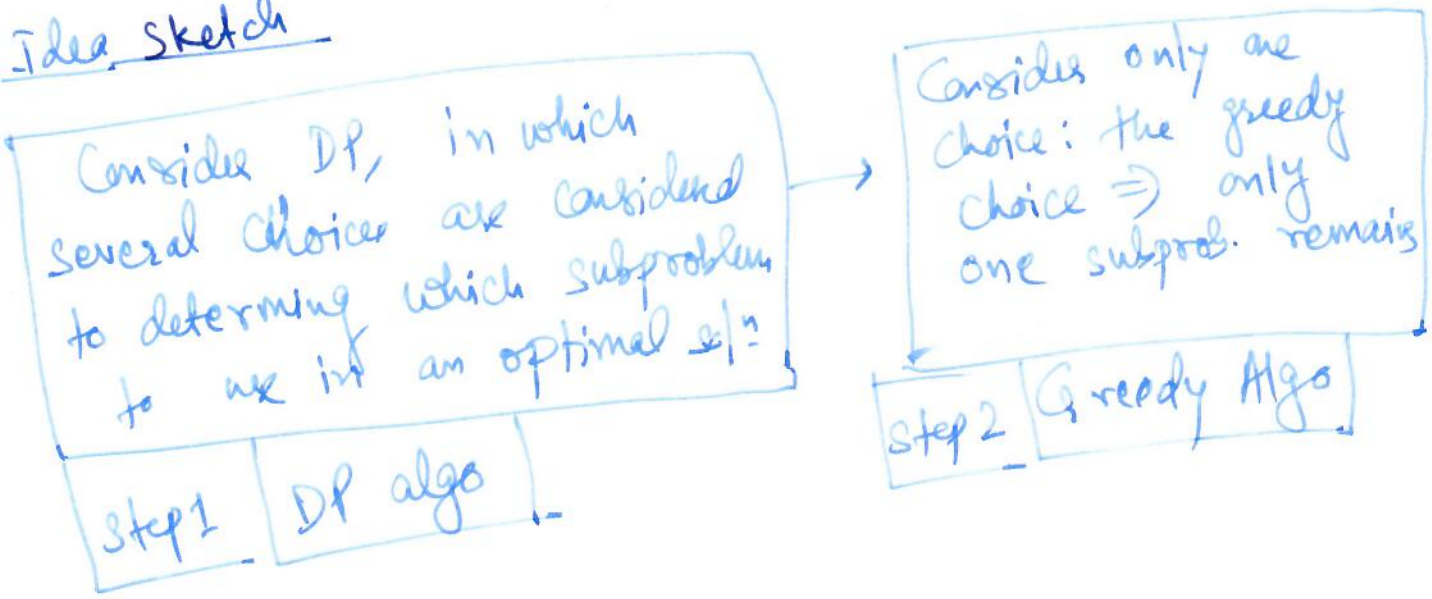
a_{11} " " " : $[12, 16)$

Time intervals of a_1, a_4, a_8, a_{11} do not overlap. Hence, a_1, a_4, a_8, and a_{11} are mutually compatible activities.

Third possibility: $\{a_2, a_4, a_9, a_{11}\}$.

Solving Activity Selection Problem

Idea sketch



optimal substructure

S_{ij} : set of activities that start after activity a_i finishes and that finish before activity a_j starts.

A_{ij} : maximum set of mutually compatible activities in S_{ij}.

Example.

Greed Algo

i	1	2	3	4	4'	5	6	7	8	9	10	11
s_i	1	3	0	5	7	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	9	10	11	12	14	16

Here: $S_{2,11} = \{a_4, a_6, a_7, a_8, a_9\}$

~~$S_{2,11} = \{a_7, a_8, a_9, a_{11}\}$~~

~~$A_{39} = \{$~~

Maximum mutually compatible set in S_{39} denoted by A_{39} is

$A_{39} = \{$

$S_{2,11}$ = set of all activities that start after a_2 finishes and finishes before a_{11} starts

$\{$

a_2 finishes at : 5

a_{11} starts at : 12

Find a_i for which time interval $t_i \subset [5, 12]$

t_i : time interval of activity a_i

Greedy Algo

$t_1 = [1,4) \not\subset [5,12)$

$t_2 = [3,5) \not\subset [5,12)$

$t_3 = [0,6) \not\subset [5,12)$

$t_4 = [5,7) \subset [5,12)$ ✓

$t_{4'} = [7,8) \subset [5,12)$ ✓

$t_5 = [3,9) \not\subset [5,12)$

$t_6 = [5,9) \subset [5,12)$ ✓

$t_7 = [6,10) \subset [5,12)$ ✓

$t_8 = [8,11) \subset [5,12)$ ✓

$t_9 = [8,12) \subset [5,12)$ ✓

$t_{10} = [2,14) \not\subset [5,12)$

$t_{11} = [12,16) \not\subset [12,16)$

Hence,

$S_{2,11} = \{a_4, a_{4'}, a_6, a_7, a_8, a_9\}$

$A_{2,11}$ = set of maximally mutually compatible activities.

~~Possible~~

Mutually Compatible sets are

$\{a_4, a_8\}$, $\{a_4, a_9\}$, ~~$\{a_6\}$~~, $\{a_4, a_{4'}, a_9\}$, $\{a_4, a_{4'}, a_8\}$,

Maximum Mutually Compatible set A:

$A_{2,11} = \{a_4, a_{4'}, a_9\}$.

or $\{a_4, a_{4'}, a_8\}$.

Let a_k be some activity in A_{ij}.

S_{ik} : activities that start after activity a_i finishes and finish before activity a_k starts

S_{kj} : activities that start after activity a_k finishes and that finish before activity a_j starts.

Let $A_{ik} = A_{ij} \cap S_{ik}$

$A_{kj} = A_{ij} \cap S_{kj}$

$\Rightarrow A_{ij} = A_{ik} \cup \{a_k\} \cup \{A_{kj}\}$.

$\Rightarrow |A_{ij}| = |A_{ik}| + |A_{kj}| + 1$

Q when $|A| = |B| + |C|$, BCA, CCA ?

Claim: Optimal solution A_{ij} must also include optimal solutions to the two subproblems S_{ik} and S_{kj}.

Proof (Apply cut-and-paste argument!)

If we could find a set A'_{kj} of maximal mutually compatible activities in S_{kj} with $|A'_{kj}| > |A_{kj}|$

Greedy Algo

then we could use A_{ki}' rather than A_{kj}, in a sol^n to the subproblem for S_{ij}.

$$\Rightarrow |A_{ik}| + |A_{kj}'| + 1 > |A_{ik}| + |A_{kj}| + 1$$
$$= |A_{ij}|$$

a contradiction to the assumpt. that A_{ij} is an optimal sol^n.

Similar argument holds for S_{ik}.

$c[i,j]$ = size of the optimal sol^n for the set S_{ij}.

$$c[i,j] = c[i,k] + c[k,j] + 1.$$

if k is known.

If k is not known, then, we need to find k as follows:

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \phi, \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \phi. \end{cases}$$

DP approach

1) develop recursive algo & memoize it.

or

2) develop bottom-up approach & fill table entries.

Q Is there anything else? Can we do better?

A. ~~yes we can! Make Greedy choice!~~

Q What if we could choose an activity that adds to optimal soln without having to first solve all the subproblems?

For activity selection problem: Consider only one choice, the greedy one.

* Choose an activity that leaves the resource available for as many other activities as possible.

* one of the activities must be the first one to finish.

* Intuition: Choose the activity in S that has earliest finish time.

* Since activities ordered ~~with~~ w.r.t finish time: activity that finished first is a_1.

Greedy Algo.

* After making greedy choice a_1, only one remaining subprob. to solve: finding activities that start after a_1 finishes.

Rk (1) No activity can finish before a_1 starts.
(2) All compatible activity must start after a_1 finishes.

Let $S_k = \{ a_i \in S : s_i \geq f_k \}$
= set of activities that start after a_k finishes ($\because s_i \geq f_k$).

Optimal substructure: if a_1 is in the optimal sol^n, then optimal sol^n to original problem = activity a_1 + all activities in optimal sol^n to subprob. S_1

Q Is our greedy choice of choosing the 1st activity to finish always part of optimal solution?

Theorem Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k.

Proof Let A_k = max. size subset of mutually compatible activities in S_k.

a_j : activity in A_k with earliest finish time.

If $a_j = a_m$, we are done, since we then have that a_m is in some max-size subset of mutually compatible activities of S_k.

If $a_j \neq a_m$, let the set $A_k' = A_k - \{a_j\} \cup \{a_m\}$ be A_k but substituted a_j by a_m.

Activities in A_k' are disjoint, because the activities in A_k are disjoint (since A_k is a set of max mutually compat. activity), and since $A_k' \subset S_k$, and since $a_m \& a_j \in S_k$ a_m must be the first activity in A_k' to finish.

$\Rightarrow |A_k'| = |A_k|$

$\Rightarrow$ A_k' is a maximum-size subset of mutually compatible activities of S_k, and it includes a_m.

□

In line: $A_k' = A_k - \{a_j\} \cup \{a_m\}$

it shows that it is possible to keep max. mutually compatible set by introducing $a_m \in S_k$ that finishes first, and replacing a_j which had earliest finish time in A_k.

Example.

RK: ① We could solve ASP with DP, but don't need. Greedy choice works: keeps optimal sol'n.

② Does not need to work bottom-up as in DP, can work top-down. Greedy algos typically have top-down design.

Recursive greedy Algorithm

Arrays

s: start time
f: finish time
n: no. of activities (size of the prob.)

* assume that n input activities are already ordered by increasing finish time.
- add a_0 with $f_0 = 0$

S_0 : entire set of activities.

Recursive-Activity-Selector (s, f, k, n)

1. $m = k+1$
2. while $m \le n$ and $s[m] < f[k]$
3. $\quad m = m+1$
4. if $m \le n$
5. $\quad$ return $\{a_m\} \cup$ Recursive-Activity-Selector (s, f, m, n)
6. else return ϕ.

Lines 2-3 : look for 1st activity in S_k to finish
examine $a_{k+1}, \ldots, a_n$ until it finds first activity a_m that is compatible with a_k;

Example (same as the one started with),

Greedy Algorithms

Elements of greedy strategy

Steps

1. Determine the optimal substr of the problem.
2. Develop a recursive solution.
3. Show that if we make the greedy choice, then only one subprob. remains.
4. Develop a recursive algo that implements the greedy strategy
5. Convert the recursive algo to an iterative algo.

Design greedy algo according to the sequence

1. Cast the opt. problem as one in which we make a choice and a left with one subprob. to solve.
2. Prove that there is always an optimal sol^n to the original prob. that makes the greedy choice
3. Demonstrate optimal substr. by showing that having made the greedy choice, what remains is a subprob. with the property that if we

Combine an optimal sol^n to the subprob. with the greedy choice we have made, we arrive at an optimal sol^n to the orig. prob.

Optimal Substr.

Greedy Choice Property: We can assemble a globally optimal sol^n by making locally optimal (greedy) choices. Make a choice that looks best in current problem, without considering results from subproblems.

Choices for Dynamic Vs. Greedy

Dynamic	Greedy
1) Make choice at each step; choice depends on solutions to subprobs.	1) Make whatever choice seems best at the moment, then solve the subprob. that remains.
2) Solve from smaller subprob. to larger subprob. (even in top-down memoiz, smaller subprob are solved first)	2) Choice in greedy algo may depend on choice so far, but cannot depend on future choices or sol^n to subprobs.
3) DP solves subprobs. before making the 1st choice	3) Greedy makes 1st choice before solving any subprob.
	4) By preprocessing the input, often greedy choices can be made quickly.

Greedy

Greedy or dynamic?

0-1 knapsack problem

Thief robbing a store finds n items. The ith item is worth v_i dollars and weighs w_i pounds, where v_i and w_i are integers. The thief wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack, for some integer W. Which items should he take? This is called 0-1 knapsack problem because for each item, the thief must either take it or leave it behind. he cannot take fractional amount of an item or take an item more than once.

Fractional knapsack problem

Setup is same as 0-1 knapsack problem, but the thief can take fractions of items, rather than having to make a binary (0-1) choice for each item.

1) item in (0-1): gold biscuit
2) item in frac: gold dust

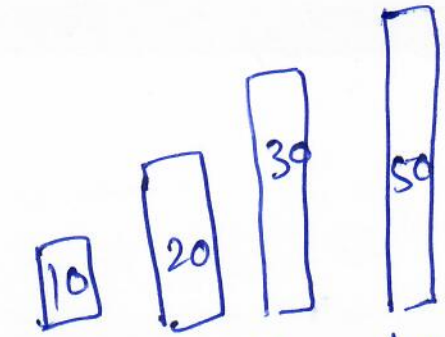
Fractional knapscack → can solve by greed strategy.

0-1 knapsack: can't solve with greedy.

Greedy approach for fractional knapsack.

1) Compute value per pound v_i/w_i for each item
2) Thief begins by taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted, and he can still carry more, he takes as much as possible of the item with the next greatest value per pound, and so on.

Ex. Fractional knapsack has greedy choice property.



$60 $100 $120 knapsack

+

= $220

= $160

$80
+
$100
+
$60
= $240.

Huffman Codes

(If time permits!)

Matroid

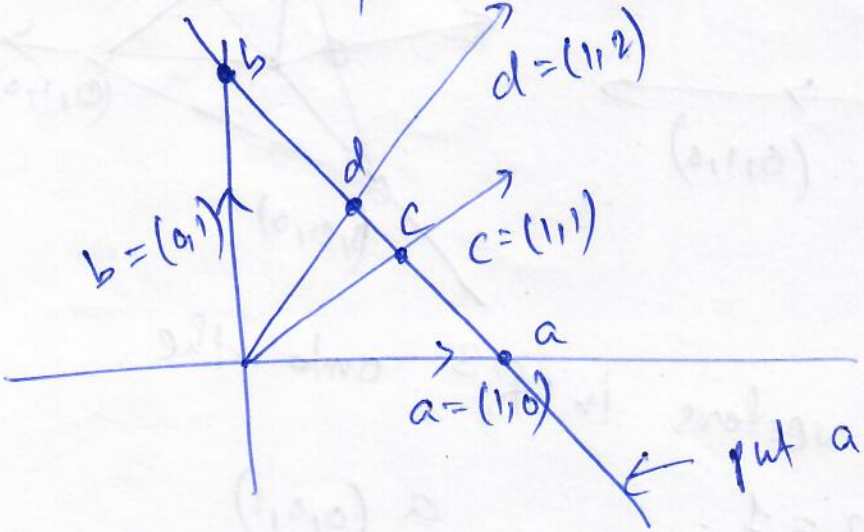
Matroid theory uses linear and abstract algebra, graph theory, combinatorics and finite geometry.

$$A = \begin{matrix} a & b & c & d \end{matrix}\begin{bmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 2 \end{bmatrix}$$

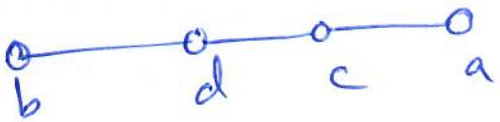
$a = (1,0)$, $b = (0,1)$, $c = (1,1)$ and $d = (1,2)$.

Q Which subsets of columns of A are linearly independent? or linearly dependent?

- Every pair of vectors form L.I. subset of R^2.
- Any subset of three of these form L.D. set.



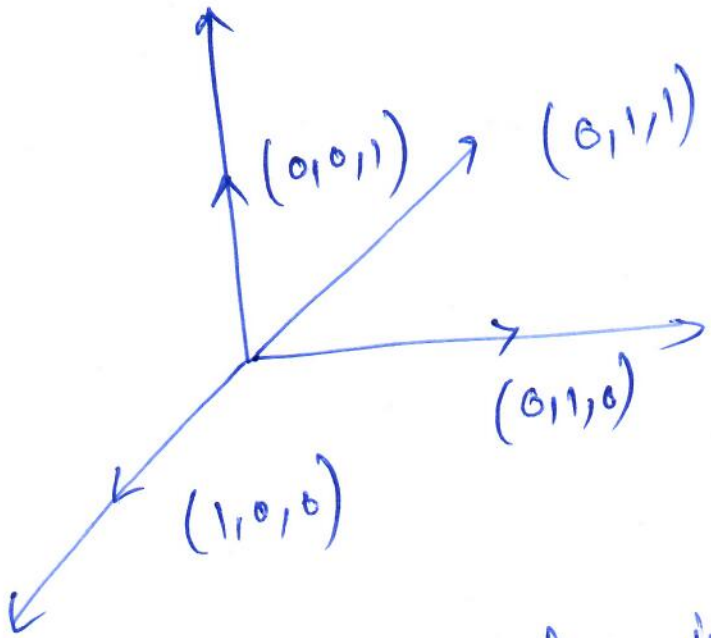
1) Length of a vector doesn't matter.
2) Can replace a vector by its negative without changing our picture.

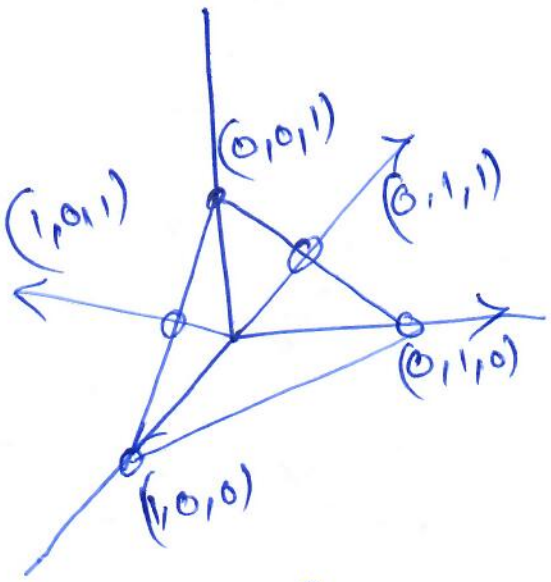


picture of a Matroid

- Any three colinier points are L.D.

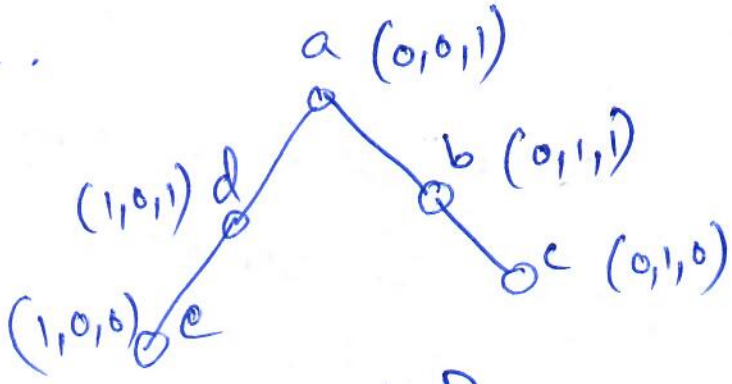
Example 2. $B = \begin{bmatrix} a & b & c & d & e \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix}$





Projecting vectors in $\mathbb{R}^3$ onto the plane $x+y+z=1$.

Corresponding Matroids



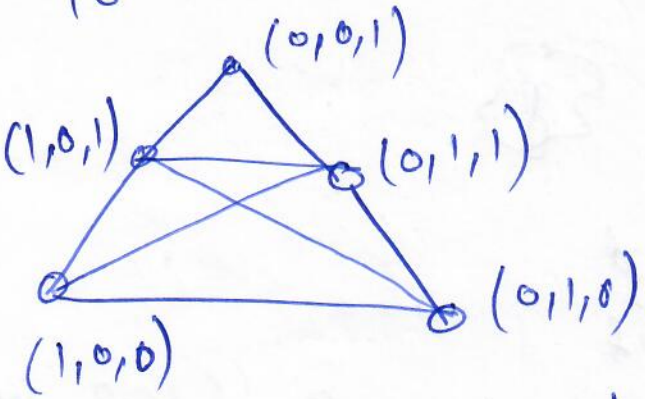
- Any three colinier points are L.D.

Matroids

Remarks.

i) Don't draw line segments connecting two points, for example we don't draw, ec, eb points ~~because~~ if they are the only two points on that line.

To reduce clutter.



Matroid with two-point lines drawn.

Def^n (Matroid)

Example $E = \{a, b, c, d\}$

$I = \{\phi, \{a\}, \{b\}, \{c\}, \{d\}, \{a,b\}, \{c,d\}\}$ see def^n of Matroid after 2 pages

Answer. Cond^n (3) is violated. are not in I.

Let $A = \{c\}$, $|A| = 1$

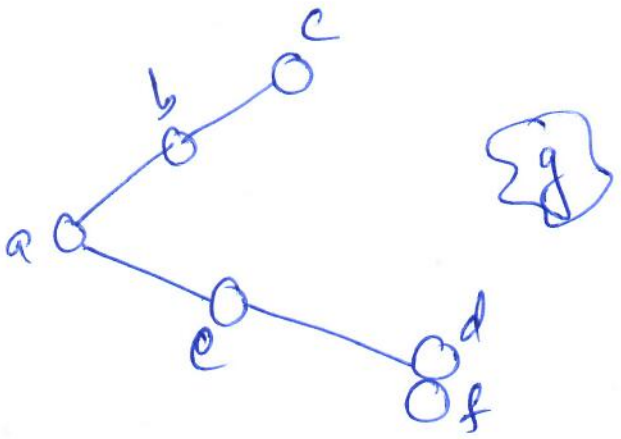
$B = \{a, b\}$, $|B| = 2$.

$|A| < |B|$, but $A \cup \{a\} = \{a, c\}$ or $A \cup \{b\} = \{c, b\}$

Eg.

$$C = \begin{array}{c} \begin{matrix} a & b & c & d & e & f & g \end{matrix} \\ \begin{bmatrix} 1 & 0 & 1 & -1 & 0 & 2 & \infty \\ 1 & -1 & 0 & 0 & 1 & 0 & \infty \\ 0 & 1 & 1 & 2 & 2 & -4 & 0 \end{bmatrix} \end{array}$$

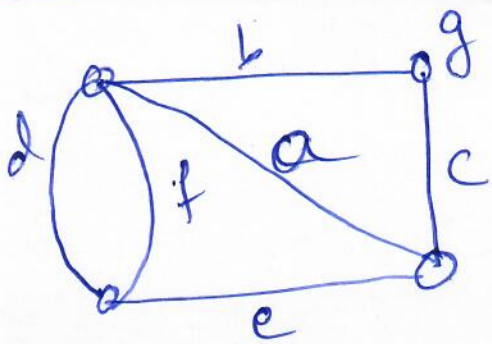
- C is rank 3.
- Matroid is planar.

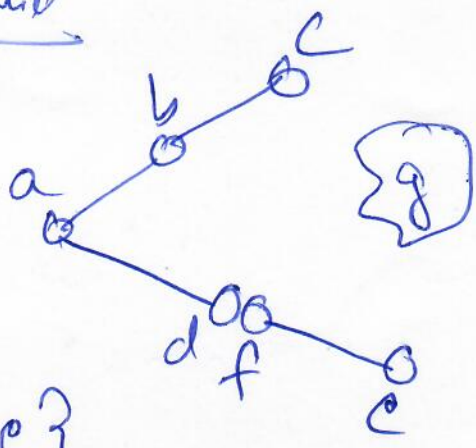


Determine ind. set of a simple rank 3 matroid

1) Empty set is an ind. set.
2) Every point is ind.
3) Every pair of pt. is an ind. set
4) A triple of pt. is an ind. set if & only if these pts are not collinear.
5) No set with more than 3 pts is ind.

Example: Graphic Matroid





$E = \{a, b, c, d, e\}$

Th. If G is a connected graph with n vertices, then the rank of the matroid $M(G)$ is $n-1$, the no. of edges in a spanning tree.

Q Do all Matroids come from graphs?

A. No. Matroids that do arise as cycle Matroids of graphs are called graphic.

Matroids and Greedy Methods

Matroid theory: Situations where greedy method yields optimal sol^n.

Matroid: 1) a combinatorial str.
2) does not covers all cases (activity selectn, or Huffman-coder)

↑ means: not all greedy algo may have Matroid str.!

Matroids

$M = (S, \mathcal{I})$ satisfying:

1) S is a finite set
2) $\mathcal{I}$, non-empty family of subsets of S, called independent subsets of S, s.t if $B \in \mathcal{I}$ and $A \subseteq B$, then $A \in \mathcal{I}$. In this case, $\mathcal{I}$ is called hereditary. $\phi \in \mathcal{I}$.
3) if $A \in \mathcal{I}$, $B \in \mathcal{I}$, and $|A| < |B|$, then there exists some element $x \in B - A$ s.t. $A \cup \{x\} \in \mathcal{I}$. M satisfies exchange property.

Introduce Matrix Matroid

Graphic matroid $M_G = (S_G, \mathcal{I}_G)$ defd in terms of a given undirected graph $G = (V, E)$:

- S_G = set of edges E of G.
- If $A \subseteq E$, then $A \in \mathcal{I}_G$ if and only if A is cyclic. A set of edges A is

independent if and only if the subgraph $G_A = (V, A)$ forms a forest.

Th. If $G = (V, E)$ is an undirected graph, then $M_G = (S_G, I_G)$ is a matroid.

Pf. Clearly, $S_G = E$ is a finite set.

Claim 1. I_G is hereditary; since a subset of a forest is a forest: removing edges from an acyclic set of edges cannot create cycles.

Claim 2 M_G satisfies the exchange property.

$G_A = (V, A)$, $G_B = (V, B)$ are forests of G and that $|B| > |A|$. That is, B contains more edges than A does.

we claim that any $F = (V_F, E_F)$ contains exactly $|V_F| - |E_F|$ trees.

$$|E_F| = \sum_{i=1}^{t} e_i$$

- v_i: no. of vertices of ith tree
- e_i: no. of edges of ith tree

$$= \sum_{i=1}^{t} (v_i - 1) = \sum_{i=1}^{t} v_i - t = |V_F| - t,$$

⇒) $t = |V_F| - |E_F|$.

⇐) G_A contains $|V| - |A|$ trees

G_B " $|V| - |B|$ "

Since $|B| > |A|$, forest G_B has fewer trees than forest G_A does, forest G_B must contain some tree T whose vertices are in two different trees in forest G_A. Since T is connected, it must contain an edge (u,v) s.t $u \& v$ are in different trees in G_A. Since (u,v) connects vertices in two different trees in G_A, we can add the edge (u,v) to G_A without creating a cycle. Therefore, M_G satisfies the exchange property. Hence M_G is a matroid ∎

Def^n (extension) Given $M = (S, I)$, we call an element $x \notin A$ an extension of $A \in I$, if we can add x to A while preserving independence; i.e. x is an extension of A if $A \cup \{x\} \in I$.

Example. If

Consider graphic matroid M_G.

If A is an independent set of edges, then edge e is an extension of A if and only if e is not in A and the addition of e to A does not create a cycle.

Def" (maximal) If A is an independent subset in a matroid M, we say A is maximal if it has no extensions.

Th. All maximal independ subsets in a matroid have the same size.

Pf. Spse to the contrary that A is a maximal independent subset of M and there exists another larger independent subset B of M.

From exchange property:

for some $x \in B - A$, we can extend A to a larger independ set $A \cup \{x\}$, contradicting the assumpt. that A is maximal. ☐

Example M_G, for connected, undirected graph G. Every maximal ind. subset of M_G must be a tree with $|V|-1$ edges that connects all vertices of G. Such a tree is called spanning tree.

Def^n (weighted matroid)

$M=(S,\mathcal{I})$, is weighted if $\exists$ a weight fn., $w(x)>0$ to each $x \in S$.

for $A \subseteq S$:

$$w(A) = \sum_{x \in A} w(x).$$

(imp)

Greedy algo on a weighted matroid

Finding a greedy Algo	$\Leftrightarrow$	find a maximum-weight independent subset in a weighted matroid. Given $M=(S,\mathcal{I})$, find ind. set $A \in \mathcal{I}$ s.t $w(A)$ is maximized.

Def^n (optimal subset of matroid) A subset of a matroid is called optimal subset if it is independent and has maximum possible weight.

optimal subset $(\Rightarrow)$ maximal ind. subset

Example. MST problem

- Given $G = (V, E)$
- length fn w s.t $w(e)$ is length of edge e.

(reserve weight for Matroid, and length for original edge weight of graph)

Goal for MST: Find a subset of the edges that connects all of the vertices & has minimum total length (weight of edges).

Formulate MST interms of problem of finding an optimal subset of a Matroid

Consider M_G, a matroid, with wgt. fn w', $w'(e) = w_0 - w(e)$, w_0 is larger that the max. length of any edge.

- we ensure that weight fn is > 0
- optimal subset is a spanning tree of minimum total length in original graph.

Each max ind. subset A corresp. to a spanning tree with $|V|-1$ edges.

since $w'(A) = \sum_{e \in A} w'(e)$

$$= \sum_{e \in A} (w_0 - w(e))$$

$$= (|V|-1)\, w_0 - \sum_{e \in A} w(e)$$

$$= (|V|-1)\, w_0 - w(A),$$

for any ind. subset A, an ind. subset that maximizes $w'(A)$ must minimize $w(A)$.

Greedy (M, w)

1) $A = \phi$
2) sort $M.S$ into monotonically decreasing order by weight w
3) for each $x \in M.S$, taken in monotonically decreasing order by weight $w(x)$
   if $A \cup \{x\} \in M.I$
   $A = A \cup \{x\}$

return A.

- A is always independent by induction.

Let $n = |S|$.

sort time: $O(n \lg n)$

Line 4 executes n time.

Each execution of line 4 requires a check whether $A \cup \{x\}$ is ind.

If each check take $O(f(n))$,

Total cost $= O(n \lg n + nf(n))$.

Lemma. (Matroids Exhibit the Greedy Choice Property)

Given $M = (S, \mathcal{I})$ a weighted matroid with weight w and that S is sorted into monotonically decreasing order by weight. Let x be the first element of S s.t $\{x\}$ in independent, if any such x exists. If x exists, then there exists an optimal subset A of S that contains x.

Pf. If no such x exists, then the only ind. subset is the empty set and the lemma is vacuously true.

Let B be any nonempty optimal subset. Assume $x \notin B$; otherwise, $A = B$ gives an optimal subset of S that contains x.

Claim: No element of B has weight greater than $w(x)$ (it can be at most equal).

Pf: If $y \in B$, then $\{y\} \subseteq B$, $B \in I$, and I being hereditary, $\{y\} \in I$. It is given that x is the 1st element of S s.t. $\{x\}$ is independent.

~~Now $y \neq x \Rightarrow w(y) \leq w(x)$~~

Elements of S are ordered by weight, and $y \neq x \Rightarrow w(y) \leq w(x)$. Since, y was arb. $w(x) \geq w(y)$ for any $y \in B$.

Construct the set A as follows. Begin with $A = \{x\}$. Since x is such that $\{x\}$ is independent (given), $A = \{x\}$ is independent. Our next goal is to enlarge the set A by and keeping it independent.

Matroids

we have $|A| < |B|$. If $|B| = |A|$, then we replace ~~$\{x\}$ in B~~ the existing element of B by A. If $|A| < |B|$, then we use the exchange property to repeatedly find a new element of B that we can add to A until $|A| = |B|$, while preserving independence of A. At that point, A and B are the same except that A has x and B has some other element y.

That is, $A = B - \{y\} \cup \{x\}$ for some $y \in B$,

and so

$$w(A) = w(B) - w(y) + w(x)$$
$$\geq w(B) \quad \left[\text{since } w(x) - w(y) \geq 0\right]$$

Because the set B is optimal, set A, which contains x, must also be optimal.

Remark: If an element is not an option initially, then it cannot be an option later.

Lemma M2

Let $M = (S, I)$ be any matroid. Let $x \in S$. Let x be an extension of some independent subset A of S, then x is also an extension of ϕ.

Pf Since x is an extension of A, we have that $A \cup \{x\}$ is independent. Since I is hereditary, $\{x\}$ must be independent. Thus, x is an extension of ϕ. ∎

Cor. Let $M = (S, I)$ be any matroid. If x is an element of S such that x is not an extension of ϕ, then x is not an extension of any independent subset A of S.

Pf Contrapositive of Lemma above.

Rk: Any element that cannot be used immediately can never be used.

Lemma M3 Matroids exhibit the optimal substructure Property

Let x be the 1st element of S chosen by GREEDY for the weighted matroid $M = (S, I)$. The remaining problem of finding a maximum

weight independent subset containing x reduces to finding a maximum-weight independent subset of the weighted matroid $M' = (S', I')$, where

$$S' = \{y \in S : \{x, y\} \in I\},$$

$$I' = \{B \subseteq S - \{x\} : B \cup \{x\} \in I\},$$

and the weight fn for M' is the weight fn for M, restricted to S'.

Pf If A is any maximum weight independent subset of M containing x, then

$$A' = A - \{x\}$$

is an independent subset of M'. (subsets of ind. set are ind. Hereditary)

Conversly, any independent subset A' of M' yields an independent subset

$$A = A' \cup \{x\} \quad \text{of } M.$$

In both cases: $w(A) = w(A') + w(x)$,

a maximum-weight soln in M containing x yields a maximum-weight soln in M' and vice-versa.

Th (Correctness of the greedy algorithm on Matroid)

If $M = (S, I)$ is a weighted matroid with weight fn w, then Greedy (M, w) returns an optimal subset

Pf. Corollary $\Rightarrow$ any elements that greedy ignores / passes over initially because they are not extensions of ϕ can be forgotten about, since they can never be useful.

(M1) Lemma 1 $\Rightarrow$ If however, greedy do select the 1st element x, then the algo. does not make any error since these exists an optimal subset containing x.

Finally, Lemma 1 (M3) $\Rightarrow$ the remaining problem is one of finding an optimal subset in the matroid M' that is the contraction of M by x.

After greedy sets

$$A = \{x\}$$

all the remaining steps of algo acts on

$$M' = (S', I'),$$

because B is independent in M' if and only if $B \cup \{x\}$ is independent in M,

$$\forall\ B \in I'.$$

Hence, overall operation of greedy will find a maximum-weight independent subset for M.

Graph Algorithms

Representations of graphs

$G = (V, E)$

Two ways:

1) Adjacency lists
2) Adjacency matrix

1) or 2) applies to both directed and undirected graphs.

For sparse graphs: adjacency-list is compact repr.

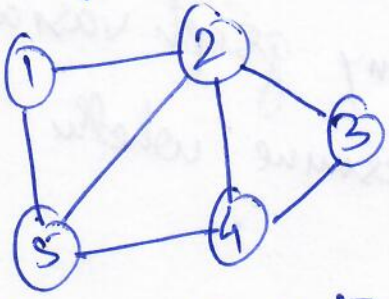
* we assume input graph in adjacency-list form.

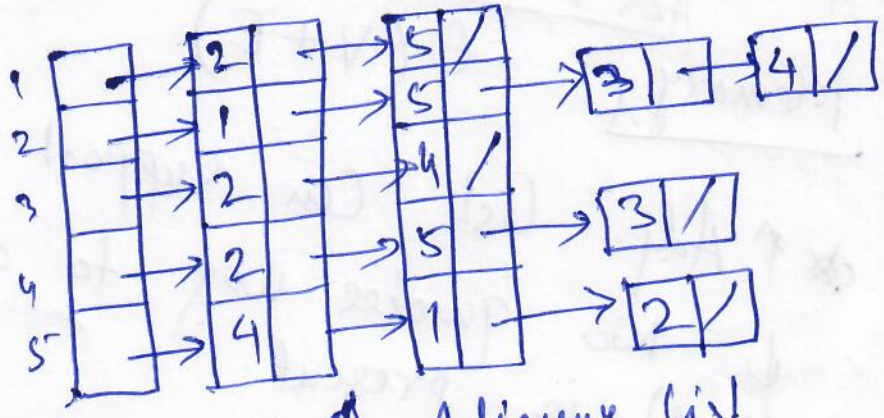
For dense graph: adjacency matrix.

Adjacency list representation: $G = (V, E)$

For each $u \in V$, the adjacency list Adj[u] contains all the vertices v such that there is an edge $(u, v) \in E$.

Adj[u] = all the vertices adjacent to u in G.





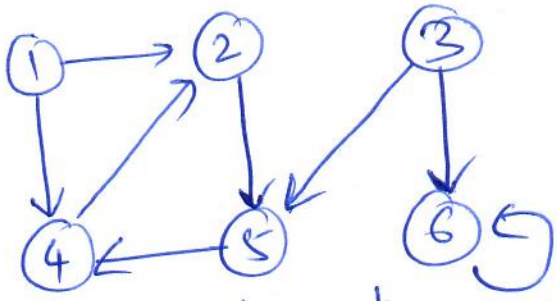
Adjacency List

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

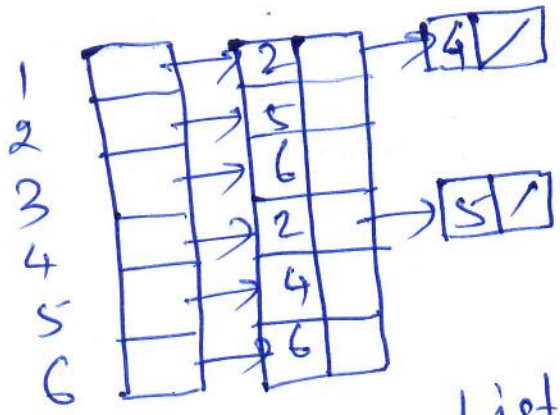
Length of adj lists: $2|E|$

Adjacency Matrix

Undirected graph



Directed graph



Adjacency List

length of adj lists = |E|

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency Matrix

~~For undirected graph,~~ for Adj list $\Theta(V+E)$

Memory

* ↑ Adj. list can support many graph variants.
  ↓ No quickest way to determine whether (u,v) is present.

Adjacency Matrix Repr.

$G = (V, E)$, vertices numbered: $1, 2, \ldots, |V|$.

$A = (a_{ij}) \in \mathbb{R}^{|V| \times |V|}$ s.t.

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Memory: $\Theta(V^2)$

Undirected graph: Adj. matrix is symm. (Can store only entries above diagonal).

For weighted graph:

$$a_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

Check if 0 is suitable.

Breadth-first Search

Given $G = (V, E)$, a source vertex s, explore the edges of G to discover every vertex that is reachable from s. It computes the distance from s to each

* Works on directed and undirected graph

```
BFS(G, s)
1.  for each vertex u ∈ G.V - {s}
2.      u.color = WHITE
3.      u.d = ∞
4.      u.π = NIL
5.  s.color = GRAY
6.  s.d = 0
7.  s.π = NIL
8.  Q = ϕ
9.  ENQUEUE(Q, s)
10. while Q ≠ ϕ
11.     u = DEQUEUE(Q)
12.     for each v ∈ G.Adj[u]
13.         if v.color == WHITE
14.             v.color = GRAY
15.             v.d = u.d + 1
16.             v.π = u
17.             ENQUEUE(Q, v)
18.     u.color = BLACK
```

Example: See page 596, Cormen

Running time: $O(V+E)$

Shortest paths

$\delta(s, v)$: shortest path distance from s to v, i.e., minimum no. of edges

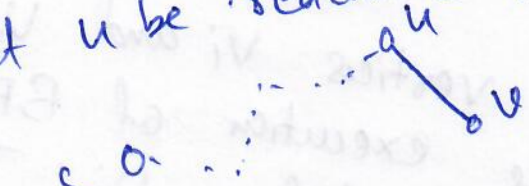
reading

Lemma 22.1

Let $G = (V, E)$ be directed or undirected graph, $s \in V$ be an arbitrary vertex. Then for any edge $(u, v) \in E$,

$$\delta(s, v) \leq \delta(s, u) + 1. \quad \rightarrow (1)$$

Pf. Let u be reachable from s.



Shortest path from s to v cannot be longer than shortest path from s to $u + 1$. That is

$$\delta(s, v) \not> \delta(s, u) + 1.$$

i.e., $\delta(s, v) \leq \delta(s, u) + 1.$

If u is not reachable from s, then

$$\delta(s, u) = \infty.$$

$\delta(s, v) \leq \delta(s, u) + 1$ holds.

reading So,

Lemma 22.2

Let $G = (V, E)$ be a directed or undirected graph, and suppose that BFS is run on G from a given source vertex $s \in V$. Then upon termination, for each vertex $v \in V$ the value $v.d$ computed by BFS satisfies $v.d \geq \delta(s, v)$.

Lemma 22.3 ← ready

Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \ldots, v_r \rangle$ where v_1 is the head of Q and v_r is the tail. Then $v_r.d \le v_1.d + 1$ and $v_i.d \le v_{i+1}.d$ for $i = 1, 2, \ldots, r-1$.

← ready

Cor. 22.4 Suppose that vertices v_i and v_j are enqueued during the execution of BFS, and that v_i is enqueued before v_j. Then $v_i.d \le v_j.d$ at the time that v_j is enqueued.

← ready

Th. 22.5 (Correctness of breadth-first search)

$G = (V, E)$, directed or undirected. BFS run from source vertex $s \in V$. Then, during its execution, BFS discovers every vertex $v \in V$ that is reachable from the source s, and upon termination, $v.d = \delta(s, v)$ $\forall v \in V$. Moreover, for any vertex $v \ne s$ that is reachable from s, one of the shortest paths from s to v is a shortest path from s to $v.\pi$ followed by the edge $(v.\pi, v)$.

Pf. Assignment.

Graph Algorithms

Depth-first search

* seach deeper in the graph.

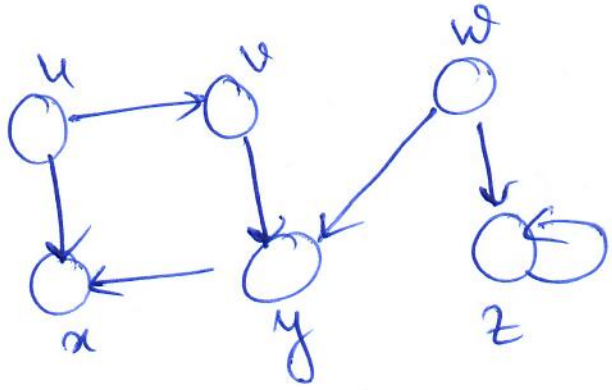
```
DFS(G)
1.    for each vertex u ∈ G.V
2.        u.color = WHITE
3.        u.π     = NIL
4.    time = 0
5.    for each vertex u ∈ G.V
6.        if u.color == WHITE
7.            DFS-VISIT(G,u)
```

```
DFS-VISIT(G,u)
1.    time = time+1
2.    u.d = time
3.    u.color = GRAY
4.    for each v ∈ G.Adj[u]
5.        if v.color == WHITE
6.            v.π = u
7.            DFS-VISIT(G,v)
8.    u.color = BLACK
9.    time = time+1
10.   u.f = time
```

Graph Algorithms

Example

Consider the following graph:

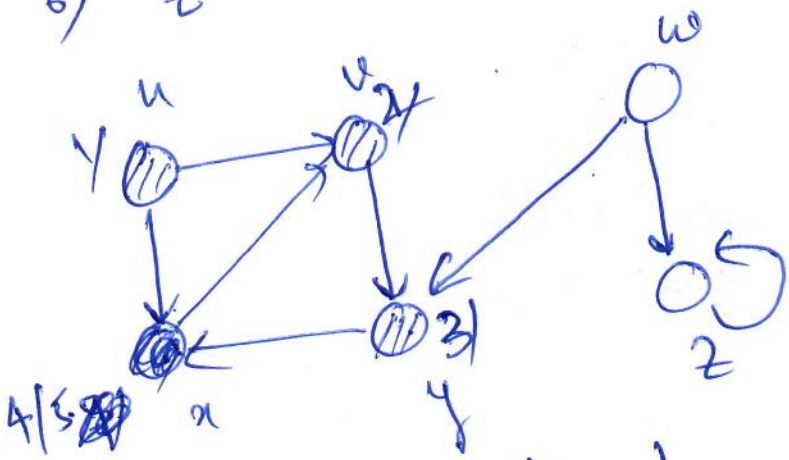


Adjacency List:

1) u → v x
2) v → y
3) w → y z
4) x → v
5) y → x
6) z → z

u → v x
v → y
y → x
x → v

Note: See Cormen for this example!



(111) : grey

: black.

(1) DFS-VISIT (G, u)

1) start with u.

2) Adjacency of u:

u → v, x.

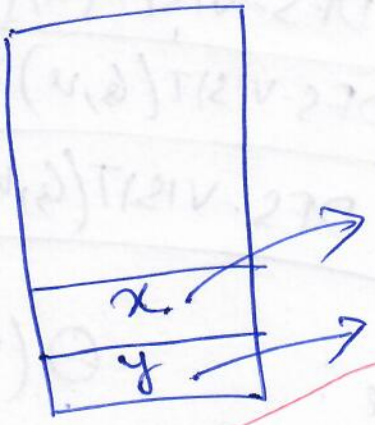
(3) ~~→ DFS-VISIT (G, v)~~

~~→ DFS-VISIT (G, x)~~

Recursive fns

↓

FIFO ← stack.



1) Udaykiran Singh (2016128)

2) 2016125

3) 2016037

4) 2016055

5) 2016123

6) 1062

7) 1113

8) 1154

1114

1154

2016 1406

1004

1124

1092

1094

1232

1404

1036

1000

1020

1027

1033

1047

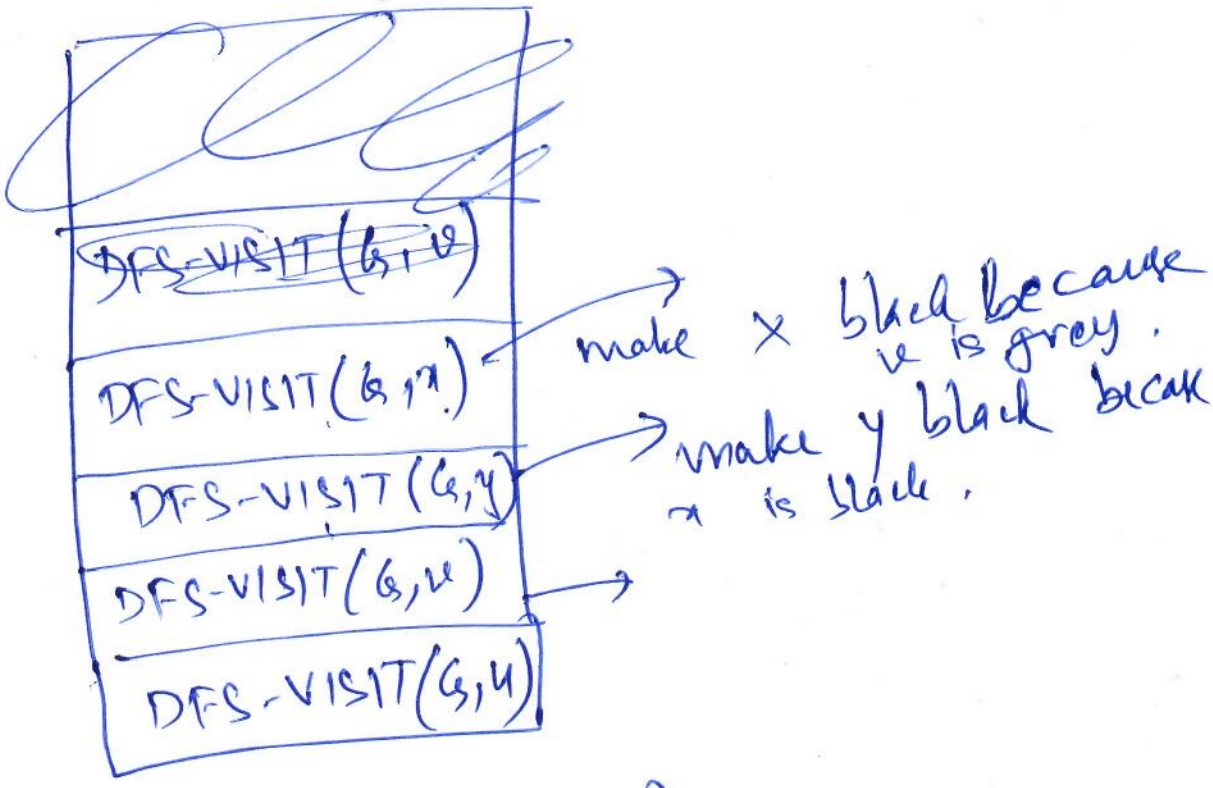
1063

Step ① Start with vertex u.

Step ② DFS-VISIT(G,u)

u → v, x (loops 4, 5, 6, 7)

DFS-VISIT(G, v)



Running time: $\Theta(V+E)$.

↗ See Cormen for details!

Graph Algorithms

Strongly Connected Components (not covered in class!)

Decompose a directed graph into its strongly connected components.

strongly connected components: A maximal set of vertices $C \subseteq V$ s.t. for every pair of vertices u and v in C, we have

$u \rightsquigarrow v$ and $v \rightsquigarrow u$

that is, u and v are reachable from each other.

Sketch of Algo to find strongly connected components

Consider $G = (V, E)$ and

$G^T = (V, E^T)$,

where $E^T = \{(u, v) : (v, u) \in E\}$.

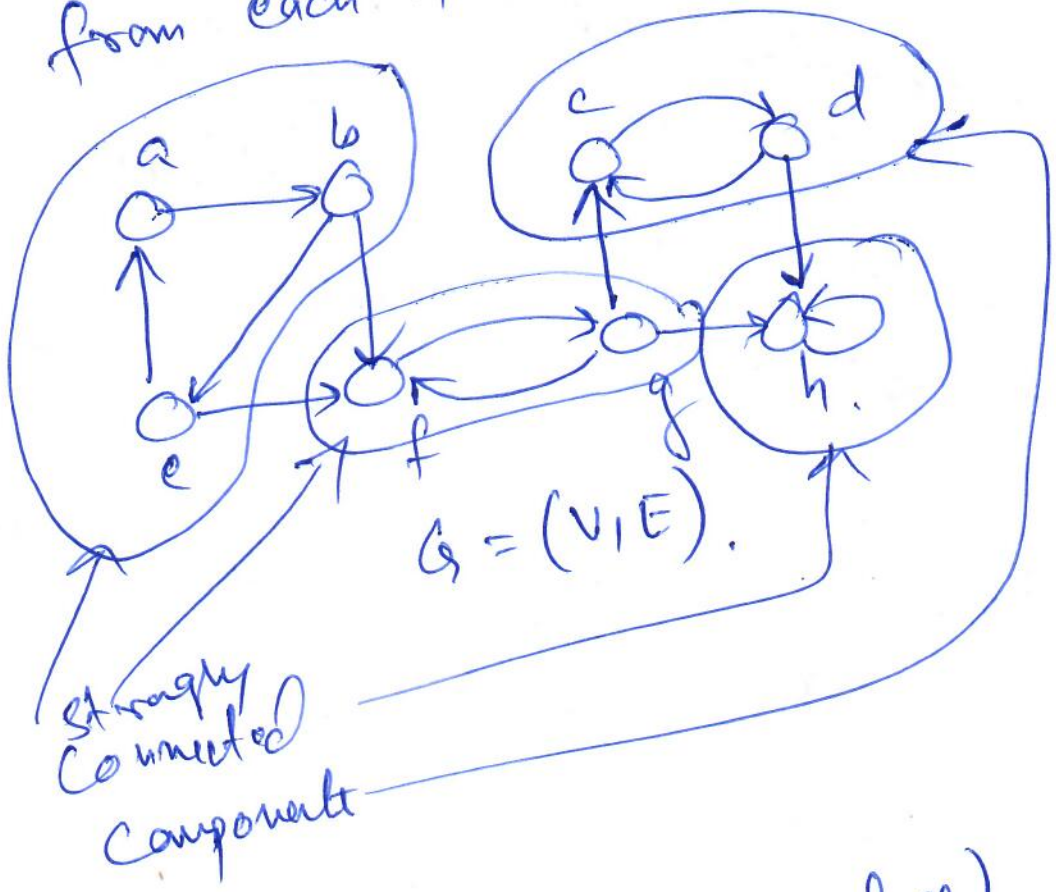
E^T: edges with directions reversed.

Time for G^T: $O(V+E)$.

Lemma. G and G^T have exactly the same strongly connected components.

If. u & v are reachable from each other if and only if they are reachable from each other in G^T.

"Not the same numbers"



~~Draw Transpose (reverse edges)~~

STRONGLY-CONNECTED-COMPONENTS (G).

1. Call DFS(G) to compute finishing times $u.f$ for each vertex u.
2) Compute G^T
3) Call DFS(G^T), but in the main loop of DFS consider the vertices in order of decreasing $u.f$
4) Output the vertices of each tree in the dept-first forest formed in line 3 as a separate strongly Connected Component.

Algo to find Strongly Connected Comp.

1) Call DFS(G) to compute finish times u.f for each vertex u.
2) compute G^T
3) Call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing u.f
4) output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component.

Component Graph:

$$G^{SCC} = (V^{SCC}, E^{SCC}).$$

Let G has strongly connected components $C_1, C_2, \ldots, C_k$.

$V^{SCC} = \{v_1, v_2, \ldots, v_k\}$.

v_i : represents each strongly connected component.

Lemma. Component graph is a DAG.

Let $C \neq C'$ be distinct strongly connected component in directed graph $G = (V, E)$, let $u, v \in C$, let $u', v' \in C'$, and suppose that G contain a path

$$u \rightsquigarrow u'.$$

Then G cannot also contain a path

$$v' \rightsquigarrow v.$$

Pf.

Minimum Spanning Trees

24.09.17 (P1)

Motivation: To interconnect a set of n pins, we can use arrangement of $n-1$ wires, each connecting two pins. The one that uses least amount of wire is usually most desirable.

$G = (V, E)$, V: set of pins, E: set of possible interconnections betn pairs of pins. For each edge $(u,v) \in E$, we have weight $w(u,v)$ specifying the cost to connect u & v.

Goal: Find a acyclic subset $T \subseteq E$ that connects all of the vertices and total weight

$$w(T) = \sum_{(u,v) \in T} w(u,v)$$

is minimized.

* Two algorithms: Kruskal's and Prim's
* Each of them are greedy

Kruskal's algorithm

```
MST-KRUSKAL(G, w)
1. A = ∅
2. for each vertex v ∈ G.V
       MAKE-SET(v)
3.
4. sort the edges of G.E into nondecreasing order
       by weight w
5. for each edge (u,v) ∈ G.E, taken in nondecreasing
                          order by weight.
```

if FIND-SET(u) ≠ FIND-SET(v)
A = A ∪ {(u,v)}
UNION(u,v)

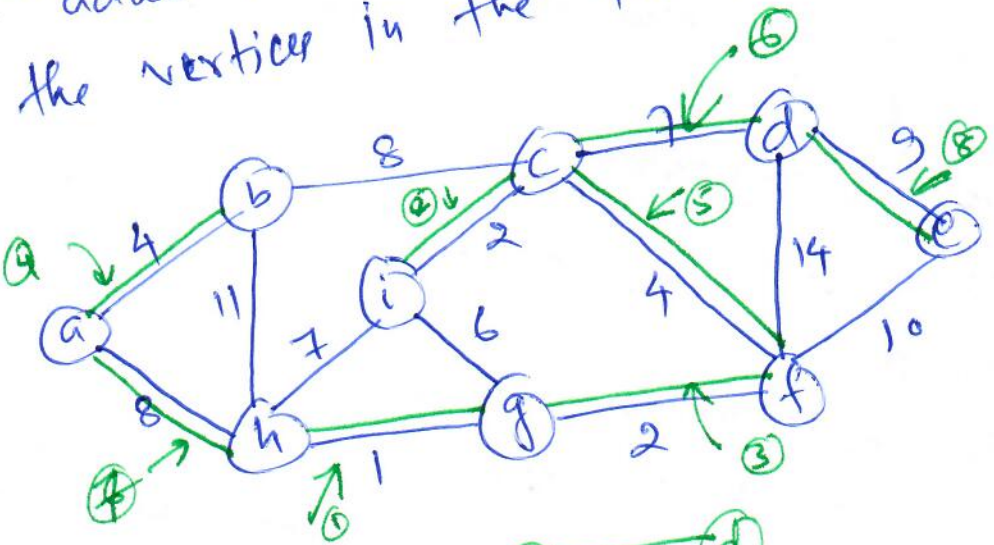
return A

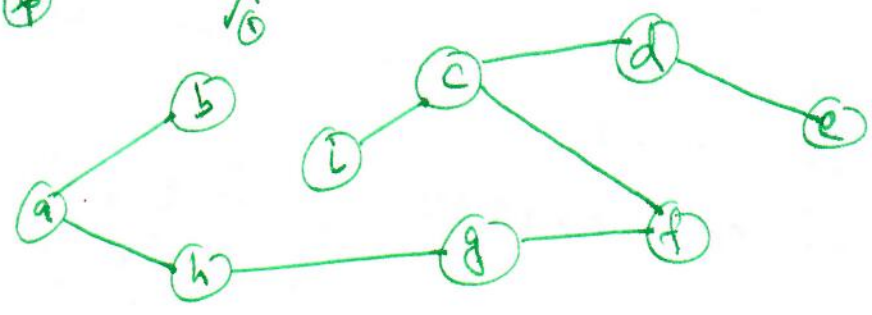
Key steps

(1) Line 1-3 in algo initialize the set A to empty set and create |V| trees, one containing each vertex.

(2) The for loops in line 5-8 examines edges in increasing order of weight, from lowest to highest. The loop checks for each edge (u,v), whether the endpoints u and v belong to the same tree. If they do, then the edge (u,v) cannot be added the forest without creating a cycle, and edge is discarded. Otherwise, line 7 adds the edge (u,v) to A, and line 8 merges the vertices in the two trees.

Example





Minimum Spanning Trees

Running Time of Kruskal's Algorithm

* Depends on how disjoint-set data str. is implemented.
* Assume disjoint-set-forest implementation with union-by-rank and path-compression.

- Line 1: $O(1)$ time
- Line 4: time to sort edges: $O(E \lg E)$.
- Lines 5-8: $O(E)$ Find-Set and Union operations
- Line 3: $|V|$ Make-Set ops.
- Total: $O((V+E)\alpha(V))$, α is a slowly growing fn.

Since $|E| \geq |V|-1$ (G is connected)

disjoint set ops take: $O(E\alpha(V))$

$\alpha(|V|) = O(\lg V) = O(\lg E)$

Total run time: $O(E \lg E)$.

Since $|E| < |V|^2$, $\lg|E| = O(\lg V)$.

Total run time: $O(E \lg V)$

Minimum Spanning Trees

Prim's Algorithm

* Edges in set A always form a single tree.

Key steps.

1) Tree starts from arbr. root vertex r and grows until the tree spans all the vertices in V.

2) Each step adds to the tree A a light edge that connects A to an isolated vertex - one on which no edge of A is incident.

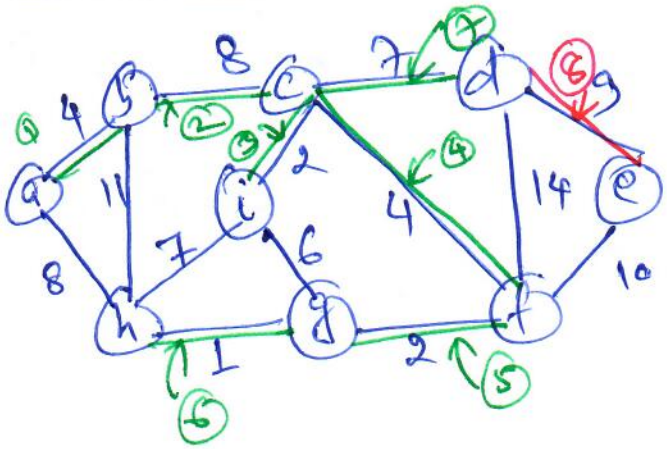
* Need a fast way to select a new edge to add to the tree formed by the edges in A.

```
MST-PRIM(G, w, r)
1.  for each u ∈ G.V
2.      u.key = ∞
3.      u.π   = NIL
4.  r.key = 0
5.  Q = G.V
6.  while Q ≠ ∅
7.      u = Extract-Min(Q)
8.      for each v ∈ G.Adj[u]
9.          if v ∈ Q and w(u,v) < v.key
10.             v.π   = u
11.             v.key = w(u,v)
```

Q: Min. priority Queue.
Based on a key attribute

v.key: min weight of any edge connecting v to a vertex in the tree.

v.key = ∞, if there is no edge.



Running Time

* Depends on how min-priority queue Q is implemented.
* Assum Q is a binary min-heap.

Lines 1-5: $O(V)$ using Build-Min-Heap

Line 6: it executes $|V|$ times, extract-min takes $O(\lg V)$ time, total time: $O(V \lg V)$

Line 8-11: for loop executes $O(E)$ times

test for membership in Q in line 9 can be implemented in constant time by keeping a bit for each vertex that tells whether or not it is in Q, and updating the bit when the vertex is removed from Q.

Line 11: assignment involves an implicit Decrease-key operation on the min-heap. A bin-min-heap takes $O(\lg V)$ time for this.

Total time: $O(V \lg V + E \lg V) = O(E \lg V)$

Rk. Asymptotic run time of Prim's algo can be improved by using Fibonacci heaps.

Define min-heap here.

Single-Source Shortest Paths

Problem: Given a weighted, directed graph $G = (V, E)$, with weight fn: $w : E \to \mathbb{R}$ mapping edges to real-valued weights.

The weight $w(p)$ of path $p = \langle v_0, v_1, \ldots v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

shortest-path weight $\delta(u, v)$ from u to v.

$$\delta(u,v) = \begin{cases} \min\{ w(p) : u \overset{p}{\rightsquigarrow} v \} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise.} \end{cases}$$

shortest path: Any path p with weight $w(p) = \delta(u, v)$

Variants of Shortest path Problem

1) **Single-destination shortest-path problem**:
find a shortest path to a given destination vertex t from each vertex v.
Reverse edges too reduce it to a single source problem.

(2) Single-pair shortest-path problem:
Find a shortest path from u to v for given vertices u and v. If we solve the single-source problem with source vertex u, we solve this problem.

(3) All-pairs shortest-paths problem: Find a shortest path from u to v for every pair of vertices u and v.
- Can solve this by running a single source alg. once from each vertex, but can solve it faster.

Optimal substr. of a shortest path

* shortest path betⁿ two vertices contains other shortest paths within it.
* Hence, DP or greedy method might apply.
* Dijkstra's algo is a greedy algo
* Floyd-Warshall algo is a DP algo.

Lemma 24.1 (Subpaths of shortest paths are shortest paths)
Given weighted, directed graph $G = (V, E)$ with weight fⁿ $w: E \to \mathbb{R}$, let

Single-Source Shortest Paths (Graph Algorithms)

$p = \langle v_0, v_1, \ldots v_k \rangle$ be a shortest path from vertex v_0 to vertex v_k and for any i and j s.t. $0 \le i \le j \le k$, let $p_{ij} = \langle v_i, v_{i+1}, \ldots v_j \rangle$ be the subpath of p from vertex v_i to vertex v_j. Then p_{ij} is a shortest path from v_i to v_j.

Proof: Decompose path p into:

$$v_0 \overset{p_{oi}}{\rightsquigarrow} v_i \overset{p_{ij}}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k \longrightarrow (SP1)$$

we have,

$$w(p) = w(p_{oi}) + w(p_{ij}) + w(p_{jk}).$$

Let there be a path p_{ij}' from v_i to v_j that is shorter than the path p_{ij}, meaning

$$w(p_{ij}') < w(p_{ij}). \longrightarrow (SP2)$$

If we cut-paste this path p_{ij}' in (SP1) then we have:

$$v_0 \overset{p_{oi}}{\rightsquigarrow} v_i \overset{p_{ij}'}{\rightsquigarrow} v_j \overset{p_{jk}}{\rightsquigarrow} v_k \longrightarrow (SP2)$$

Now,

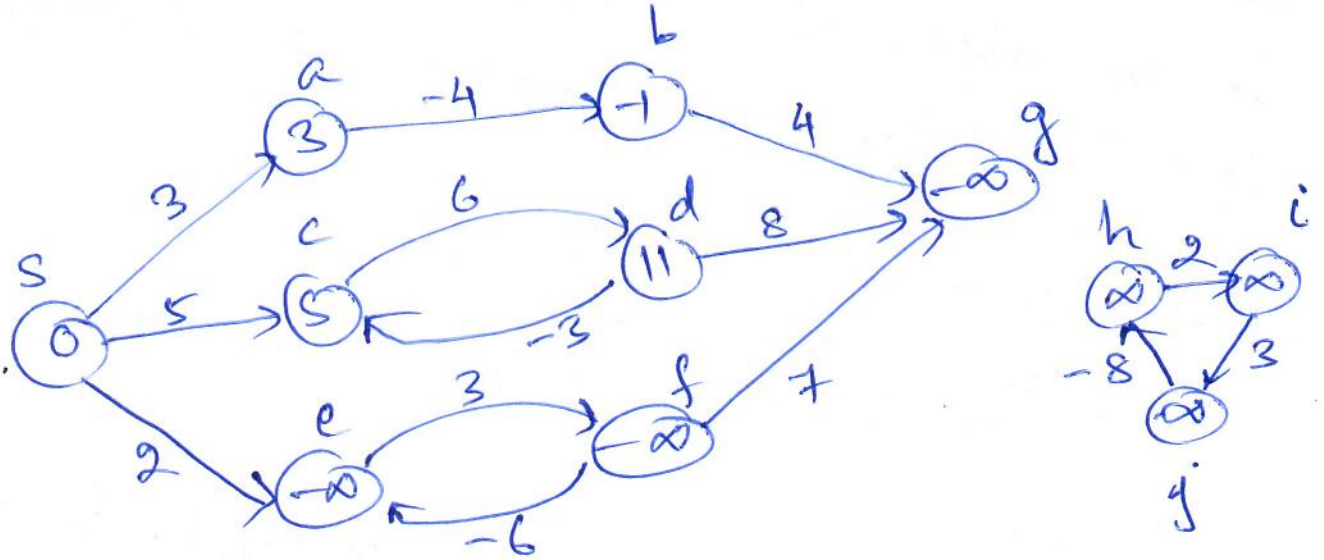
$$w(p') = w(p_{oi}) + w(p_{ij}') + w(p_{jk})$$

$$\overset{(SP2)}{<} w(p_{oi}) + w(p_{ij}) + w(p_{jk})$$

That is, we found a path p' which is shorter than the shortest path p, a contradiction to our hypothesis.

Negative weight Edges and Shortest paths

Consider



1) $\delta(s,a) = w(s,a) = 3$

2) $\delta(s,b) = w(s,a) + w(a,b) = 3 + (-4) = -1$

Paths from s to c: $\langle s,c\rangle$, $\langle s,c,d,c\rangle$, $\langle s,c,d,c,d,c\rangle$ and so on.

cycle $\langle c,d,c\rangle$ has weight $= 6 + (-3) = 3 > 0$

3) $\delta(s,c) = w(s,c) = 5$.

Similarly

4) $\delta(s,d) = w(s,c) + w(c,d) = 11$.

Single-Source Shortest Path

Paths from s to e : $\langle s,e\rangle$, $\langle s,e,f,e\rangle$, $\langle s,e,f,e,f,e\rangle$, and so on.

weight of $\langle s,e\rangle = 2$

" $\langle s,e,f,e\rangle = -1$

" $\langle s,e,f,e,f,e\rangle = -4$.

⋮

5) Hence $\delta(s,e) \xrightarrow{\pi} -\infty$. (tends to / equal to.)

6) Similarly $\delta(s,f) = -\infty$

7) Since $\delta(s,f) = -\infty$, $\delta(s,g) = -\infty + 7 = -\infty$.
(optimal substr. property).

8) $\delta(s,h) = \delta(s,i) = \delta(s,j) = \infty$, because they are not reachable from source s.

~~Cycles~~

Remarks

1) Dijkstra's algo assume edge weights are nonnegative

2) Bellman-Ford allow negative-weights and produce correct answer as long as no negative weight

cycles reachable from source. Typically algo. can detect -ve weight cycle.

Cycles and shortest path

1) Non-zero weight cycles lead to ~~-ve~~ $-\infty$ weight or they are useless.

2) This leaves 0-weight cycles. We can repeatedly remove these cycles from the path until we have shortest path that is cycle-free.

3) Restrict attention to shortest paths of $|V|-1$ edges.

Representing Shortest Paths

* Wish to compute vertices on shortest-paths.
* Given $G = (V, E)$, maintain for each $v \in V$ a predecessor $v.\pi$ that is either another vertex or NIL.
* Consider predecessor subgraph $G_\pi = (V_\pi, E_\pi)$ induced by π values.

$$V_\pi = \{ v \in V : v.\pi \neq \text{NIL} \} \cup \{ s \}$$

$$E_\pi = \{ (v.\pi, v) \in E : V_\pi - \{ s \} \}.$$

Single Source Shortest Path

* π values (Graph Algo) produced by algos have the property that at termination G_π is a shortest paths tree. A rooted tree containing a shortest path from source s to every vertex that is reachable from s.

Let the shortest path tree rooted at s be $G' = (V', E')$. It is directed.

$V' \subseteq V$, $E' \subseteq E$.

1) V' is the set of vertices reachable from s in G.
2) G' forms a rooted tree with root s, and
3) for all $v \in V'$, the unique simple path from s to v in G' is the shortest path from s to v in G.

RK Shortest paths are not unique.

Relaxation

For each vertex $v \in V$, maintain an attribute $v.d$, an upper bound on the weight of a shortest path from source s to v. Here $v.d$ = shortest path estimate

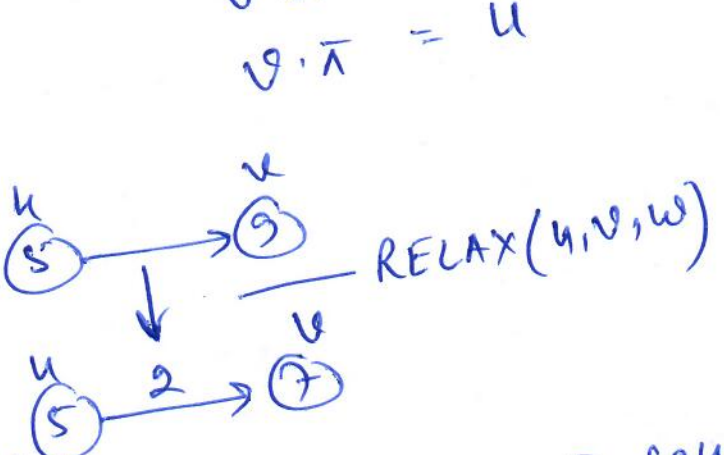
INITIALIZE-SINGLE-SOURCE (G, s)

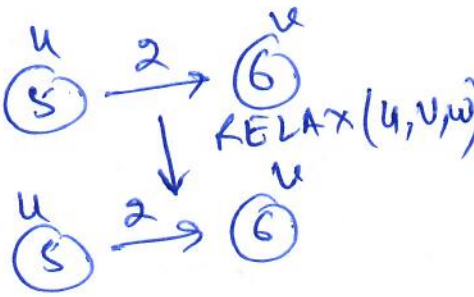
1. for each vertex $v \in G.V$
2. $\quad v.d = \infty$
3. $\quad v.\pi = NIL$
4. $s.d = 0$

Relaxing an edge (u, v) means testing whether we can improve the shortest path to v found so far by going through u and if so updating v.d and v.π.

RELAX (u, v, w)

1) if $v.d > u.d + w(u,v)$
2) $\quad v.d = u.d + w(u,v)$
3) $\quad v.\pi = u$





Each Algo

*) calls INITIALIZE-SINGE-SOURCE
2) Repeatedly relax edges.

Single-Source Shortest (Graph Algos)

Rk:
1) Dijkstra's Algo & shortest paths algo for directed acyclic graphs relax each edge exactly once.
2) Bellman-Ford algo relaxes each edge $|V|-1$ times.

Properties of shortest paths and Relaxation

1) **Triangle inequality** (Lemma 24.10)
For any edge $(u,v) \in E$,
$$\delta(s,v) \leq \delta(s,u) + w(u,v)$$

2) **Upper-bound property** (Lemma 24.11)
We always have
$$\delta(s,v) \leq v.d \quad \forall v \in V,$$
and once $v.d = \delta(s,v)$, $v.d$ never changes.

3) **No-path property** (Cor 24.12)
If there is no path from s to v, then we always have $v.d = \delta(s,v) = \infty$.

Convergence Property: (Lemma 24.14)

If $s \rightsquigarrow u \rightarrow v$ is a shortest path in G for some $u, v \in V$, and if $u.d = \delta(s,u)$ at any time prior to relaxing edge (u,v), then $v.d = \delta(s,v)$ at all times afterwards.

Path-relaxation property: (Lemma 24.15)

If $p = \langle v_0, v_1, \ldots v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the order $(v_0,v_1), (v_1,v_2), \ldots (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$.

This property holds regardless of any other relaxation steps that occur, even if they are intermixed with relaxation of the edges of p.

Predecessor-subgraph property (Lemma 24.17)

Once $v.d = \delta(s,v)$ $\forall v \in V$, the predecessor subgraph is a shortest-paths tree rooted at s.

Algos for shortest path

Single-Source Shortest-Path (Graph Algo)

Assume the following conventions

1) for any real $a \neq -\infty$, $a + \infty = \infty + a = \infty$
2) for any real $a \neq \infty$, $a + (-\infty) = (-\infty) + a = -\infty$
3) G is stored in adjacency list representation

Bellman-Ford Algorithm

It solves single-source shortest-paths problem in case in which edge weights may be negative.

Given a weighted, directed graph $G = (V, E)$ with source s & wt. fn $w : E \to \mathbb{R}$, the BF algo. returns:

1) boolean indicating whether or not there is a negative-weight cycle that is reachable from source.
2) If there is no negative-wt. cycle reachable, the algo produces shortest paths and their weights.

BELLMAN-FORD Algo

BELLMAN-FORD (G, w, s)

```
1. INITIALIZE-SINGLE-SOURCE (G, s)
2.    for i = 1 to |G.V| - 1.
3.        for each edge (u, v) ∈ G.E
4.            RELAX (u, v, w)
5.    for each edge (u, v) ∈ G.E
6.        if v.d > u.d + w(u, v)
7.            return FALSE
8.    return TRUE
```

Algo relaxes edges, progressively decreasing an estimate $v.d$ on the weight of a shortest path from the source s to each vertex $v \in V$ until it achieves the actual shortest-path weight $\delta(s, v)$.

Steps

(1) Line 1 : initialize the d & π values of all vertices.

(2) Line 2 : Make $|V|-1$ passes over the edges of the graph. Each pass consists of relaxing each edge of the graph once.

(3) Lines 5-8 : check for a negative weight cycle.

Single Source Shortest (Graph Algorithms)

Cost: $O(VE)$

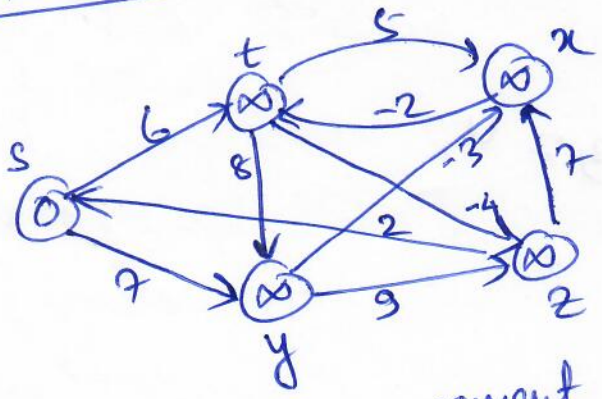
Pf Initialization: $\Theta(V)$ (Line 1)

Lines 2-4: $|V|-1$ passes over the edges $= (|V|-1)\,\Theta(E)$
$= \Theta(VE)$

Lines 5-7: $O(E)$.

Total: $\Theta(V) + \Theta(VE) + O(E)$
$= O(VE)$

Example: Bellman-Ford Algo



Consides the arrangement of edges in the following orders:

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x),
(z,s), (s,t), (s,y)

1st pass

(t,x) $x.d \not> t.d + w(t,x)$ $\infty \not> \infty + 5$

(t,y) $y.d > t.d + w(t,y)$? $\infty \not> \infty + 8$

(t,z) $z.d = \infty \not> \infty + 4$

(x,t) ×

(y,x) ×

(y,z) ×

(z,x) ×

(z,s) ×

(s,t) ✓ $t.d = s.d + w(s,t) = 6$

(s,y) ✓ $y.d = 0 + 7 = 7$

2nd pass

(t,x) ✓ $x.d = 11$

(t,y) ×

(t,z) ✓ $z.d = 2$

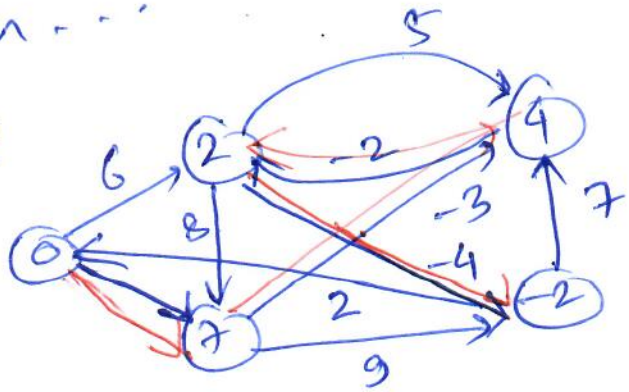
(x,t) ×

(y,x) ✓ $x.d = 7 - 3 = 4.$

(y,z) ✓ ~~$z.d =$~~

and so on...

Final Answer



Single-Source Shortest (Graph Algo)

Lemma 24.2

Let $G = (V, E)$ be a weighted, directed graph with source s and weight fn $w: E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, after the $|V|-1$ iterations of the for loop of lines 2-4 of BELLMAN-FORD, we have

$$v.d = \delta(s, v)$$

for all vertices v that are reachable from s.

Pf We recall the path relaxation property:

Path-relaxation Property: If $p = \langle v_0, v_1, \ldots v_k \rangle$ is a shortest path from $s = v_0$ to v_k, and we relax the edges of p in the orders $(v_0, v_1), (v_1, v_2), \ldots (v_{k-1}, v_k)$, then $v_k.d = \delta(s, v_k)$. Other edge relaxation can be "intermixed".

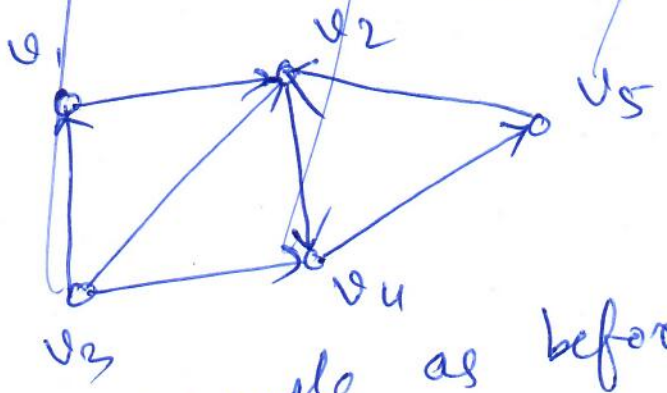
Considers any vertex v that is reachable from s, and let $p = \langle v_0, v_1, \ldots, v_k \rangle$, where $v_0 = s$ and $v_k = v$, be any shortest path from s to v. Because shortest paths are simple, p has at most $|V|-1$ edges, so $k \le |V|-1$. Each of the $|V|-1$ iterations of the for loop relaxes all $|E|$ edges. Among the edges relaxed in the ith iteration is (v_{i-1}, v_i), for $i = 1, 2, \ldots, k$. By path-relaxation property, $v.d = v_k.d = \delta(s, v)$. ∎

~~Cor. 24.3~~

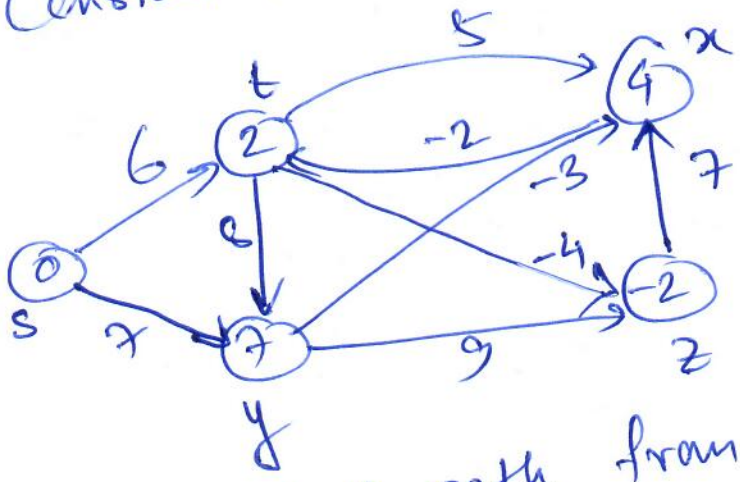
Example.

Let $V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{(v_3, v_1), (v_3, v_4), (v_5, v_2)\}$



Consider the example as before:



The shortest path from s to t is:

(s, y), (y, x), (x, t), i.e, $\langle s, y, x, t \rangle$

In the Bellman-Ford algo, we decide to relax the edges in the following order:

(t, x), (t, y), (t, z), (x, t), (y, x), (y, z),

(z, x), (z, s), (s, t), (s, y).

Since Shortest paths are simple paths, shortest path ~~cannot contain~~ can contain atmost $|V|-1$ edges.

Here, for this example $|V|-1 = 5-1 = 4$.

Hence, the Bellman-ford algo for loop (lines 2-4) runs 4 times, that is,

```
for i = 1 to 4
    RELAX (t,x,w)
    RELAX (t,y,w)
    .
    .
    RELAX (s,y,w)
end
```

Hence all relaxation happens in following order.

~~RELAX (t,x,w) RE~~

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

(t,x), (t,y), (t,z), (x,t), (y,x), (y,z), (z,x), (z,s), (s,t), (s,y)

So, (s,y), (y,x), (x,t) were relaxed in this order. By relaxation property,

~~$v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$~~

$t.d = \delta(s, t) = 2$, which is the shortest path from s to t.

Further relaxation of (s,y), (y,t), and (x,t) does not change $t.d$ (convergence property) ∎

Cor. 24.3

Let $G = (V, E)$ be a weighted, directed graph with source vertex s and weight fn $w: E \to \mathbb{R}$, and assume that G contains no negative-weight cycles that are reachable from s. Then, for each vertex $v \in V$, there is a path from s to v if and only if BELLMAN-FORD terminates with $v.d < \infty$ when it is run on G.

Th 24.4 (Correctness of the Bellman-Ford algorithm)

Let BELLMAN-FORD be run on a weighted, directed graph $G = (V, E)$ with source s and weight fn $w: E \to \mathbb{R}$. If G contains no negative-weight cycles that are reachable from s, then the algo returns TRUE, we have $v.d = \delta(s, v)$ $\forall v \in V$, and the predecessor subgraph G_π is a shortest-paths tree rooted at s. If G contains a negative weight cycle reachable from s, then the algo returns FALS

Pf. Spse that G contains no negative weight cycles that are reachable from the source s.

Claim 1): $v.d = \delta(s,v) \quad \forall v \in V$.

1) If v is reachable from s, then Lemma 24.2 proves this claim.

2) If v is not reachable from s, then the claim follows from the no-path property.

Predecessor-subgraph property implies that G_π is a shortest-paths tree.

Claim 2: BELLMAN-FORD returns TRUE.

At termination, we have for all edges $(u,v) \in E$,

$$\begin{aligned} v.d &= \delta(s,v) \\ &\le \delta(s,u) + w(u,v) \\ &= u.d + w(u,v) \end{aligned}$$

$\Rightarrow$ None of the tests in line 6 causes BELLMAN-FORD to return FALSE. Therefore, it returns TRUE.

Now, spse that G_s contains a -ve weight cycle that is reachable from source s; let this cycle be

$$c = \langle v_0, v_1, \ldots, v_k \rangle, \text{ where}$$

$v_0 = v_k$. Then,

$$\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$$

→ SSS

Assume for the purpose of contradiction that the Bellman-Ford algo returns TRUE.

$$\Rightarrow v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i) \qquad i = 1, 2, \ldots, k.$$

Summing the inequalities around cycle c gives

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i))$$

$$= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Since $v_0 = v_k$, each vertex in cycle c appears exactly once in each of the summations $\sum_{i=1}^{k} v_i.d$ and $\sum_{i=1}^{k} v_{i-1}.d$

and so $\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$ → (SS4)

Moreover, by Cor. 24.3, $v_i.d$ is finite for $i = 1, 2, \ldots, k$, because since the cycle is reachable, each vertex are reachable. Since $v_i.d$ is finite, we can cancel $\sum v_i.d$ from the inequality

$$\sum_{i=1}^{k} v_i.d \leq \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

(SS4)

$$= \sum_{i=1}^{k} v_i.d + \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

$$\Rightarrow \sum_{i=1}^{k} w(v_{i-1}, v_i) \geq 0$$

But it contradicts (SS3). Hence, BELLMAN-FORD algo returns TRUE if graph G contains no negative-weight cycles reachable from the source, and FALSE otherwise.

Dijkstra's Algo

* Solves single-source shortest path problem on a weighted directed graph $G = (V, E)$ in which all edge weights are nonnegative. $w(u,v) \geq 0 \; \forall (u,v) \in E$.

* Run time of Dijkstra's algo is lower than that of Bellman-Ford algo.

key idea

*) Maintain a set S of vertices whose final shortest-path weights from the source s have already been determined.

2) Repeatedly select the vertex $u \in V-S$ with the minimum shortest path estimate, adds u to S, and relaxes all edges leaving u.

```
DIJKSTRA (G, w, s)
1) INITIALIZE-SINGLE-SOURCE (G, s)
2)     S = ∅
3)     Q = G.V
4)     while Q ≠ ∅
5)         u = EXTRACT-MIN(Q)
6)         S = S ∪ {u}
7)         for each vertex v ∈ G.Adj[u]
               RELAX (u, v, w).
```

Line 1: initialize d & π values
Line 2: initialize the set S to empty set.
Line 3: initialize the min-priority queue Q to contain all the vertices in V.

Lines 4-8: Line 5 extracts a vertex u from Q = V-S and adds it to set S.
* Vertex u has smallest shortest path estimate of any vertex in V-S.

Line 7-8: Relax each edge (u,v) leaving u, thus updating v.d and the predecessor v.π if we can improve the shortest path to v found so far by going through u.

Th 24.6 (Correctness of Dijkstra's Algorithm)
Dijkstra's algorithm, run on a weighted, directed graph G = (V, E) with non-negative wt. fn w and source s, terminates with u.d = δ(s,u) ∀ u ∈ V.

Pf Assignment, CLRS, page 660

Run Time : 1) $O((V+E)\lg V)$ Analysis: p 66 (p 32)

- Min-priority queue with a binary min-heap

2) $O(V \lg V + E)$

- Min-priority queue with a Fibonacci heap.

(Details: CLRS, page 662).

Difference constraints and shortest Paths

1) Linear programming that reduce to finding shortest paths from a single source.
2) Solve single-source shortest paths problem by running Bellman Ford algo, thereby solving Linear prog pb.

Linear prog

Given $m \times n$ matrix A, m-vector b, n-vector c. We wish to find a vector x of n elements that maximize the objective fn $\sum_{i=1}^{n} c_i x_i$ subject to the m constraints given by

$$Ax \leq b.$$

Graph Algo

* Simplex does not run in poly time in the size of input.
* these are algo that run in poly tm.
* Single-pair shortest path & max-flow are special cases of LP.

Def^n (feasible sol^n)
Any vector x that satisfies $Ax \leq b$.

System of difference constraints

$Ax \leq b$ represent diff constraints.

representing $x_j - x_i \leq b_k$.

$1 \leq i, j \leq n$, $i \neq j$ & $1 \leq k \leq m$.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \leq \begin{bmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{bmatrix}$$

Equivalent to finding values of the unknowns x_1, x_2, x_3, x_4, x_5 satisfy

$$x_1 - x_2 \leq 0$$
$$x_1 - x_5 \leq -1$$
$$\vdots$$
$$x_5 - x_4 \leq -3$$

One sol^n $x = (-5, -3, 0, -1, -4)$.

$x' = (0, 2, 5, 4, 1)$

each compont of x' is 5 larger.

Lemma 24.8

Let $x = (x_1, x_2, \ldots x_n)$ be a sol^n to a system $Ax \leq b$ of diff. constraints and let d be any constant. Then $x + d = (x_1 + d, x_2 + d, \ldots x_n + d)$ is a sol^n. to $Ax \leq b$ as well.

Pf for each x_i & x_j

$$x_j + d - (x_i + d) = x_j - x_i$$

∎

Constraint graphs

Given $Ax \leq b$

Constraint graph : weighted, directed graph

$G = (V, E)$,

$V = \{v_0, v_1, \ldots, v_n\}$

$E = \{(v_i, v_j) : x_j - x_i \leq b_k$ is a constraint$\}$

$\cup \{(v_0, v_1), \ldots, (v_0, v_n)\}$.

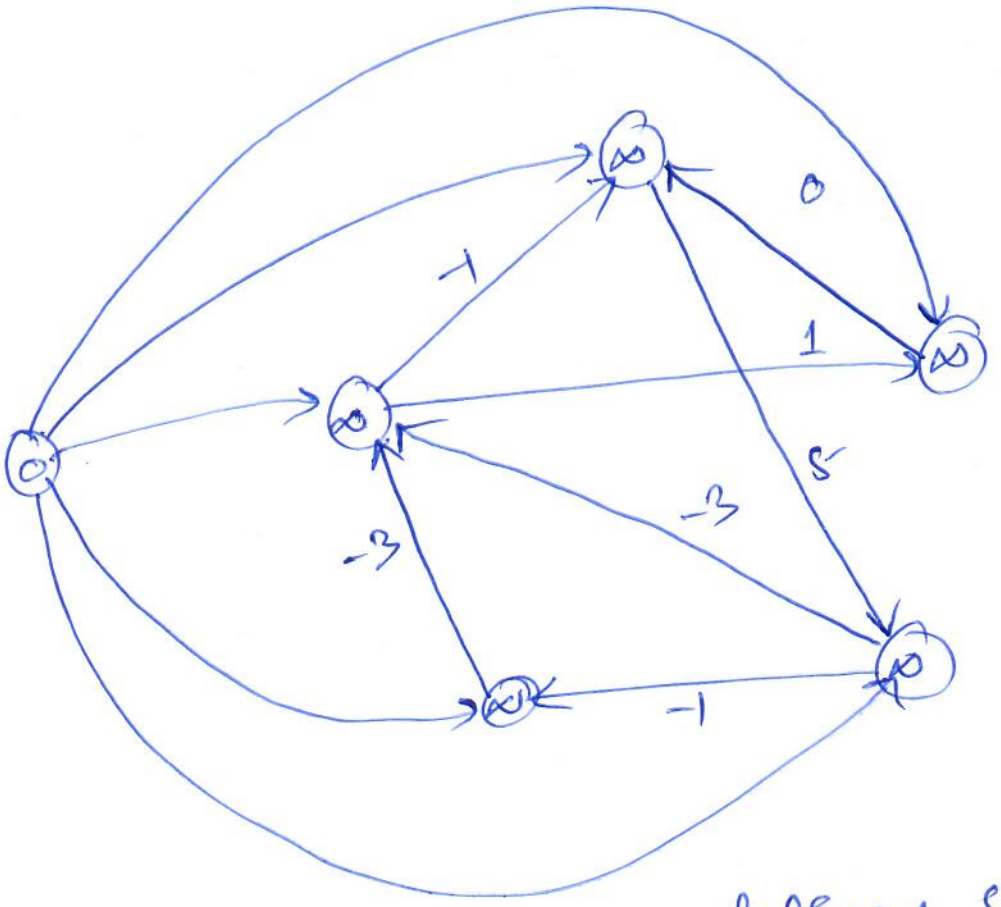
Additional vertex : v_0, to guarantee that the graph has some vertex which can reach all other vertices.

$V = v_i$ for each unknown $x_i + v_0$.

$E =$ an edge for each difference constraint + (v_0, v_i) for each x_i.

*) If $x_j - x_i \leq b_k$ then $w(v_i, v_j) = b_k$.

*) Weight of each edge leaving v_0 is 0.



Th. Given $Ax \le b$ of difference constraints, let $G = (V, E)$ be the corresponding constraint graph. If G contains no negative-weight cycles, then

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \cdots, \delta(v_0, v_n)) \quad (*)$$

is a feasible sol'n. If G contains a negative weight cycle, then there is no feasible sol'n of $Ax \le b$.

Pf. Claim: If the constraint graph contains no negative-weight cycles, then (*) gives a feasible sol'n.

Consider any edge $(v_i, v_j) \in E$.

By Δ^{le} ineq.

$$\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$$

$$\Rightarrow \delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$$

By choosing $x_i = \delta(v_0, v_i)$ & $x_j = \delta(v_0, v_j)$

the diff. constraint

$$x_j - x_i \leq w(v_i, v_j)$$

is satisfied.

claim: If the constraint graph contains a negative weight cycle, then the system of diff. constraints has no feasible soln.

W.L.O.G, Let the negative-wt. cycle be

$$c = \langle v_1, v_2, \ldots, v_k \rangle$$

where $v_1 = v_k$. [v_0 not in cycle, because no entering edge to v_0]

$$x_2 - x_1 \leq w(v_1, v_2)$$

$$x_3 - x_2 \leq w(v_2, v_3)$$

$$\vdots$$

$$x_k - x_{k-1} \leq w(v_{k-1}, v_k).$$

Assum x has a sol^n satisfying each of these k inequalities.

If we sum:

$$0 \leq w(c) \qquad \left[\begin{array}{c}\text{Note} \\ x_1 = x_k\end{array}\right]$$

But since c is a -ve wt. cycle, $w(c) < 0$, hence, a contradiction.

Complexity: m constraint + n unknowns

$\Rightarrow$ graph with $n+1$ vertices & $n+m$ edges

$$O((n+1)(n+m)) = O(n^2 + nm)$$