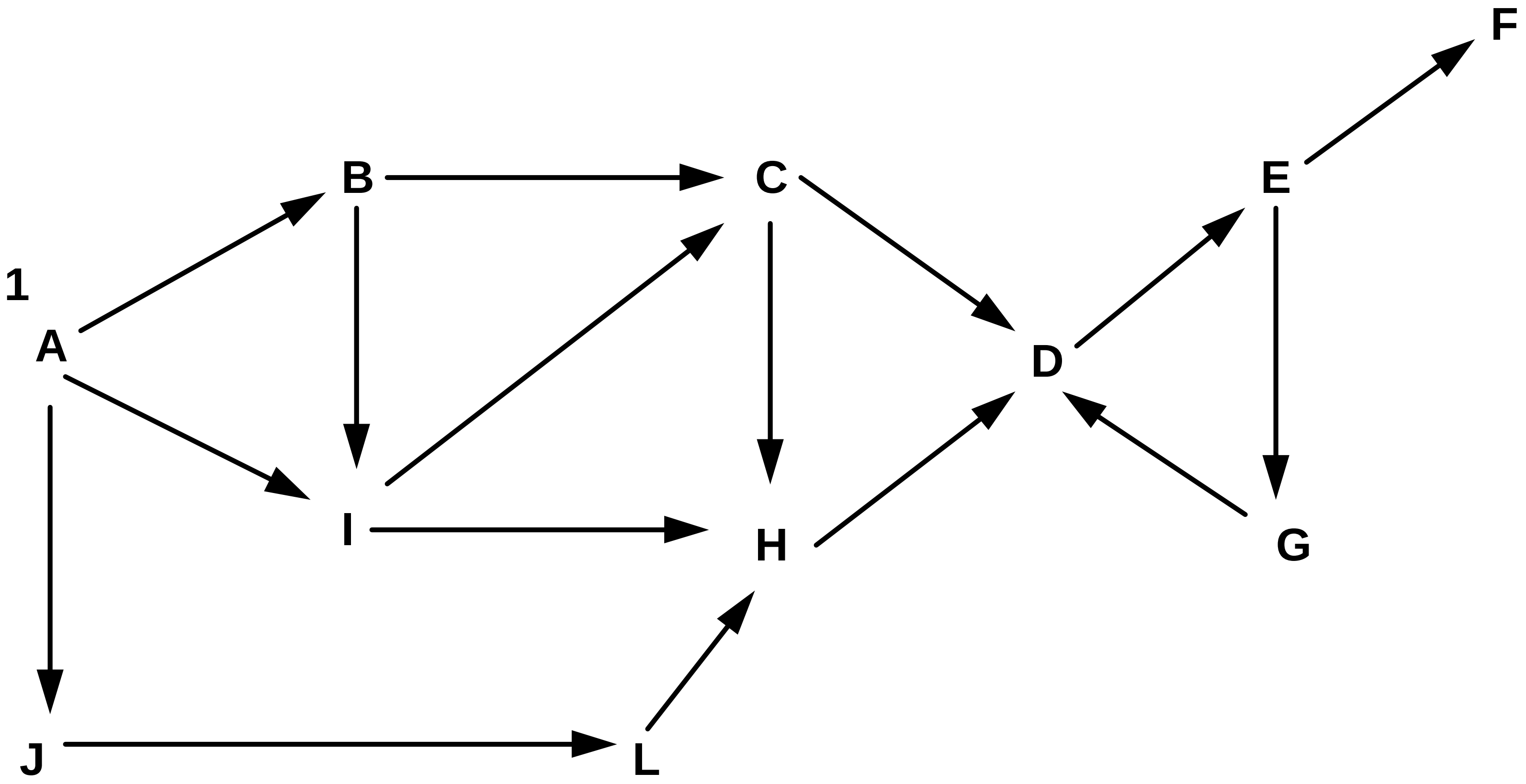
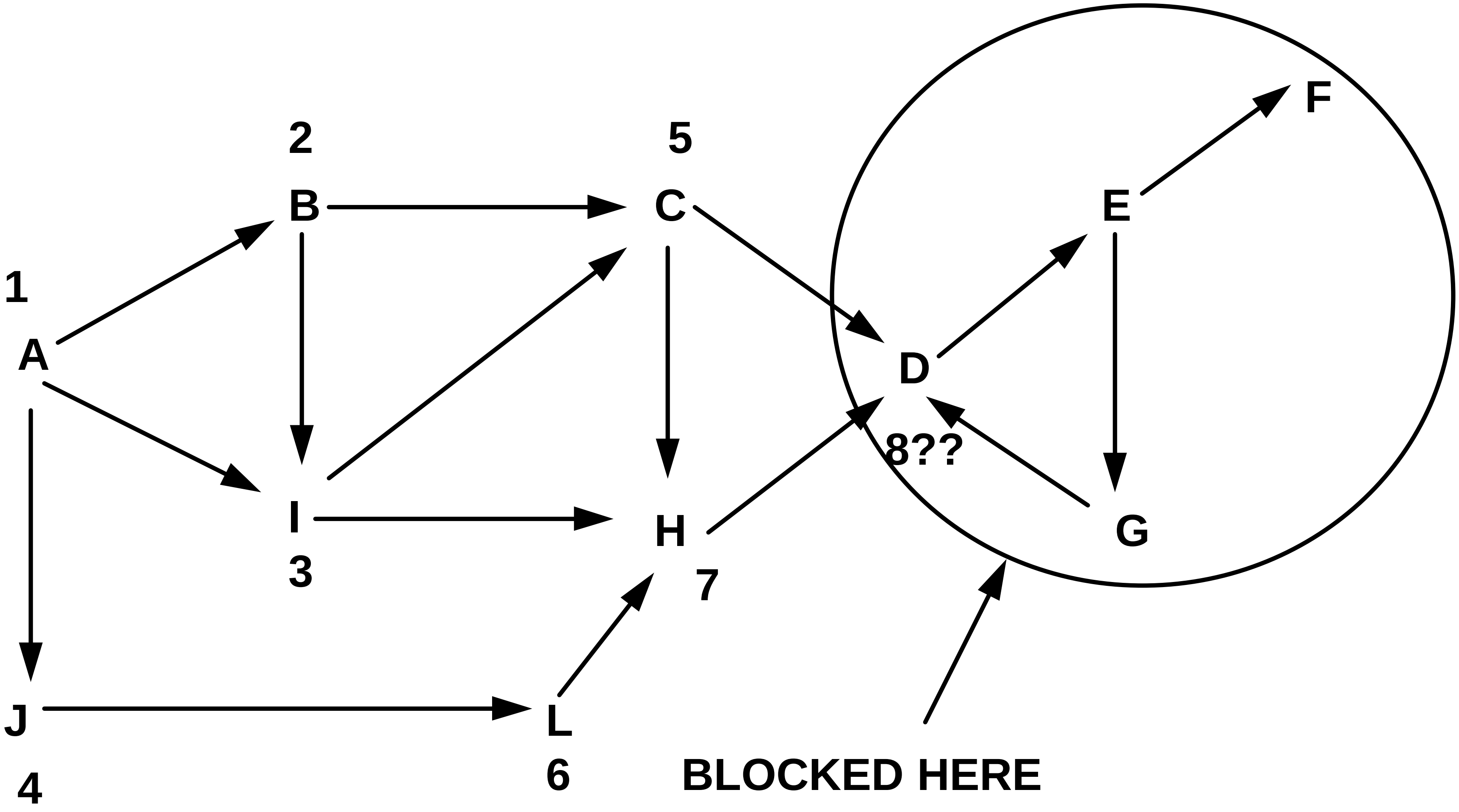
Lecture 1 - Summary

Terms: graph, vertex, edge, degree, degree sequence,
realizability, path, component, connected,
isomorphism, planarity, homeomorphism,subgraph,
induced subgraph, bipartite, complete graph,
Peterson's graph, cubic graphs, Planarity Charact:
(PL iff no subgraph homeomorphic to K5 or K3,3);
Bipartite Charcat. (Bipartite iff No odd cycles)

TOPOLOGICAL NUMBERING



TOPOLOGICAL NUMBERING



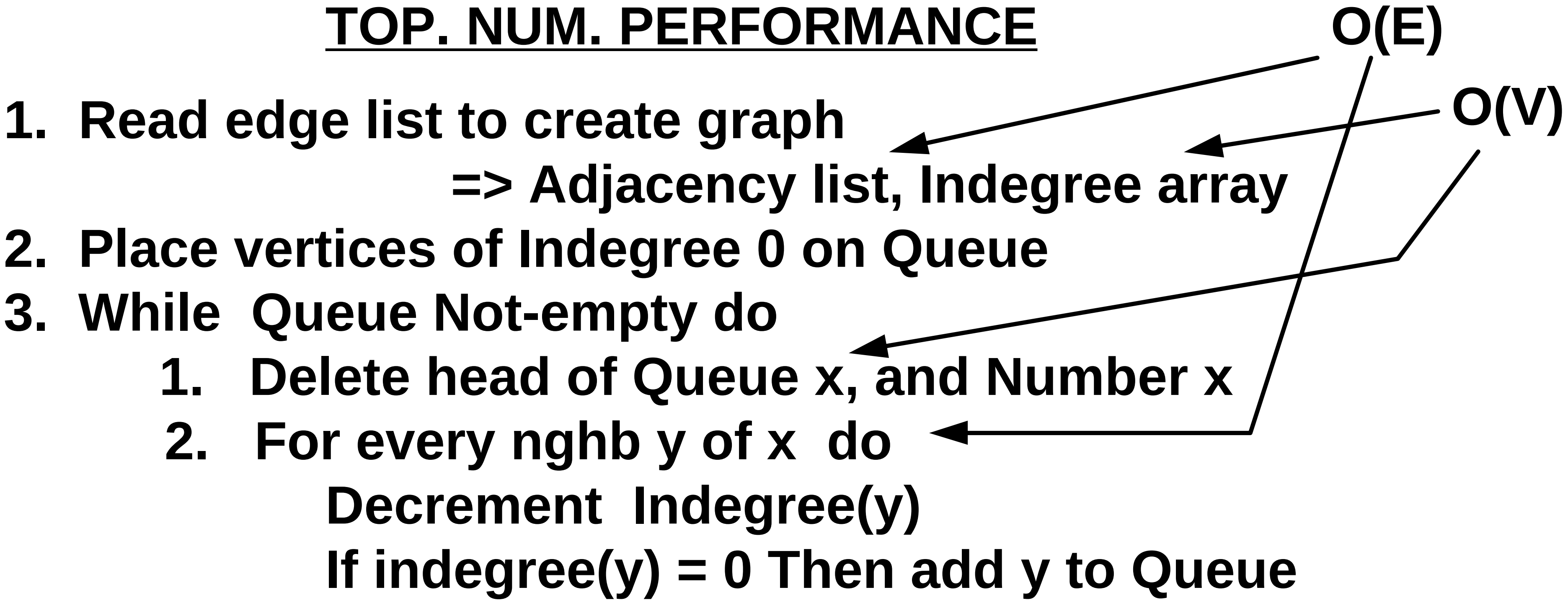
BLOCKED HERE
BECAUSE
REMAINING VERTICES ARE
EITHER ON OR ARE REACHABLE
FROM A CYCLE

TOP. NUMBERING ALGORITHM

```
1. Read edge list to create graph
                    => Adjacency list, Indegree array
2. Place vertices of Indegree 0 on Queue
3. While Queue Not-empty do
      1. Delete head of Queue x, and Number x
      2. For every nghb y of x do
            Decrement Indegree(y)
            If indegree(y) = 0 Then add y to Queue
```

On Termination:

```
  If all vertices are numbered
        Then Graph is acyclic & top. numbered
  If any vertices are leftover
        Then Graph is not Acyclic and the
        leftover vertices are exactly those
        reachable from cycles
```



TOP. NUM. PERFORMANCE

O(E)

O(V)

```
1.  Read edge list to create graph
                        => Adjacency list, Indegree array
2.  Place vertices of Indegree 0 on Queue
3.  While  Queue Not-empty do
        1.   Delete head of Queue x, and Number x
        2.   For every nghb y of x  do
                Decrement  Indegree(y)
                If indegree(y) = 0 Then add y to Queue
```

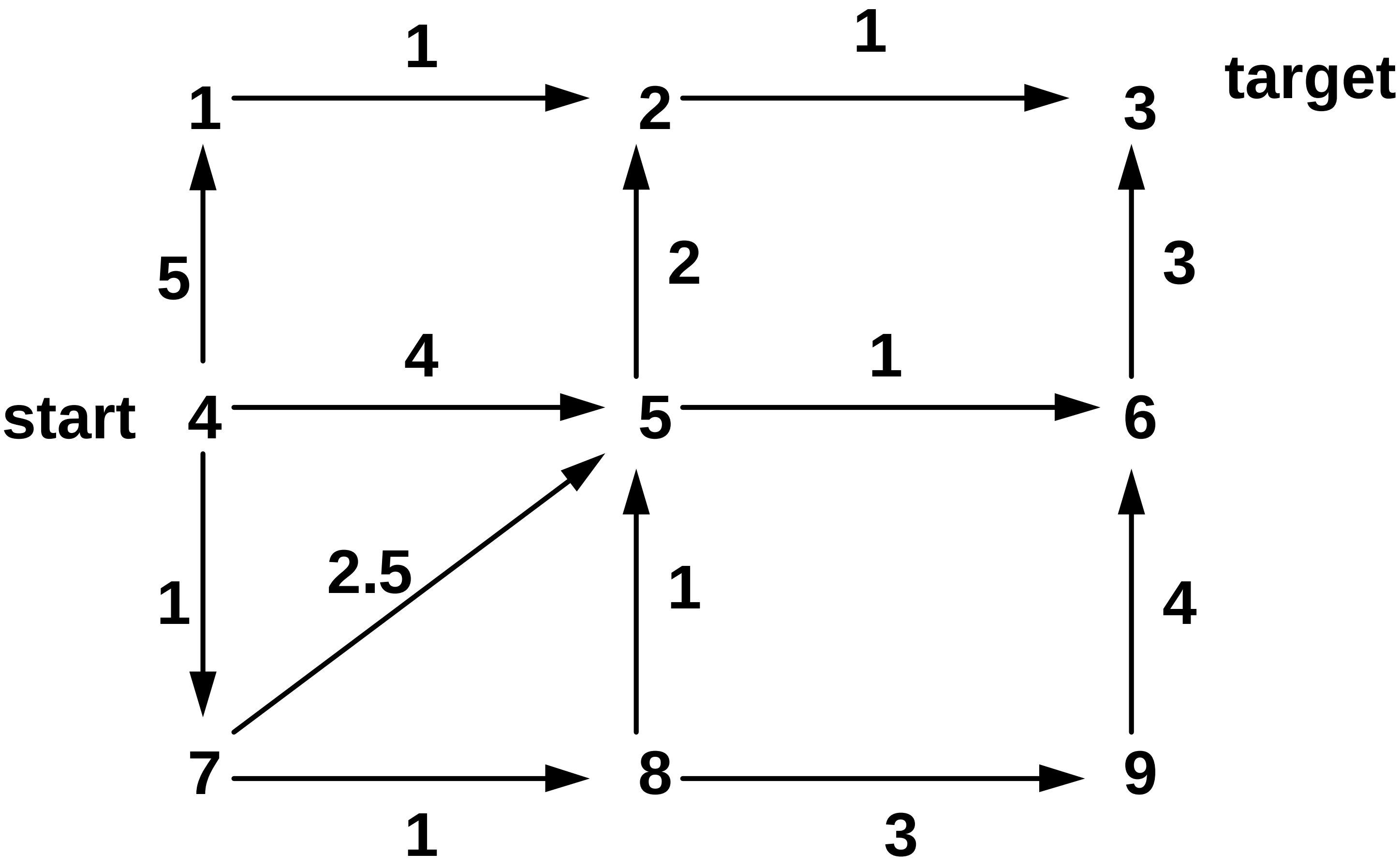
The basic actions are those repeated for neighbors y of each x - this occurs at most O(E) times - even less if the digraph is not acyclic. Thus overall cost is: O(E)

MORE PRECISELY: O(max(V,E))

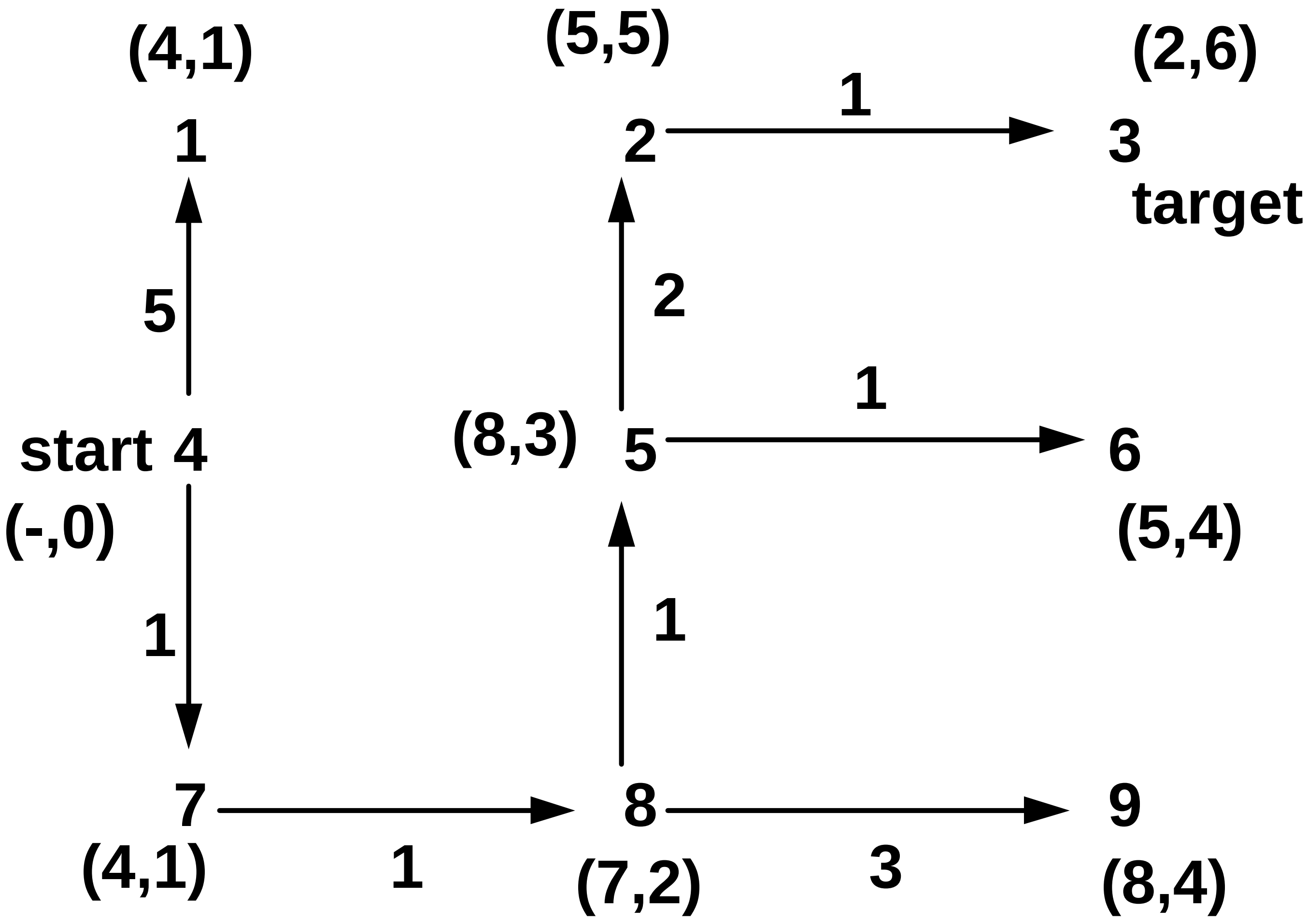
Algorithm Trace

indeg			
a	0		
b	~~1~~	0	
c	~~2~~	~~1~~	0
d	~~3~~	~~2~~	1
L	1		
f	1		
g	1		
h	~~3~~	~~2~~	1
i	~~2~~	1	
j	~~1~~	0	
L	~~1~~	0	

```
Queue
    |
    |  a
a   |
    | bj
b   | j
    | j  i
j   | i
    | i L
i   | L
    | Lc
L   | c
c   |
    | h
h   |
```



Positive weighted digraph for Dijkstra



Final Search Tree for Dijkstra Shortest Paths
(Predeccessor, Estimated dist) at each vertex

Dijkstra Shortest Path Algorithm

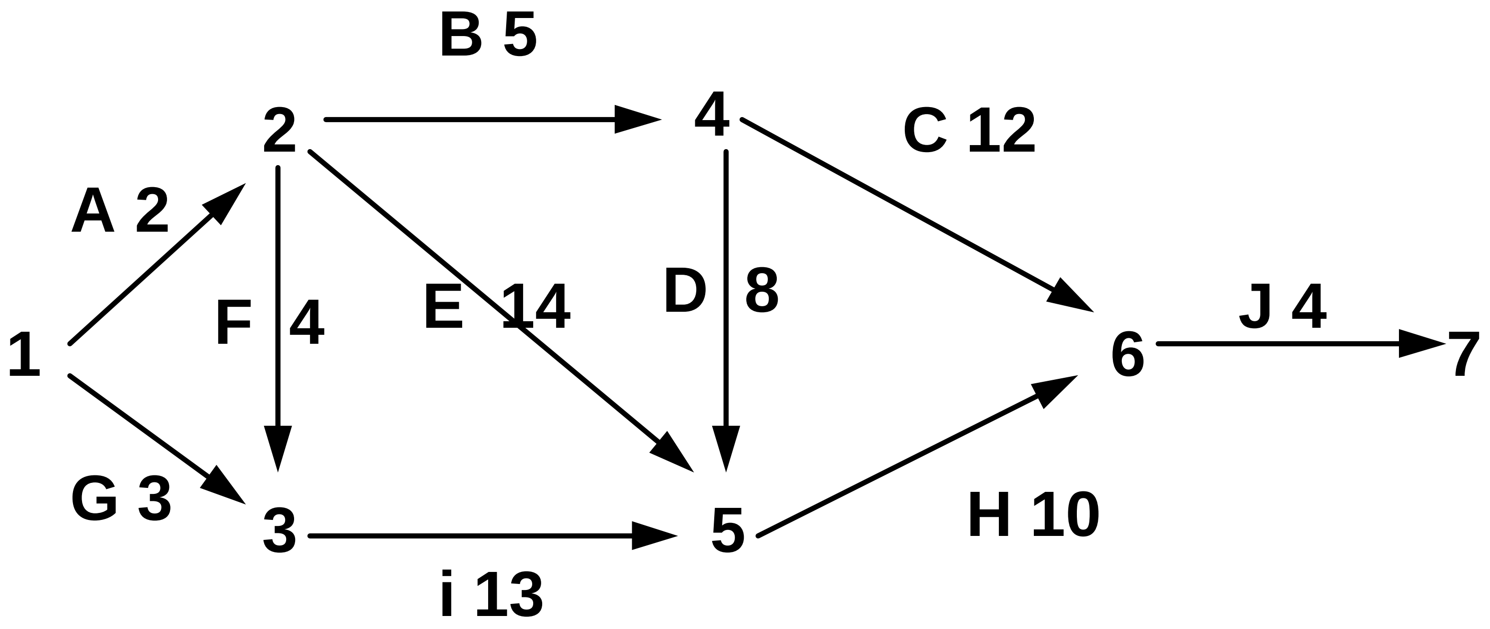
```
1.  Initialize Pred & Est: Start to (-,0), rest of vert. to (-, BIG)
    Move Start to R, rest of vert to U, and set S to empty

2.  While R non-empty do:
    1.  Move least vertex x in R to S
    2.  If x = target Then exit
    3.  Else For every nghb y of x do
            1: y in U:  Pred(y) <= x & Est(y) <= Est(x) + len(x,y)
                        Move y  to R
            2: y in R:   If Est(x)+len(x,y) < Est(y)
                         Then Pred(y) <= x & Est(y) <= Est(x)+len(x,y)
            3: y in S: no action - ignore
```

On completion: Either shortest path from start to target has been found (accessable by tracing Pred pointers back from target) or target vertex is unreachable from start.

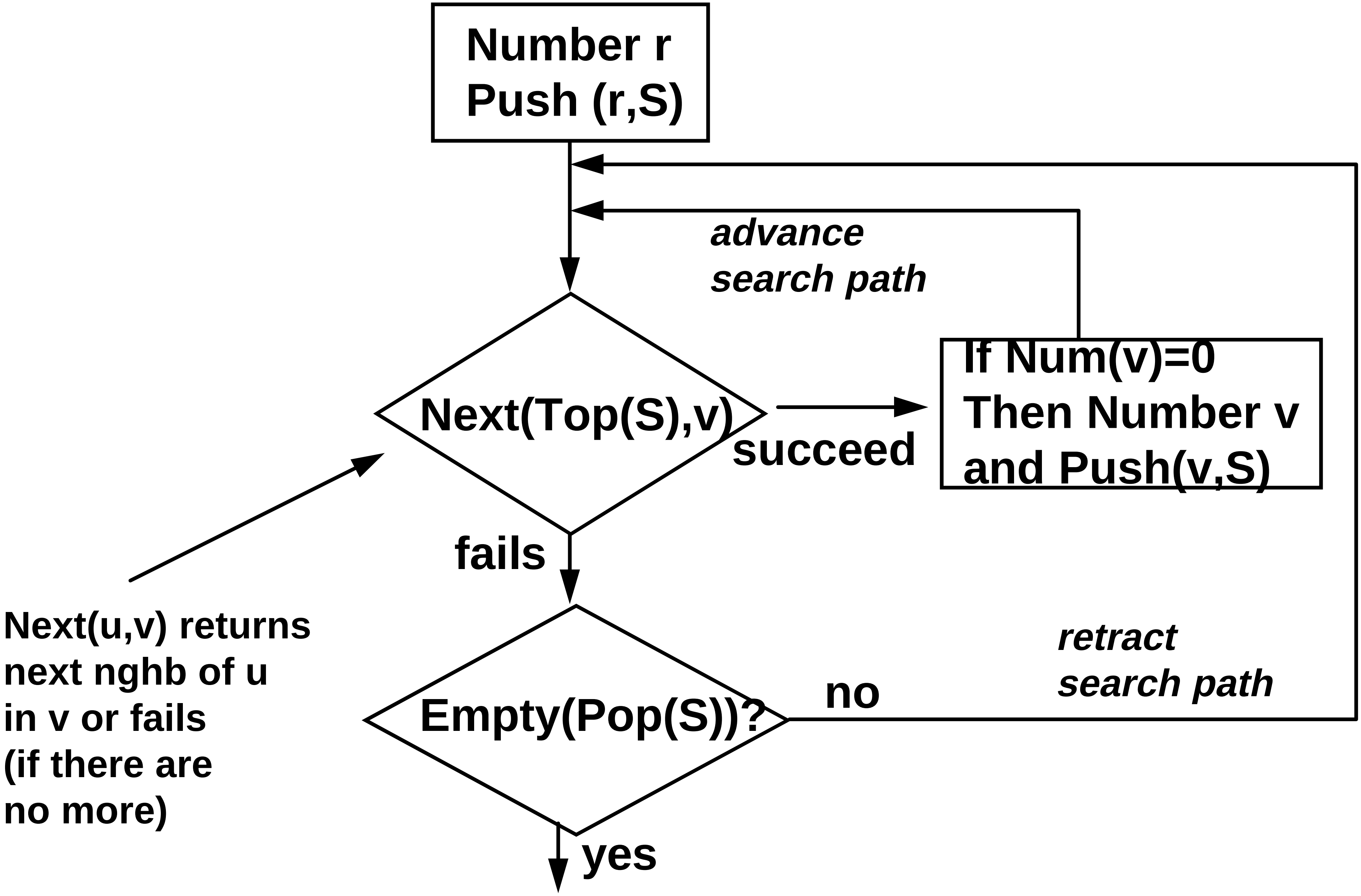
Longest paths in acyclic digraphs - Scheduling Interpretation

Precedence and duration of tasks digraph:



Shortest feasible schedule = longest path (time)

Depth First Search Algorithm - from r



Number r
Push (r,S)

Next(Top(S),v)

succeed

If Num(v)=0
Then Number v
and Push(v,S)

advance
search path

fails

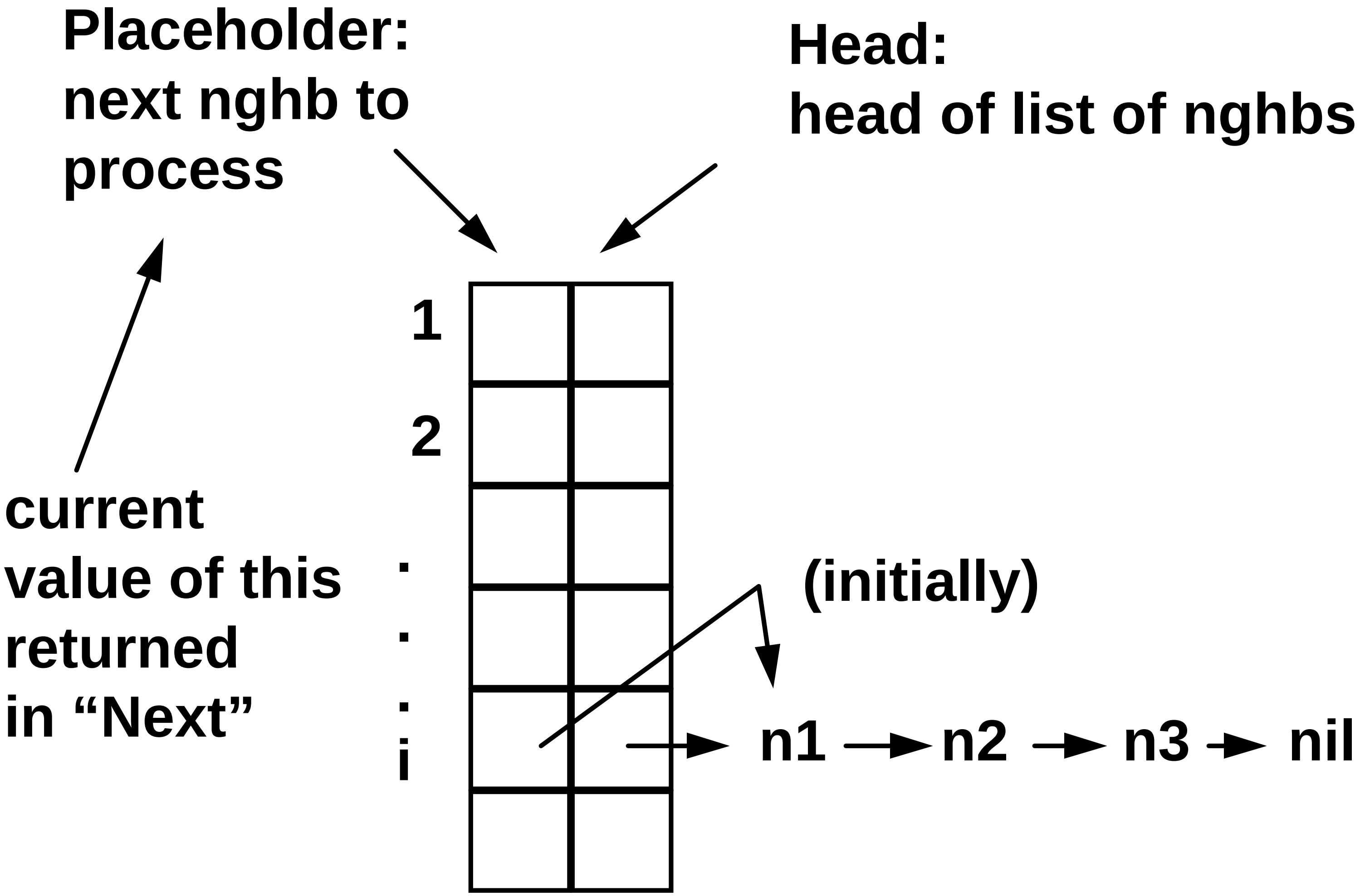
Empty(Pop(S))?

no

retract
search path

yes

Next(u,v) returns
next nghb of u
in v or fails
(if there are
no more)



Placeholder:
next nghb to
process

Head:
head of list of nghbs

current
value of this
returned
in “Next”

1

2

.

.

.

i

(initially)

n1 → n2 → n3 → nil

<u>**Data Structure for Depth First Search**</u>

(Array of adjacency lists - supplemented by pointer to next vertex to be processed in each list.)

Topologically Based Shortest/Longest Path Calc.

To calculate D(i,j):

```
1. For j < i: D(i,j) is “infinite”
2. For j = i: D(i,j) = 0
3. For j > i: D(i,j) requires:
       For m = i+1 to j do
           $D(i,m) = \max_{\text{all pred of } m}\{ D(i, pred(m)) + len(pred(m),m) \}$
```

D(i,j) calc. entails D(i,m) calc for m<j

pred(m) < m, so D(i,pred(m)) is defined by the time it is required for calculation of D(i,m) if calc. done sequentially

by top. num.

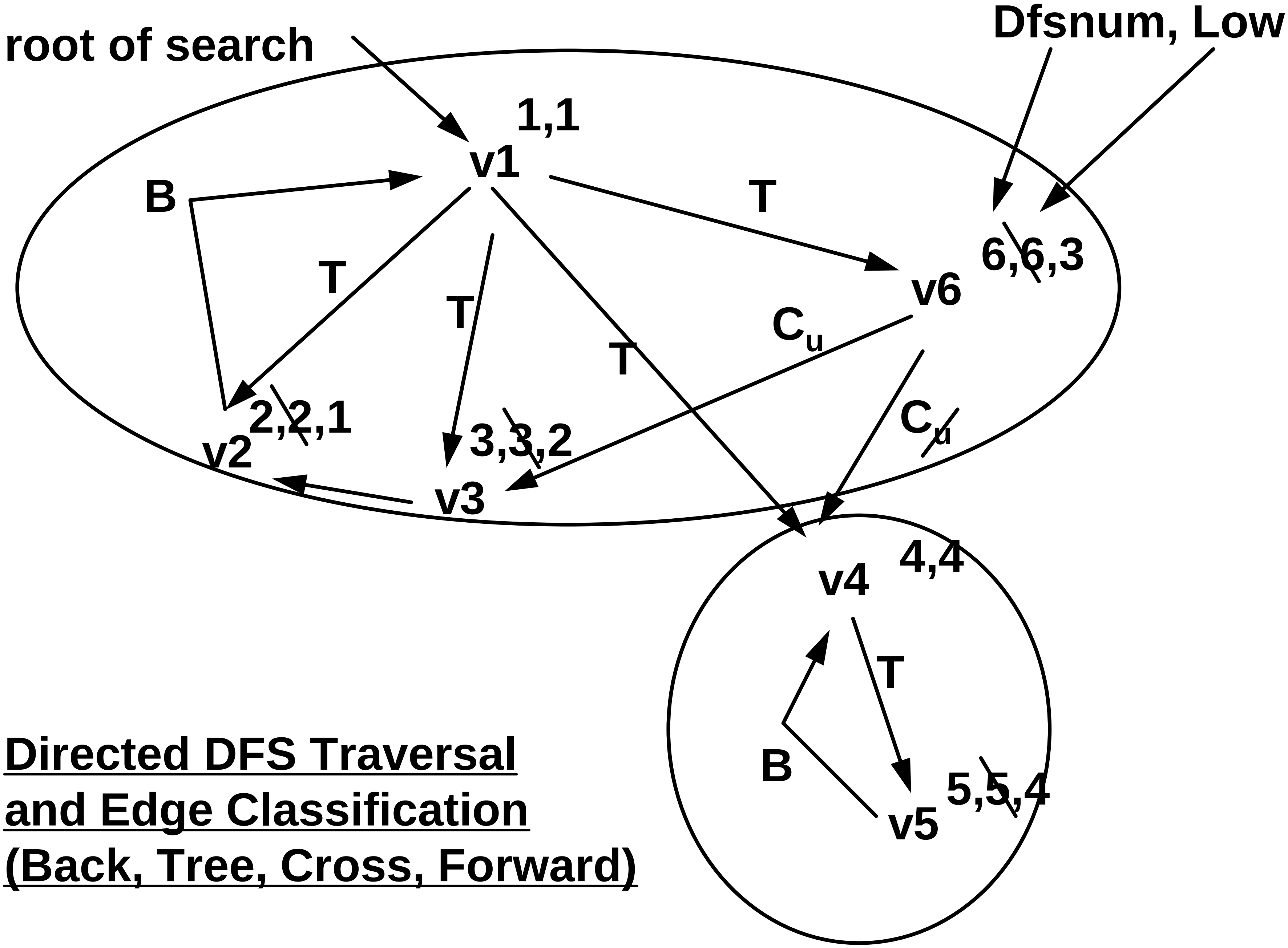
Eg: D(1,6)depends on D(1,4)&D(1,5) in task graph - both calculated by that point - earlier in the For loop.

Next (u,v)

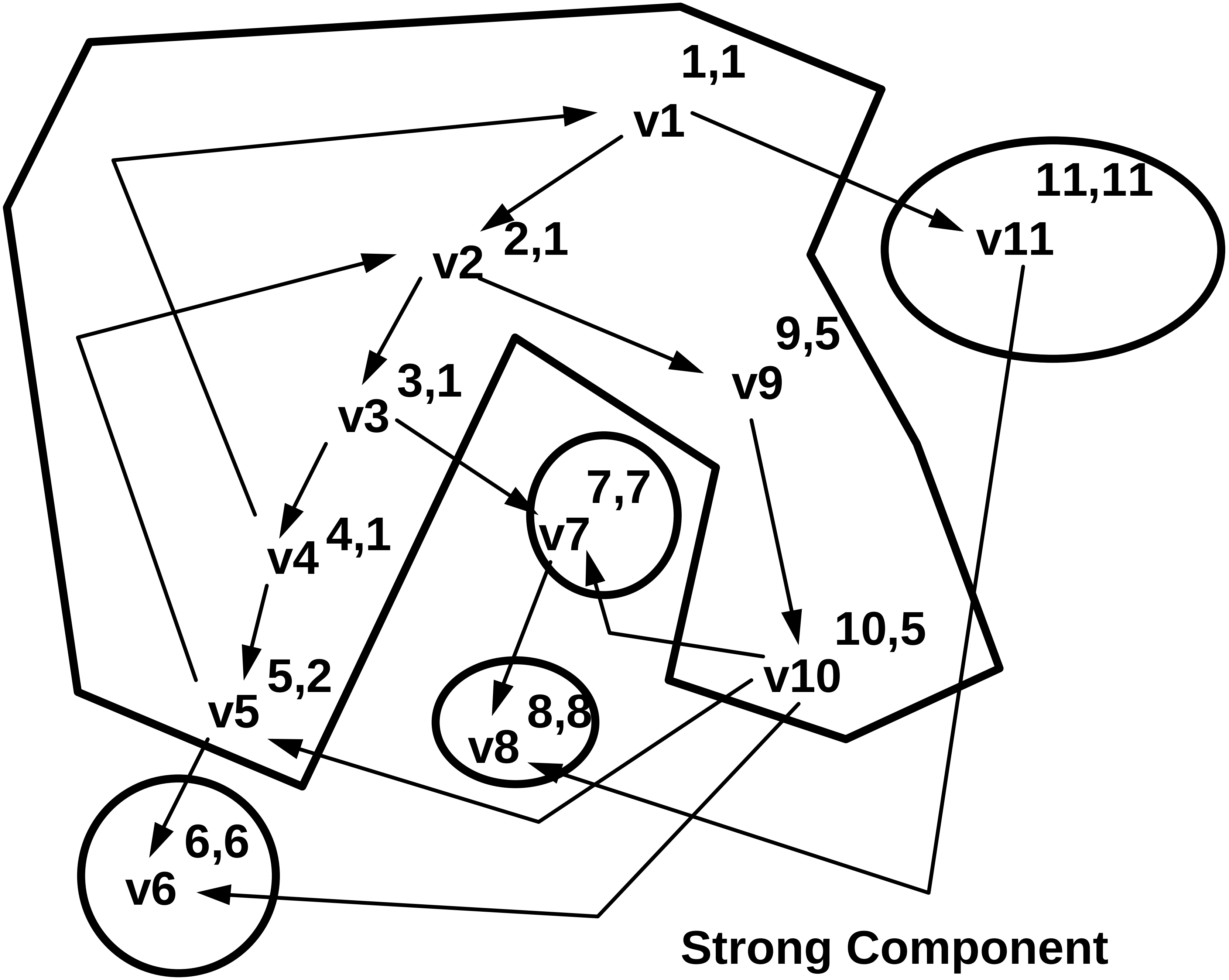
```
If Place(u) = nil

Then Next      <= false                        (*fails*)
        Place(u) <= Head(u)                    (*restart list*)

Else  Next      <= True                        (*succeeds*)
        v           <= Place(u)                (*return next nghb*)
        Place(u) <=Place(u)↑ .succ             (*advance Place*)
```

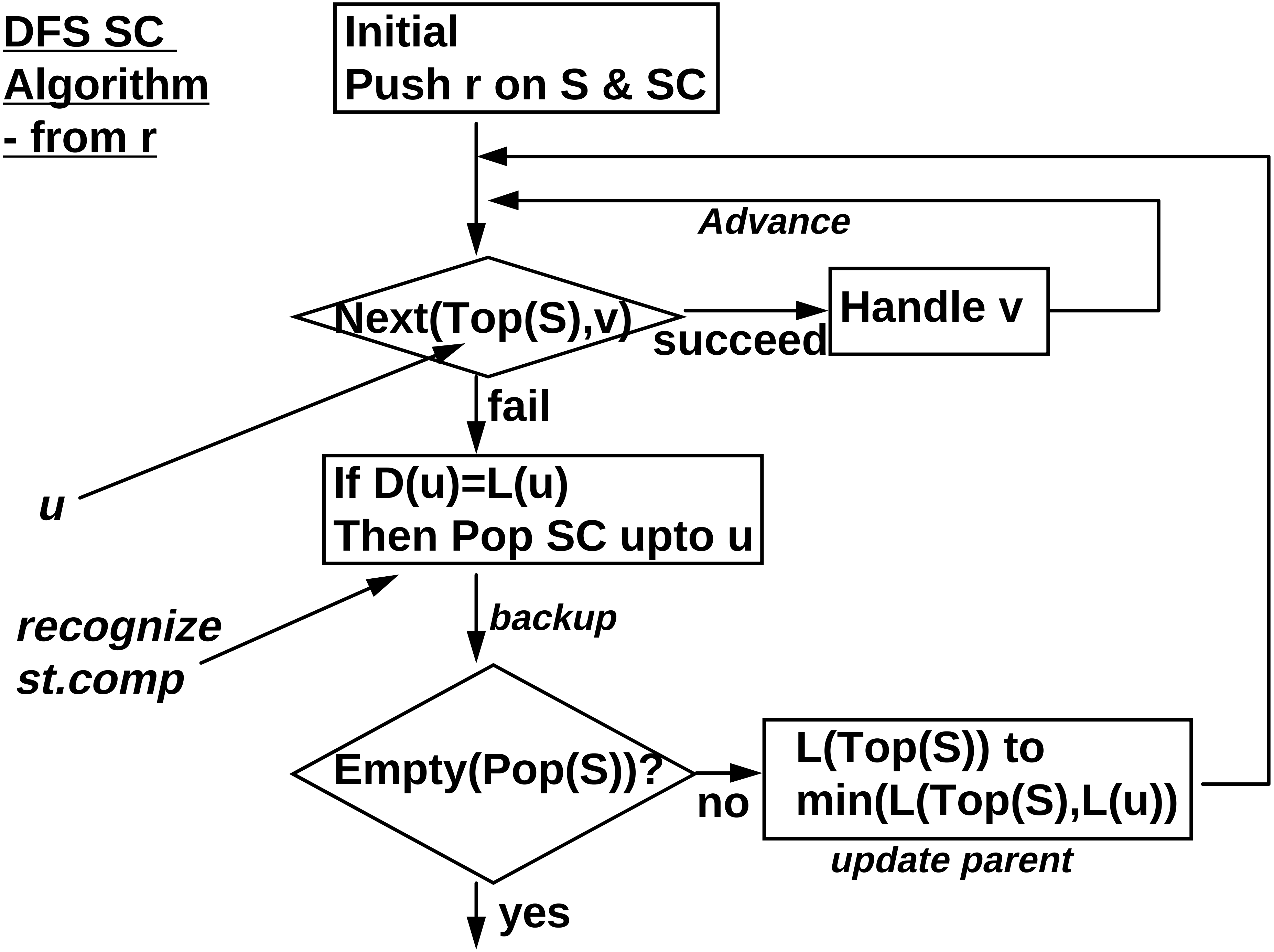


<u>Directed DFS Traversal</u>
<u>and Edge Classification</u>
<u>(Back, Tree, Cross, Forward)</u>



Strong Component
Recognition (from root v1)

DFS SC Algorithm - from r



Initial
Push r on S & SC

Advance

Next(Top(S),v)

succeed

Handle v

fail

u

If D(u)=L(u)
Then Pop SC upto u

backup

recognize st.comp

Empty(Pop(S))?

no

L(Top(S)) to
min(L(Top(S),L(u))

update parent

yes

Handle v from edge (u,v)

test

action

Tree: D(v) = 0

Number v, Push v on S and SC

Forward: D(v) > D(u)

No action

Back: D(v)<D(u) and v on S

L(u) <= min {L(v),L(u)}

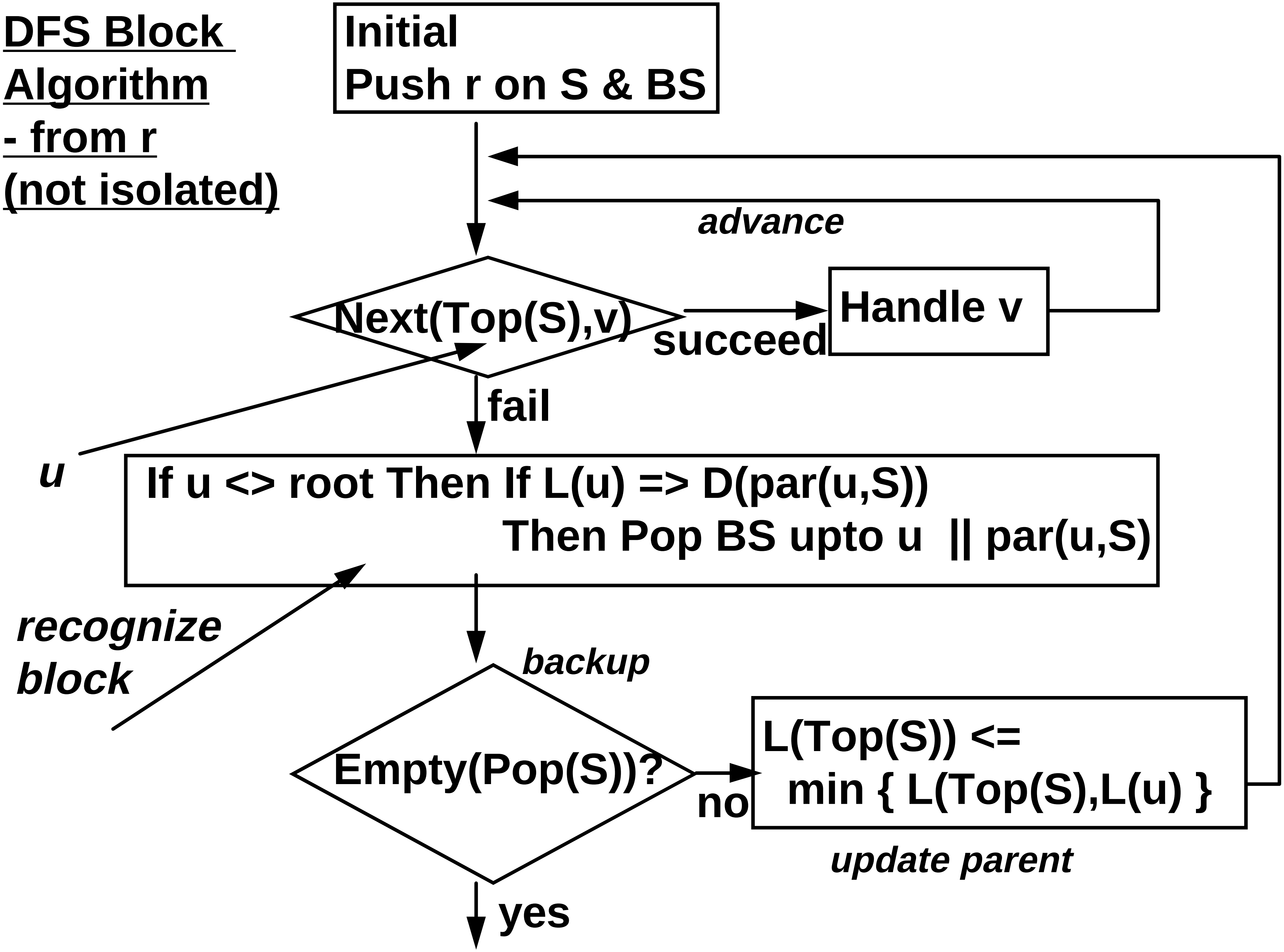
Cross: D(v)<D(u) and v not on S

If v on SC Then L(u) <= min {L(v),L(u)}

If v not on SC Then No action

can use: v on SC for both

DFS Block Algorithm - from r (not isolated)



Initial
Push r on S & BS

Next(Top(S),v)

Handle v

If u <> root Then If L(u) => D(par(u,S))
Then Pop BS upto u || par(u,S)

Empty(Pop(S))?

L(Top(S)) <=
min { L(Top(S),L(u) }

Handle v from edge (u,v)

test

action

Tree:

D(v) = 0

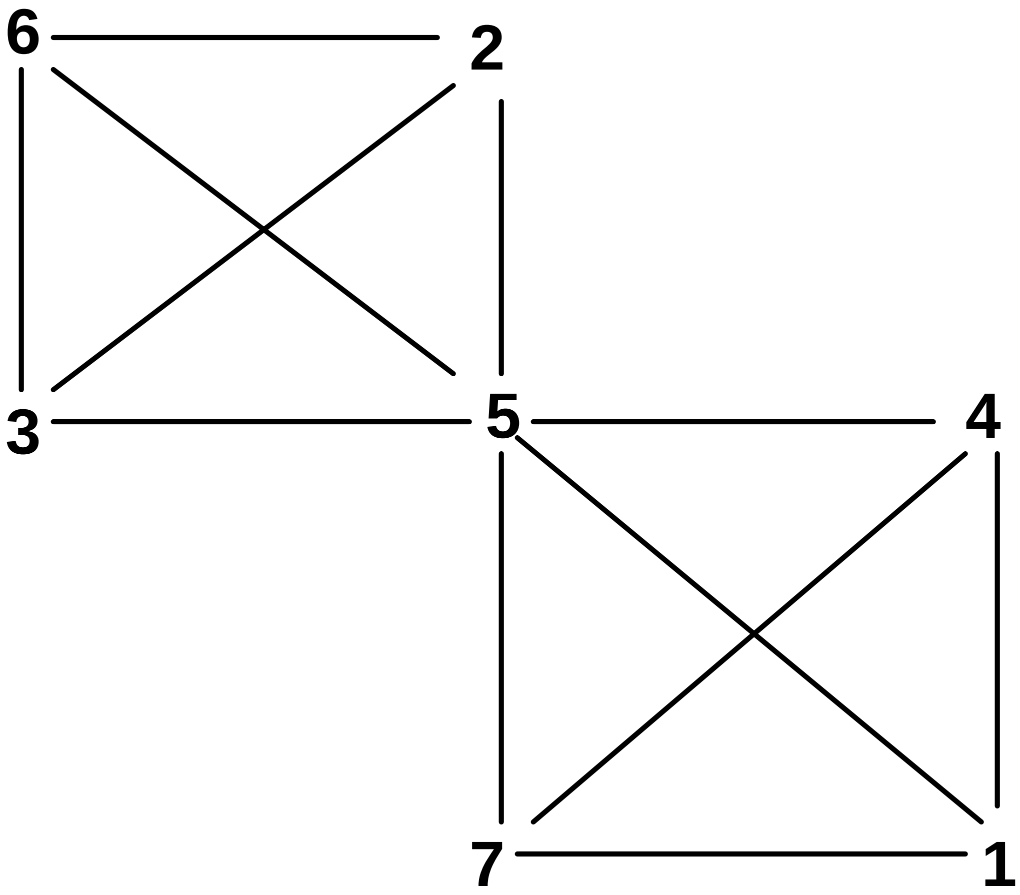
Number D(v) & L(v), Push v on S and BS

Back:

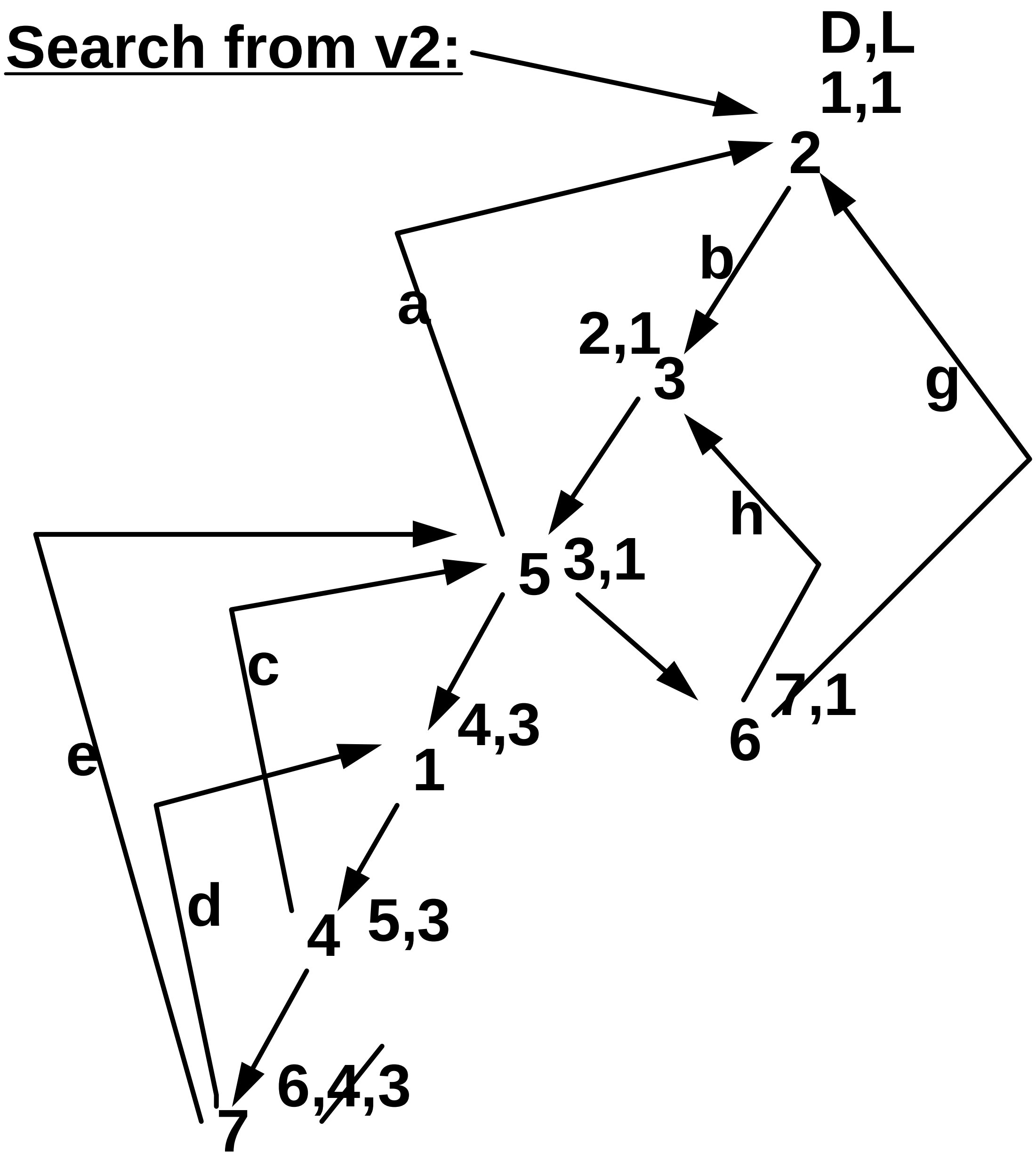
D(v) < D(u)

L(u) <= min { D(v),L(u)}

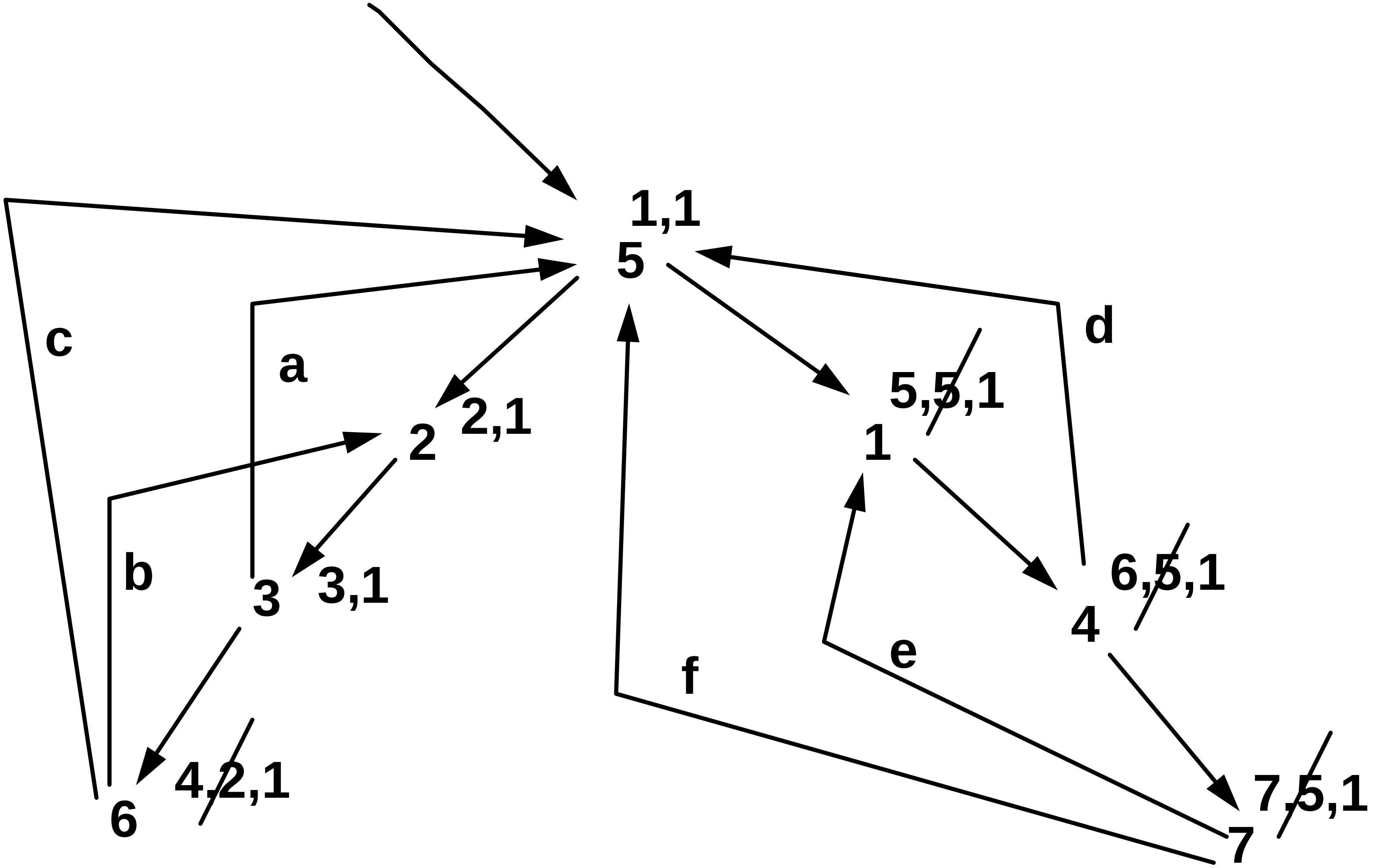
Block Example

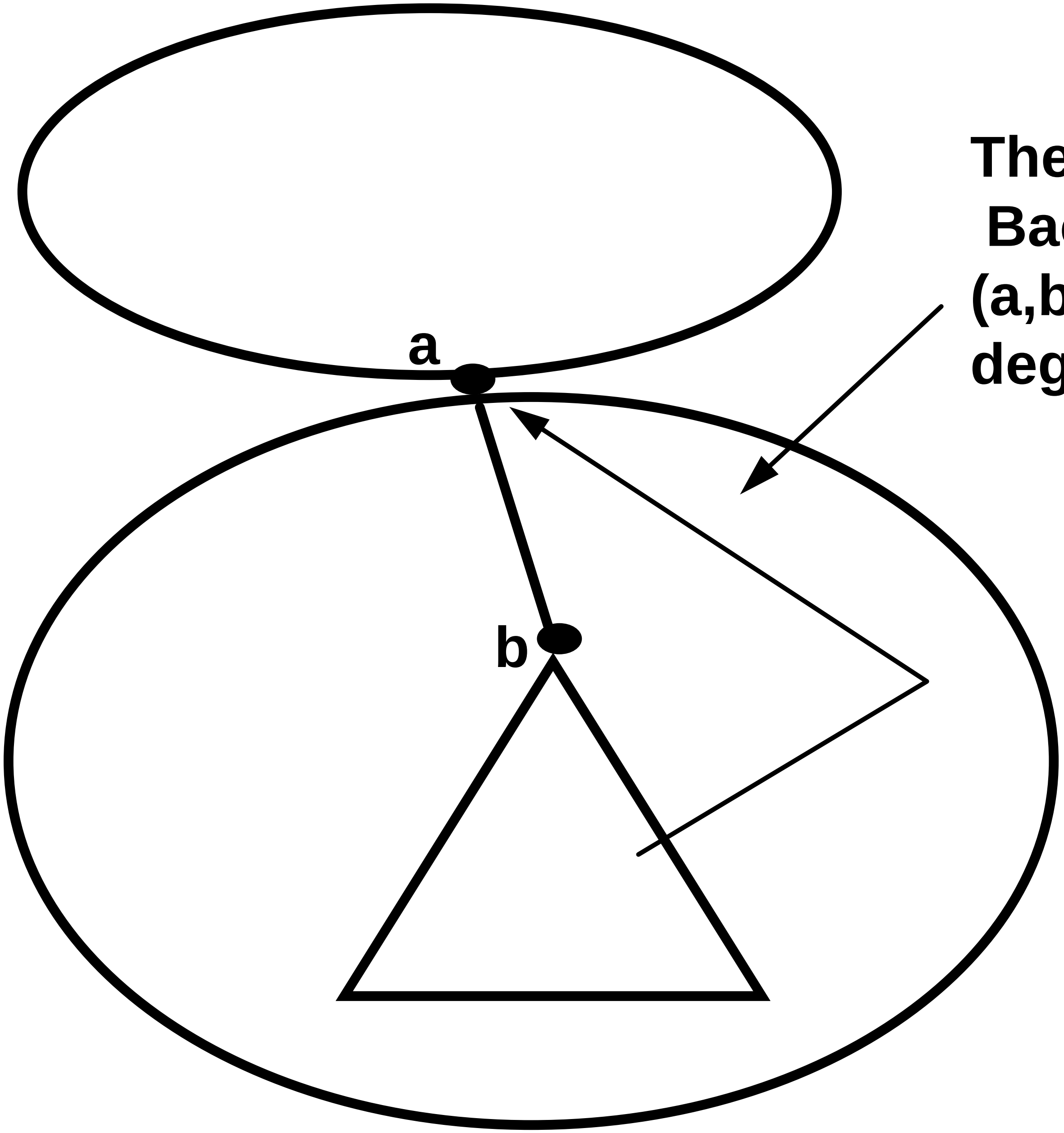


Search from v2:



Search from v5:

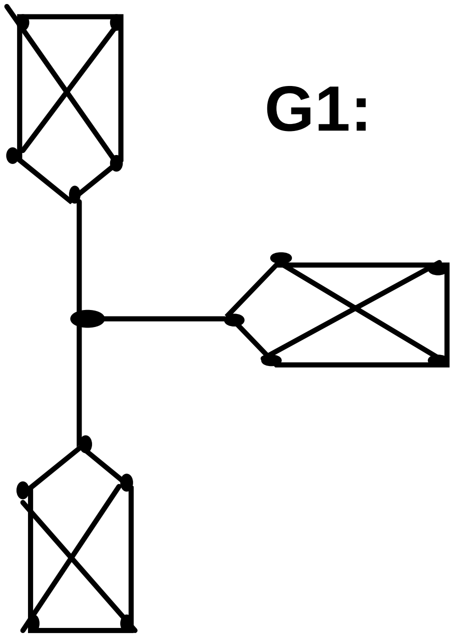


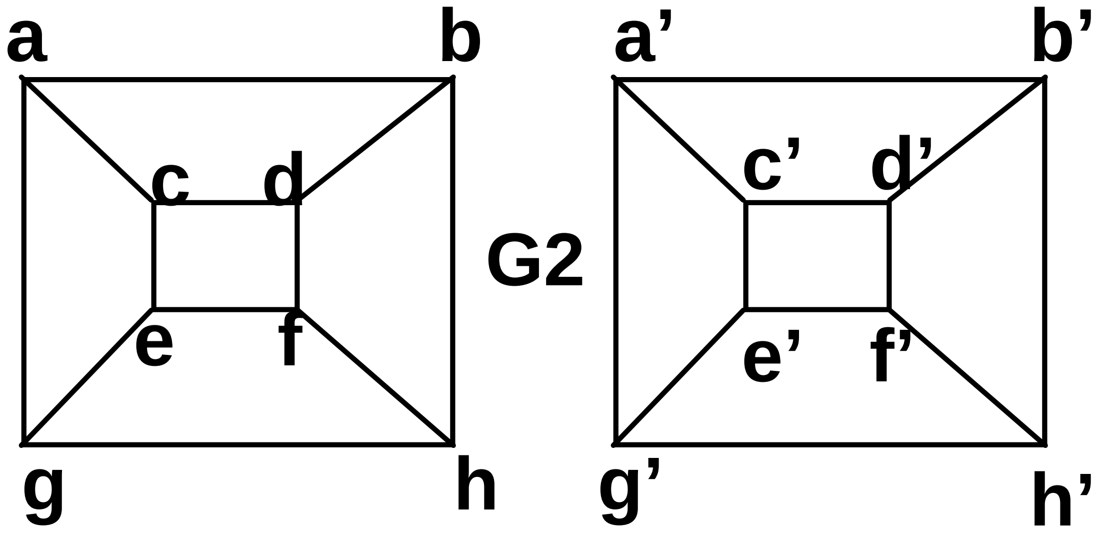


There exists such
 Back edges - unless
(a,b) itself is a
degenerate block.

Preview Questions for Exam 1:

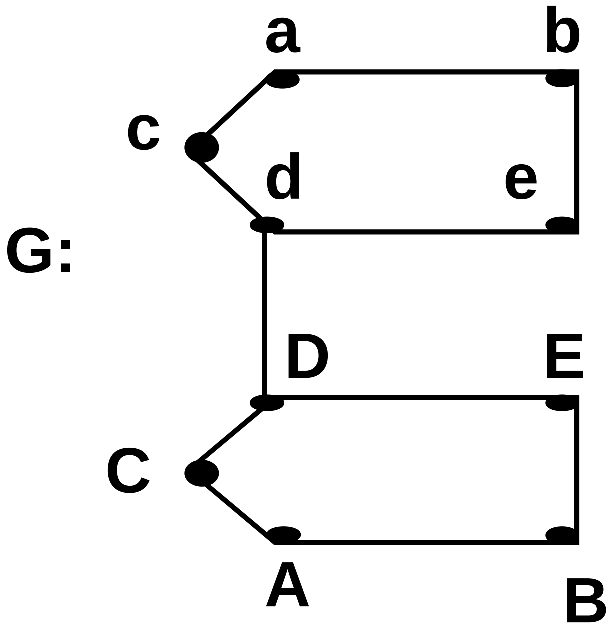
(1) Give 4 named graphical invariants G1 and G2 below agree on, and 4 they do not agree on.



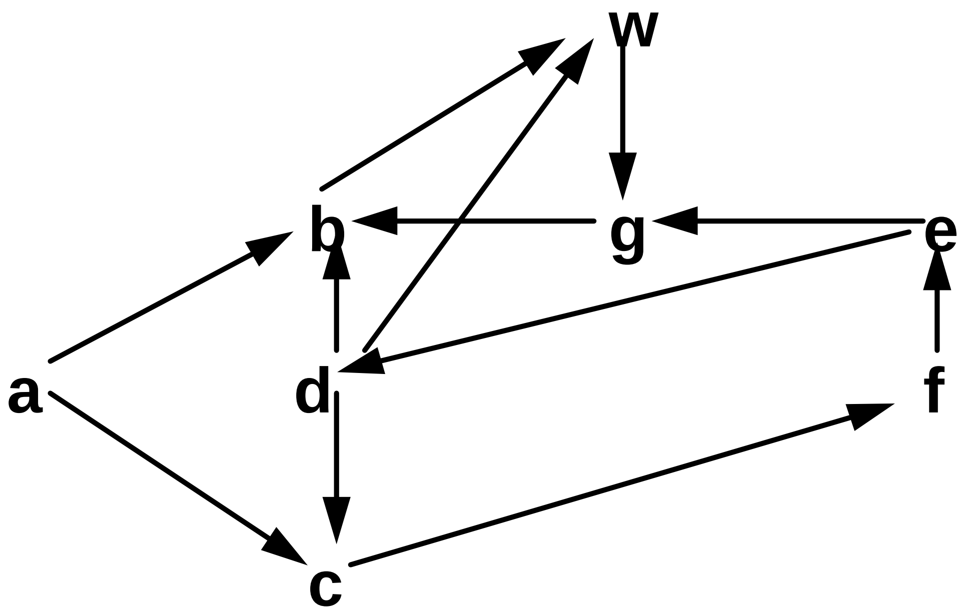


(u,u’) is an edge for every letter in G2

(2) How many isomorphisms are there of G onto itself. How many are there that map a to b? How many that map c to E?



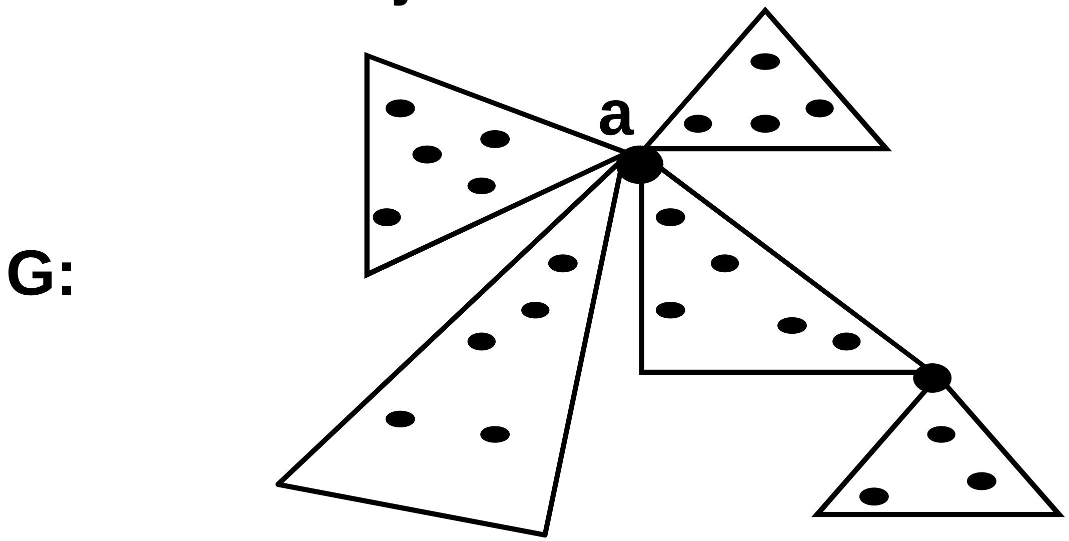
(3) Consider Dijkstra’s algorithm for dist. from a to b. Assuming merely that the edge weights are positive, what is the maximum heap (“R”) size that can occur and what are the heap elements then ? What is the minimum heap size that could occur?



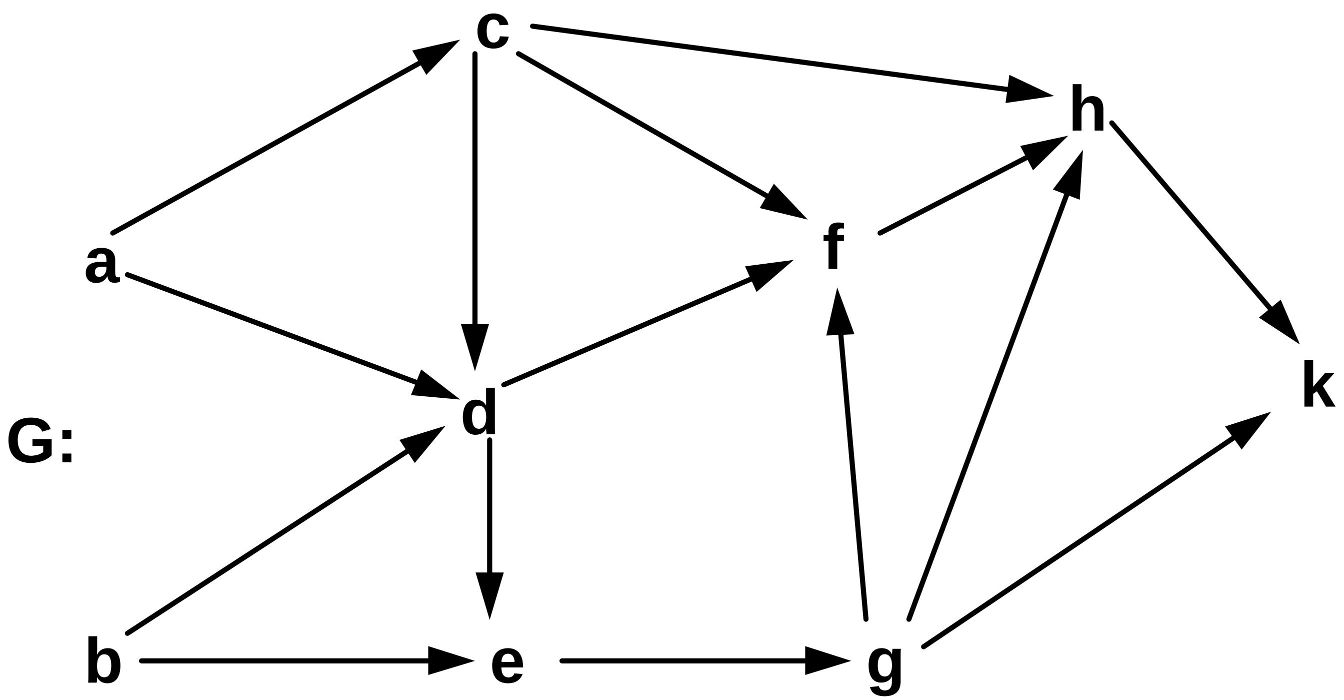
What are the maximum number of vertices that can be topologically numbered here - and what is the minimum number of edges that have to be removed so the resulting digraph is topologically number-able?

(4) For the example order 11 digraph in the handouts, what is the maximum number of cross edges that could occur in any traversal (regardless of root chosen, order of vertices on adjacency lists, etc.)?

(5) The blocks of G just are connected as shown - and the only vertices are the “critters” (●). What are the maximum and minimum possible sizes of the BS stack if the block recognition algorithm starts at a. Assume the articulation points are not adjacent.



(6) What is the maximum queue size in top. numbering algorithm for G? Which vertex (vertices)has (have) a unique topological number regardless of the order of processing of vertices?



(7) Be able to trace the operations of:

Dijkstra

Block

Depth First Search

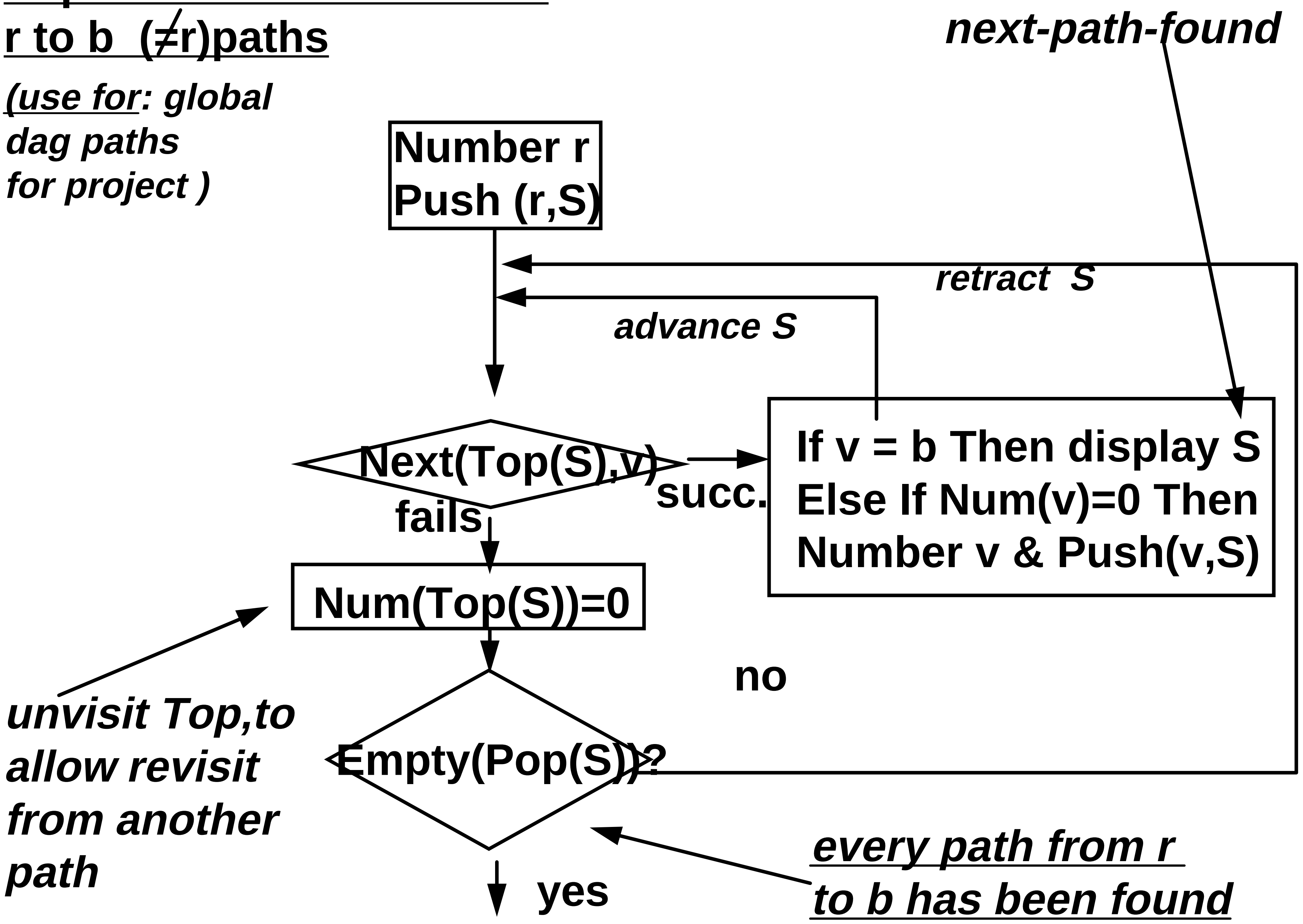
Strong Component

Top. Numbering and

Top. No.-based shortest/longest path algorithms.

Depth First Search For All r to b (≠r)paths

***(use for**: global dag paths for project)*



Number r
Push (r,S)

Next(Top(S),v)

succ.

If v = b Then display S
Else If Num(v)=0 Then
Number v & Push(v,S)

advance S

retract S

next-path-found

fails

Num(Top(S))=0

Empty(Pop(S))?

no

yes

unvisit Top,to allow revisit from another path

every path from r to b has been found

Max Matching Characterization:

M is a MM iff There exists No augmenting path for M

Definition: Augmenting path

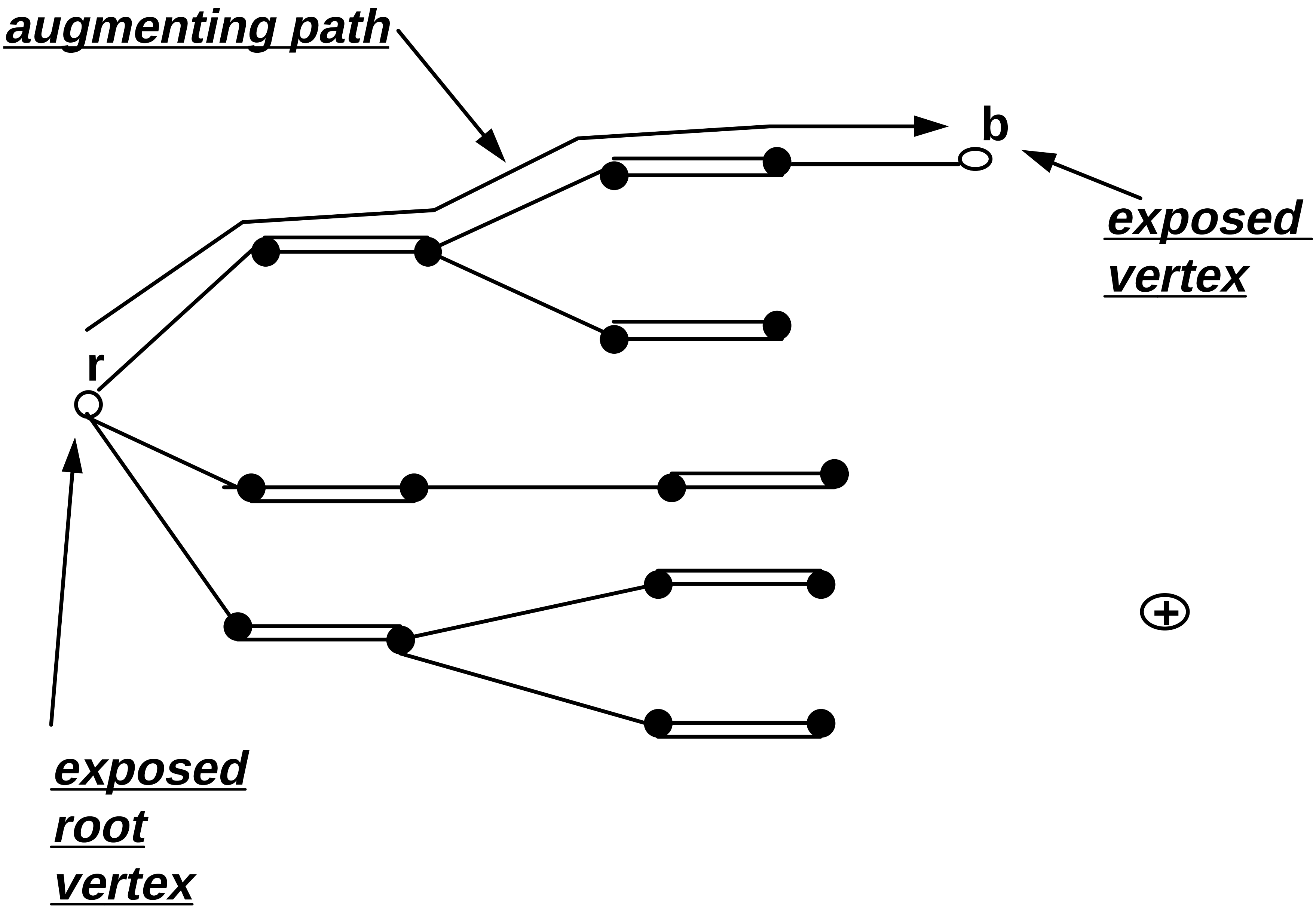
Alternating path with exposed endpoints

Proof of Characterization:

1. Suppose M’ > M.
2. Consider M $\oplus$ M’ => Paths & Cycles
3. Remove even cycles & paths so stilll M’ > M
4. There are no odd cycles.
5. There remain only odd paths - M to M, and M’ to M’.
6. Must exists some M’ to M’ - as M’ > M.
7. But: M’ to M’ path is augmenting for M !!

   contrary to assumption there were no M aug paths.
8. Therefore: there can be no M’ > M. QED

Alternating/Augmenting Search Tree



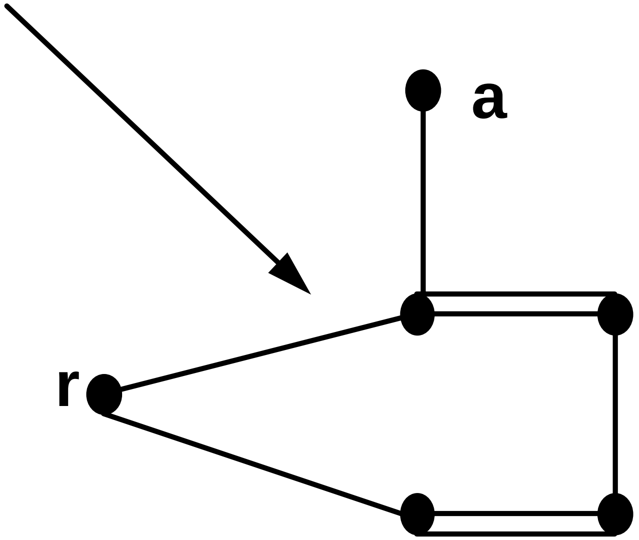
augmenting path

exposed vertex

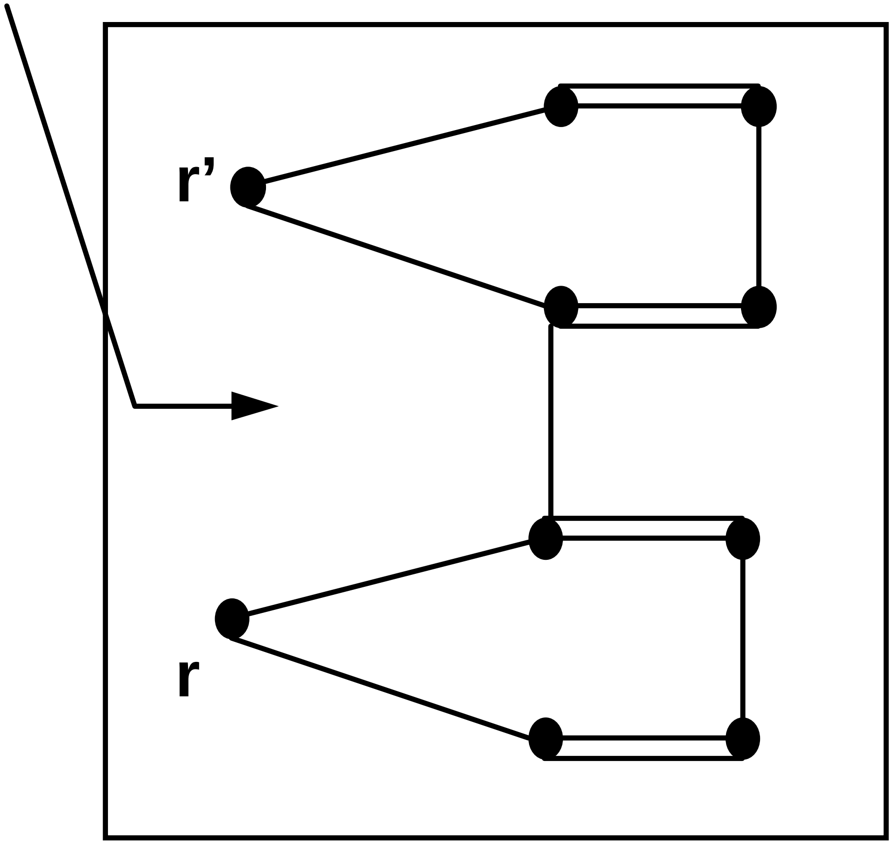
exposed root vertex

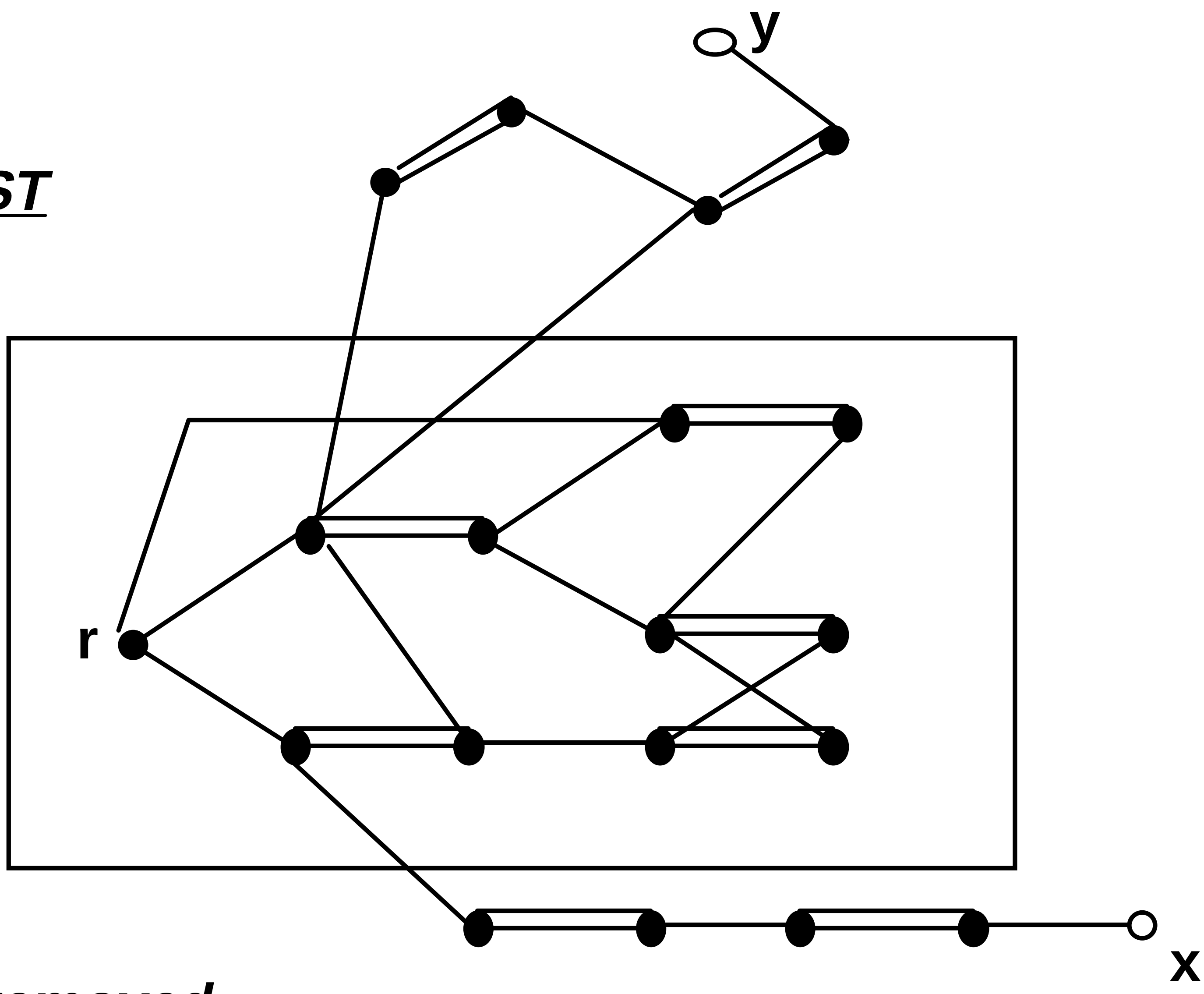
Counterexamples for Odd Cycle Constraint

may not find aug. path from r



may not find aug. path from r or r’

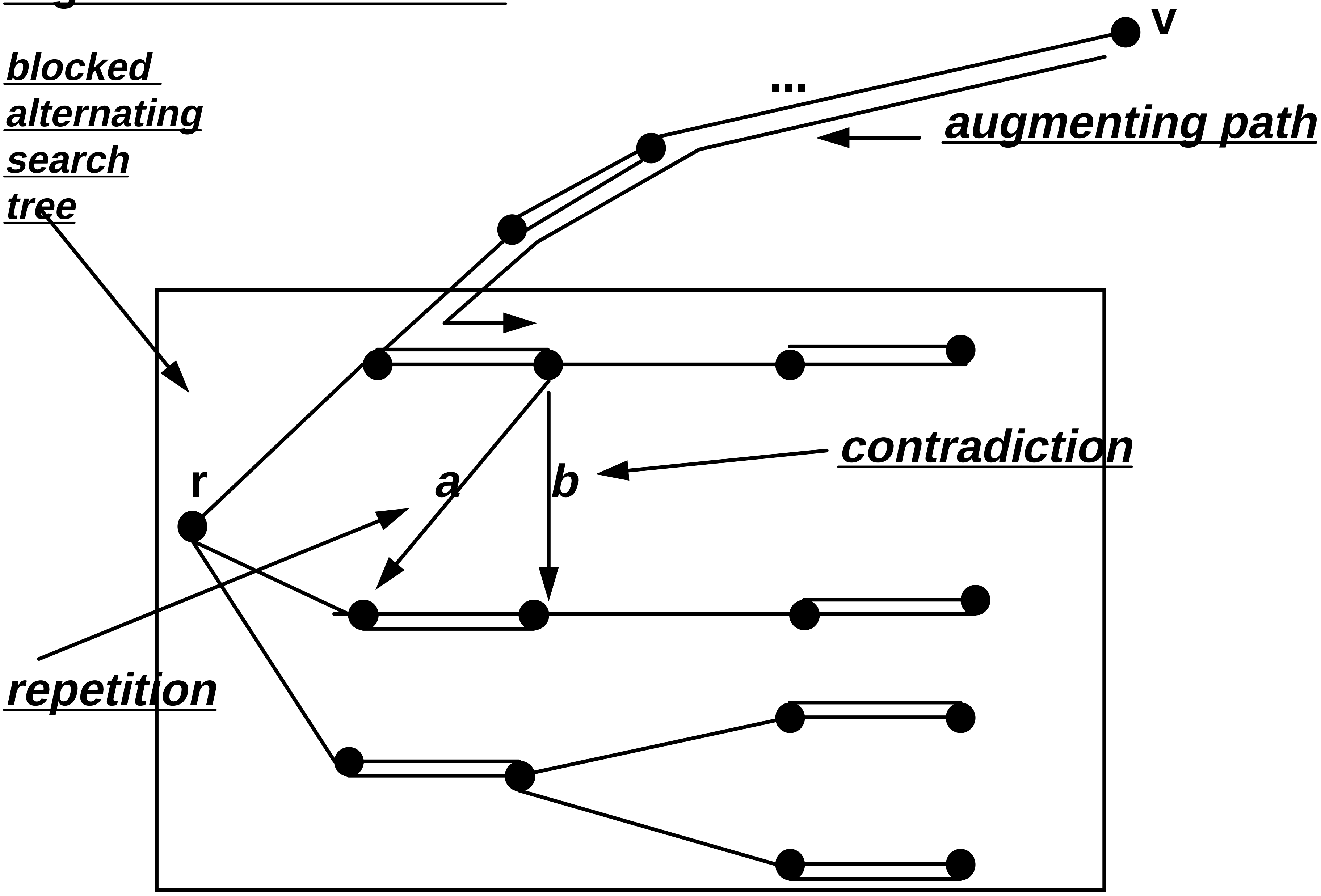




Blocked ST

can be removed
- without loss

Algorithm Correctness



blocked alternating search tree

augmenting path

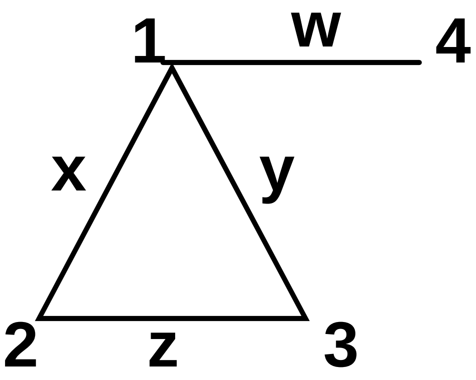
contradiction

repetition

Search Tree Blocked => No Augmenting Path (from r)

Perfect Matching - every vertex matched

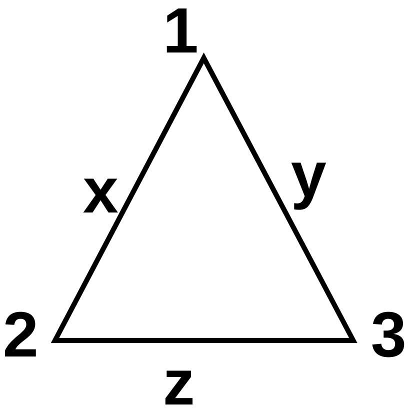
Tutte Determinant: TD



0	x	y	w
-x	0	z	0
-y	-z	0	0
-w	0	0	0

PM Condition: G has PM iff TD not identically zero.

Example



$$\begin{array}{|c|c|c|}\hline 0 & x & y \\ \hline -x & 0 & z \\ \hline -y & -z & 0 \\ \hline\end{array}$$

$$TD = (-1)(-x) \begin{array}{|c|c|}\hline x & y \\ \hline -z & 0 \\ \hline\end{array} + (-y) \begin{array}{|c|c|}\hline x & y \\ \hline 0 & z \\ \hline\end{array}$$

$$= - - x\,yz + -xyz \;\; = 0$$

Determinant Calculation

$$\begin{array}{|c|c|c|c|}\hline 0 & x & y & w \\ \hline -x & 0 & z & 0 \\ \hline -y & -z & 0 & 0 \\ \hline -w & 0 & 0 & 0 \\ \hline \end{array} = -(-w) \begin{array}{|c|c|c|}\hline x & y & w \\ \hline 0 & z & 0 \\ \hline -z & 0 & 0 \\ \hline \end{array}$$

$$= w\,(-(-z)(-zw)) = -\,w^2z^2$$

Not identically zero, so has PM.

Cost Analysis

1. Recursive calculation - from definition of determinant, costs   _________

2. Calculation by Gaussian Elimination costs _____

   (Based on   TDM => upper triangular form, by row operations that do not change det, then use fact that det of upper triangular matrix is just product of diagonals).

3. Criticism of G.E. approach - symbolic?

Exponential Complication

$$\begin{array}{cccc} x & y & z & \ldots \\ a & b & c & \ldots \\ p & q & r & \ldots \\ \ldots & & & \end{array}$$

$$\begin{array}{cccc} x & y & z & \ldots\ldots \\ 0 & b - (a/x)y & c - (a/x)z & \ldots\ldots \\ 0 & q - (p/x)y & r - (p/x)z & \ldots\ldots\ldots \end{array}$$

Resolution: random evaluation

1. **non-zero => definitely not zeo, so definitely PM**
2. **zero => almost certainly zero, so a.c. no PM**

Existential versus Constructive

Let G have PM

Select arbitrary edge (x,y) in G

If G - (x,y) has no PM

Then PM(G) = ____________________ (x,y) U PM(G - x - y)

Else PM(G) = ____________________ PM(G - (x,y))

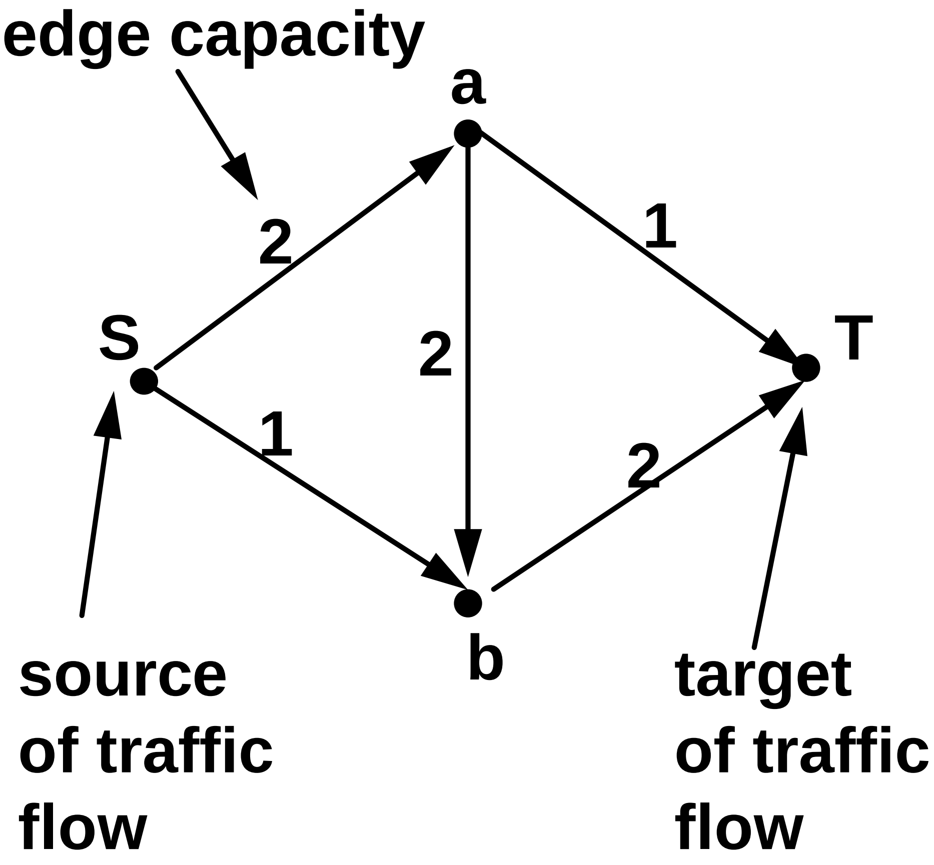
Maximum Matching Algorithm

```
MM <= False; M <= empty  (* M is the set of matched edges*)
Repeat (*Each iteration except last increments M; last
            recognizes have Max Matching - since no aug path found.*)
   Phase <= Not Done;  Reinitialize G and list of exposed vertices.
   Repeat
     If    (Find-next-exposed-vertex(u))
     Then    If  Aug-ST(u,v)      Then  Apply u to v aug path to increment M
                                         Phase <= Done
                                  Else   Flag blocked ST vertices (removes ST)
     Else     Phase  <= Done
              MM <= True
   Until Phase is Done
Until  MM is True
```

The Inner Loop searches for an exposed vertex with aug. path - any blocked ST's recognized along way are “removed” from G by merely marking their vertices as not to be scanned anymore (at least until Flags are reset in the outer loop)   -- Question: Does this flagging accomplish the desired O(EV) effect we claimed?

Flow Networks

Problem: What is the maximum traffic flow from S to T?

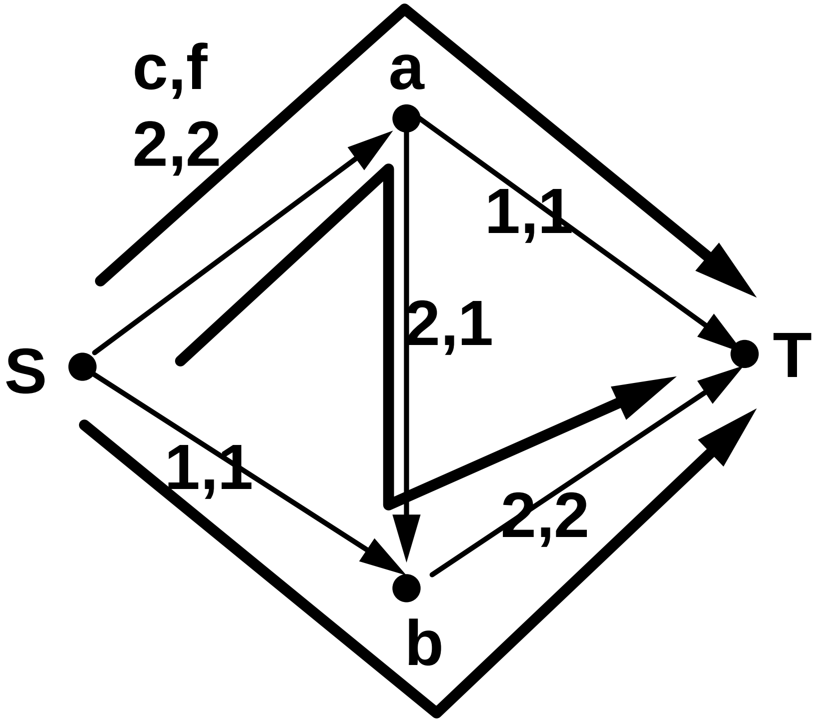


edge capacity

source of traffic flow

target of traffic flow

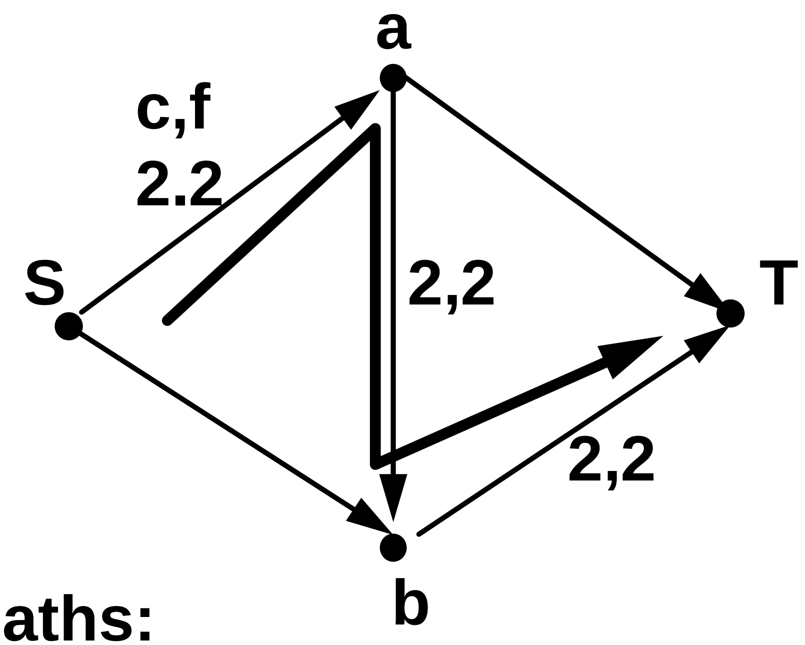
Realization of Flow by Flow Paths



Paths: SaT (- flow 1)
SbT (- flow 1)
SabT (- flow 1)

Total Path flow per edge
<= Edge Capacity on edge

Total Flow: 3 (maximum)



Paths:
SabT (- flow 2)

Total Flow: 2 (maximal)

Maximal means the flow can-not be increased by adding another path flow, without violating capacity constraints.

Conservative Model of Flow

A *conservative flow* on a digraph with edge capacities is an assignment of real valued flows to the edges so:

(1) (*Positivity*)

Each edge flow is non-negative.

(2) (*Capacity constraint*)

Each edge flow <= edge capacity.

(3) (*Conservation of Flows*)

For every vertex v other than S or T:

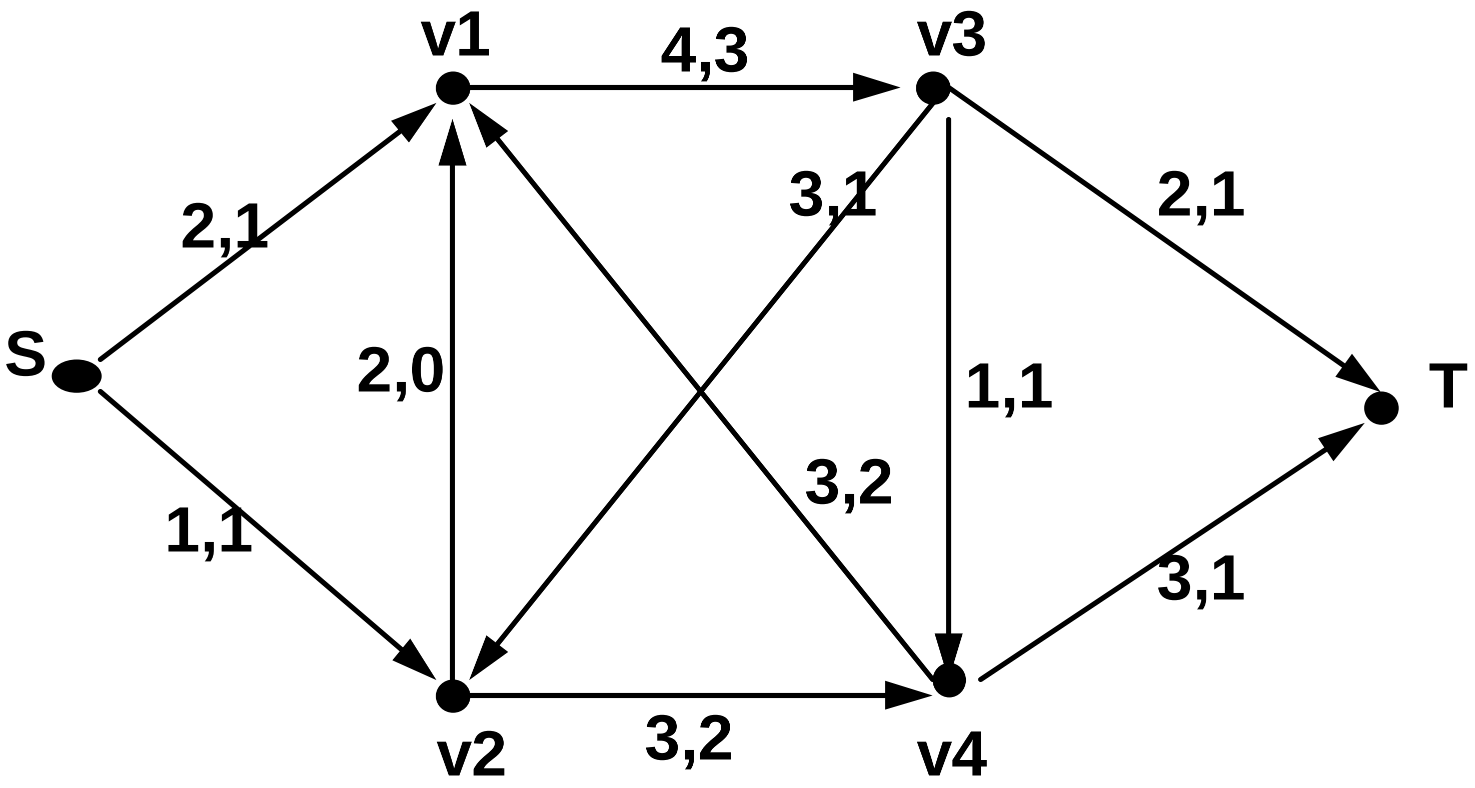
sum of flows into v = sum of flows out of v.

(4) (*Flow Value*)

netflow out of S (or netflow into T)

We can trivially obtain a valid conservative flow from a set of path flows by adding (superimposing) the values of the path flows through each edge. The value of the resulting conservative flow is the sum of the values of the path flows.

A More Complicated Conservative Flow Example



Observe: flowinto(v1)=flowout(v1)=3,etc - except for S and T.
flowvalue is defined as: flowout(S) = 2

Equivalence of Path & Conservative Flows

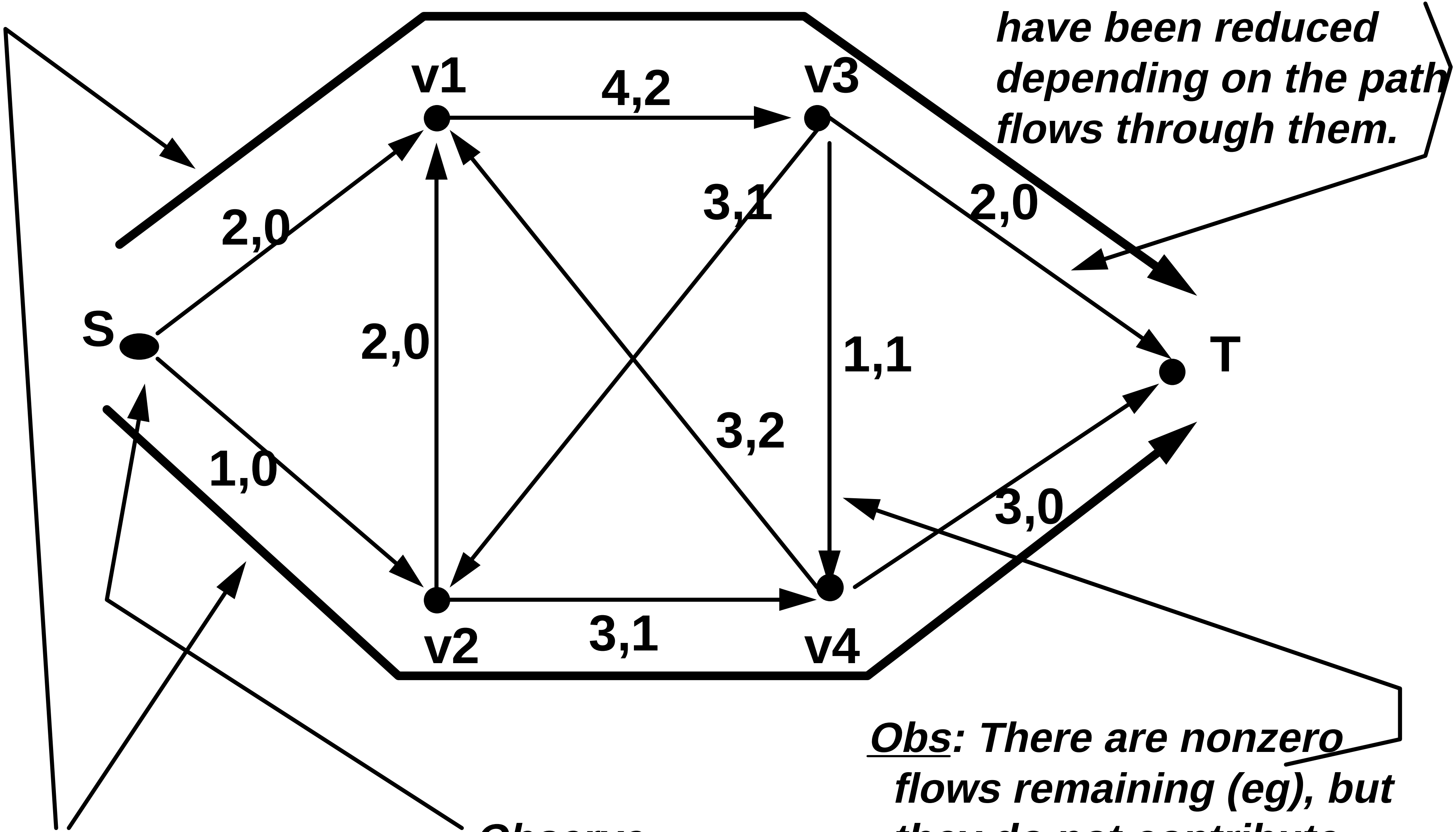
For every valid path flow, there is an equal valued conservative flow, and vice versa:

Path Flow=> Cons. Flow: merely superimpose path flow values
Cons. => Path Flow: use modified DFS to obtain flow paths of equal value

Modified DFS Procedure:

1. **Start depth first search at S, but advance path only on edges that have non-zero flows!**
2. **As usual, never extend the path to already visited vertices.**
3. **The process can never be blocked at any intermediate vertex - because the flow into & out of each such vertex is equal. If the path can enter, it can surely exit.**
4. **Thus, if there is a nonzero flow out of S, then the procedure will find a path from S to T - all of whose edges have positive flow. For simplicity, assume all flows are integers and remove a unit flow from each edge on the found path. This reduces the flowout(S) by 1.**
5. **Repeat the process until flowout(S) is zero. At that point, the procedure will have found a set of paths equal in value to the original conservative flow.**

Network after DFS path flows are removed



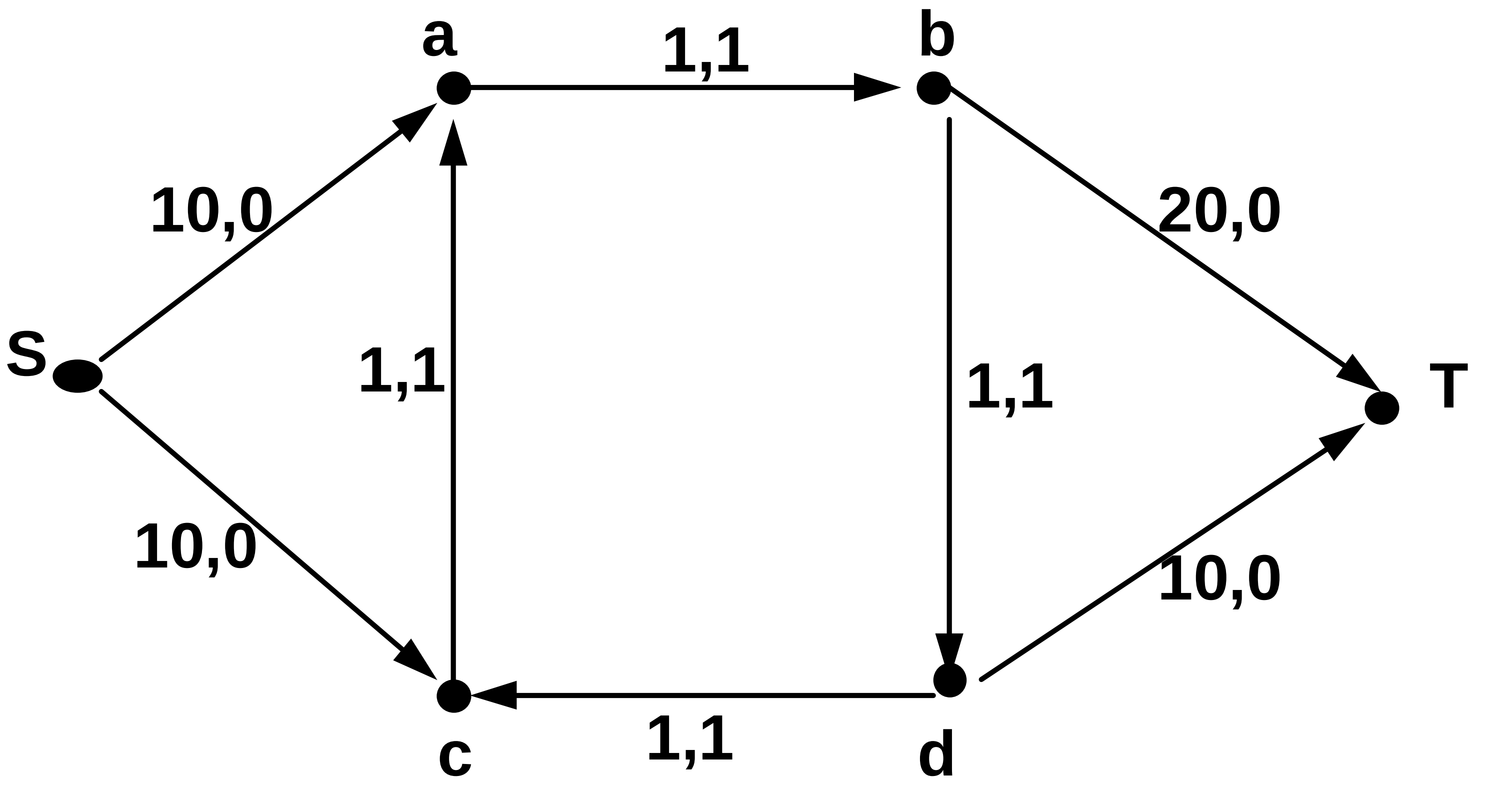
***Obs**: the edge flows(eg) have been reduced depending on the path flows through them.*

***Obs**: There are nonzero flows remaining (eg), but they do not contribute to flow value.*

***Observe**: flowout(S)=0*

Flow paths found by dfs, each of flow 1.

A Flow can be Blocked even when it is not Maximum



Flow Value = 0

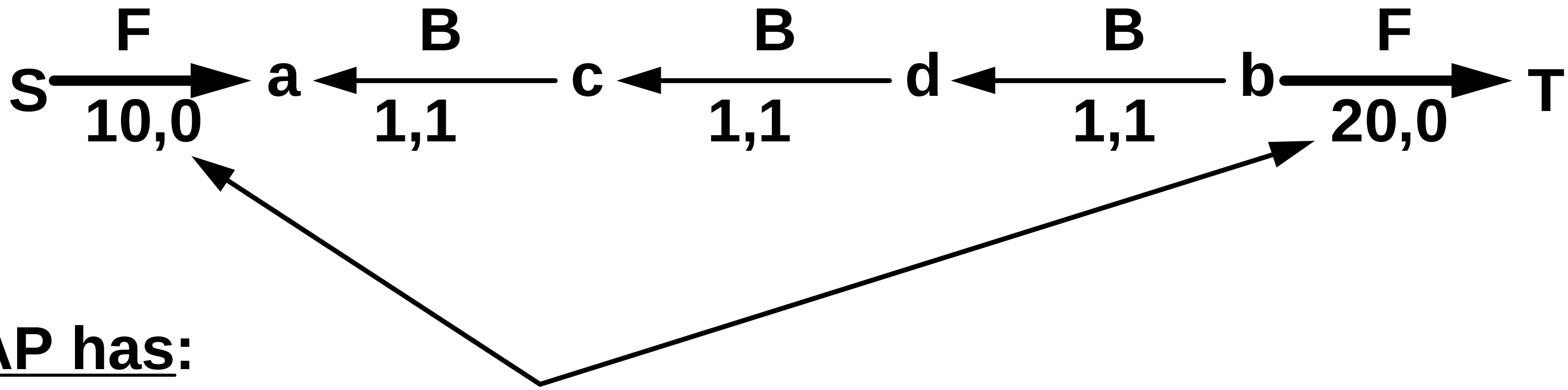
Max Flow should be 3

Nonetheless: there is no S-to-T path along which flow can be augmented.

To rectify this: introduce Flow Augmenting Paths (FAPs)

Flow Augmenting Paths

The previous example contains the Flow Augmenting“path”:



FAP has:

1. **Forward edges (F) with: flow < capacity (spare cap.)**
2. **Backward edges (B) with: flow > zero (positive flow.)**

Use such a path to augment flow by:

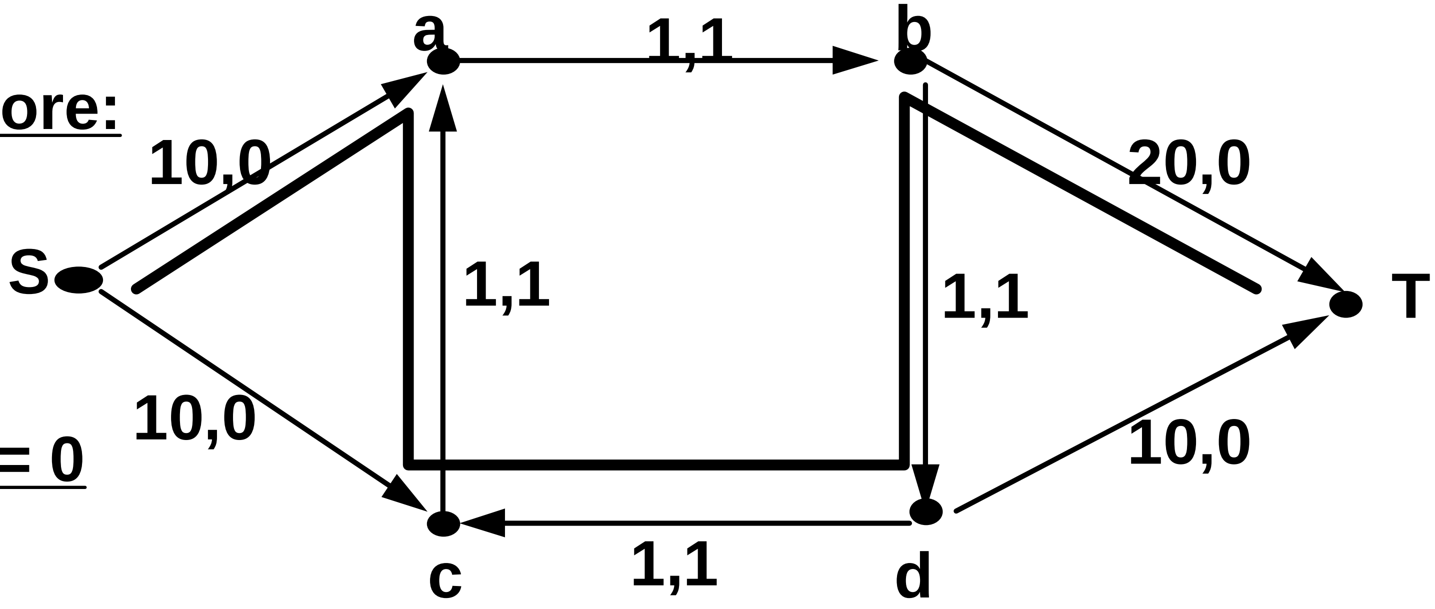
1. **Increment flow on F edges by (say) 1.**
2. **Decrement flow on B edges by 1.**

Resulting Conservative flow is valid and 1 larger than before.

Application of FAP

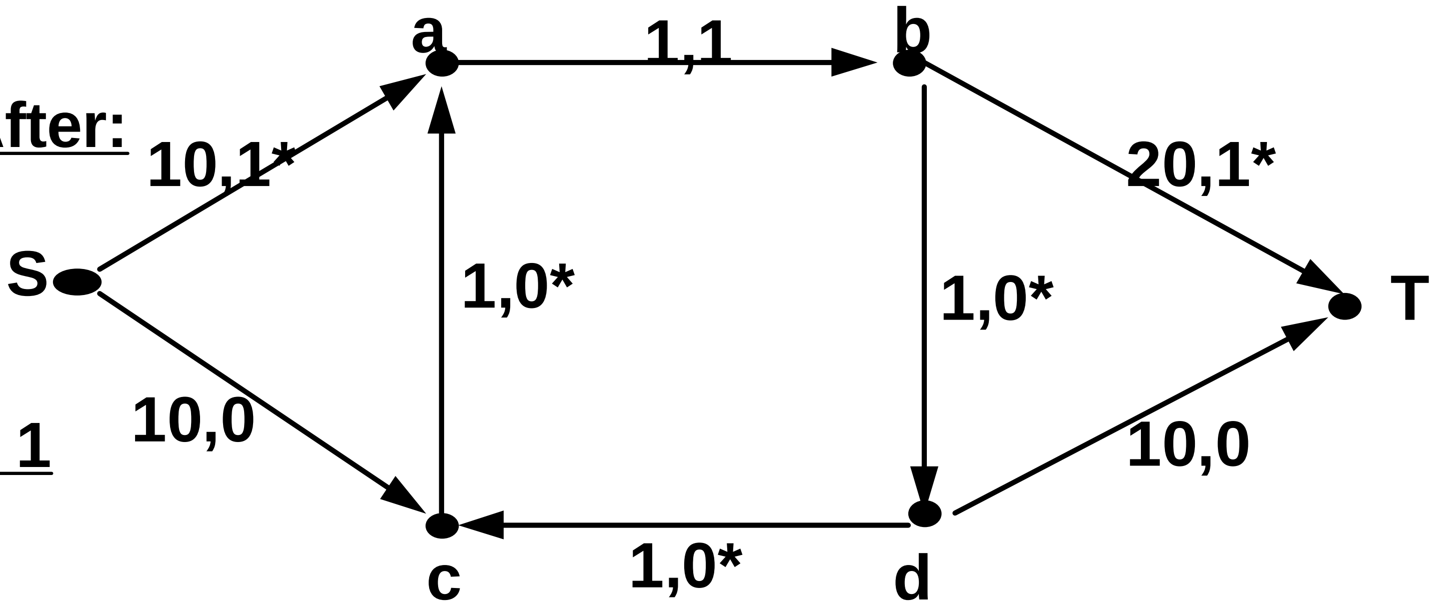
Before:

flow = 0



After:

flow = 1



How do you find a FAP

Use modified DFS:

1. **Unlike usual DFS, both outgoing and incoming edges are allowed for advancing the FAP.**
2. **A Forward edge (outgoing) can be used only if it has spare capacity; a Backward edge (incoming) can be used only if it has Positive flow.**
3. **If the search tree reaches T, then the vertices on the stack are the vertices of the FAP; otherwise, if the search tree is blocked before T, then there is no FAP.**

<u>Observe</u>:

1. **Flow Augmenting Tree Blocked iff There is no FAP**

2. **Flow Augmenting Tree Blocked iff Flow is Maximum**

The proof of (2) follows readily from:

3. **Let (X, V-X) be a cut in a flow network. Then the FlowValue = f(X,V-X) - f(V-X,X) (ie, the net flow across cut.)**

To find Max Flow, merely repeatedly generate Flow Augmenting Search Tree to find FAP. When the Flow Aug. Search Tree procedure blocks before reaching T, the flow is maximum (by (2) above.)

Floyd’s Shortest Path Problem/Algorithm

Problem: Given digraph G(V,E) with real-valued edge weights, find the shortest distance between every pair of vertices.

Assumption: G contains no negative cycles (ie, cycles whose edge weights sum to < 0).

Input: List of edges and their weights.
Output: Distance matrix D(I,J)

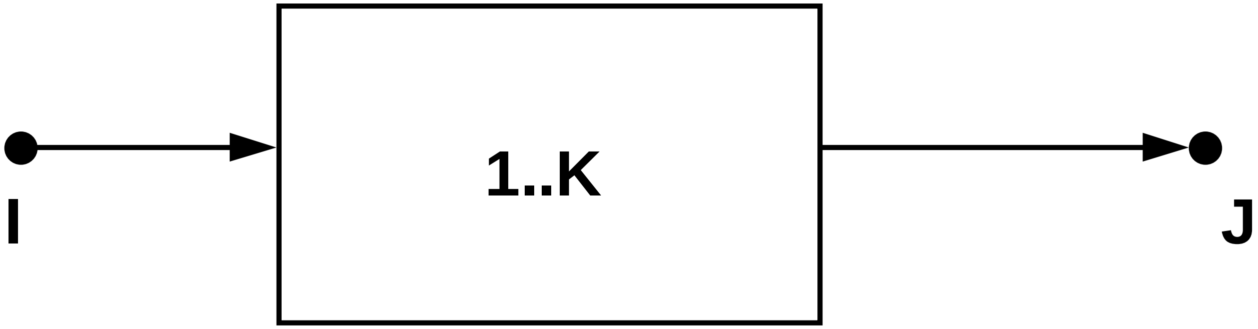
Remark: If G has cycles<0, then the algorithm gives invalid answers. Indeed, the problem itself (as opposed to merely particular procedures for its solution) then becomes NP Complete - ie, has no polynomial-time algorithms.

Performance: 0(V^3)

Idea Underlying Floyd's Algorithm

(1) Parametrize problem
(2) Build paths by Partitioning

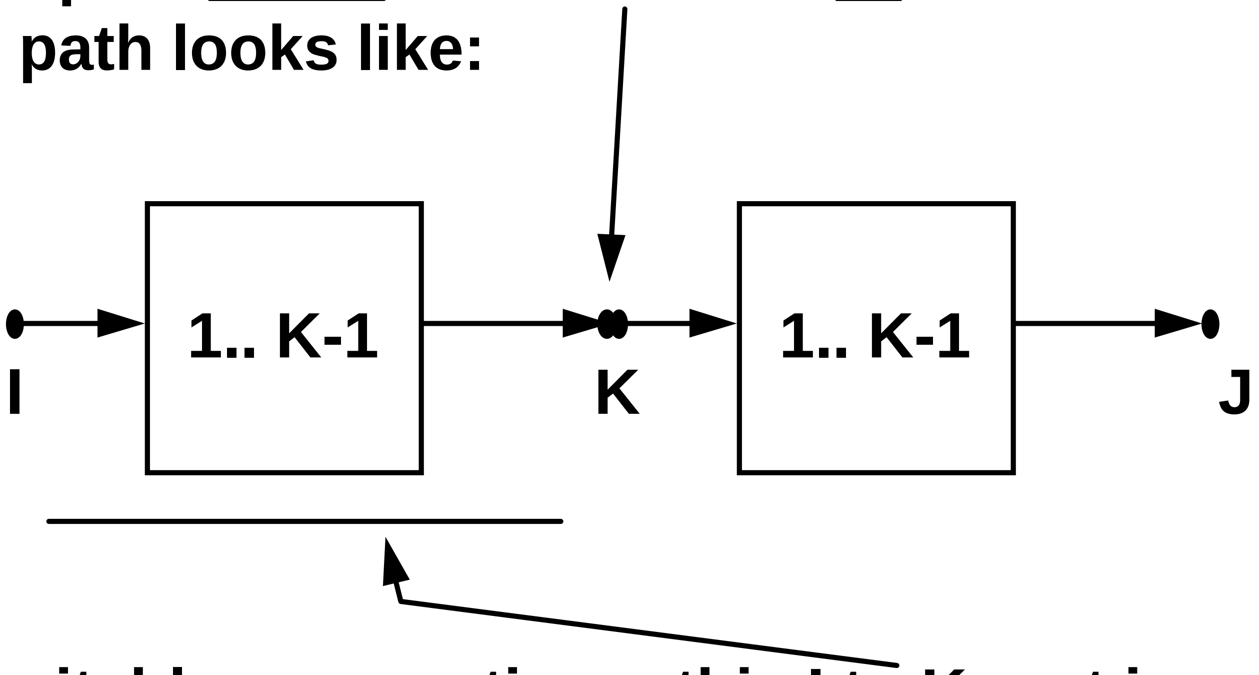
Parametrization:
A stage K shortest path from I to J is a shortest path whose internal vertices have subscripts only on 1..K. Pictorially:



The advantages of parametrization are:
(1) K is 0 => just original edge from I to J (possibly the trivial weight M "edge") .
(2) K is V => just unrestricted shortest path from I to J.
(3) Furthermore: one can easily build stage K paths out of stage K- 1 paths.

Partitioning: The Stage K Alternatives

A stage K path <u>either</u> uses vertex K <u>or</u> does not. In the first case the path looks like:

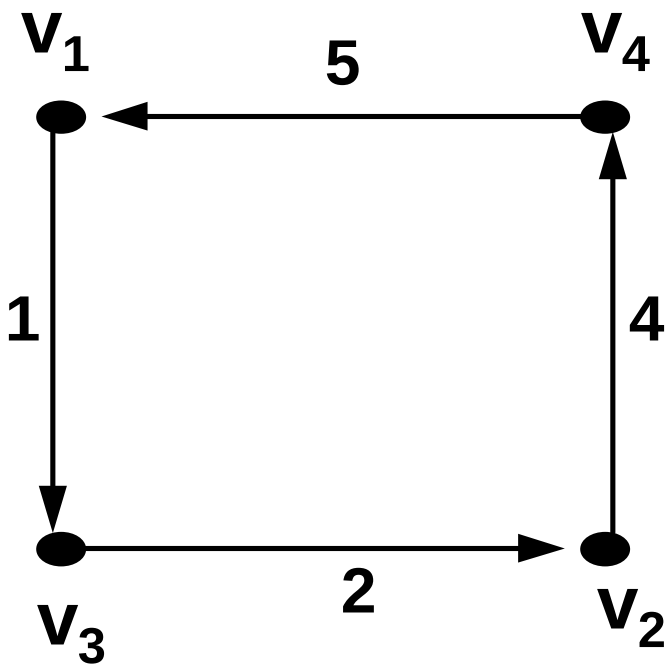


Under suitable assumptions: this I to K part is a stage K-1 shortest path from I to K, and similarly for the K to J part.

This “partition” leads to bootstrapping formula:

$$D^K(I,J) = \text{Min} \; \{ D^{K-1}(I,J) \; , \; D^{K-1}(I,K) + D^{K-1}(K,J) \}$$

Shortest Paths Example



Digraph with weights
for G(V,E)

0	M	1	M
M	0	M	4
M	2	0	M
5	M	M	0

Initial Distance Matrix D for G
(M denotes a “large” number,
reflecting absence of a
corresponding edge.)

After stage 1:

0	M	1	M
M	0	M	4
M	2	0	M
5	M	6*	0

After stage 2:

0	M	1	M
M	0	M	4
M	2	0	6*
5	M	6	0

After stage 3:

0	3*	1	7*
M	0	M	4
M	2	0	6
5	8*	6	0

After stage 4:

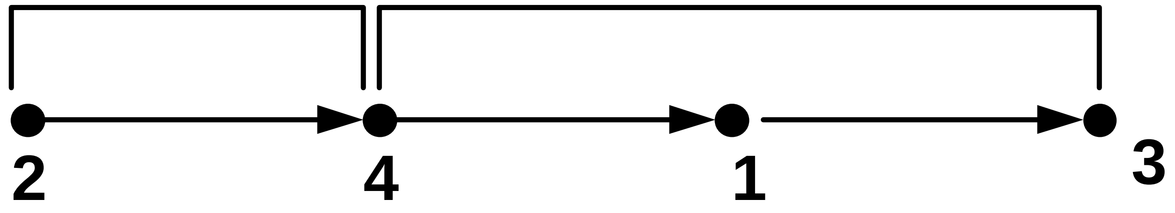
0	3	1	7*
9*	0	10*	4
11*	2	0	6
5	8	6	0

<u>Remarks on the Example</u>

(1) There are only 4 edges in the digraph, so there are only 4 real values other than 0 & M in the initial matrix.

(2) The original digraph is strongly connected (ie, there is a path between every pair of vertices), so in the final digraph there are No M’s left - as expected.

(3) The “build-up” underlying a particular calculation:



D(2,4) + D(4,3)

given as input

known by stage 1

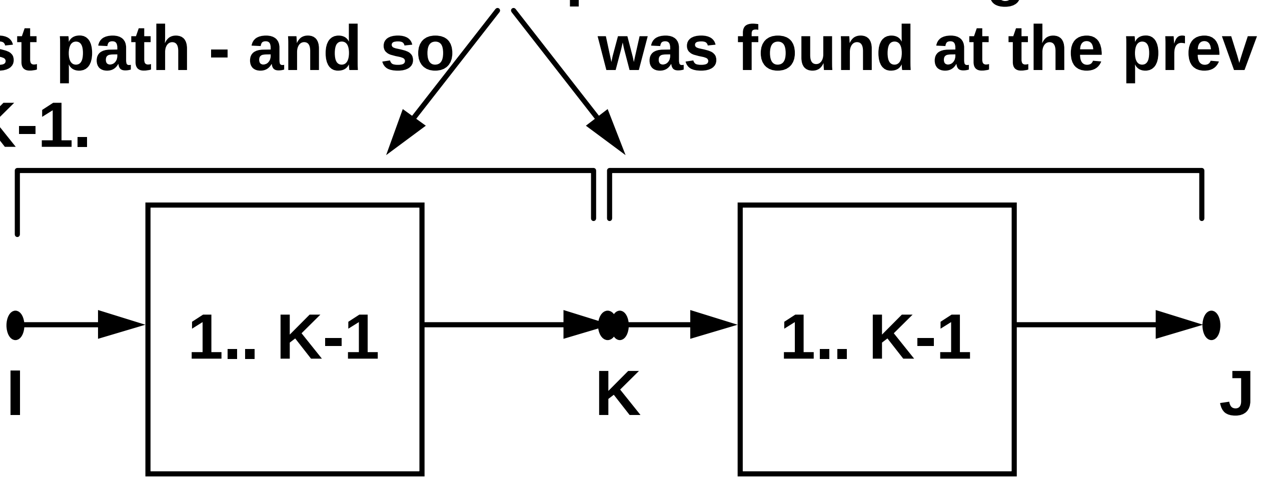
combined by stage 4: D(2,3) at stage 4

Key Points:

(1) (Correctness - or Partitionability)
A part of a shortest path is also a shortest path
iff G has no negative cycles.

(2) (Testing Cycle Restriction)
G has no negative cycles
iff some D(i,i) value eventually becomes negative.

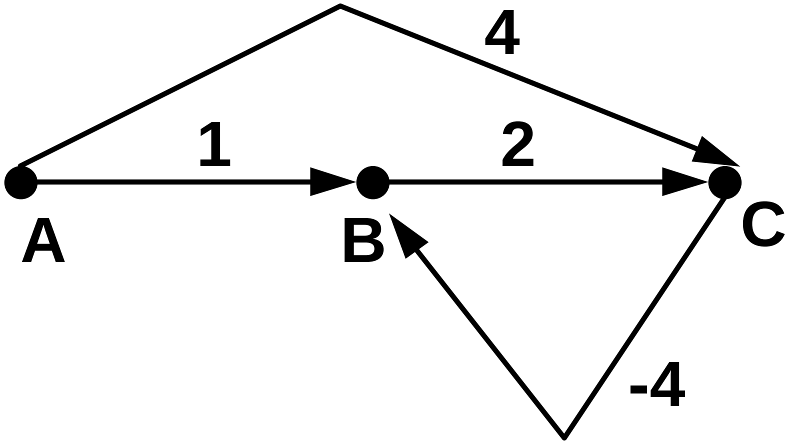
(1) => Lets us assume each part in the stage K alternative is a shortest path - and so was found at the previous stage K-1.



(2)=>Lets us test for the restriction under which the algorithm is valid - <u>during</u> the application of the algorithm itself ! If any diagonal entry becomes negative, then the digraph does not satisfy the “no cycles < 0” condition.

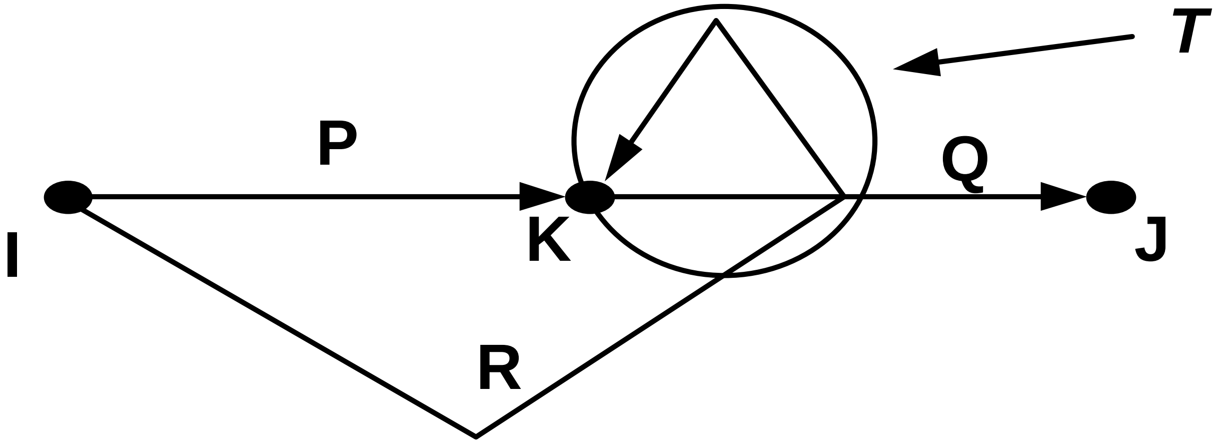
Proof of (1):

(a) If some cycles < 0, then partitioning (and so the algorithm) may fail, as shown by example:



Here: ABC is shortest path from A to C, But its leading part AB is not a shortest path from A to B (ACB is !)

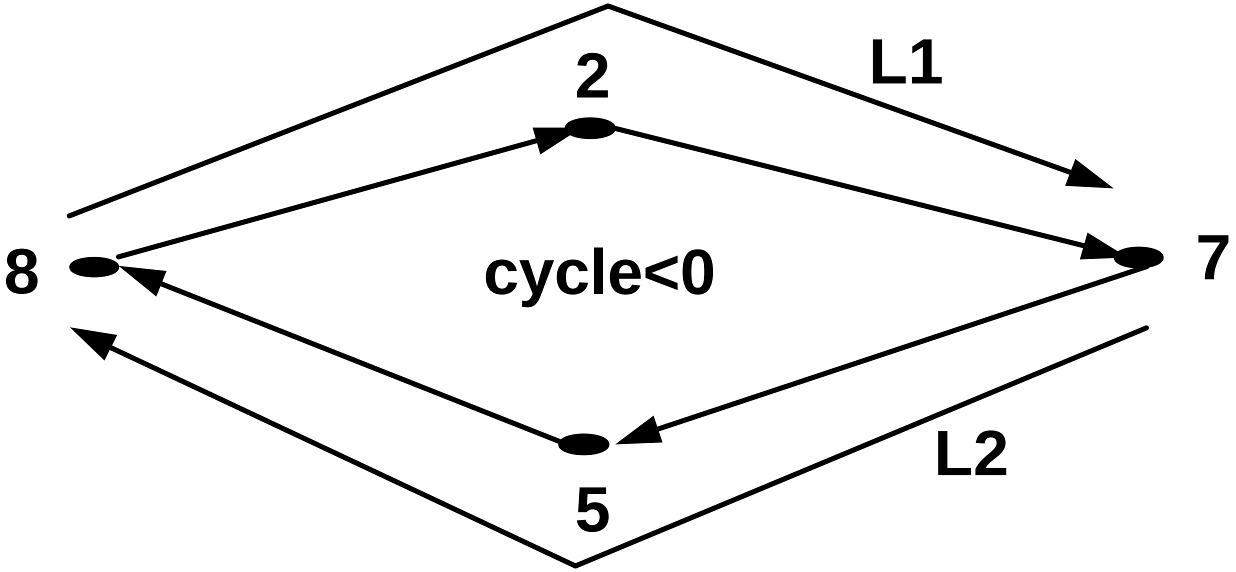
(b) If no cycle < 0, then partitioning succeeds (ie, each part of a shortest path is also a shortest path.)



If P U Q is shortest I to J path, then P must be shortest I to K path. For else if R (from I to K) were shorter, then R U Q is certainly a “walk” from I to J (- walks are paths that can intersect themselves). This walk can be transformed to a path by deleting any cycles it contains, such as T. Since by assumption cycles are all >0, removing them makes the result shorter, so the resulting path would be even shorter than P U Q, contrary to supposition that P U Q is shortest.

Proof of (2) We have to prove a statement and its converse:

(a) If some cycle < 0, then some diagonal D(I,I) becomes negative eventually (ie, by most stage V).

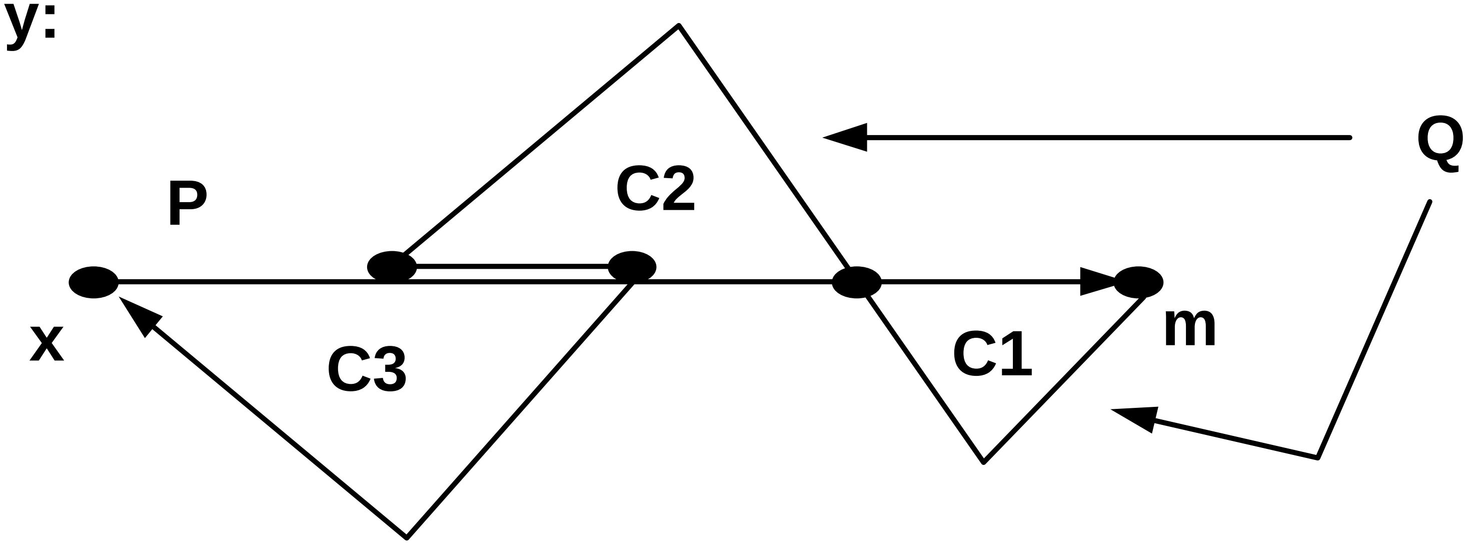


For example, consider cycle shown, and assume L1+L2<0.
By stage k=2: D(8,7)<=L1. By stage 5: D(7,8)<=L2. So
by stage 7: D(8,8) <= D(8,8) + D(7,8) <= L1+L2 < 0.

(b) If some diagonal becomes negative, then there must be a negative cycle. Let D(x,x) be first such diagonal to become negative, and let this happen at stage m. Then:

D(x,x) = D(x,m) + D(m,x) < 0,

where D(x,m) is the length of a shortest path P from x to m, while D(m,x) is the length of a shortest path Q from m to x. Pictorially:



But P U Q can always be represented as a disjoint union of cycles (eg, C1,2,3). Since the sum D(x,x)of these cycles<0, then at least one of the cycles C must be < 0.

t

Project Requirements

Input: Weighted digraph(s) G(V,E)
Pairs of vertices { (A,B) | A,B in V(G) }
Output: Shortest path and distance from A to B
using Disjkstra + Strong components partition.

Process adjacency lists in numerical order.
Use example give.

Include other cases that intelligently test algorithm.

The problem has to be done in phases:

I: Goal: Identify strong components of G(V,E)
Method: Depth first search
Associated outputs:
Final Dfs and Low for each vertex
Set of vertices in each st. component
An array Owner such that Owner(i)=k when vertex i is a member of st-component k.

II: Goal: Build associated acyclic super-digraph S(V1,E1)
Method: Construct adjacency list representation using st. components from I.
Associated outputs:
Adjacency list at each supervertex v is of form:

$(a_1,b_1,w_1) \Rightarrow (a_2,b_2,w_2) \Rightarrow (a_3,b_3,w_3) \Rightarrow \ldots$

where (a_k,b_k,w_k) indicates (a_k,b_k) is edge in G(V,E) from v to w_k. (Order these lexicographically.)

III:Goal: For each test pair of vertices A and B (from G), identify cheapest A to B path.

Method:

(1) Find next global path P in S from W_1 (= Owner(A)) to W_2 (=Owner(B)) in S(V1,E1):

$W_1 \Rightarrow a_1, b_1 \Rightarrow W_2 \Rightarrow a_2, b_2 \Rightarrow W_3 \Rightarrow \ldots \Rightarrow (a_k, b_k, w_k) \Rightarrow W_{k+1}$

(2) Evaluate the cost of the global route P by solving and summing the costs of the “local” Dijkstra problems from b_i to a_{i+1}, and the edge costs of $(a_i,b_i)_.$

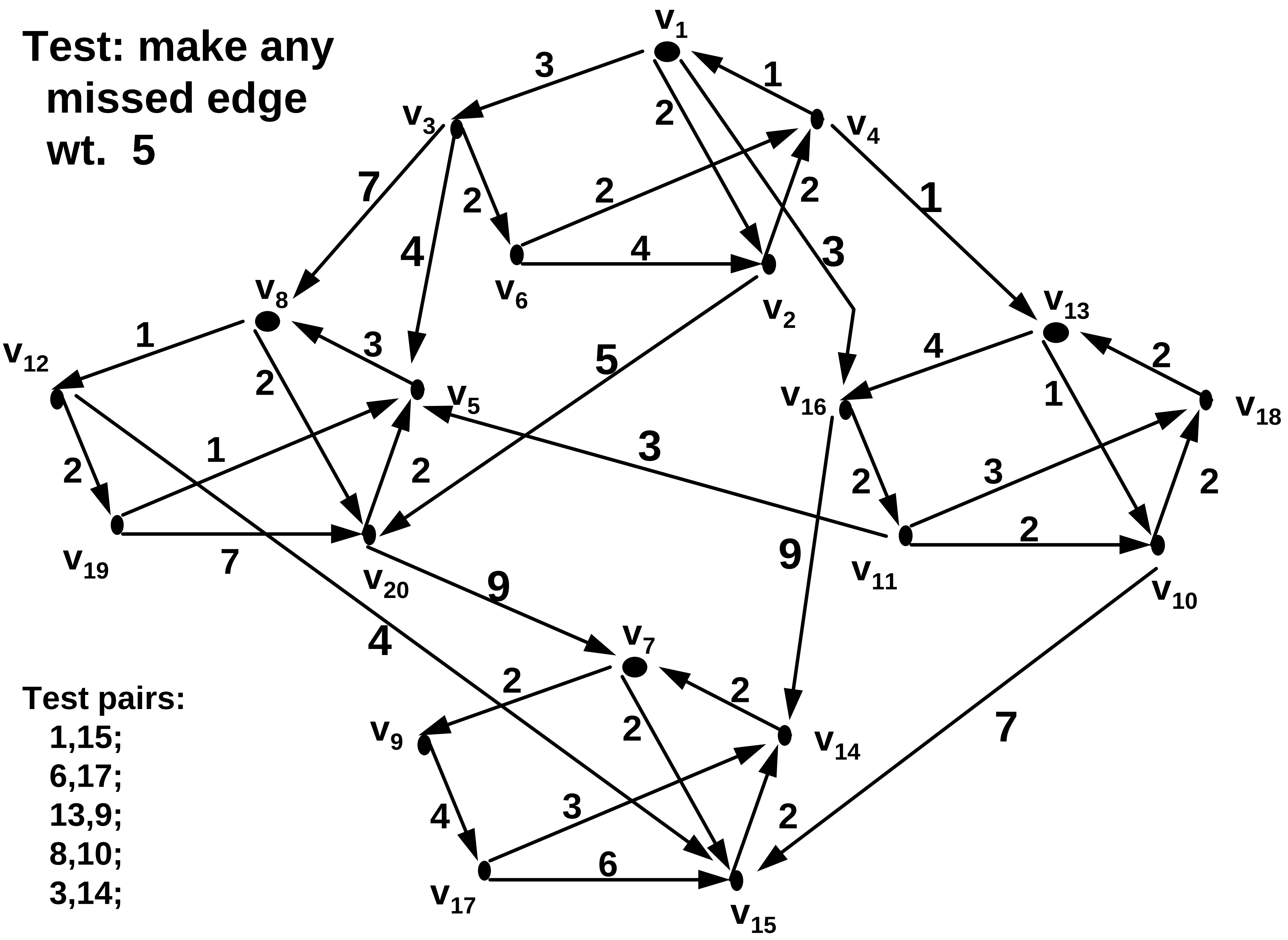
(3) Identify least cost route from (2)

Output: Structured view of shortest path:

W_1 {A =>...=>a_1 Dijkstra path} => (a_1,b_1)=>
W_2 {b_1 =>...=>a_2 Dijkstra path} => (a_2,b_2) =>.....
W_k{b_{k-1} =>...=>a_k Dijkstra path} => (a_k,b_k)=>
W_{k+1}{b_k =>...=>B Dijkstra path}

and cost of each “piece” of path.

Test: make any missed edge wt. 5

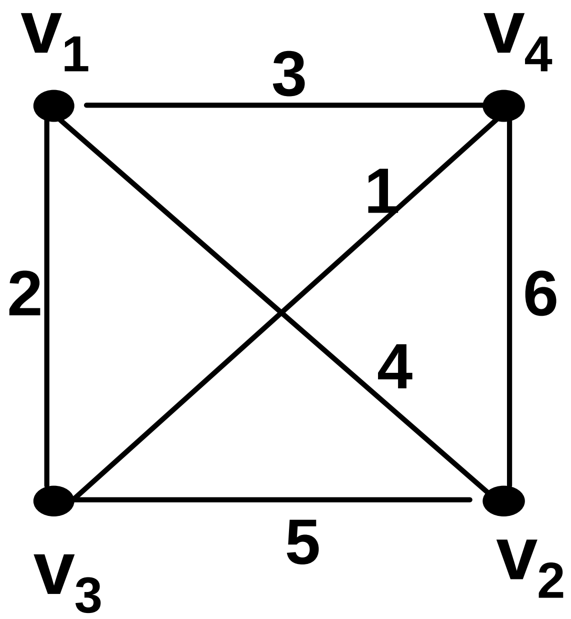


Test pairs:
1,15;
6,17;
13,9;
8,10;
3,14;

Minimum Spanning Trees

Problem: Given undirected connected graph G(V,E) with real valued edge weights, find a minimum weight connected vertex spanning subgraph.

minimum subgraph => tree (acyclic by minimum character)
Exp. G below has MST consisting of edges 1,2,and 4



Some questions:

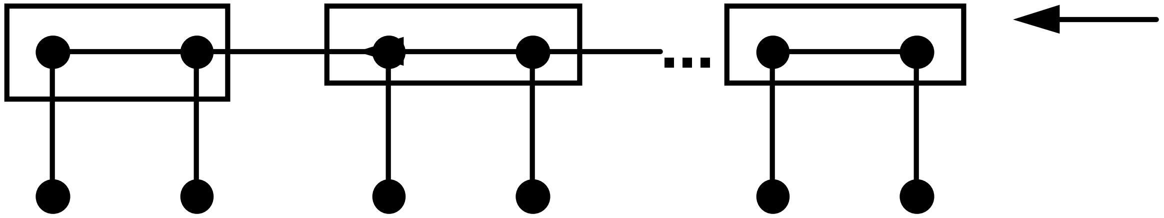
Why isn’t edge 3 in MST?
Can the largest edge ever be in MST?
How many edges are there in MST?
Is the MST unique?

Method - Greedy Algorithm

A “greedy algorithm” trys to obtain a global optimal by repeatedly seeking “local optima” restricted only by the basic problem constraint.

Exp. Maximum Matching: A greedy algorithm might merely: Scan the edge list, adding an edge to the matching obtained so far subject only to the constraint that the added edge not be adjacent to a previously added edge (ie, the basic matching constraint.)

For Maximum Matching, the greedy approach is suboptimal, as shown by the following example, where greedy approach gives solution which is only 50% optimal:



Only 50% of vertices would be matched if boxed edges were successively selected.

Greedy Algorithm for MST:

Scan the edge list, adding the next least weight edge to the subgraph generated so far, subject only to the constraint that the added edge not introduce a cycle.

Algorithm Statement:

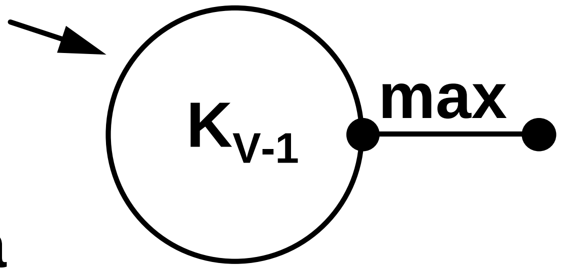
```
T <=  { }
Repeat
    1.   Get next least weight edge x
    2.   If T + x is acyclic, then add x to T
Until    | E(T) | = V - 1
```

Objectives:

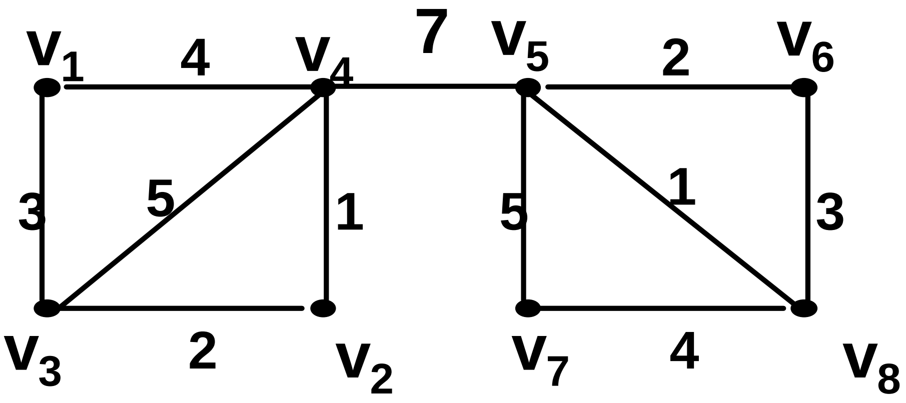
- Examine performance for implementations of step 2
- Examine performance for heap implementation of step 1
- Prove correctness of algorithm

Remarks:

1. 1st & 2nd edges are automatically in MST.
2. Max weight edge is in MST iff it is a bridge (ie, an edge whose removal disconnets G.)
3. Maximum possible number of iterations of Repeat-loop is (V-1)(V-2)/2 as shown by exp:
4. A minimum of V-1 iterations are needed, and occurs precisely when the least V-1 edges form a spanning tree.



5. Despite having equal-weight edges, graph shown has unique MST.



Algorithm Correctness:

Suppose T is tree generated by MST algorithm and S is a minimum weight spanning tree. We show that S and T have same weight, so that T is MST too.

1. If T = S, we are done. Else, suppose T and s are not equal.
2. Let x be 1st edge added by algorithm to T not also in S.
3. S + x has a cycle C. Some edge x' on C is not in T.
4. Consider S + x - x'. Distinguish three cases:

- **a. wt (x') > wt(x)**
- **b. wt (x') = wt(x)**
- **c. wt (x') < wt(x)**

Case a is impossible,else wt(S+x-x’)<wt(S) - a contradiction.

Case c is impossible too. For otherwise, x’ would have been considered by the algorithm for inclusion in T before x. By assumption all the edges in T at that point would also be in S, so - since x’ is in S and S is a tree - x’ would not cause a cycle in T at that point, and so would have been included in T - contrary to fact that x’ is not in T.

Case b is only case that can occur. But then S+x-x’ and S have the same weight. Observe that S+x-x’ has one more edge in common with T than S does, so that by repeating this “procedure” we will eventually obtain T - and in the meantime have demonstrated that T has same weight as S, which is a MST. This proves correctness of algorithm.

Testing if T+x is Acyclic

Methods:

1. Flag array
2. Circular lists embedded in an array
3. Unary Tree embedded in an array

1. Initially Flag(i)=i for every i. If edge (i,j) added and i<j, then every vertex with Flag = to j, is reset to i.

2. Initially, Flag(i)= - i for every vertex i. If edge (i,j) added and i<j, then list vertex j lies on is appended to end of list vertex i lies on. (Head of each list is identified by having -v, where "-" signals a head of list, with tail at v.)

First compare methods 1 and 2 for performance.

Function	Test for Cycle	Form Union of Components	Total
Frequency	O(E) times	O(V) times	
Flag	O(1)	O(V)	O(V^2)
Circular List	O(V)	O(1)	O(EV)

Even though Flag & List approach at first appear the same (O(1) & O(V) or the reverse), the frequency of the Test and Union operations required by the algorithm indicate the simpler Flag approach is better overall.

3. The Unary Tree approach embeds a “parent pointer” tree in an array UT. Initially UT(i)=i, for every i.
Test and Union are simple to implement. Thus (i,j)forms a cycle iff i and j eventually trace back to same root. If the roots x and y of the trees for i and j are different, then we join the trees with roots x and y, merely by setting UT(y) to x .

Thus the Unary Tree becomes a cluster of trees as edges are added. The performance of the Test operations depends on how long it takes to find a root.

Performance of UT:

1. **Test: O(height of UT)**
   This may degenerate to O(V) eg: adding edges (5,4), (4,3), (3,2), (2,1) gives 5=>4=>3=>2=>1 path of length 5.)
2. **Union: O(1) - just make one root point to another**

To guarantee satisfactory Height bound for UT use:

Small-Large Rule: Add tree with fewer vertices under root of tree with more vertices.

Requires: maintain count of number of vertices in roots.

Claim: SL Rule guarantees UT height =O(log V)

Proof:

First make some simple transformations in claim statement:

If order <= V Then ht <= $\log_2(V)$ is same as:

If ht > $\log_2(V)$ Then order > V is same as:

If ht > k Then order > 2^k

We will work with the last statement for convenience.

The proof is by induction on h.

Assume that for k < h: If ht > k Then order > 2^k

Prove that for k = h: If ht = k Then order > 2^h

Let T be the least order tree of height h constructed according to the SL Rule.

Thus T = X U Y, where X and Y are the UT's that were combined to form T. Assume X <= Y.

Since both X and Y < T, then by the definition of T, ht(X) and ht(Y) < h.

Note: ht(Y) = h-1: else the tree T formed from X U Y would have height <= h-1 (contrary to supposition).

Thus since: ht(Y) = h-1: by induction: Y => $2^{(h-1)}$.

Since: X > Y, then X => 2^{h-1} also.

Thus: T = X U Y => 2^{h-1} + 2^{h-1} => 2^{h} , as required.