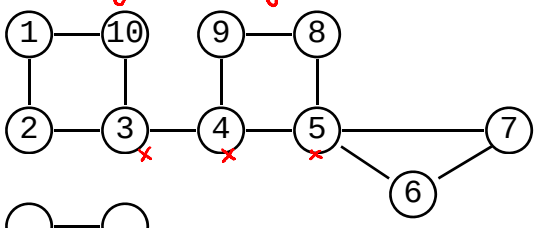
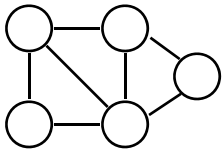
Biconnectivity

Let G = (N,E) be a connected, undirected graphs

A node a ∈ N is an <u>articulation point</u> if there are v and w different from a such that every path from v to w goes through a



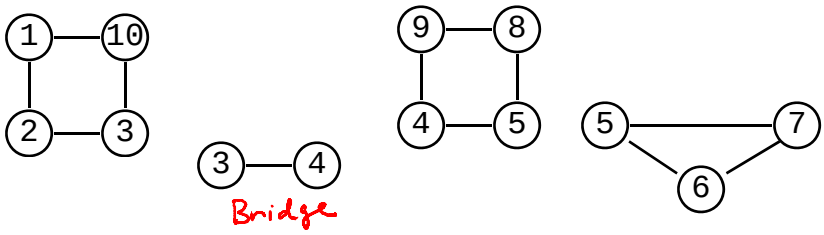


No articulation points

Biconnected Component

Maximal subgraph that is still connected after removal of any one node

Express in terms of edges



Articulation point must lie in two components

DFS and Articulation Points

Tree, back edges in DFS give clue to articulation points

If a is an articulation point

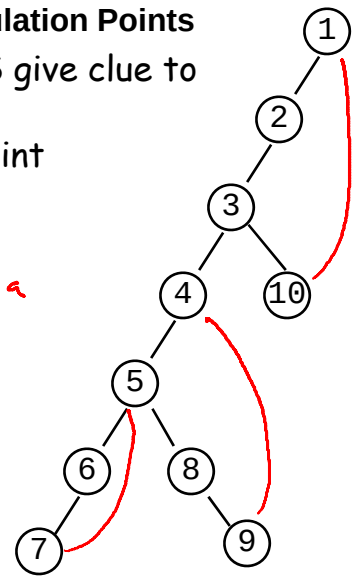
that splits v and w, then

v is an ancestor

w is a descendent of a

or

vice versa



DFS and Articulation Points 2

Lemma:

G = (N,E) is connected, undirected ~~pa~~ graph

S = (N,T) is DFS Tree of G

Node a is an articulation point of G if and only if

1. a is the root of S and

   a has two or more children

or

2. a is not the root and a has some child c in S where there is no back edge from c or any descendent to a proper ancestor of a.

Low Numbers

For graph G = (N,E), let

T = Tree edges from a DFS

B = Back edges

For simplicity, assume DFNum[v] = v

How high (lowest DFNum) can we hop from a node on a back edge?

LOW[v] =

min({v} ∪

{w | there is {x,w} in B where x is a descendent of v}

include v ~~itself~~ itself

Calculating Low

If {x,w} in B and w < v then (DFNum(w) < DFNum(v))

w is an ancestor of v.

Also, v is an articulation point if for some child c of v,

LOW[c] ≥ v

Calculate LOW during DFS

LOW[v] is minimum of

1. v
2. LOW[c] for all children of v
3. w where {v, w} ∈ B

Biconnected Components Algorithm

```
Count – Global, = 0
Parent[v] – parent of node v in T
BI(v)
  mark v
  DFNum[v] ← Count, increment Count
  Low[v] ← DFNum[v]
```

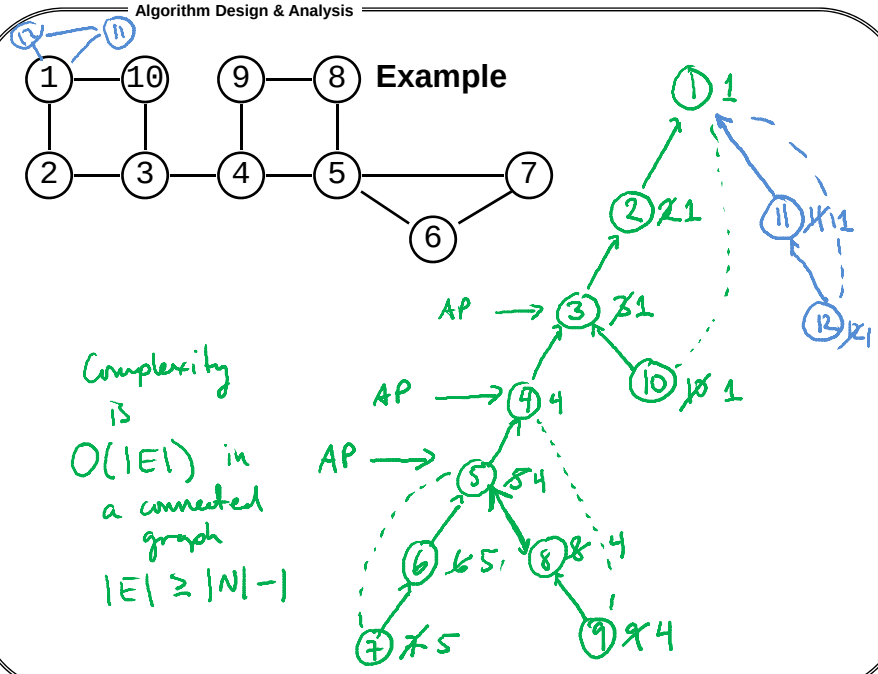

Algorithm Cont.

```
for each edge {v,w} do
  if w not marked then
    add {v,w} to T
    PARENT[w] ← v
    BI(w)

    if LOW[w] ≥ DFNum[v]
      and v not root then
      Print (v is an ~~also~~ articulation point)
    LOW[v] ← min(Low[v], Low[w])
  else if PARENT[v] ≠ w then
    LOW[v] ← min(LOW[v], DFNum[w])
```

Need a check at end if root is an articulation point.

Example



Complexity is $O(|E|)$ in a connected graph

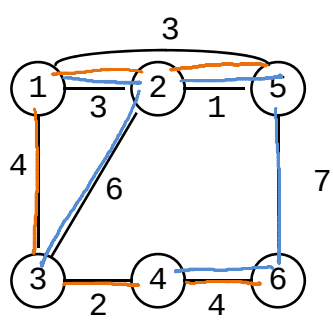
$|E| \geq |N| - 1$

Minimum Cost Spanning Tree

G = (N, E)

<u>Cost function</u> c : $E \to \mathbb{R}$ gives <u>edge weights</u>

A spanning tree is a graph S = (N, E') where E ⊇ E' and S is a tree



cost = 21

cost = 14

Cost of a Spanning Tree

Cost of a spanning tree $S = (N, E')$ is

$$\sum_{e \in E'} c(e)$$

Want minimum cost

<u>Lemma</u>: Let $S = (N, E')$ be a spanning tree for $G = (N, E)$. Then

a. for v_1, v_2 in N there is a unique simple path v_1 and v_2

b. adding an edge from $E - E'$ to S causes a unique simple cycle

Spanning Forest

A set of trees

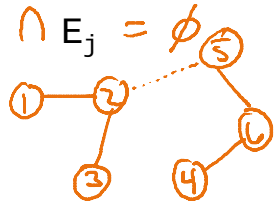
$\{(N_1, E_1), (N_2, E_2), \ldots, (N_k, E_k)\}$

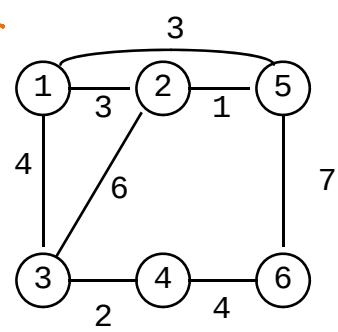
Where

$N = N_1 \dot\cup N_2 \dot\cup \cdots \dot\cup N_k$ $\quad N_k \cap N_\ell = \emptyset$

$E_i \subseteq E$

$E_i \cap E_j = \emptyset$





Expanding a Spanning Forest

Lemma: Suppose we have a spanning forest of $k > 1$ trees. Let $E' = E_1 \cup E_2 \cup \cdots \cup E_k$

Let $e = \{v,w\}$ be a lowest cost edge in $E - E'$ such that $v \in N_1$, $w \notin N_1$.

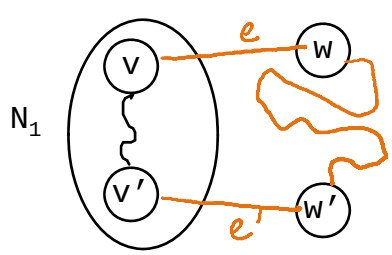
Then there is a spanning tree for G that includes $E' \cup \{e\}$ and has minimal cost among trees that include E'.

What kind of algorithm is this leading us towards?

Proof of Lemma

Suppose not. Let spanning tree $S'' = (N, E'')$, $E' \subseteq E''$, $e \notin E''$ have lower cost than any spanning tree that contains E' and e.

Adding e to S'' causes a cycle



$c(e) \leq c(e')$

$e' \in E''$

Proof of Lemma 2

S″ must have an edge e′ = {v′,w′}
with $v' \in N_1$ $w' \notin N_1$

We know c(e) ≤ c(e′)
Let S = S″ with e′ removed and e added
S has no cycles
e created a cycle, removing e′ broke it
S is still a tree

S costs no more than S″

Greedy Strategy

Take least cost edge e_1 in E. By last lemma,
there is some spanning tree that contains e_1
[Consider an initial spanning forest
$\{(\{v_1\},\varnothing), (\{v_2\},\varnothing), \ldots, (\{v_n\},\varnothing)\}$]

Iterative Step

Let

$\{(N_1,E_1),(N_2,E_2),\ldots,(N_k,E_k)\}$

be a spanning forest that is a subset of a MST

Add the lowest cost edge that connects two trees in the forest

By the last lemma, the result is a spanning forest that is a subset of a MST

Kruskal's Algorithm

```
KRUSKAL(G(N,E))
   S ⇐ ∅          Spanning forest edges
   NS ⇐ {{v1},{v2},...,{vn}}
   Q ⇐ a priority queue of edges m∈E by increasing weight
   while |NS| > 1 do
      {v,w} ← EXMIN(Q)
      if Find(v) ≠ Find(w) then
         S ⇐ S ∪ {{v,w}}
         Union( FIND(v) , FIND(w) )
```

Node sets of the trees in the forest

Example

Q	c	NS					
✓{2,5}	1	{1}	{2}	{3}	{4}	{5}	{6}
✓{3,4}	2	{1}	{2,5}	{3}	{4}	{6}	
✓{1,2}	3	{1}	{2,5}	{3,4}	{6}		
✗{1,5}	3	{1,2,5}	{3,4}	{6}			
✓{1,3}	4	{1,2,3,4,5},	{6}				
✓{4,6} STOP	4	{1,2,3,4,5,6}					
{2,3}	6						
{5,6}	7						

Complexity

Priority Queue: sort, make a list $O(|E| \lg |E|)$

Union/Find

Union's = $n-1$ $n = |N|$

Find's = $2|E|$ at most $O(|E| \cdot G(|E|))$

Can do a little better—

keep a heap of edges by cost

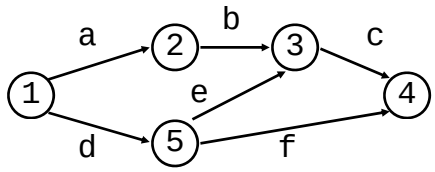
$O(d \lg |E|)$

the number of edges we consider before we find a free

Path Problems

Directed graphs with labels on edges

1 ⇝ 4 $(a \cdot b \cdot c) + (d \cdot e \cdot c) + (d \cdot f)$



Shortest Path

Labels could be non-negative integers

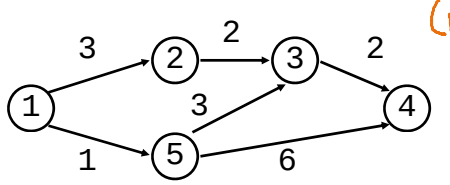
Use for finding paths of minimum distance

- $\cdot$ as addition
- $+$ as min

$(3+2+2)$ min

$(1+3+2)$ min

$(1+6)$



Connectivity

Labels could be Boolean values

Use for reachability

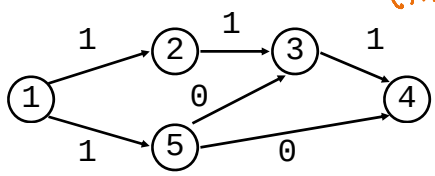
• is and

+ as or

$(1 \wedge 1 \wedge 1) \vee$

$(1 \wedge 0 \wedge 1) \vee$

$(1 \wedge 0)$

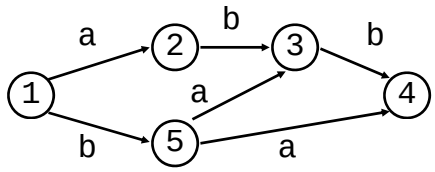


Finite Automata

Labels could be symbols from an alphabet

(as in a finite automaton). Use for

finding strings that take you from one node to another.



$(abb) \cup (bab) \cup (ba)$

• concat

+ union

~~[illegible]~~ will need * in general

Path Algorithms

Can produce generic algorithms, just need definitions for $\cdot$, $+$

Can take advantage of identities $1 \vee \text{anything} = 1$

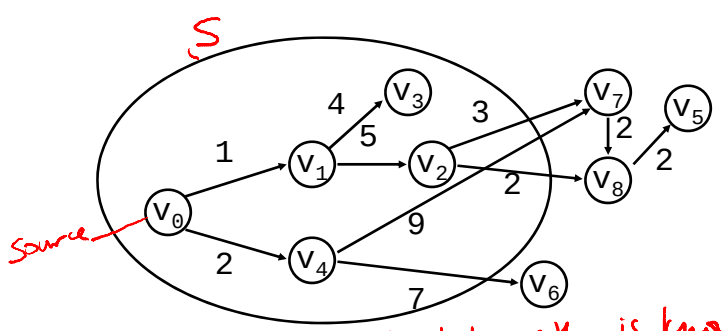
Two kinds of algorithms

1. Single source from one node v to all the rest
2. All pairs: from v to w for every pair v, w.

Can do 2. with multiple calls to 1., but there might be cheaper ways

Dijkstra's Algorithm

Single-source, shortest path in a digraph



S— Nodes for which shortest path is known

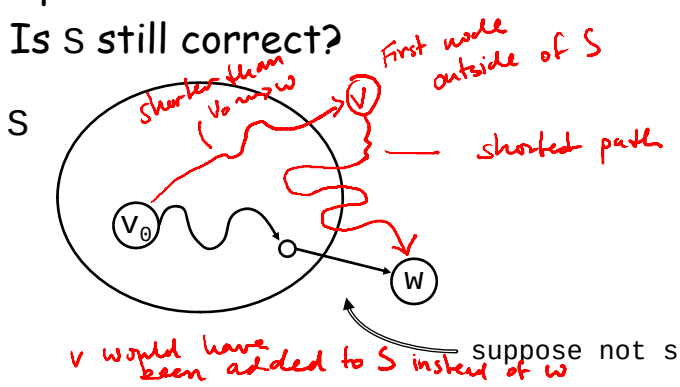
D—array of distances to nodes from v_0 only passing through nodes in S

General Step

Add node w not in S with minimum D value to S.

Update distances to nodes in N-S using w

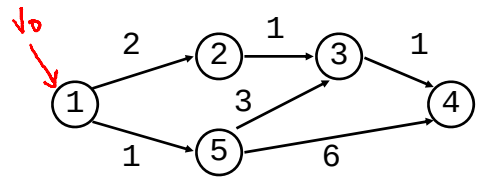
Is S still correct?



Full Algorithm

```
G = (N,E)
lab(v,w) - label of v->w,   set to +∞ if there is no edge v->w
   represent with matrix
S ← {v0}
D[v0] ← 0
D[v] ← lab(v0,v) for rest
while S ≠ N do
   choose w in N-S with minimum D[w] value
   S ← S ∪ {w}
   for all v in N-S do
      D[v] ← min(D[v], D[w]+lab(w,v))
```

Example



maintain an array D

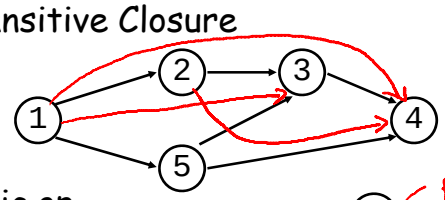
$O(|N|^2)$

S	D[1]	D[2]	D[3]	D[4]	D[5]
{1}	0	2	∞	∞	1
{1,5}		2	4	7	1
{1,2,5}		2	3	7	
{1,2,3,5}			3	4	
{1,2,3,4,5}				4	
	0	2	3	4	1

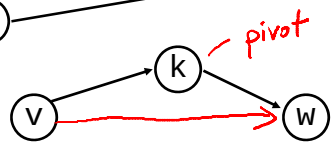
Warshall's Algorithm

All pairs reachability

Transitive Closure

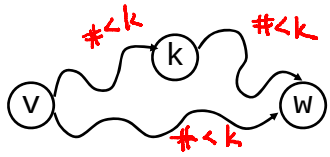


Basic op



Consider each k only once

Paths that go through nodes numbered < k



is there a path with nodes numbered ≤ k

Algorithm

|N|

```
for k = 1 to n do
  for v = 1 to n do
    if a[v,k] then
      for w = 1 to n do
        if a[k,w] then a[v,w] ← 1
```

a[ ] adjacency matrix initially

a[i,j] = 1 means i → j exists

$O(n^3)$

a[i,j]

Example

$$
\begin{array}{c|ccccc}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 1 & 1 & \cancel{0}\,1 & \cancel{0}\,1 & 1 \\
2 & 0 & 1 & 1 & \cancel{0}\,1 & 0 \\
3 & 0 & 0 & 1 & 1 & 0 \\
4 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 0 & 1 & 1 & 1
\end{array}
$$

useful to set diagonal to 1

pivot on 1 no new paths

pivot on 2 1→2 get 1→3

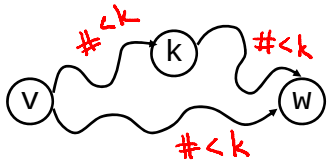
pivot on 3 1→3, 2→3, 5→3 get 1→4, 2→4

pivot on 4 no new paths

pivot on 5 1→5 no new paths

Floyd's Algorithm

All-pairs shortest path

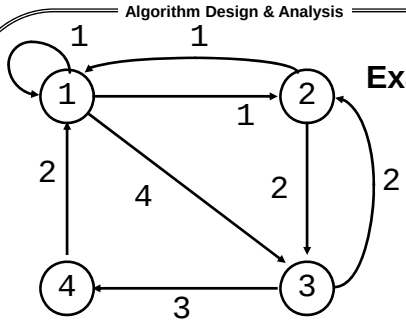


shortest distance through nodes numbered < k

```
for k = 1 to n do
  for v = 1 to n do
    if d[v,k] ≠ ∞ then
      for w = 1 to n do
        d[v,w] ← min(d[v,w], d[v,k]+d[k,w])
```

matrix d initially is the edge labels or ∞ if the edge doesn't exist.

Example



iteration of outer loop

$d[i,j]^0$

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 4 & \infty \\ 2 & 1 & 0 & 2 & \infty \\ 3 & \infty & 2 & 0 & 3 \\ 4 & 2 & \cancel{\infty}\ 3 & \cancel{\infty}\ 6 & 0 \end{array}$$

Example 2

$$d[i,j]^0 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 4 & \infty \\ 1 & 0 & 2 & \infty \\ \infty & 2 & 0 & 3 \\ 2 & \infty & \infty & 0 \end{pmatrix} \end{array}$$

$$d[i,j]^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 4 & \infty \\ 1 & 0 & 2 & \infty \\ \infty & 2 & 0 & 3 \\ 2 & 3 & 6 & 0 \end{pmatrix} \end{array}$$

Example 3

$$d[i,j]^1 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 4 & \infty \\ 1 & 0 & 2 & \infty \\ \infty & 2 & 0 & 3 \\ 2 & 3 & 6 & 0 \end{pmatrix} \end{array}$$

$$d[i,j]^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & \underline{3} & \infty \\ 1 & 0 & 2 & \infty \\ 3 & \underline{2} & 0 & 3 \\ 2 & 3 & \underline{5} & 0 \end{pmatrix} \end{array}$$

Example 3

$$d[i,j]^2 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 3 & \infty \\ 1 & 0 & 2 & \infty \\ 3 & 2 & 0 & 3 \\ 2 & 3 & 5 & 0 \end{pmatrix} \end{array}$$

$$d[i,j]^3 = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 3 & 6 \\ 1 & 0 & 2 & 5 \\ 3 & 2 & 0 & 3 \\ 2 & 3 & 5 & 0 \end{pmatrix} \end{array}$$

$$\| \; d[i,j]^4$$