Graph Alg'ms

undirected
directed

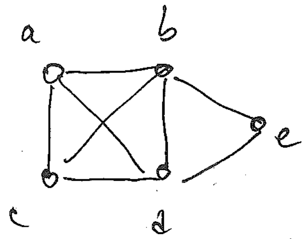
Def: A grah $G=(V,E)$

V = set of vertices

E = set of edges

edge is a pair of vertices (u,v)

(undirected graphs, usually denoted uv)



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

$E = \{ab, ac, ad, bc, bd, be, cd, de\}$

Applications

- models relationships between objects

Basic Terminologies

Def for undirected graph,

if $e = uv \in E$, u and v are adjacent (neighbors)

e is incident to u and to v

degree of u = $\deg(u)$ = # vertices adjacent to u

Fact. $\sum_{u \in V} \deg(u) = 2m$ (m is # of ~~vertices~~ edges in the graph)

Proof: for each vertex u, mark its incident edges

marks = $\sum \deg(u)$.

= $2m$ because each edge is marked twice

Def. for directed graphs,

out-deg(u) = # out-neighbors from u

in-deg(u) = # in-neighbors to u

Fact $\sum_{u \in V}$ in-deg$(u) = m = \sum_{u \in V}$ out-deg(u)

Def A path is a sequence $\langle v_0, v_1, \dots v_k \rangle$

with $(v_0, v_1) \in E$, $(v_1, v_2) \in E$ ... $(v_{k-1}, v_k) \in E$

it's a path from v_0 to v_k of length k

it's a simple path if all the vertices $\langle v_0 \dots v_k \rangle$ are all distinct

it's a cycle if $v_0 = v_k$ and all edges are distinct

if $\exists$ path from u to v, then v is reachable from u.

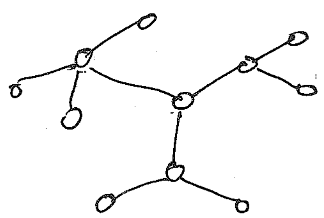
For undirected graph, G is connected,

if every vertex v is reachable from every vertex u.

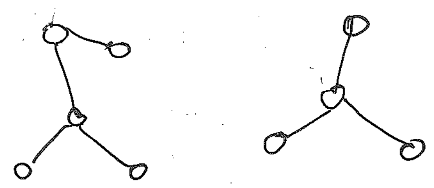
G is acyclic if there is no cycle.

An undirected, connected, acyclic graph is a tree

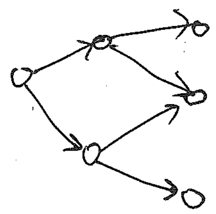
$(m = n-1)$

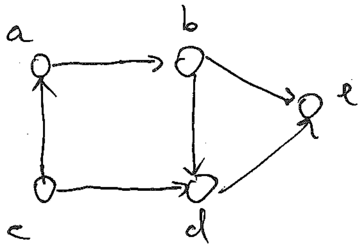


An undirected, acyclic graph is a forest.



A directed, acyclic, graph is a dag





Graph Alg'ms

usually n = # vertices $\Rightarrow$ $\underbrace{n \lesssim m}_{\text{no isolated vertices}} \leq n^2$

m = # edges

How to represent a graph?

a graph is said to be

$\begin{cases} \text{dense} & (m \text{ closer to } n^2) \\ \text{sparse} & (m \text{ closer to } n) \end{cases}$

option 1: adjacency matrix

$$A[u,v] = \begin{cases} 1 & (u,v) \in E \\ 0 & \text{otherwise} \end{cases}$$

space: $\Theta(n^2)$ great if graph is dense

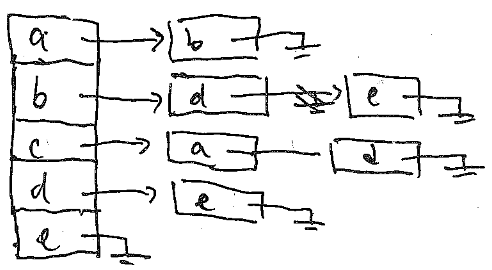
bad if graph is sparse

Option 2: adjacency lists

for each $u \in V$,

Store a linked list

Adj[u] = all out neighbors of u



space $\Theta(n + \sum_{u \in V} \text{out-deg}(u)) = \Theta(n+m)$

(good for sparse graph, at least as good as option 1 in terms of asymptotic bound)

Basic Problems

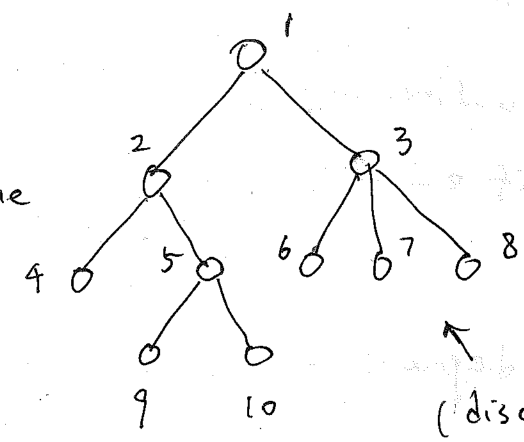
- Given a graph, is there a path from a given vertex s to vertex t ?
- find all vertices reachable from s.

Two Basic Alg'ms

- breadth-first-search (BFS)
- depth-first-search (DFS)

Ex. warm-up: trees

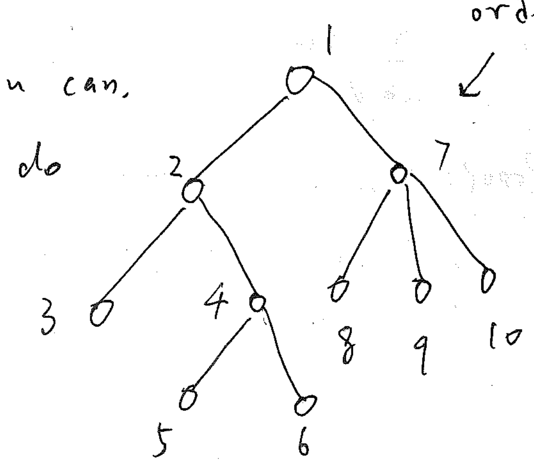
BFS: visit vertices at one level, then next level, etc...



(discovery order)

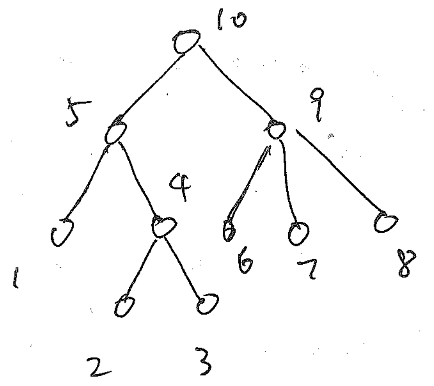
DFS: go as deep as you can, when ~~back~~ stuck, do backtrack.

(like pre-order traversal)

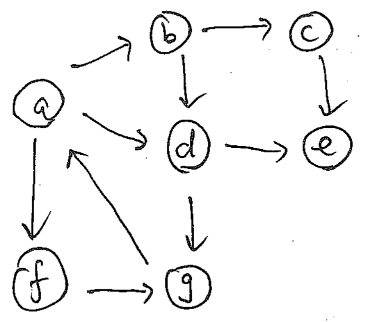


DFS (finish order)

like: post-order traversal



Ex. general graphs



take source = a.

idea: same idea for trees

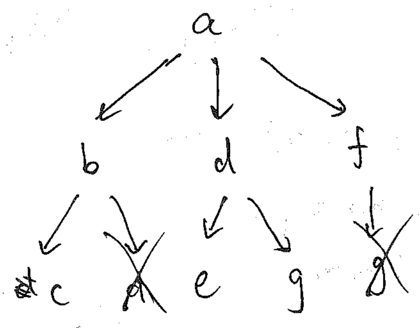
but don't visit vertices

that you have seen before.

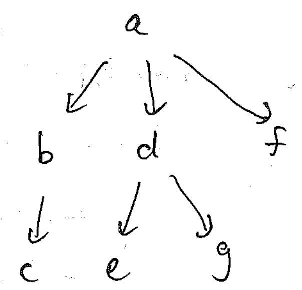
BFS

"BFS tree"

subgraph of G.



=>



back: (g, a)

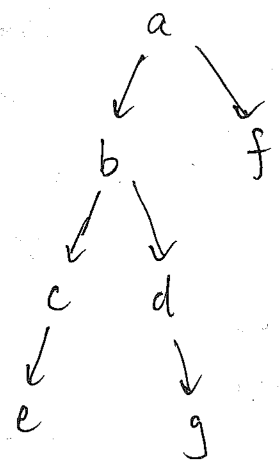
forward: none

cross: (c, e) (f, g)

(b, d)

DFS

"DFS tree"



back: (g, a)

forward: (a, d)

cross: (f, g) (d, e)

Def. An edge $(u, v) \in E$ is called

— a <u>tree edge</u> if it is in tree

— a <u>back edge</u>

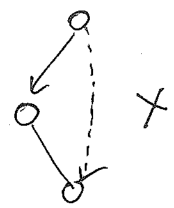
if v is an ancestor of u

— a <u>forward edge</u>

is v is a descendant of u

— a <u>cross edge</u> otherwise

<u>Facts</u> For BFS on ~~undir / undir~~ directed graph, no forward edges



(suppose $\exists$ a forward edge, the ~~vertex~~ lower

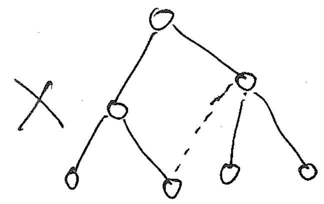
should be some levels above. first visited)

For BFS on undirected graph,

no forward / back edges

For DFS on undirected graph

no cross edges



Implementation of BFS:

```
BFS(G, s)    // idea: use a queue Q
1.  for each v ∈ V, do mark v "undiscovered".
2.  insert s to Q, mark s "discovered"
3.  while Q ≠ ∅,
4.      remove head u of Q
5.      for each v ∈ Adj[u] do
6.          if v is "undiscovered"
7.              insert v to tail of Q
8.              mark v "discovered", π[v] = u  → trace
                d[v] = d[u] + 1  → level
```

adjacency list representation

order of discovery elements in Q

Analysis: line 5-8: $O(|Adj[u]|)$

$= O(\text{out-deg}(u))$

each vertex inserted / deleted once.

$$T(n) = O(n + \sum_{u \in V} \text{out-deg}(u)) = O(n+m)$$

DFS (G, s) // idea - use ~~Stack~~ Recursion (is already a stack)

```
mark s "discovered"
for each v ∈ Adj[s] do
        if v is "undiscovered"
            DFS(G, v)
mark s "finished"
```

DFS (G) // explores the whole graph

```
for each v ∈ V do mark v "undiscovered"
for each v ∈ V do
    if v is "undiscovered" then DFS(G, s)
```

Analysis: each vertex acts as "s" once

⇒ total time $\Theta(n+m)$

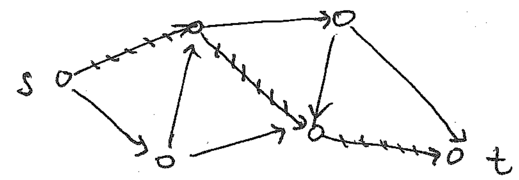
Application of DFS / BFS

- is t reachable from s?
  just run BFS / DFS from s,
  check if t is marked "discovered"
- is undirected graph connected?
  just run BFS / DFS from any fixed s
  check if all vertices marked "discovered"
- find all connected components of undirected graph?
  run BFS / DFS on whole graph
- is undirected graph acyclic?
  run BFS / DFS, check non-tree edges
- is directed graph acyclic?
  run DFS. check back edges
  ("discovered" but not "finished")

All above applications can be done in <u>Linear time</u>!

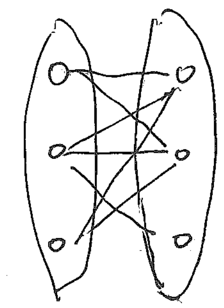
<u>Unweighted Shortest Path</u>

Given s, t ∈ V, find a path from s to t of shortest length



Alg'm: run BFS from s
return path from s to t in BFS tree
($\pi[v] = u$ for retrieve
$d[t]$ stores the length)

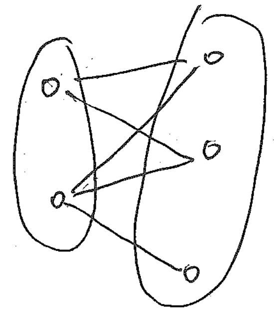
<u>Bipartiteness</u> (2-coloring problem)



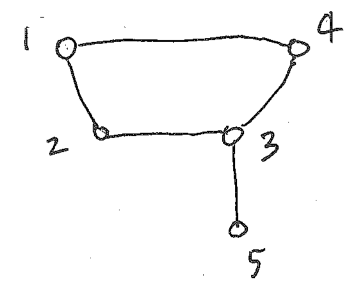
yes

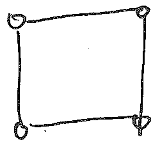
Given a undir graph $G = (V, E)$
check whether G is <u>bipartite</u>, ie.
V can be partition into V_1, V_2
s.t $\forall uv \in E$, $u \in V_1, v \in V_2$
or $u \in V_2, v \in V_1$

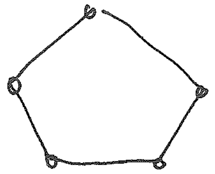
yes



←



 Yes

 No ← odd-length cycle

Collary : G is bipartite $\Longleftrightarrow$ G doesn't have odd-length cycles

Alg'm : // Assume G connected

$O(n+m)$ 1. run BFS from any vertex s

2. for each $v \in V$

$O(n)$ color v green iff v on even level

color v red iff v on odd level

3. for each $uv \in E$

$O(m)$ if (u,v) same color , return No // odd-length cycle

4. return Yes

Total time : $O(n+m)$

Generally, eg. 3-coloring is hard to solve,

proven only have exponential time algorithm.

Topological Sort

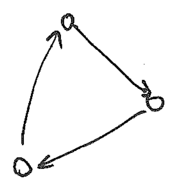
Given a directed graph $G=(V,E)$

return a vertex ordering s.t.

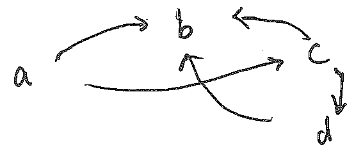
$\forall (u,v) \in E \Rightarrow u$ appears before v

Note : output not unique possibly

answer may not exist



eg.



answer: a, c, d, b

$a \to c \to d \to b$

Application : job scheduling

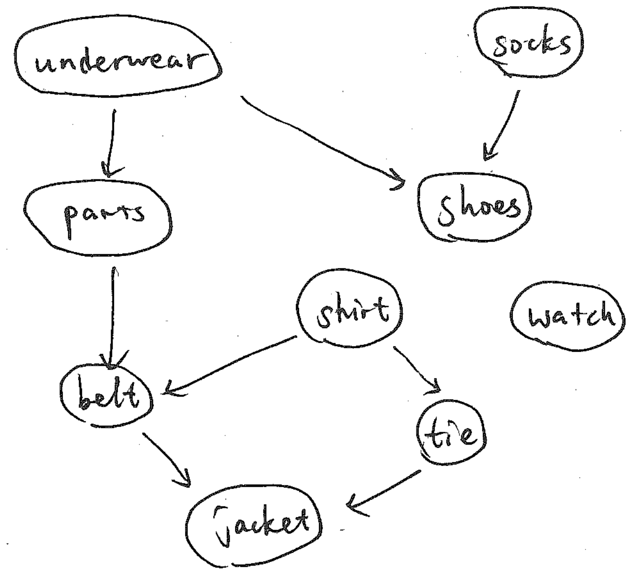
Alg'm : // Assume G is acyclic, or say a dag

1. run DFS (G) // on whole graph

$O(m+n)$

2. return reverse of finish order

Ex. "Prof. Bumstead's schedule"



DFS trees:

watch 1

shoes 2

shirt 6 → tie 4 → jacket 3

shirt → belt 5

socks 7

underwear 9 → pants 8

(1~9 finish order)

answer: 9 → 1 order

Note: Topological Sort is generalized sorting

$O(m+n)$ linear time not better than $O(n\log n)$

number sorting alg'ms,

since $m=n^2$ in this sence, yielding $O(n^2)$.

Correctness Proof: // Assume G is a dag

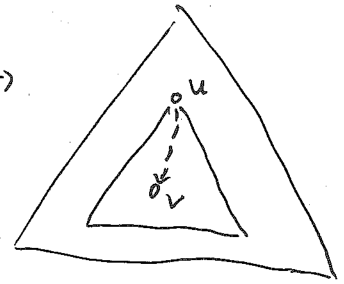
(sketch) $\forall (u,v) \in E$,

case A: u is discovered first before v.

$\hookrightarrow$ v is finished before u

$\Downarrow$

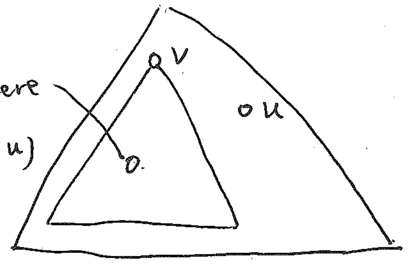
u appears first in output



case B: v is discovered before u.

$\therefore$ v is finished before u.

u cannot be here (or (u,v), (v,u) $\Rightarrow$ cycle)



$\therefore$ Proof is done!

~~Collora Colo~~ Corollary of Alg'm:

topological sort exists $\iff$ G is a dag.

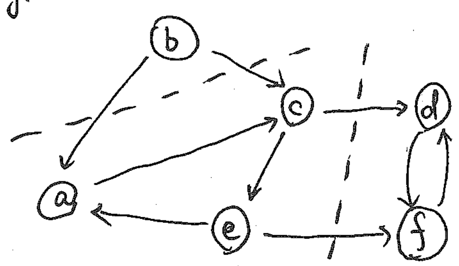
Strongly Connected Components

Given directed graph $G = (V, E)$,

partition V into components s.t.

u, v in same component $\iff \exists$ path $u \leadsto v$, $v \leadsto u$.

e.g.



3 components:

{b} {a, c, e} {d, f}

Application:

— simplify or condense a graph

good for visualization reachability

no cycle $\Rightarrow$ dag (nice for further analysis)

History: Purdom '68 $O(n^2)$

Munro '71 $O(n \log n + m)$

Tarjan '72 $O(n+m)$

Kosaraju '78

Sharir '81

$O(n+m)$ ← simpler

Kosaraju, Sharir's Alg'm: // idea: by Magic!

1. run DFS(G)

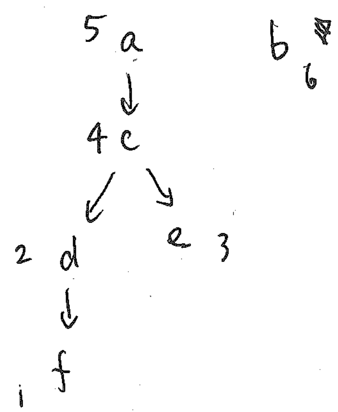
   number vertices in finish order

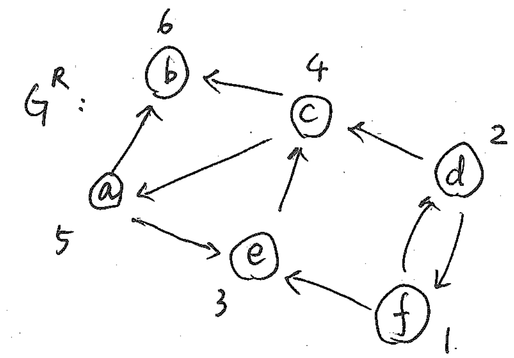
2. form "reverse" graph G^R

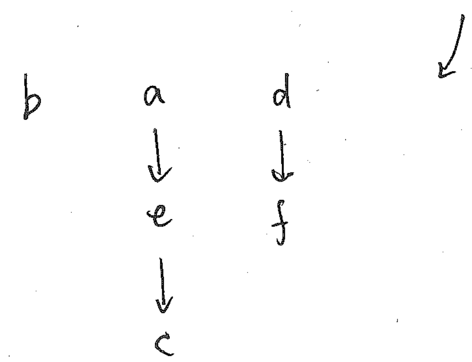
   run DFS(G^R), preferring higher-numbered vertices

   return DFS trees

Ex.







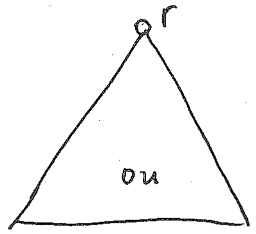
Correct partitioning!

Correctness Proof:

Take DFS tree T of G^R

let r be root

u be any node in T



in G^R

Proof in "5 easy steps":

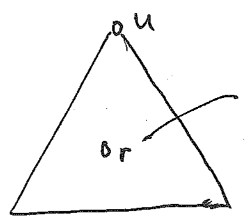
1. r has higher number than u. // preference rule

2. $\exists$ path $u \rightsquigarrow r$ in G // $\exists$ $r \rightsquigarrow u$ in G^R

3. $\exists$ path $r \rightsquigarrow u$ in G

(if not, case A: u is discovered before r.

DFS trees in G

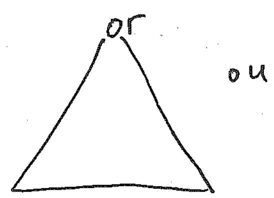


by step 2.

$\Rightarrow$ r is finished before u,

case B: r is discovered before u.

DFS trees in G



by negation of statement ($\nexists$ $r \rightsquigarrow u$ in G)

$\Rightarrow$ r is finished before u,

case A, B $\Rightarrow$ r has smaller number

(Contradiction!)

)

4. $u, v \in T$. $\Leftrightarrow$ $\exists$ path $u \rightleftarrows v$ ($u \rightleftarrows r \rightleftarrows v$)

5. $u \rightleftarrows v$ $\Rightarrow$ u, v in same tree

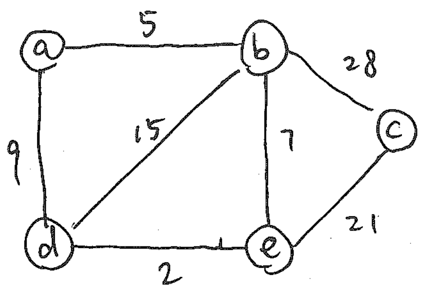
Minimum Spanning Trees

Given a weighted undirected graph

$G = (V, E)$ $w : E \to \mathbb{R}^+$ weight function

find a connected subgraph T using all vertices while minimizing total weight.

e.g.



one sol'n: $5 + 9 + 7 + 28$
$= 49$

optimal sol'n: $5 + 7 + 2 + 21$
$= 35$

Observation: the optimal subgraph must be acyclic.

$\Rightarrow$ answer is connected, acyclic, undirected $\Rightarrow$ tree.

Idea 0 - brute force: exponential!

Idea 1 - greedy: this works!

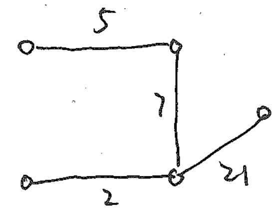
Kruskal's Alg'm (1956): high-level version

naively, $O(mn)$ time
(pre-sort weights)

1. $T = \phi$
2. repeat {
3. pick next smallest-weight edge e
4. if $T \cup \{e\}$ doesn't contain a cycle, // $O(n)$ naively (DFS / BFS)
5. insert e to T

}

skipped: 9, 15, 28



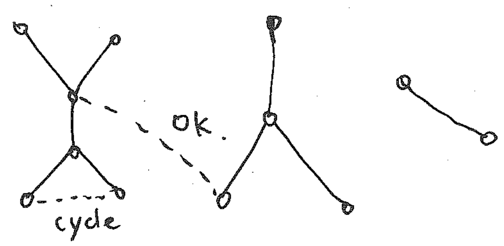
Simplest to do by HAND!

Implementation of Kruskal : detailed version

1. sort the edges in increasing weight — $O(m\log m) = O(m \log n)$ ↑ sorting bottleneck
2. create set $\{v\}$ $\forall v \in V$
3. for each edge $e = uv$ in sorted order
4. if u & v belong to different sets — $O(\alpha(n))$
5. print uv, union 2 sets

line 4–5: need data structure: operate on disjoint sets.

general picture of T at any moment:



forest. no cycle $\Updownarrow$ edge joining 2 tree

1) find a ptr to set containing v.

2) union 2 sets

$\Rightarrow$ the "union-find" problem

from book: union & find in $O(\alpha(n))$

$\alpha(n)$ is very slow-growing function ← $\alpha(n) < 5$ for all remotely practical value of n

$\alpha(n) = o(\log n) \Longrightarrow \alpha(n) = o(\log^* n)$

total time: $O(m\log n + m\alpha(n)) = O(m \log n)$

(may exist faster alg'm ...)

Ackermann function

btw, $\alpha(n)$ is inverse function of $A(x,x)$

$$A(m,n) = \begin{cases} n+1 & m=0 \\ A(m-1,1) & m>0,\ n=0 \\ A(m-1, A(m,n-1)) & m>0,\ n>0 \end{cases}$$

naively $O(mn)$ total time $\Longrightarrow$

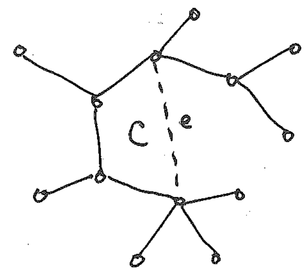
Correctness Proof (Assume weights are distinct)

Key Lemma.

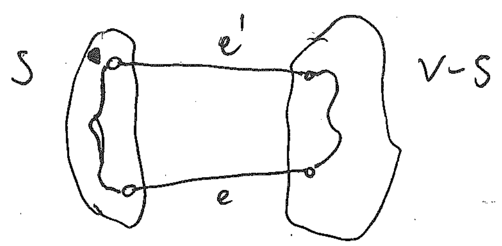
Given any $S \subseteq V$, the smallest-weight edge between S & $V-S$ must be in MST., T^*

Proof: by contradiction, suppose $e \notin T^*$, e is the smallest-weight edge

1. $T^* \cup \{e\}$ contains a cycle C.



2. C must contain another edge e' between S & $V-S$



3. $T^* \cup \{e\} - \{e'\}$ is a tree

by def. $w(e) < w(e')$

$w(T^*) + w(e) - w(e') < w(T^*)$

Correctness of Kruskal :

each edge uv inserted by Kruskal must be in MST by Lemma.

(one tree as S, other part of forest $V-S$)

Prim's Alg'm (1957) : high-level version

1. $T = \phi$, $S = \{s\}$
2. while $S \neq V$ do {
3. pick smallest-weight edge uv with $u \in S$, $v \in V-S$
4. insert uv to T, v to S

}

Correctness : by Lemma again directly!

Implementation of Prim: detailed version

// maintain $key[v] = \min_{u \in S} w(uv) \quad \forall v \in V-S$

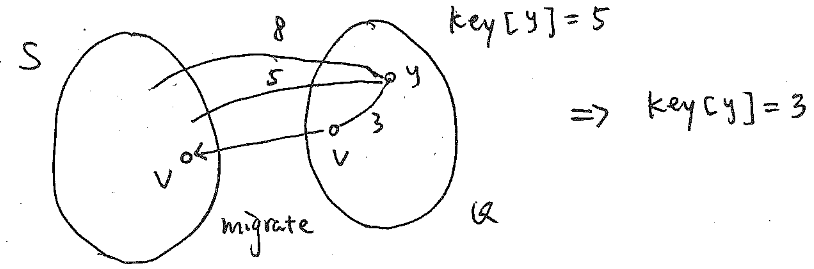
// $\pi[v]$ = the $u \in S$ attaining min

// $Q = V - S$ (alias)

1. $Q = V$
2. $key[v] = \infty \quad \forall v \in V - \{s\}$, $key[s] = 0$
3. while $Q \neq \phi$ {
4. pick $v \in Q$ with smallest key. $u = \pi[v]$
5. pick uv & remove v from Q
6. for $y \in Adj[v]$ do {
7. if $y \in Q$ & $w(vy) < key[y]$,
8. $key[y] = w(vy)$, $\pi[y] = v$

    }

}



Analysis:

Option 1: no data structure

line 4: $O(n) \times n$

line 8: $O(\sum \deg(v)) = O(m)$

total: $O(n^2 + m) = O(n^2)$

no log factor,

good for dense graphs

Option 2: standard heap

line 4-5: delete-min: $O(\log n) \times n$

line 8: change-key: $O(\log n) \times O(m)$

total time: $O(n\log n + m \log n) = O(m \log n)$

fast for sparse graph

option 3: A2 Q3

delete-min: $O(\sqrt{n})$

decrease-key: $O(1)$ change-key happens to be decrease-key

total time: $O(n^{\frac{3}{2}} + m)$

option 4: Fibonacci heap // best known

delete-min $O(\log n)$

decrease-key $O(1)$

at least as good as all previous methods

total time: $O(n\log n + m)$

for large m, faster than sorting in Kruskal

Can we do better?

Yao '75 $O(m \log\log n)$

Fredman. Tarjan '85 $O(m \log^* n)$ '86 $O(m \log(\log^* n))$

Chazelle '97 $O(m\,\alpha(n))$

Karger, Klein, Tarjan '94 $O(m)$ randomized

Shortest Paths

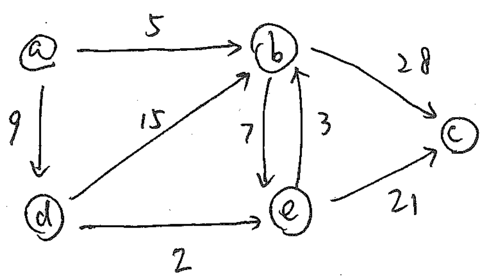
Given a weighted directed / undir graph

$G=(V,E)$, $w: E \to \mathbb{R}^+$, $s, t \in V$.

find path p from s to t,

minimizing $w(p) = \sum_{e \in p} w(e)$

e.g.



optimal: 32

$a \to d \to e \to c$

Special Case 0: unweighted

BFS $\Rightarrow$ $O(m+n)$

Special Case 1: dag (negative weights OK)

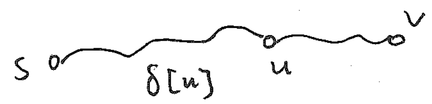
idea: DP

subproblem: $\forall v \in V$, $\delta[v]$ = weight of shortest path from s to v.

answer: $\delta[t]$

base case: $\delta[s]=0$

recursive formula: choices over next-to-last vertex u in optimal path



$$\delta[v] = \min_{\{u : (u,v) \in E\}} \left(\delta[u] + w(u,v) \right)$$

$O(mn)$ naively

evaluation order: topological sort! $\longrightarrow$ only works for dag

$(O(m+n))$

analysis: $O(\underbrace{n}_{\text{table}} + \underbrace{\sum \text{in-deg}(v)}_{m}) = O(n+m)$

also works for longest path:

1) min $\to$ max   2) negate the weights

General Case?

naive greedy - take smallest-weight out of s : fail!

" " " into t : fail!

need a clever greedy idea...

Dijkstra's Alg'm (1959): high-level version

// assume positive weights

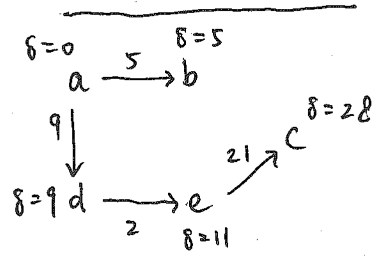
// solve single-source, all-destination problem from s to v, $\forall v \in V$

// compute $\delta[v]$ = weight of shortest path from s to v, $\forall v \in V$.

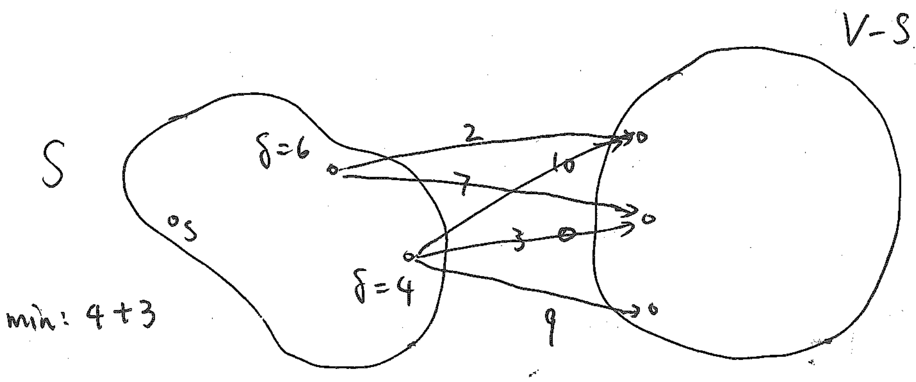
1. $S = \{s\}$, $\delta[s] = 0$
2. while $S \neq V$ do {
3. pick edge (u,v) $u \in S$, $v \in V-S$ minimizing $\delta[u] + w(u,v)$
4. insert v to S, $\delta[v] = \delta[u] + w(u,v)$

}

"shortest path tree"



VERY similar to Prim's !!!



min: 4+3

Implementation of Dijkstra : detailed version

similar to Prim's

change line 7-8:

```
7.    if y ∈ Q & key[v] + w(v,y) < key[y] {
8.        key[y] = key[v] + w(v,y) , π[y] = v
      }
```

Analysis :

no data structure	$O(n^2)$ time
heap	$O(m \log n)$ time , $m \gtrsim n$. sparse
fibonacci heap	$O(n \log n + m)$ time

Remark :

Dijkstra is generalization of BFS (unweighted shortest path)

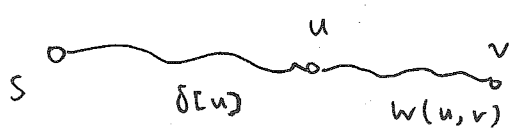
queue $\longrightarrow$ priority queue

Correctness Proof

Claim : if (u,v) is the edge from S to $V-S$ minimizing $\delta[u] + w(u,v)$, then $\delta[v] = \delta[u] + w(u,v)$

Proof : $(\leq)$ there is a path from s to v of weight $\delta[u] + w(u,v)$

(minimum weight $\leq \delta[u] + w(u,v)$)



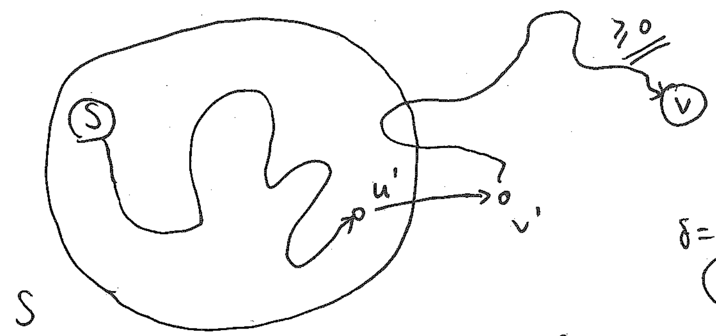
$(\geq)$ for any path p from s to v,

p must have an edge (u',v') with $u' \in S$, $v' \in V-S$.

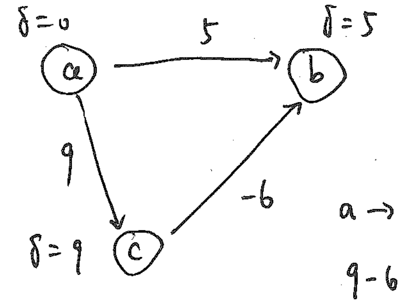
$\Rightarrow w(p) \geq \delta[u'] + w(u',v') + 0 \leftarrow$ non-negative weight

$\geq \delta[u] + w(u,v)$ by def.

by Claim, each iteration of Dijkstra is correct.



s = a



$a \to c \to b$

$9 - 6 < 5$!

NOT work for negative weights :

($O(mn)$ at least) $\hookrightarrow$ preprocess to non-negative

All-Pairs Shortest Paths

Given weighted dir / undir graph $G = (V, E)$,

find shortest path between all pairs of vertices.

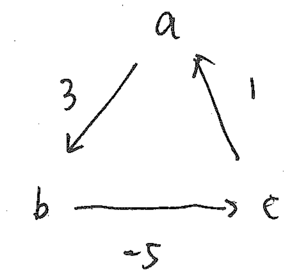
(output size $\Omega(n^2)$)

assumption - negative weights allowed

(but no negative-weight cycle)

min-weight :

$-\infty$



Method 0 - Run Dijkstra for every source vertices

$\Rightarrow O(n(n\log n + m))$ total time

if no negative weights

this is actually very good for sparse graphs.

DP Method 1:

Define subproblems $(i = 1 \dots n,\ j = 1, \dots n) + (k = 1, 2, \dots n-1)$

$D[i,j,k]$ = min weight of shortest path from i to j among paths of length $\leq k$

answer: $D[i,j,n-1]\ \forall i,j$

(optimal paths can't have repeated vertices! since repeated vertices $\Rightarrow$ cycles (negative weights))

base case:

$$D[i,j,1] = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ 0 & \text{if } i=j \\ \infty & \text{else} \end{cases}$$

recursive formula: n choices for next-to-last vertex l in optimal path

$$D[i,j,k] = \min\left(D[i,l,k-1] + w(l,j)\right) \quad \forall l \in \{1..n\}$$

evaluation order: increasing order of k

retrieve optimal paths: π array

analysis: $O(n^3)$ entries $O(n)$ each entry

total time: $O(n^4)$

DP Method 2: (reduce # entries)

Same definitions of subproblems yet different recursive formula:

(not incremental linearly, but by doubling k)

n choices for middle vertex l in optimal path

$$D[i,j,k] = \min\left(D[i,l,\tfrac{k}{2}] + D[l,j,\tfrac{k}{2}]\right) \quad \forall l \in \{1, \dots n\}$$

evaluation: try $k = 1, 2, 4, 8 \dots n-1$.

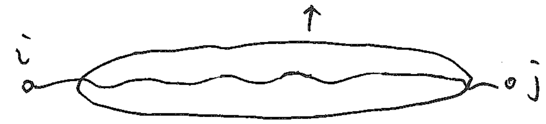
only $\log n$ different values of k

total time: $O(n^2 \log n) \times O(n) = O(n^3 \log n)$

DP Method 3: Floyd, Warshall's Alg'm (1962)

define subproblems: $(i,j = 1 \dots n,\ k = 0 \dots n)$

$D[i,j,k]$ = min weight of shortest path from i to j. among paths whose intermediate vertices from $\{1 \dots k\}$.



answer: $D[i,j,n]\ \forall i,j = 1 \dots n$

base case:

$$D[i,j,0] = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ 0 & \text{if } i=j \\ \infty & \text{else} \end{cases}$$

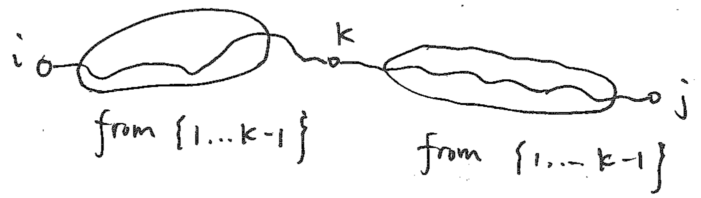
recursive formula:

case 1. not use vertex k in optimal path

$$D[i,j,k] = D[i,j,k-1]$$

case 2. use vertex k in optimal path

(claim: only used once)



$$D[i,j,k] = D[i,k,k-1] + D[k,j,k-1]$$

Hence:

$$D[i,j,k] = \min \begin{cases} D[i,j,k-1] \\ D[i,k,k-1] + D[k,j,k-1] \end{cases}$$

evaluation order: increasing k

analysis: total time: $O(n^3) \times O(1) = O(n^3)$!

space: $O(n^2)$ if don't need to retrieve

Remark: 1. comparing to Dijkstra, not faster than $O(n(n\log n + m))$, especially for sparse graphs

2. but is much simpler, smaller constant factors

Can we do better?

for unweighted, undir graph,

$O(n^{2.373})$ time by matrix multiplication!

$$c[i,j] = \sum_{\ell} c[i,\ell] \times c[\ell,j]$$

$$\Downarrow$$

$$D[i,j,\_] = \min_{\ell} (D[i,\ell,\_] + D[\ell,j,\_])$$

general weighted case?

$O\left(\frac{n^3}{(\log n)^{1/3}}\right)$ ... $O\left(\frac{n^3}{(\log n)^2}\right)$ ... $O\left(\frac{n^3}{2^{\sqrt{\log n}}}\right)$

1976 Chan 2007 2014

but $O(n^{2.999\cdots})$?

OPEN!

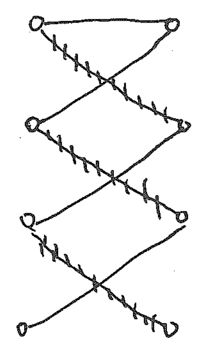
List of Other Graph Problems:

1. max matching:

Given undir graph $G=(V,E)$

find largest matching $M \subseteq E$

s.t. no 2 edges in M share a vertex

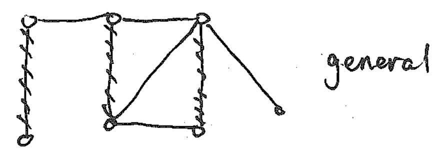


Can be solved in polynomial time:

bipartite case:

$O(m\sqrt{n}) \rightarrow O(n^{2.373}) \rightarrow \tilde{O}(m^{10/7}) \cdots$

1973 2004 2013

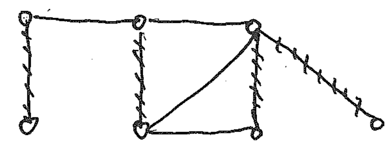


2. min edge cover:

find smallest subset $S \subseteq E$

s.t. every vertex in V is

incident to some edge in S.



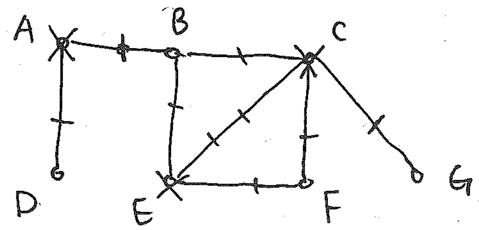
proved: can be reduced to max matching.

3. min vertex cover:

find smallest subset $S \subseteq V$

s.t every edge in E is

incident to some vertex in S.



HARD yet FUNDAMENTAL!

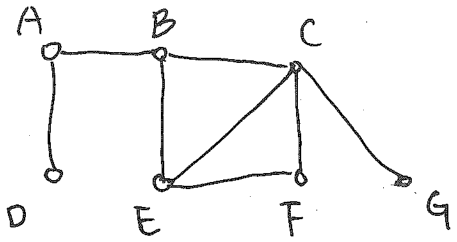
no poly time alg'm till now...

$\{A, C, E\}$, min = 3

4. <u>max independent set:</u>

find largest subset $S \subseteq V$

s.t. no two vertices in S are adjacent



e.g. $\{A, C\}$. $\{A, E, G\}$

$\{B, D, F, G\}$ max = 4 (compliment to <u>min vertex cover</u>)

claim: equivalent to <u>min vertex cover</u>

<u>independent set</u>

↳ $u \in S$ and $v \in S$

$\Rightarrow uv \notin E$

↳ $uv \in E$

$\Rightarrow u \in S$ or $v \in S$

$\Vert$

$u \notin S$ and $v \notin S$

$\Rightarrow uv \notin E$

So, just flip S!

Application: e.g. "disjoint intervals" is a special case

A5 Q3 also a special case

Remark: 4 $\Leftrightarrow$ 3, no poly alg'm to solve general case.

5. <u>max clique:</u>

find largest <u>complete subgraph</u> (every vertex is adjacent to all other vertices)

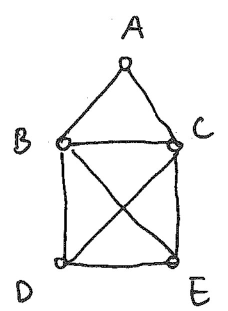
i.e. largest subset $S \subseteq V$. s.t.

$u \in S, v \in S \Rightarrow uv \in E$

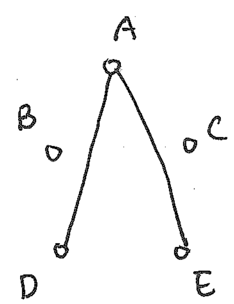
indep set in G^c:

$\{B, C, D, E\}$

G:



G^c:



claim: equivalent to <u>max indepent set</u>

S is a clique $\Leftrightarrow$ S in an indep set in G^c

6. <u>min vertex coloring:</u>

color the vertices with min # colors

s.t. $uv \in E \Rightarrow u$ & v have different colors

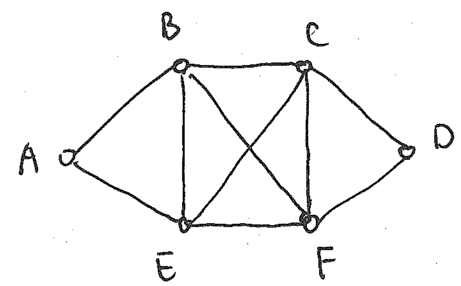
2 colors $\Rightarrow O(m+n)$ by BFS / DFS

3 colors $\Rightarrow$ HARD!

4 colors $\Rightarrow$ guaranteed by Four Color Theorem

7. <u>Euler cycle:</u>

does there exist a cycle that uses every <u>edge</u> exactly once?



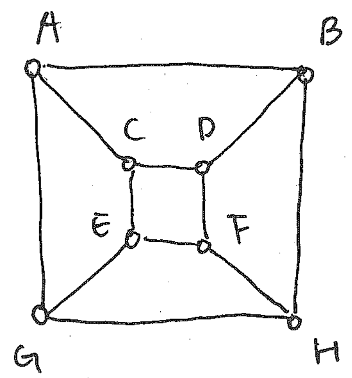
ABEFCBFDCEA

answer exist $\Leftrightarrow$ all vertex have even degree

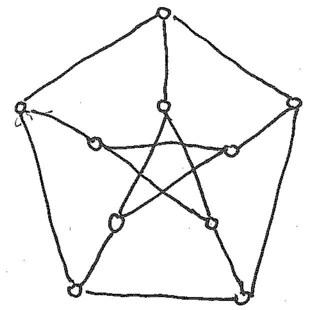
$O(m+m)$ time.

8. <u>Hamiltonian cycle:</u>

does there exist a cycle that uses every <u>vertex</u> exactly once?



ABHGEFDCA



No!

← famous Peterson Graph.

Knight Tour problem related.

9. Traveling Salesman Problem:

Given weighted graph, find cycle that visits all vertices minimizing total weight.

(sounds similar to shortest path and MST, but believed to be HARD!)

ETC... useful graph problems are countless.

General Question:

how to prove (know) that a problem is hard?

Theory of NP-Completeness

Def. P = all "easy" problems

all decision problems that can be solved in POLYNOMIAL time.

(answer: YES or NO)

(nondeterministic)
↳ NP = all decision problems that can be verified in POLYNOMIAL time.

e.g. Hamiltonian Cycle is NP:

given cycle, can verify whether it works in polytime

Vertex Cover is NP:

(Decision problem: given K, decide whether there exists a vertex cover of size $\le K$)

given subset, can verify whether it is a vertex cover of size $\le K$.

FACT. $P \subseteq NP$

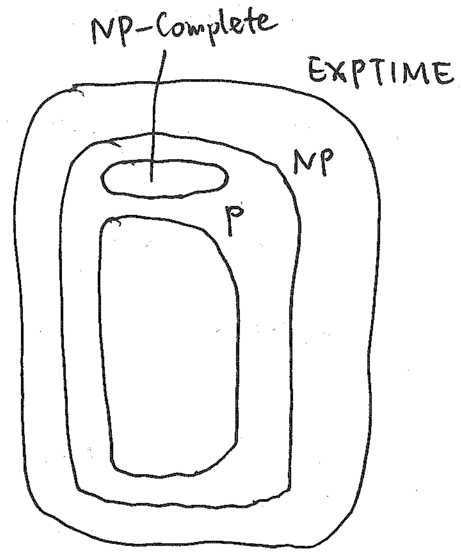
$NP \subseteq EXPTIME$

(brute force: try all possible)

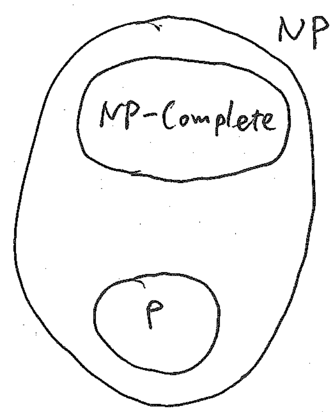
"Million-Dollar" Conjecture (1970)

$P \neq NP$ (general consensus though)

most famous OPEN problem.



Def. Problem X is NP-Complete if X is a "hardest problem" in NP.



Def. Problem X reduces to Problem Y if we can design an alg'm for X using an alg'm for Y.

e.g. selection reduces to sorting

can solve Y efficiently ⟹ can solve X efficiently

i.e. X is "easier than or as easy as" Y.

Def. Problem X is NP-Complete

if X is a "hardest problem" in NP → strong definition

ie. 1) X is NP

2) Every problem in NP reduces to X.

Fact. if $P \neq NP$, then any NP-Complete problem cannot be solved in polytime.

Does an NP-Complete problem exist? YES

Problem ~~Satisficati~~ Satisfiability (SAT)

Given Boolean formula F, decide whether there exists ~~and an~~ assigments of variables to make F true.

e.g. $(x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$

$x_1 = \text{true}$, $x_2 = \text{false}$ $\Rightarrow$ $F = \text{true}$

verify SAT is easy : substitute variables and evaluate

solve SAT is hard : brute force ? ...

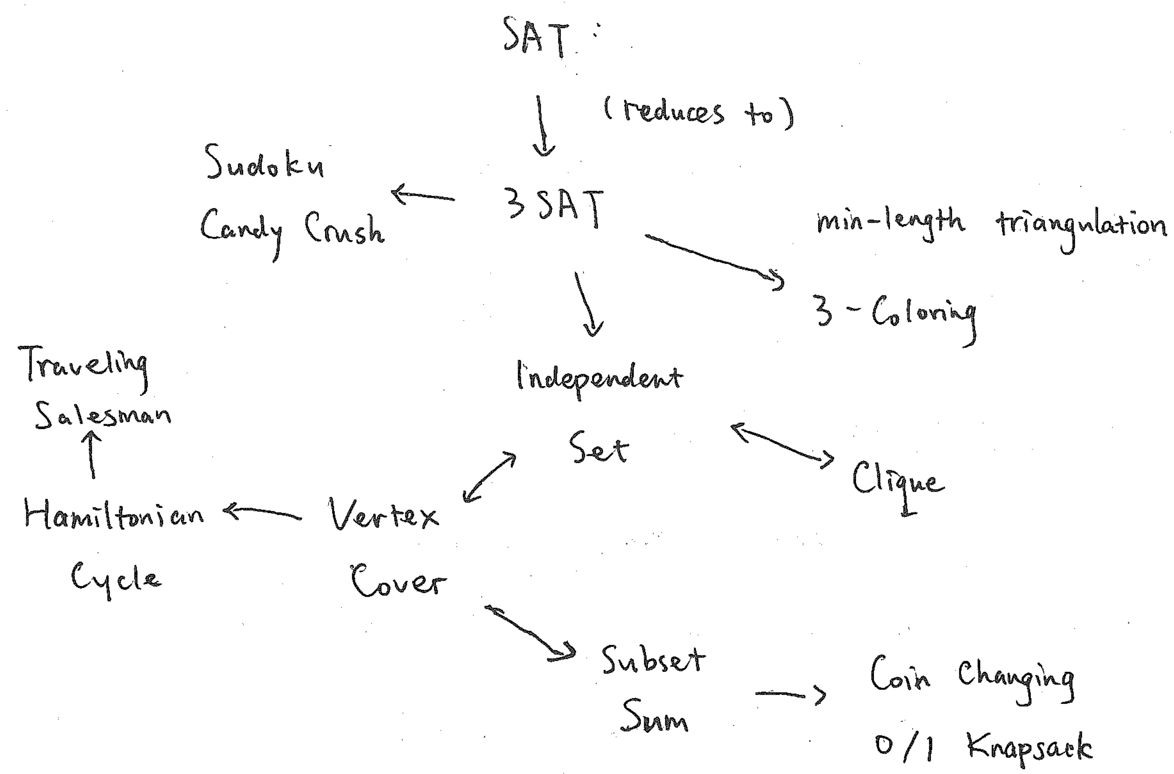
Cook's Theorem (1971) : SAT is NP-Complete

Proof Idea : everything computable reduces to logic!

Are there more NP-Complete problems? YES! thousands!

Most problems are either P or NP-Complete!

Roughly only 10 problems are not P nor NP-Complete ...



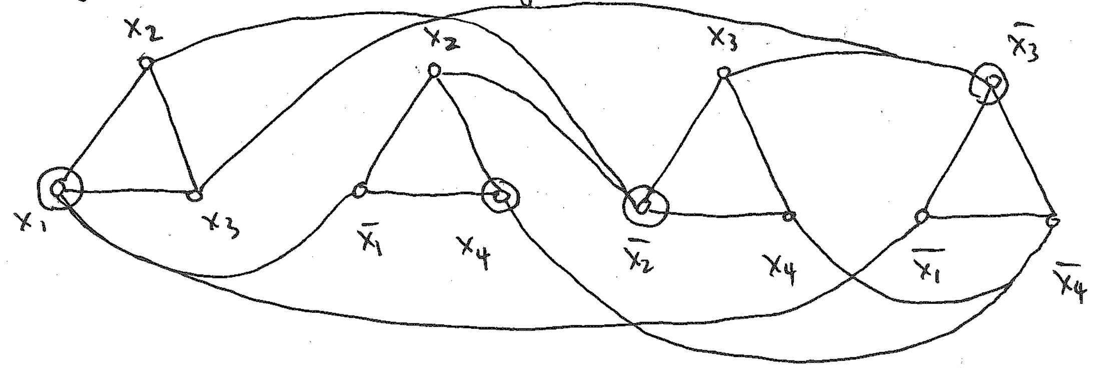
Example Reduction from 3SAT to Independent Set (Sketch)

Given Boolean formula for 3SAT

$$F = (x_1 \vee x_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee x_4)$$
$$\wedge (\bar{x}_2 \vee x_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_3 \vee \bar{x}_4)$$

decide whether $\exists$ assignment to make F true.

Alg'm: convert F to a graph :



run Indep Set : x_1, x_4, $\bar{x}_2$, $\bar{x}_3$

$\Rightarrow$ solution to 3SAT : $x_1 = \text{true}$ $x_2 = \text{false}$, $x_3 = \text{false}$, $x_4 = \text{true}$

⇒ If you could solve Indep Set,

then I could solve 3SAT.

⇒ If 3SAT is hard, then Indep Set is hard.

Want to prove A is hard,

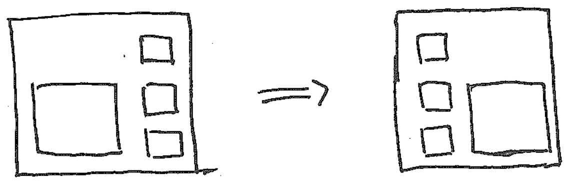
try reducing A to some known hard problem B. X

try reducing known hard problem B to A. ✓

Beyond NP: Examples

- "Warehouse Problem!"

e.g.



exponential number of moves ← also Tower of Hanoi

(can't verify in polytime)

"PSPACE - Complete"

- test whether 2 regexps are equivalent.

"EXPSPACE - Complete"

- "HALTING Problem"
  - Given a string P representing a C++ program,
    decide if P always halts. (terminates)

e.g. "main(int n) { while (n>0) n -= 2; }" ⇒ yes.

"main (int n) { while (n ≥ 1)
if n even, n = n/2;
else n = 3n+1; }."

Turing's Theorem (1936)

Halting problem can't be solved by any alg'm.

it's undecidable.