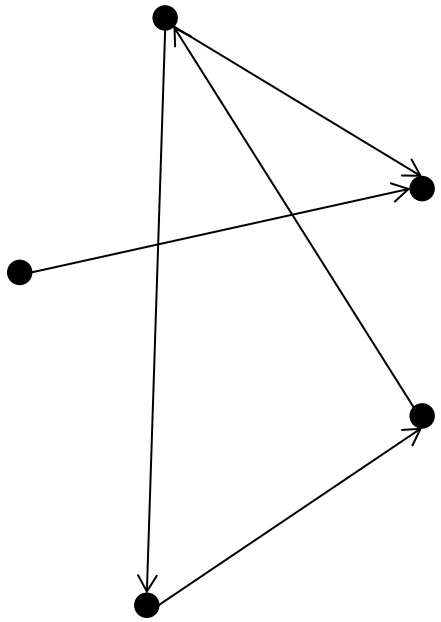
# Incremental cycle detection

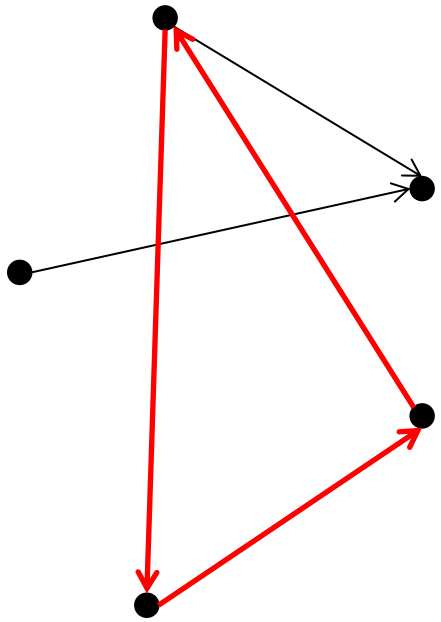
# The problem

- Start with n singleton vertices
- Add m edges one by one
- If a cycle is created report and stop

# Example



# Example

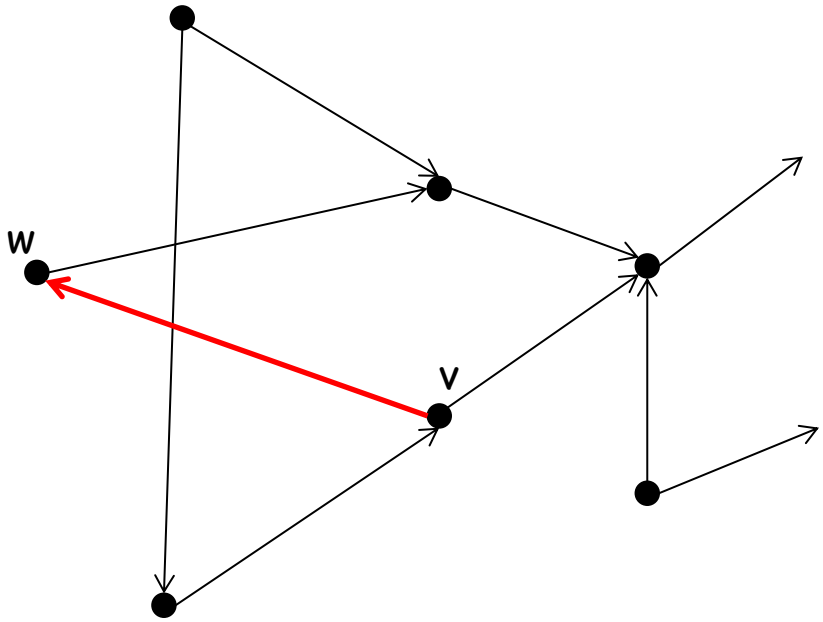


# Straightforward solution

When an edge (v,w) is added, search (DFS,BFS) from w.

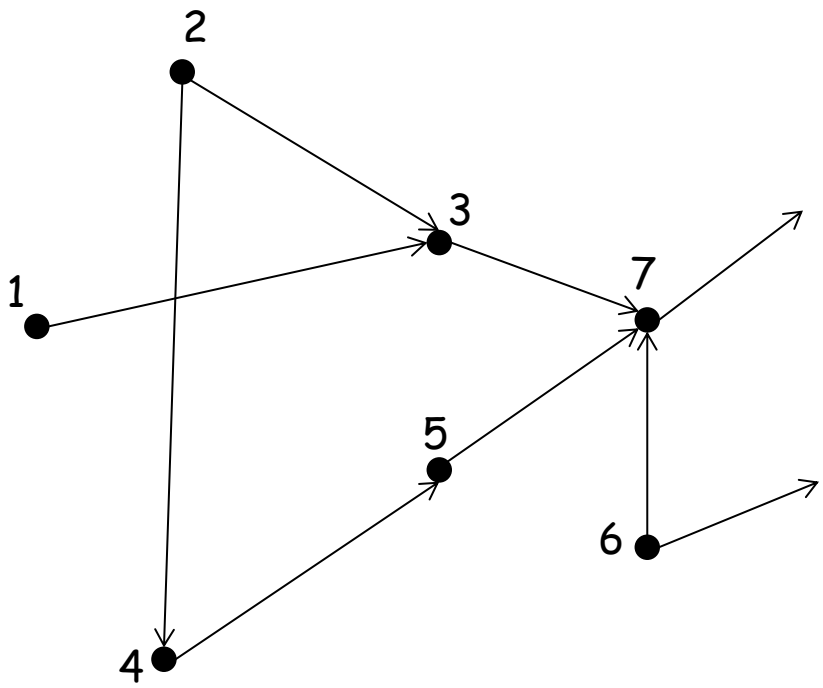
If v is reached report a cycle

Total work $O(m^2)$



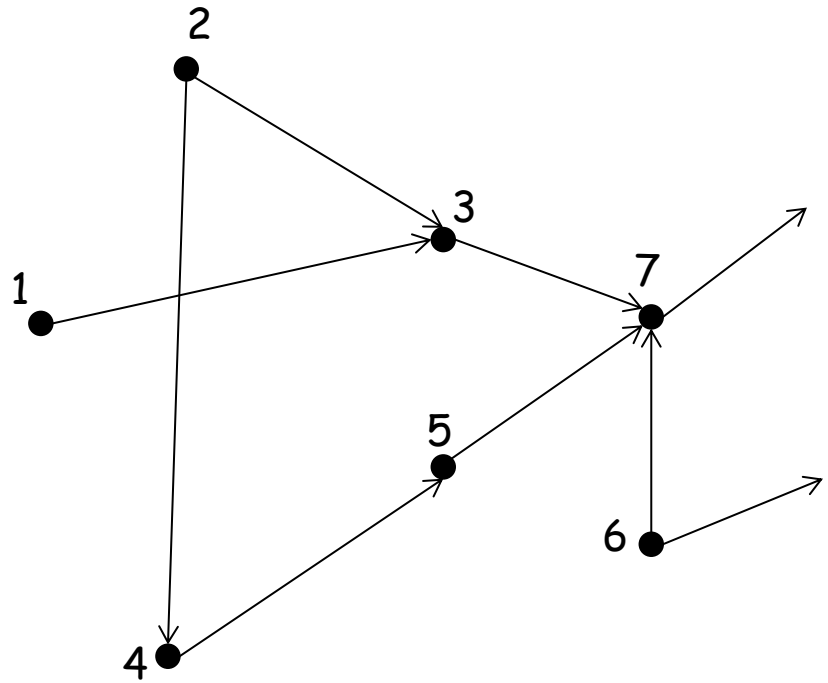
# Topological order

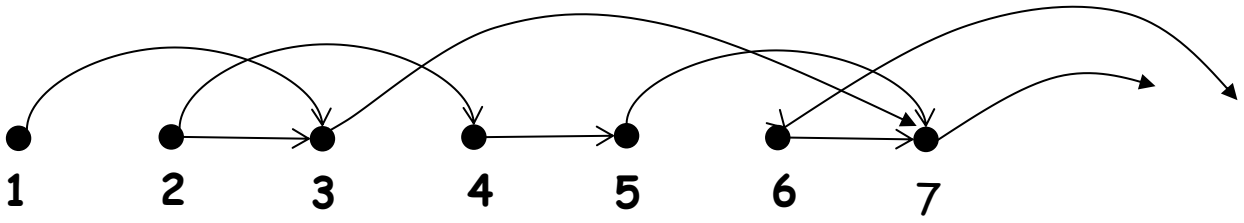
Maintain a topological order of the graph



# Topological order

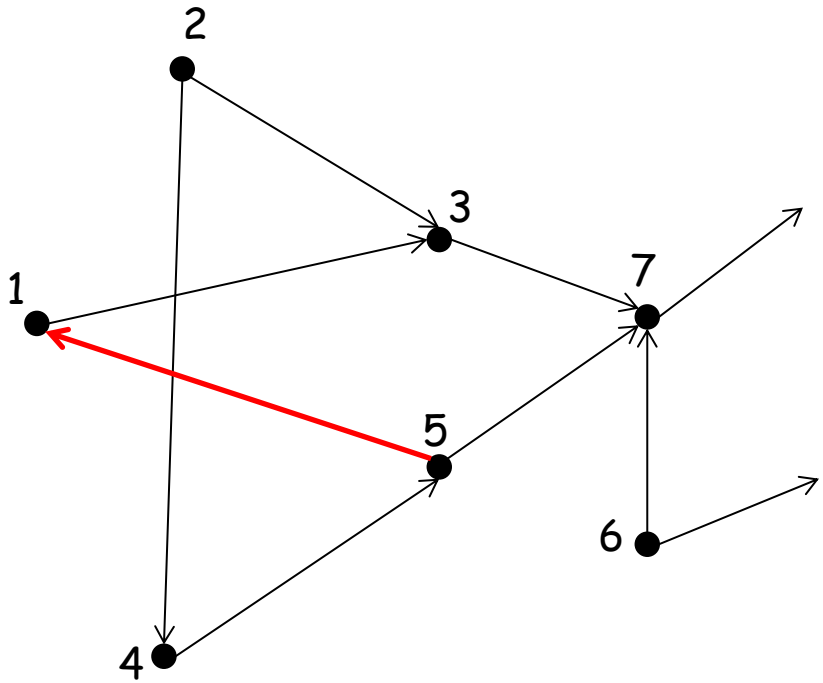
Maintain a topological order of the graph

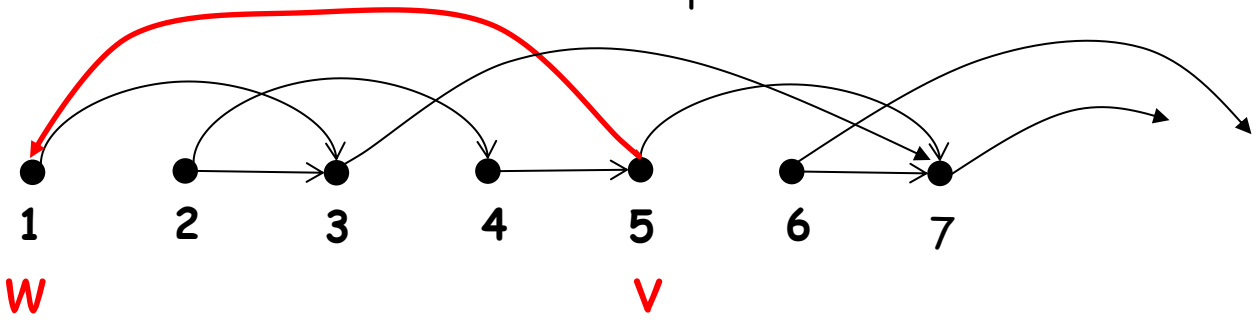


# Topological order

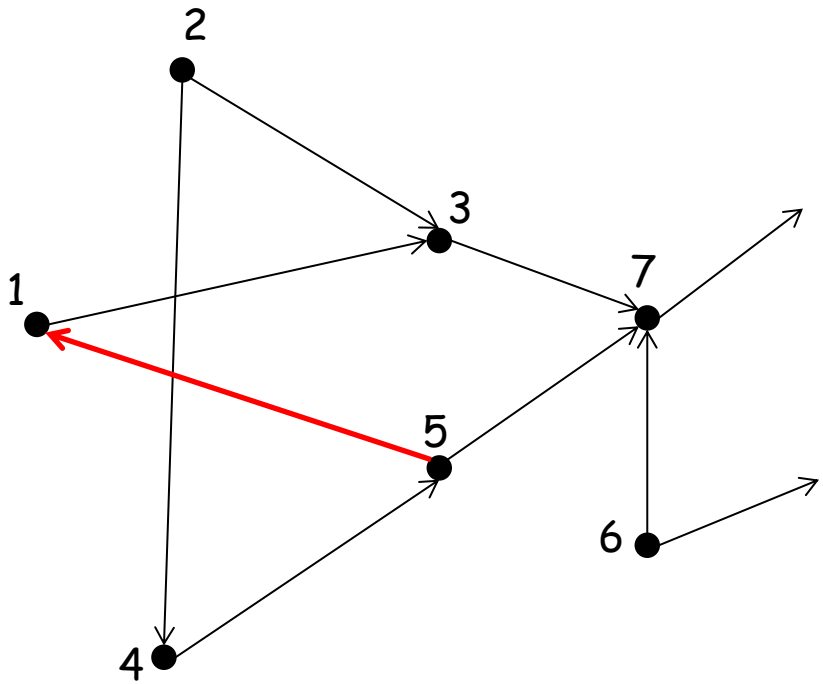
Use the topological to prune the search

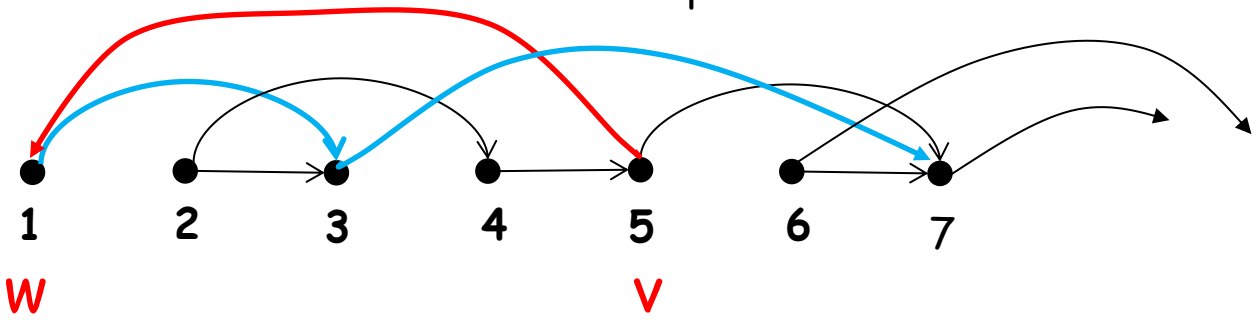


# Topological order

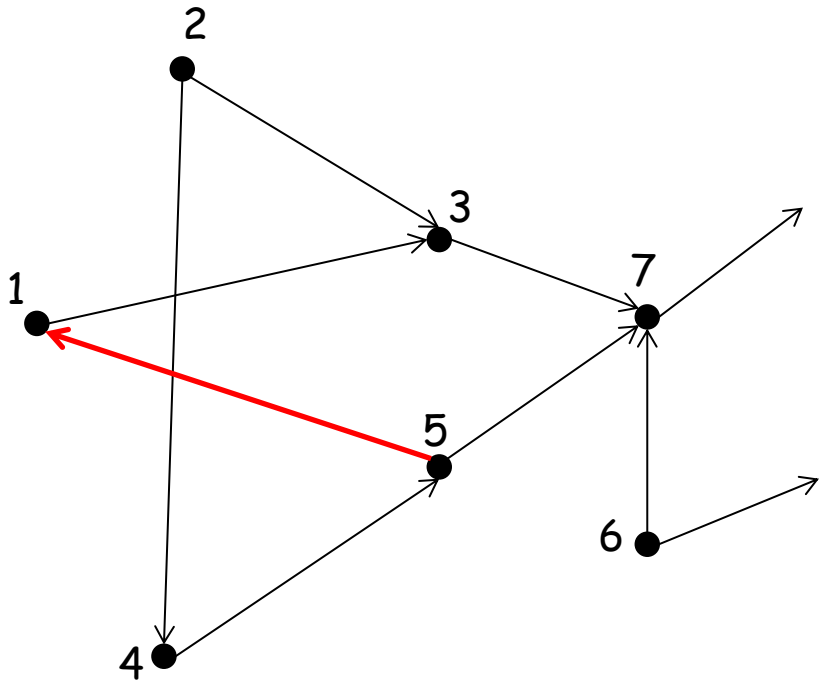
Use the topological to prune the search

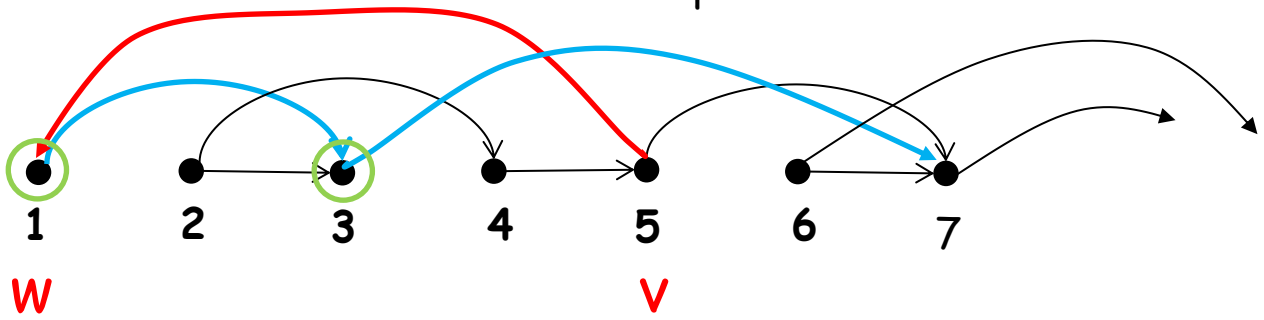


# Topological order

Need to fix the topological order

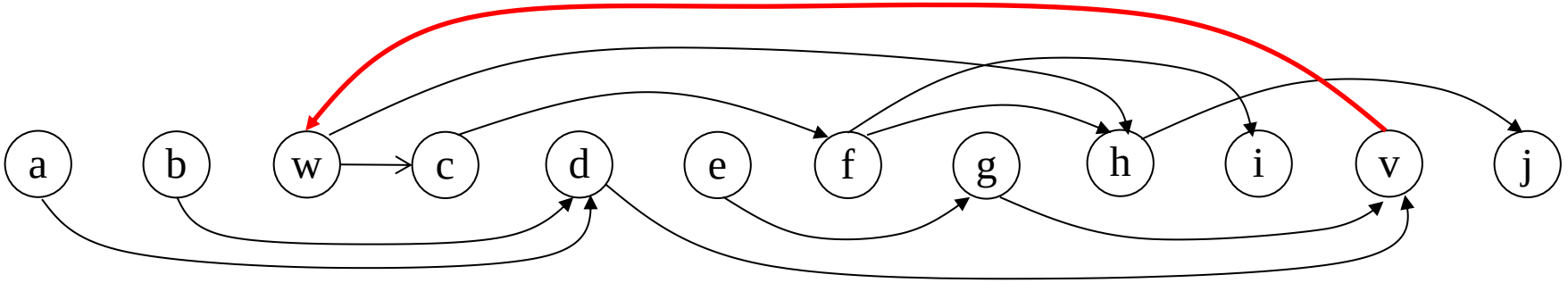


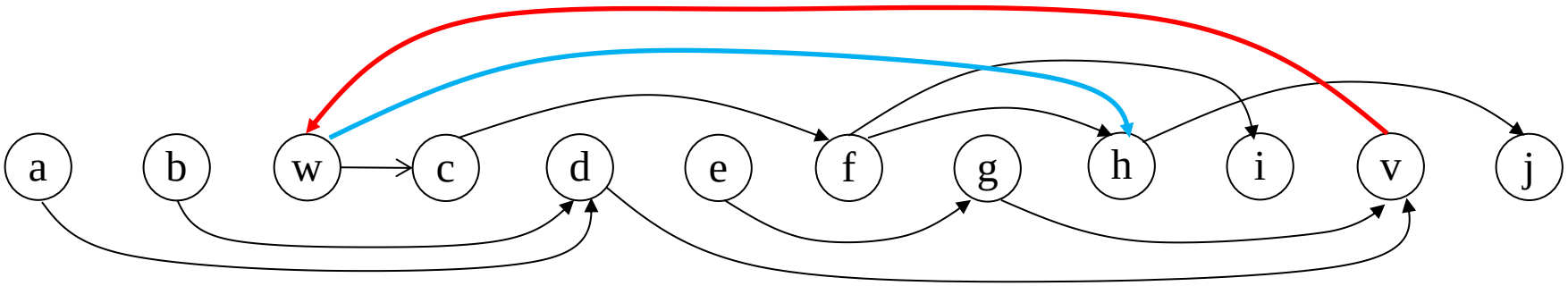
# Order maintenance

- Keep the topological order in an order maintenance data structure
- Move the vertices between v and w including w right after v, keeping them in topological order

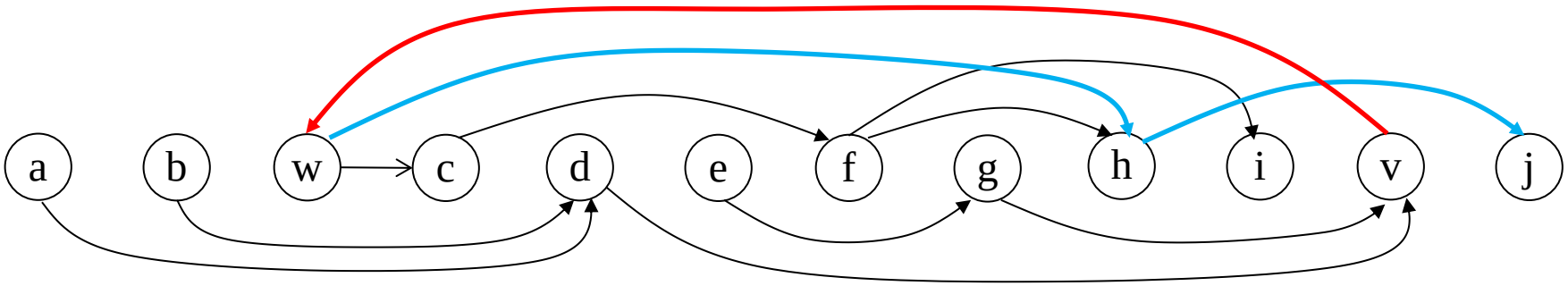
# Another example



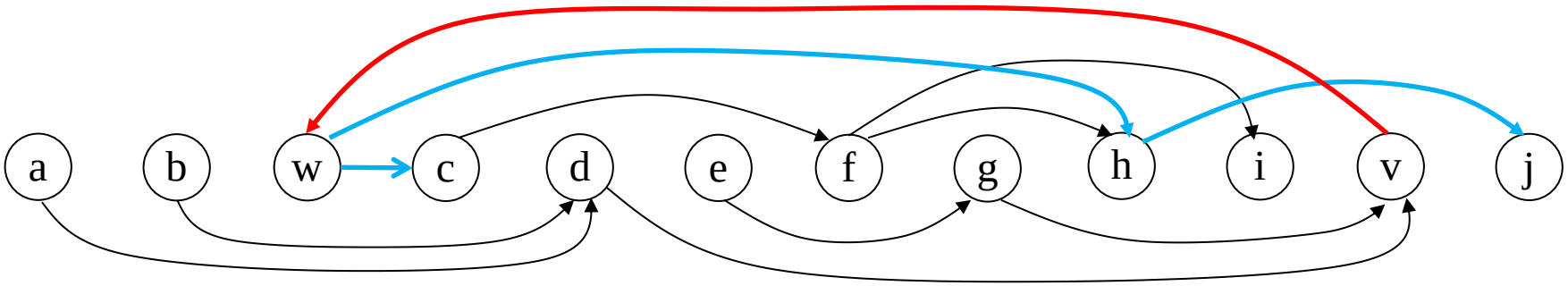
# Another example



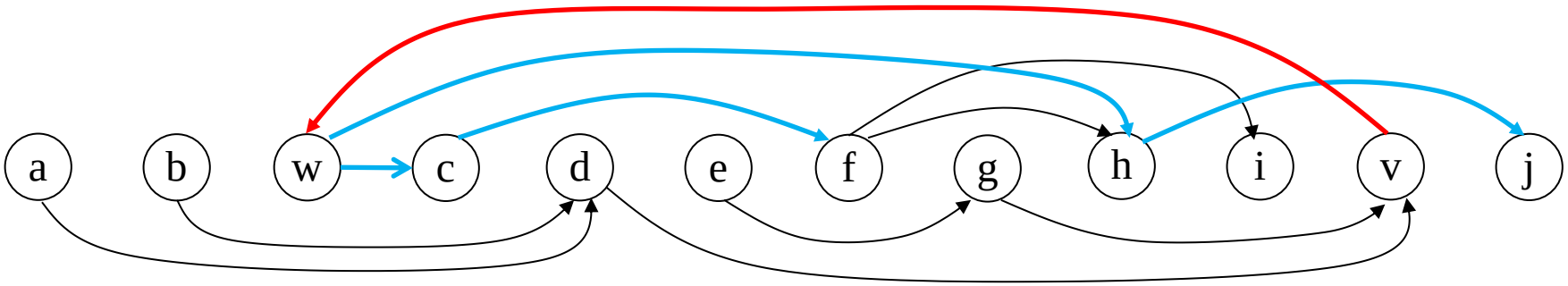
# Another example



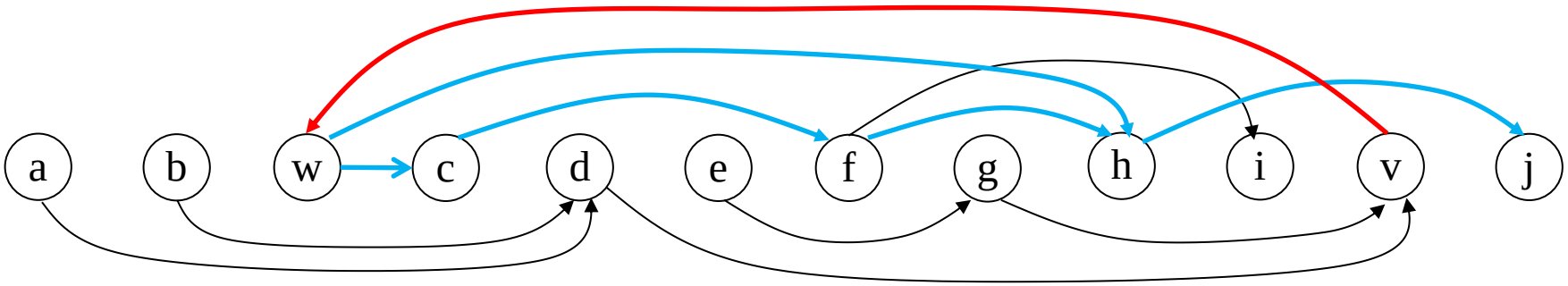
# Another example



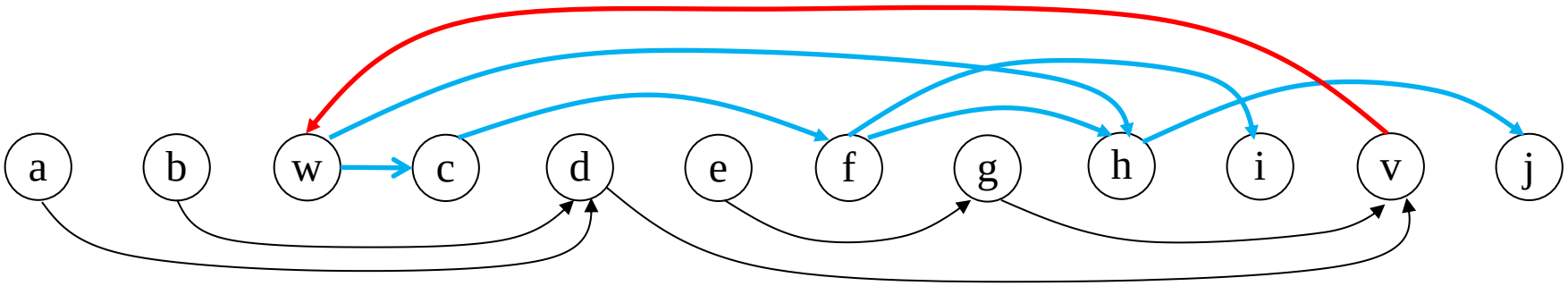
# Another example



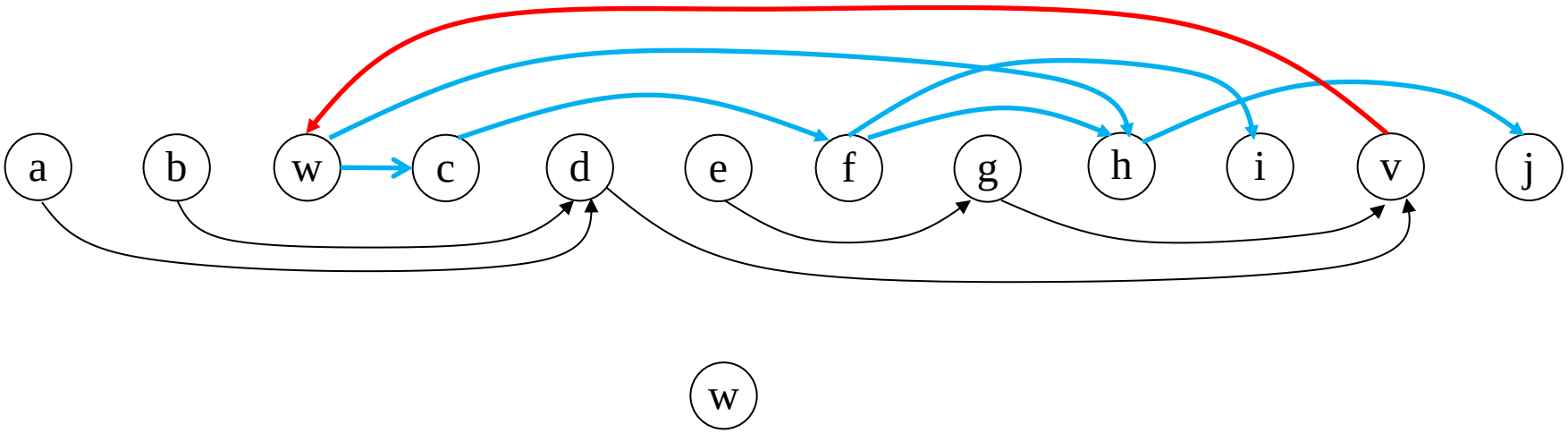
# Another example



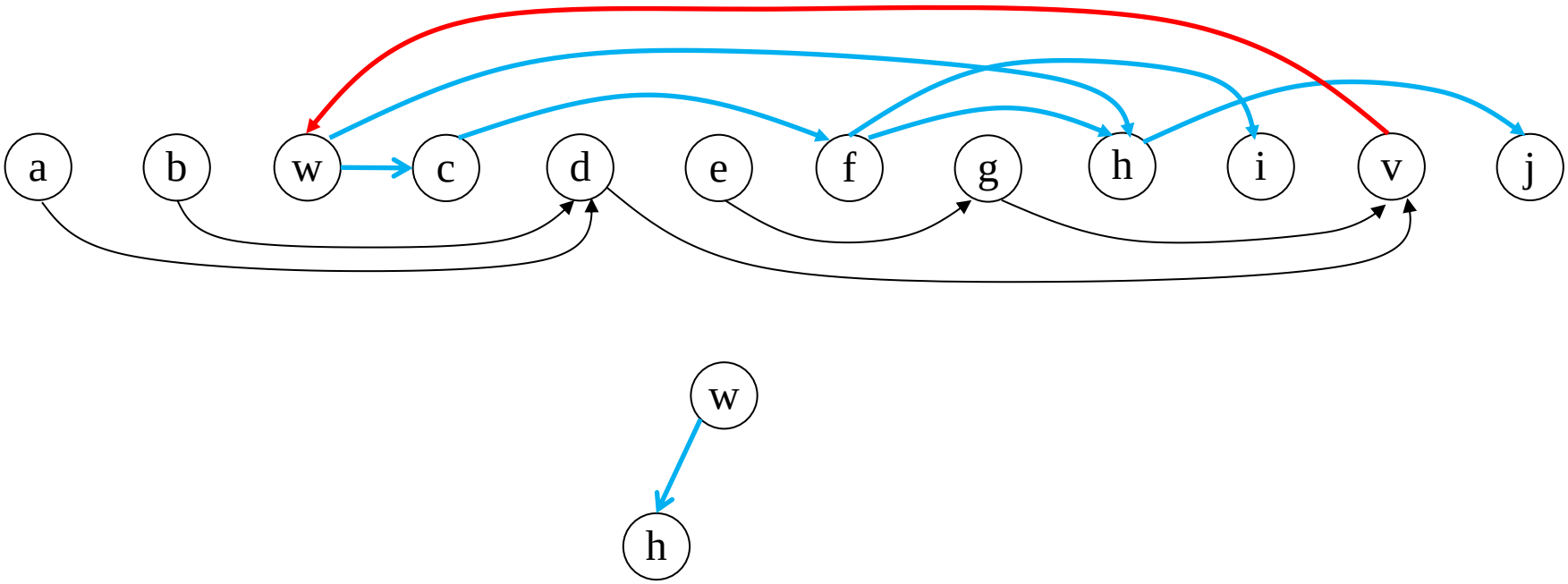
# Another example



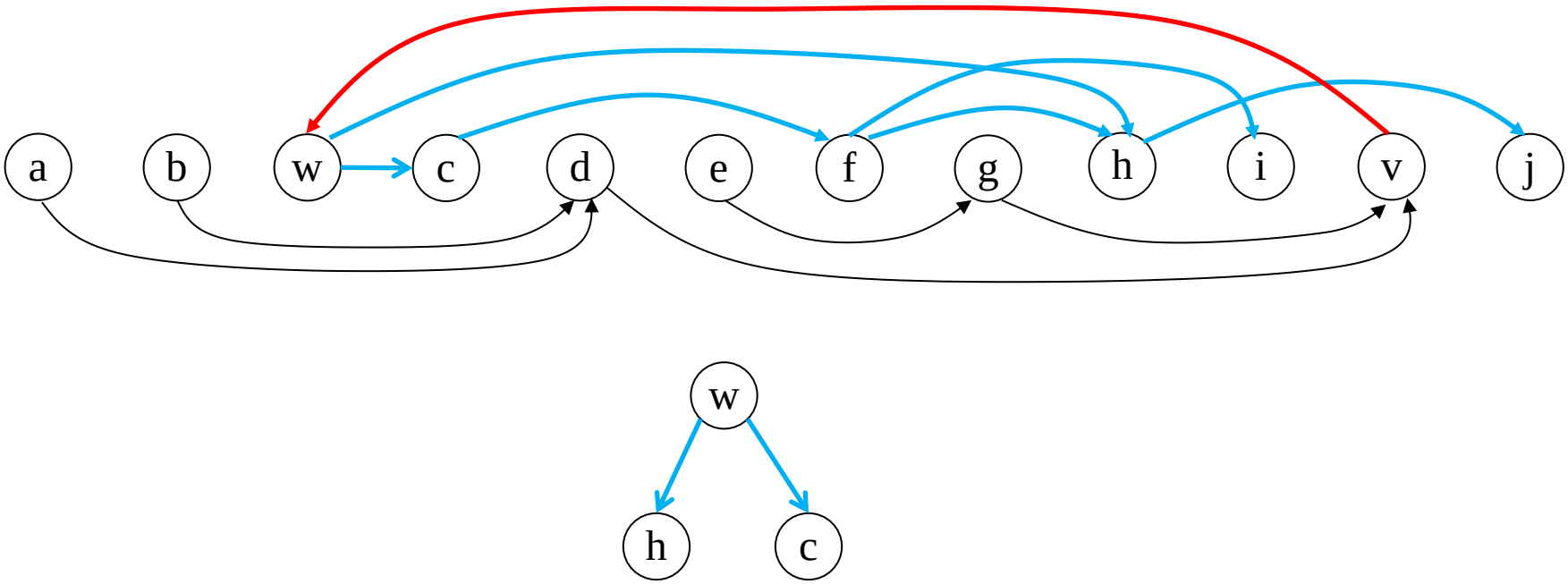
# Another example



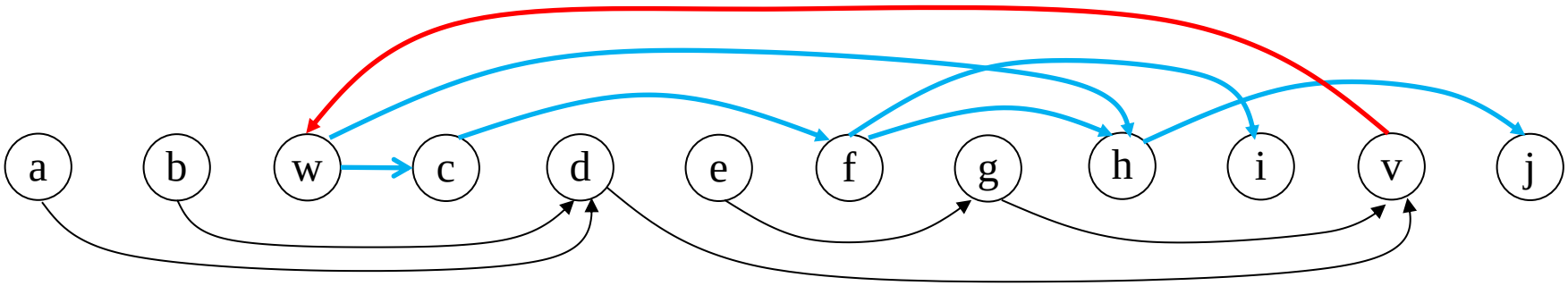
# Another example

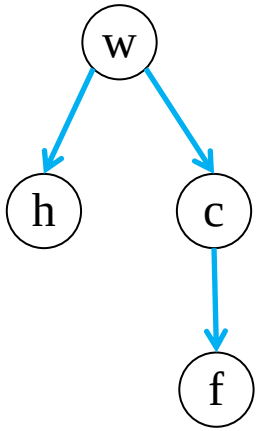


# Another example

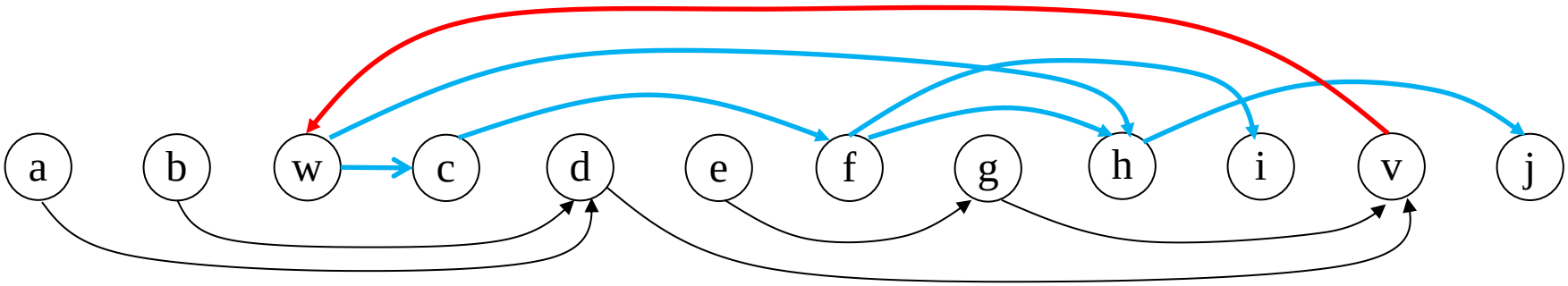


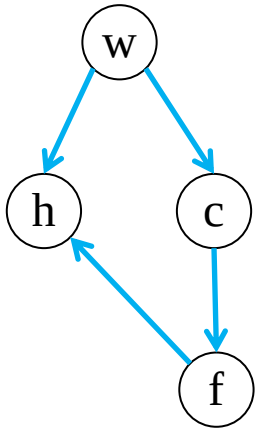
# Another example



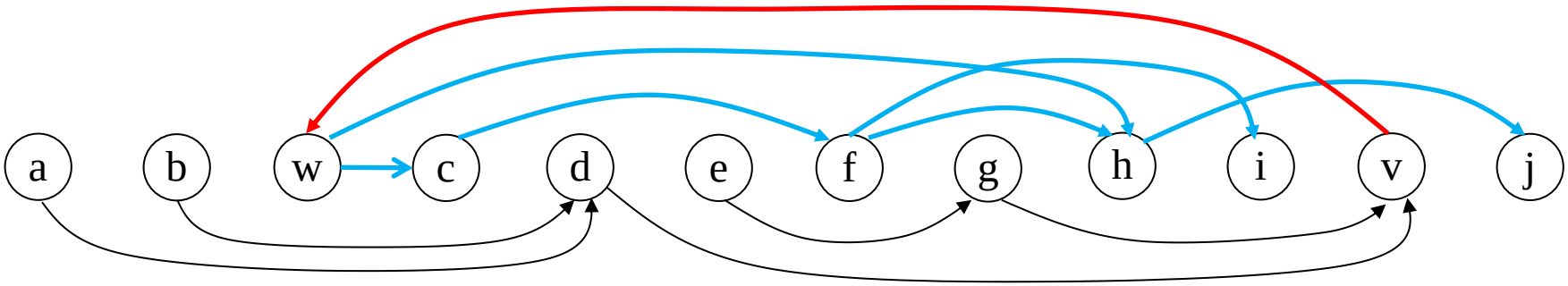



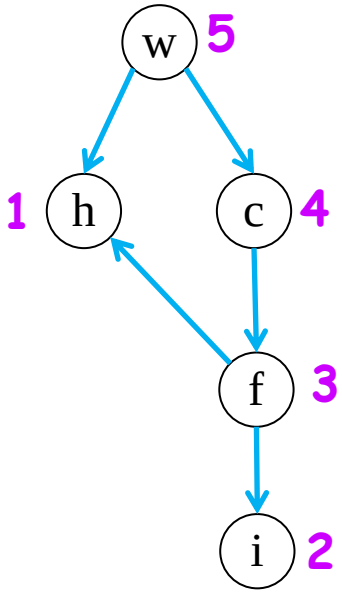
# Another example



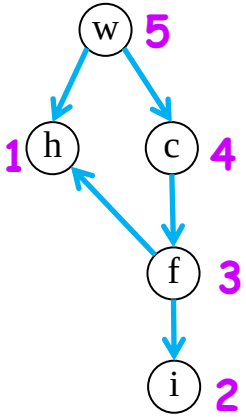



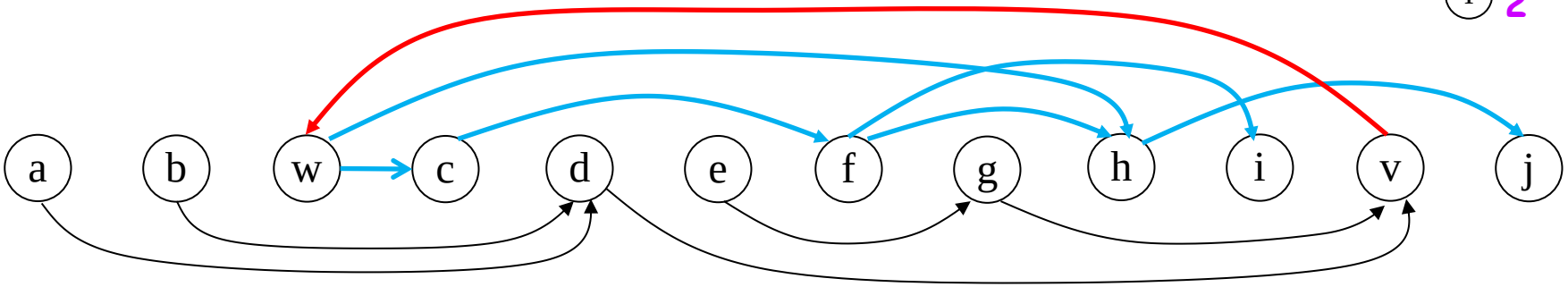
# Another example



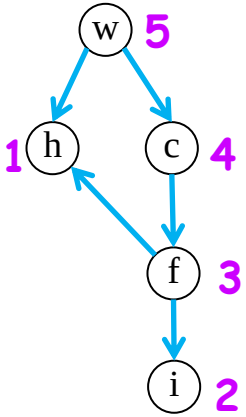



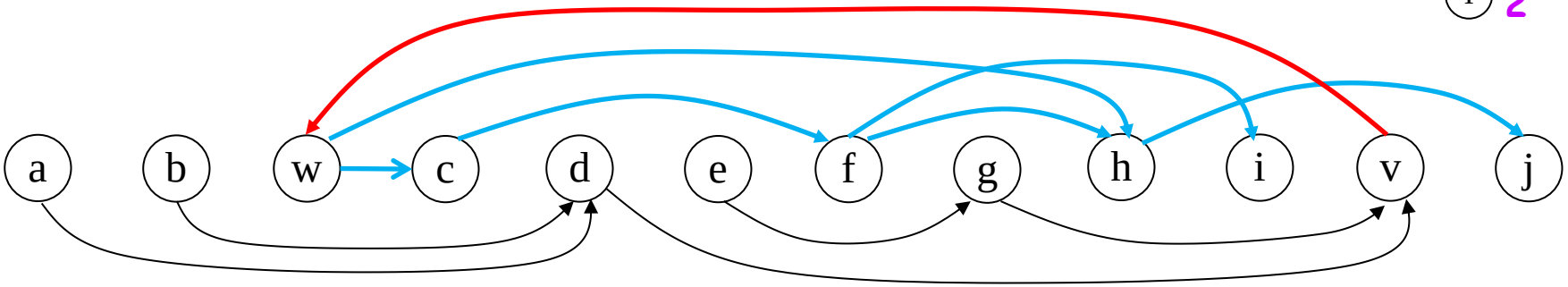
# Another example

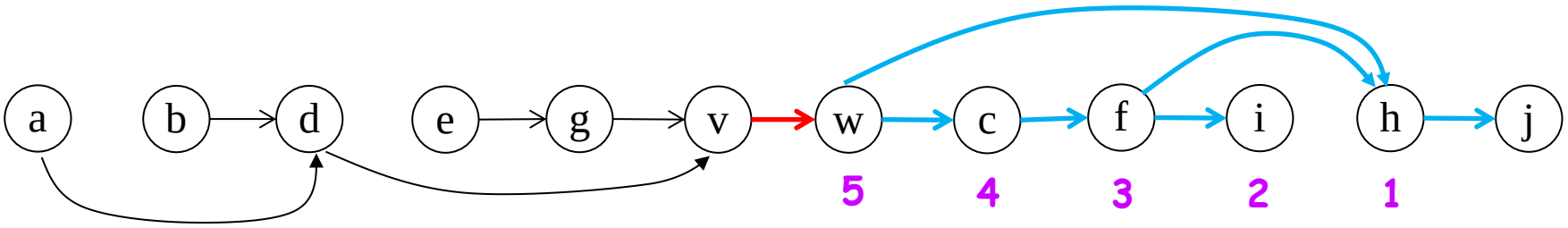


# Another example

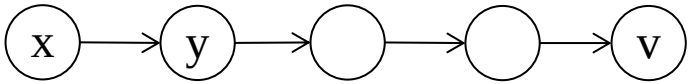


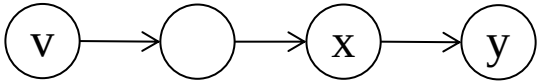



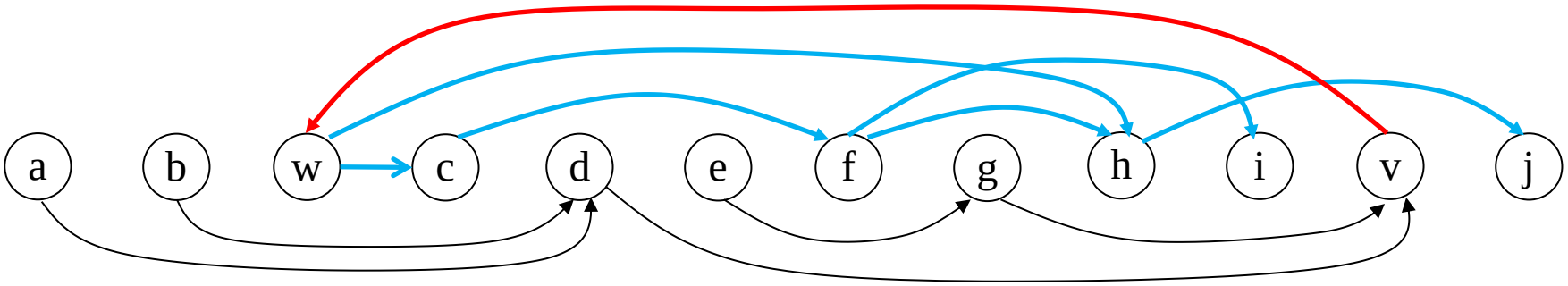
# Analysis

- Define an arc (x,y) and a vertex v to be related if there is a path containing them both



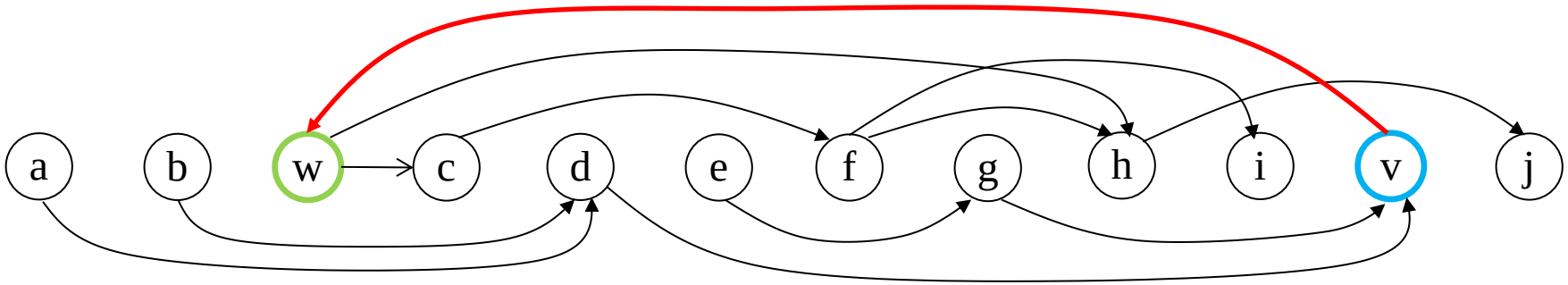
# Analysis




For each arc (x,y) traversed by the search, v was not related to (x,y) before the search, and becomes related to the arc after the search

➔ O(nm) time

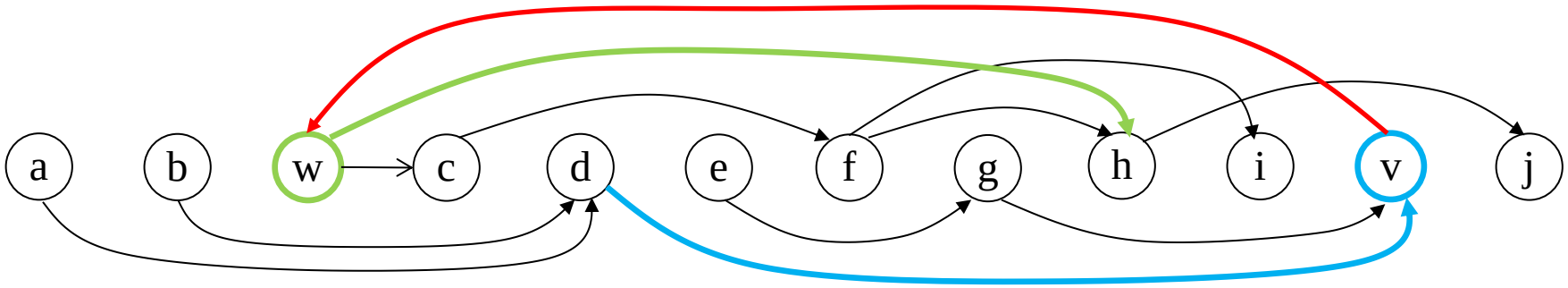
# Two-way search




F={w}

B={v}

# Two-way search

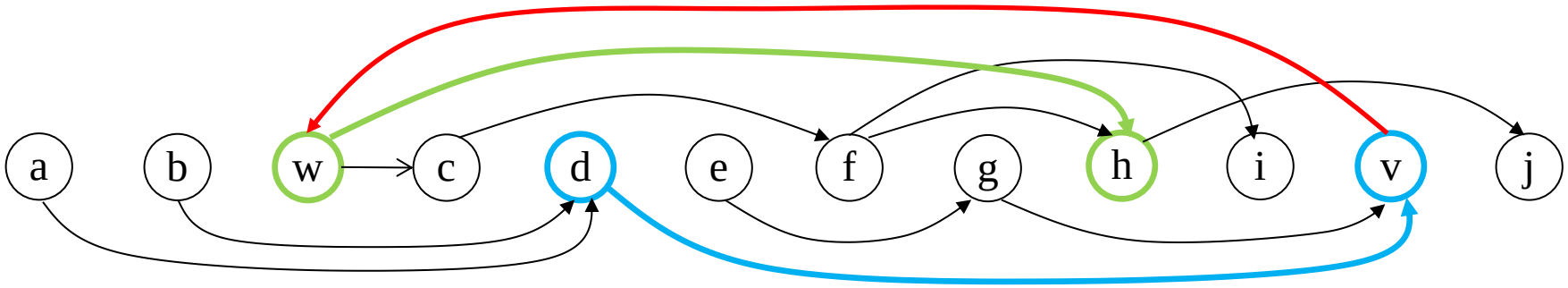



F={w}

B={v}

Traverse compatible pairs of arcs (u,v) and (x,y) such that u < y

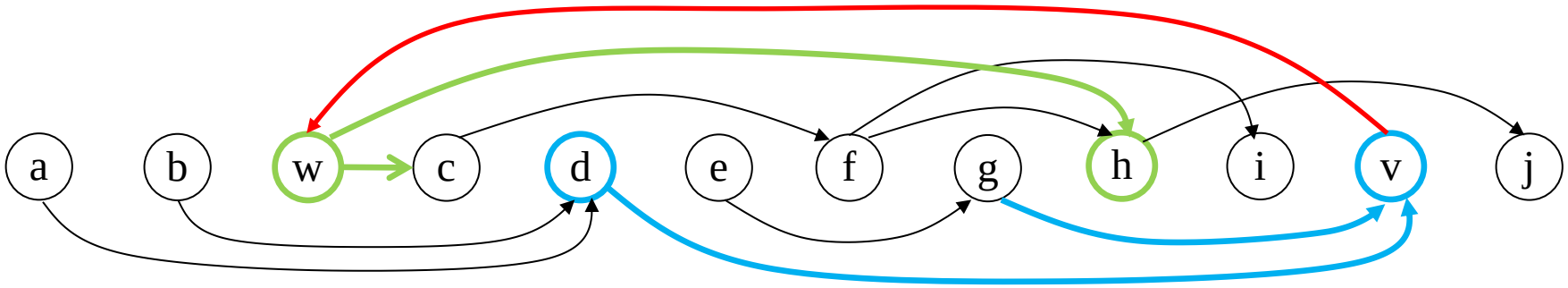
# Two-way search




F={w,h}

B={v,d}

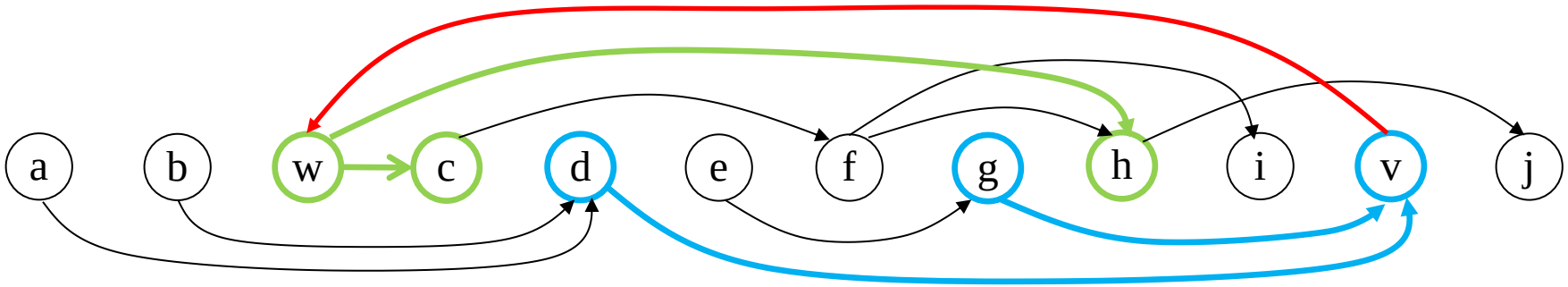
# Two-way search




F={w,h}

B={v,d}

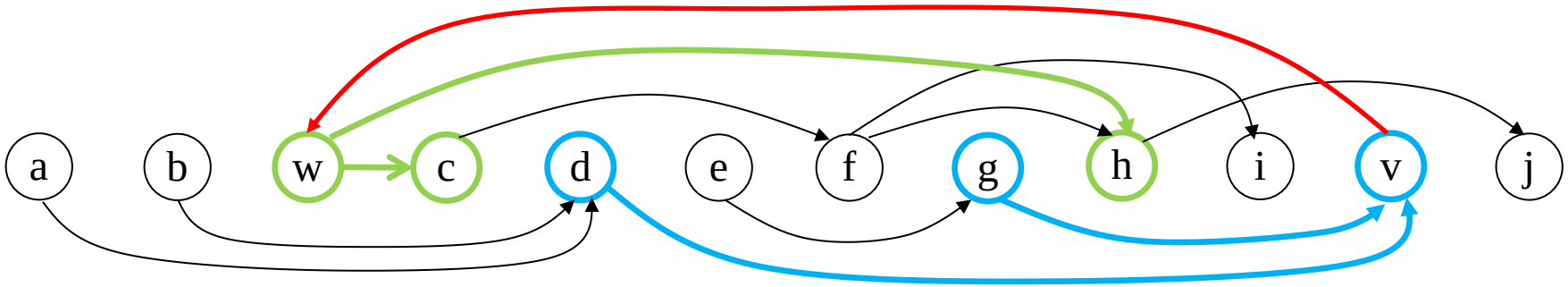
# Two-way search




F={w,h,c}

B={v,d,g}

# Two-way search

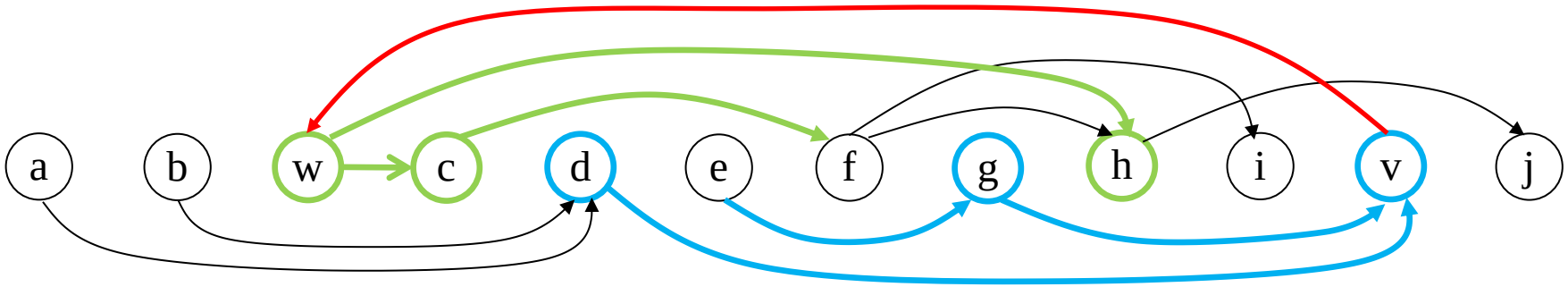



F={w,c,h}

B={v,g,d}

Order **F** from left to right and **B** from right to left

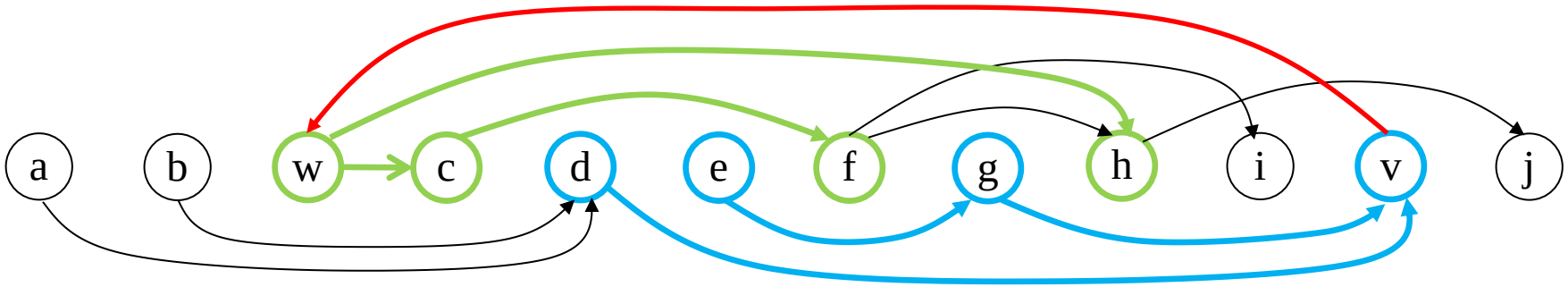
# Two-way search




F={w,c,h}

B={v,g,d}

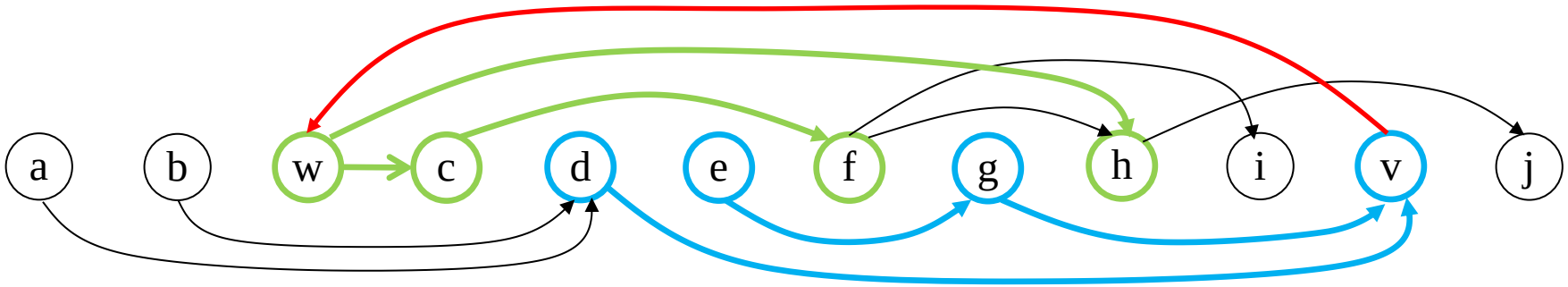
# Two-way search




F={w,c,f,h}

B={v,g,e,d}

# Two-way search

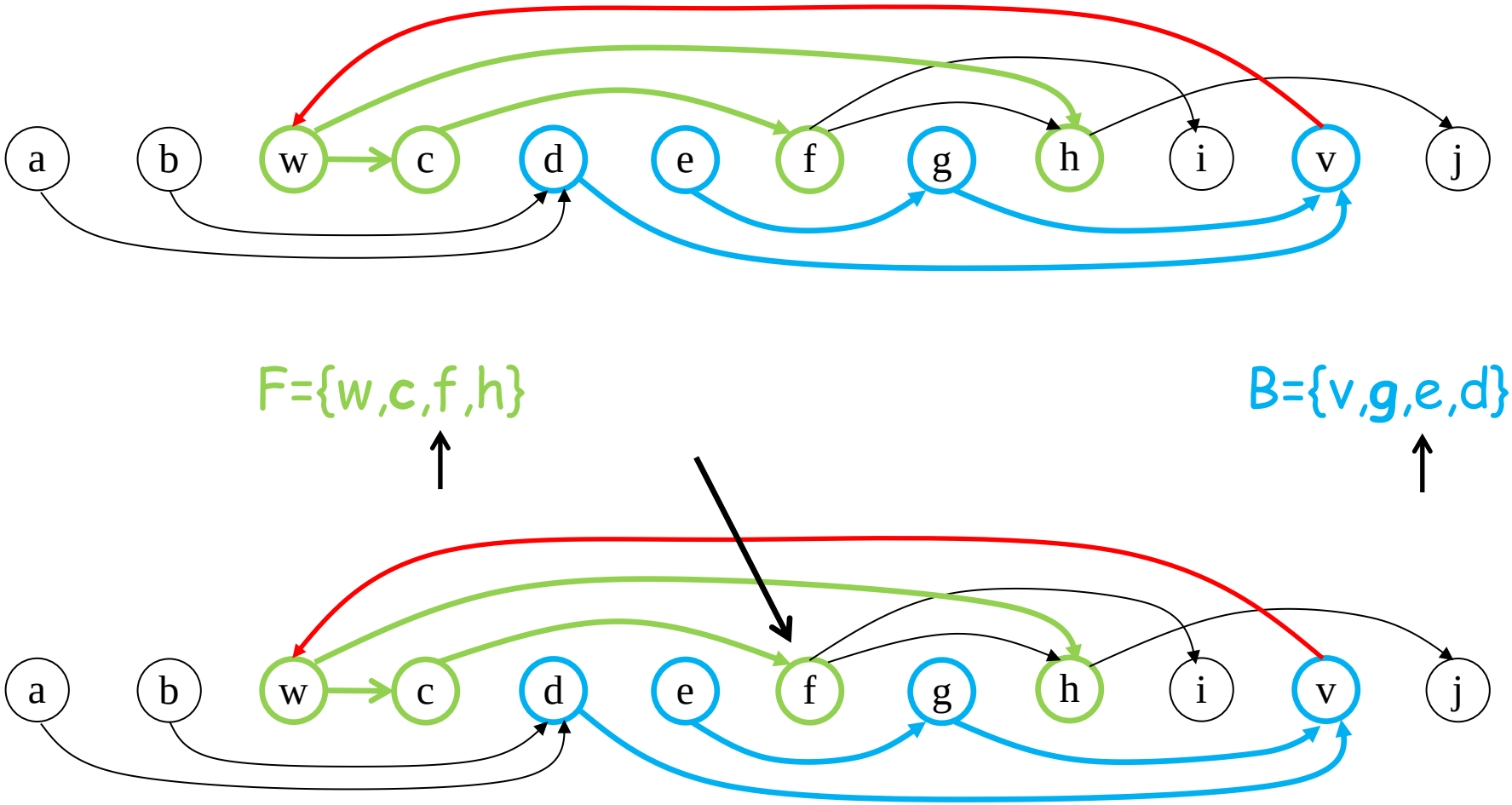



F={w,c,f,h}

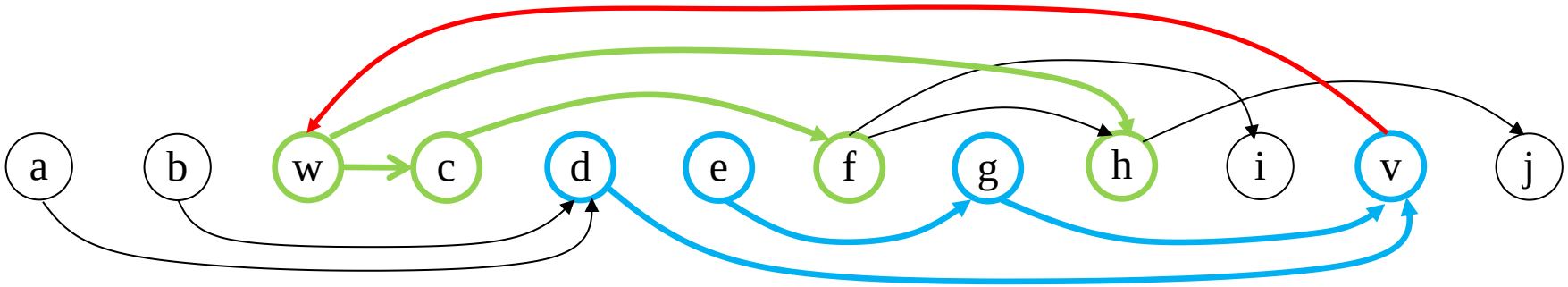
B={v,g,e,d}

Stop when there are no more compatible pairs of arcs, or when the two searches collided, in which case a cycle is detected

# Reordering

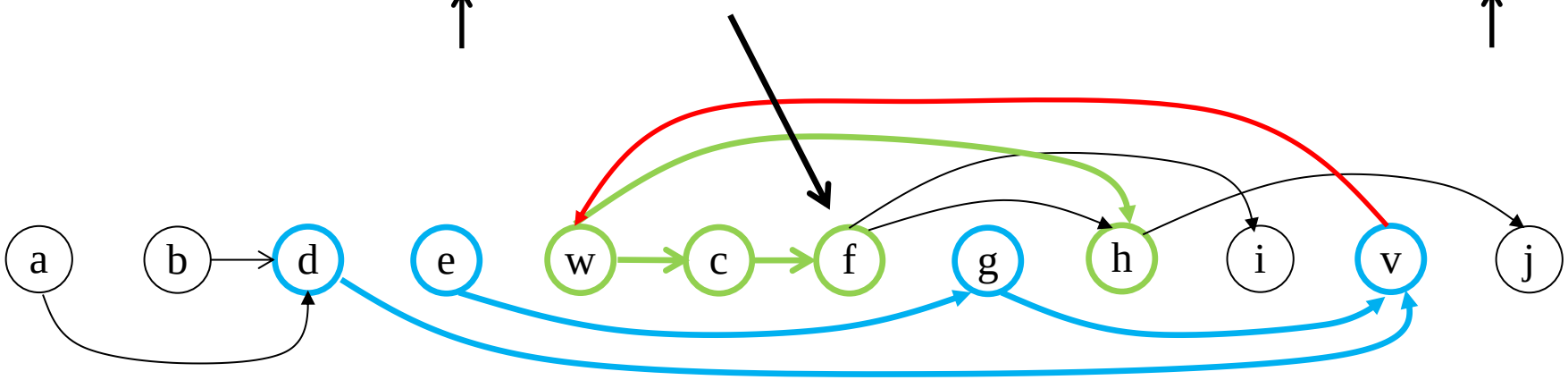


# Reordering

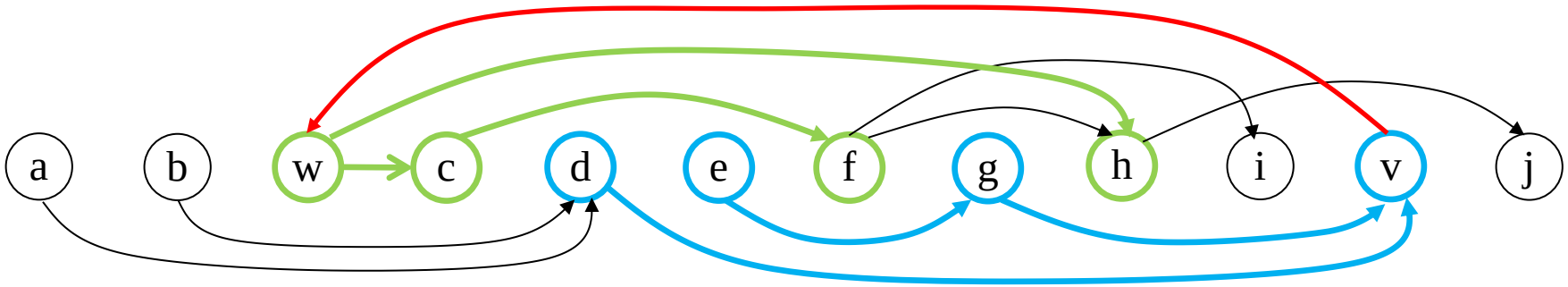



F={w,c,f,h}

B={v,g,e,d}

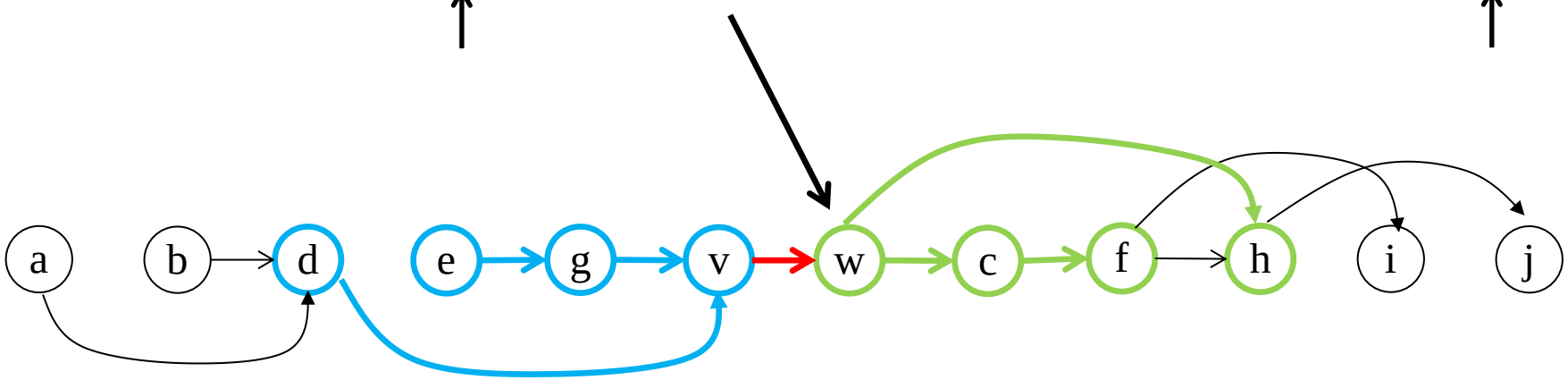


# Reordering

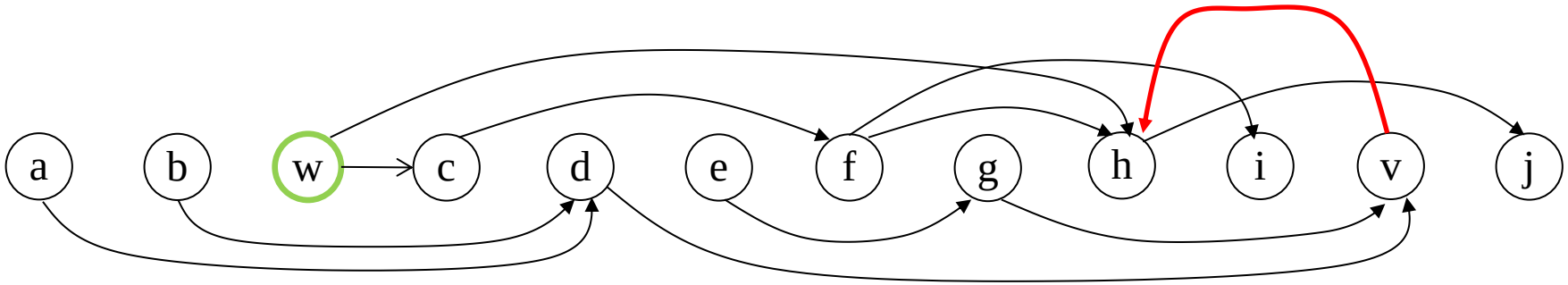



F={w,c,f,h}

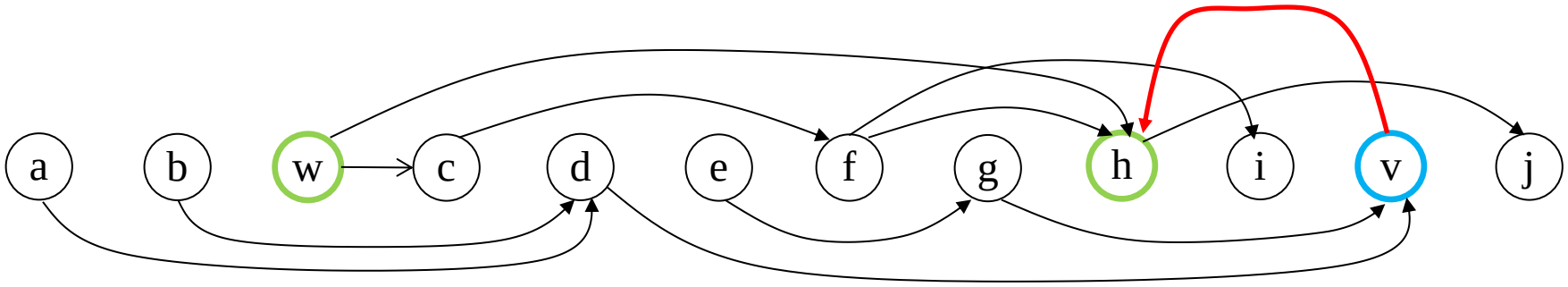
B={v,g,e,d}



# Two-way search

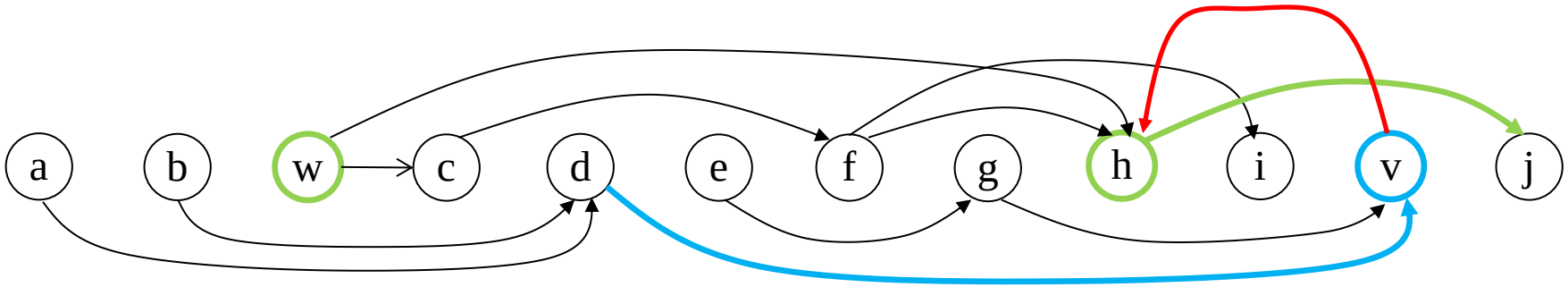


# Two-way search



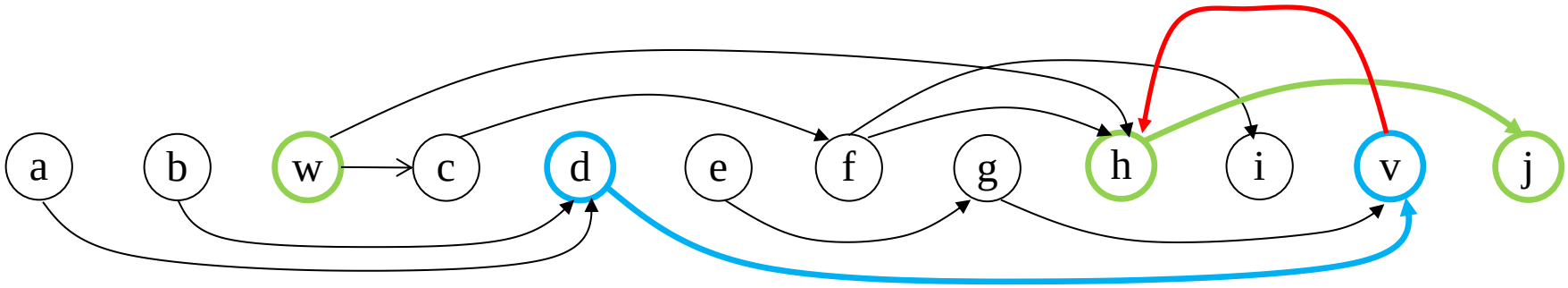

F={h}  B={v}

# Two-way search



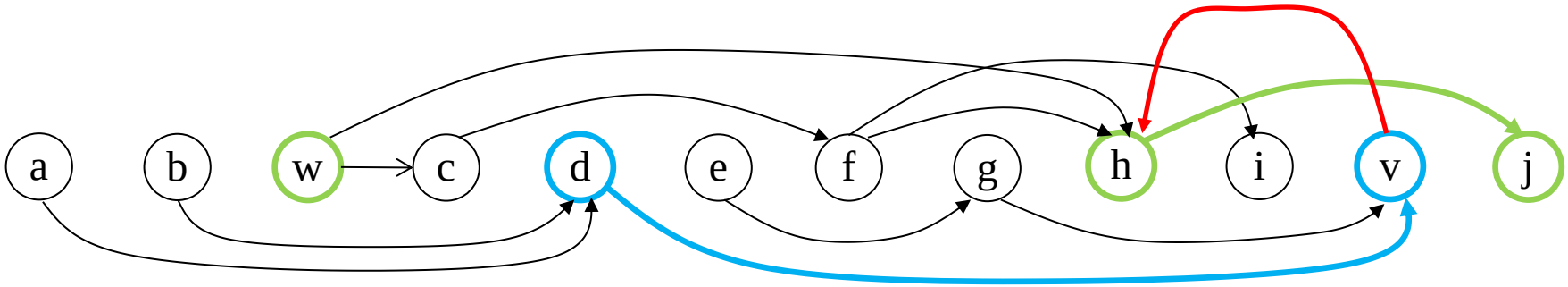

F={h}  B={v}

# Two-way search



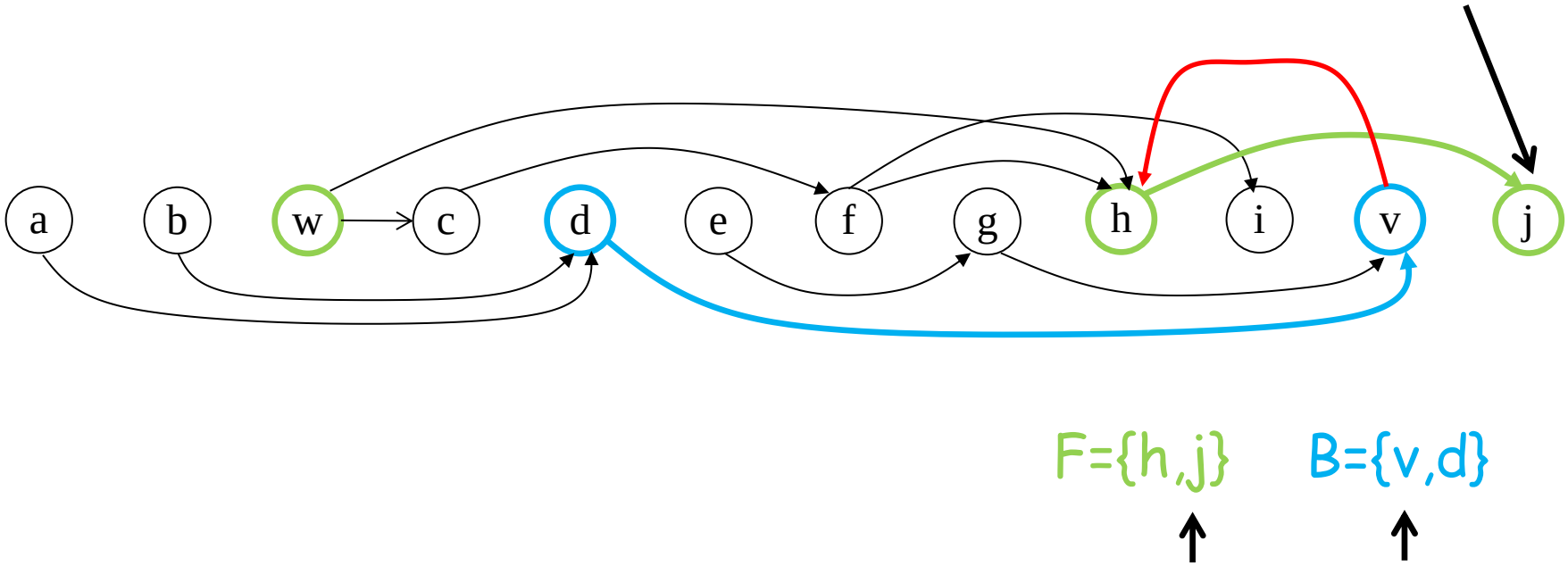

F={h,j}  B={v,d}

# Two-way search




F={h,j}  B={v,d}

# Reordering




F={h,j}  B={v,d}

Since j > v we just do reordering as in the one way version, putting all green guys straight after v

# Correctness

- Case analysis to show that topological order is maintained if a cycle is not created (homework)

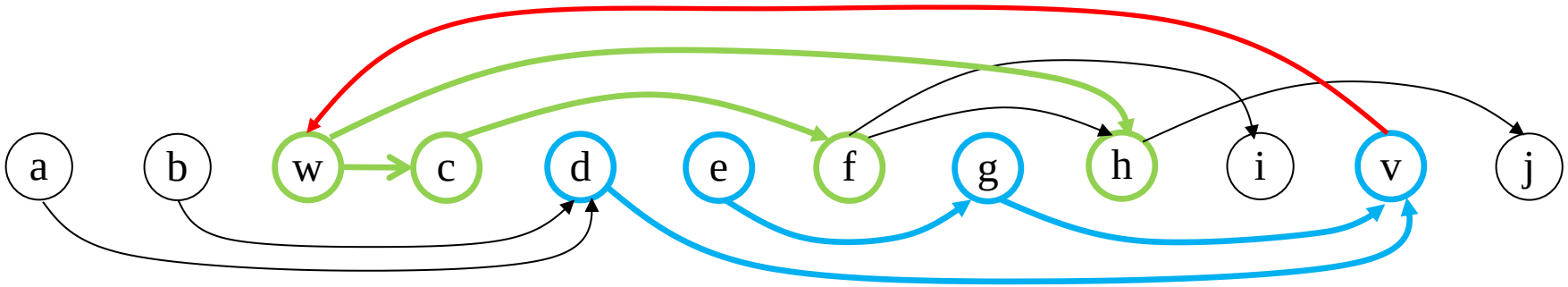
# Efficiency

**Thm: When inserting a sequence of m arcs, two-way search traverses $O(m^{3/2})$ arcs**

Proof: Define (x,y) and (u,v) to be related if there is a directed path containing both of them

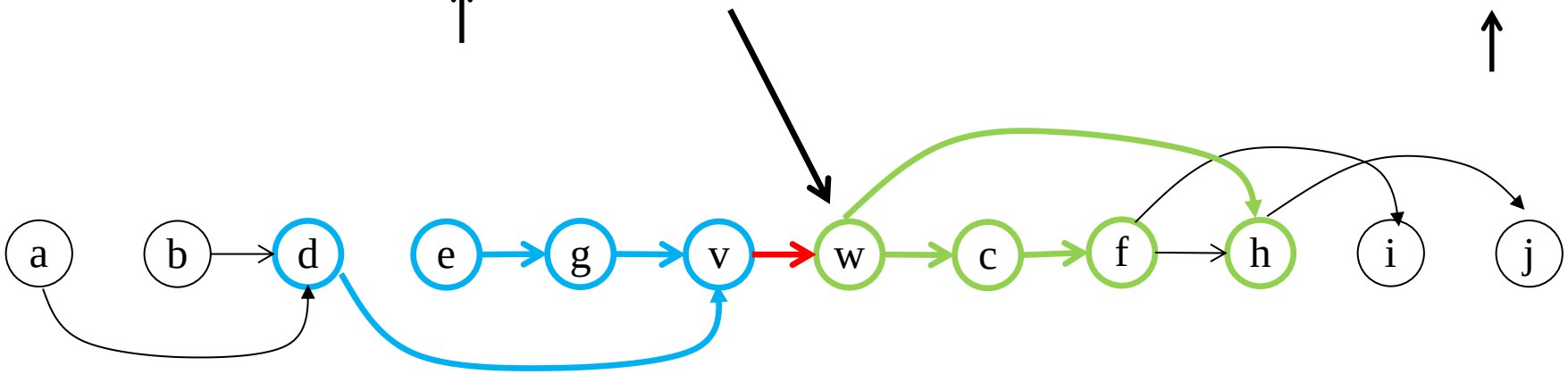
Assuming no cycle is formed. All pairs of green and blue edges that we traversed are not related before the insertion of (v,w) and related after

# All pairs of green-blue arcs become related




F={w,c,f,h}

B={v,g,e,d}



# Efficiency

**Thm: When inserting a sequence of m arcs, two-way search traverses $O(m^{3/2})$ arcs**

Proof (cont): We insert m arcs. Insertion i triggers a search of $k_i$ pairs of arcs.

Small insertion $\equiv k_i \leq m^{1/2}$

Large insertion $\equiv k_i > m^{1/2}$

Total arc scanned by small insertions = $m \times m^{1/2} = m^{3/2}$

# Large insertions

Proof (cont): Let $j_1, j_2, \ldots$ be the large insertions
Insertion $j_i$ contributes $(k_{j_i})^2$ new related pairs. So

$$\sum_i (k_{j_i})^2 \leq \binom{m}{2}$$

But

$$m^{1/2} \sum_i (k_{j_i}) \leq \sum_i (k_{j_i})^2$$

So

$$\sum_i (k_{j_i}) \leq m^{3/2}$$

# Conclusion

Thm: When inserting a sequence of m arcs, two-way search traverses $O(m^{3/2})$ arcs

All searches traverse $O(m^{3/2})$ pairs

It takes $O(\log(n))$ time to traverse a pair (we have to maintain F and B as heaps..)

Total time ➔ $O(m^{3/2}\log(n))$

# One way search for dense graphs

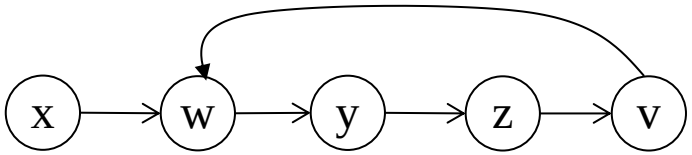
- For each vertex u, maintain k(u) ≤ size(u) (= # predecessors of u)

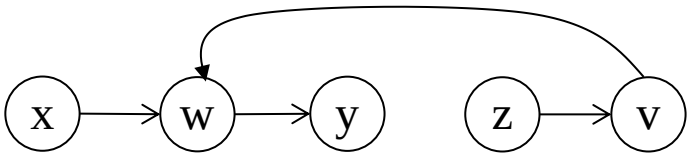
- For each arc (u,v), k(u) < k(v)

➔ k(u) is a weak topological numbering

# Inserting an arc (v,w)

- If k(v) < k(w) do nothing, otherwise

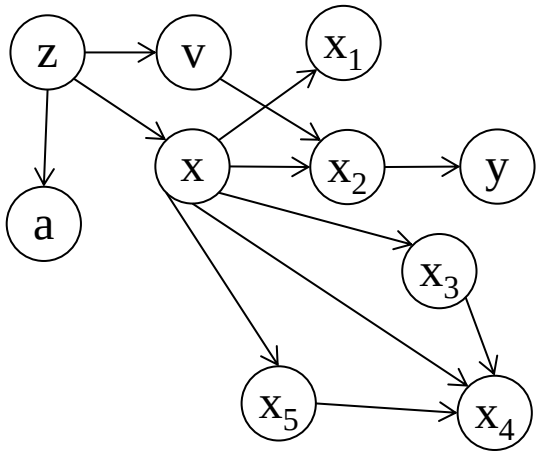



- If a cycle is formed we need to detect it. As we search for it we increase k() values of vertices

# Avoid looking at large neighbors

- Each vertex maintains a heap of its outgoing neighbors

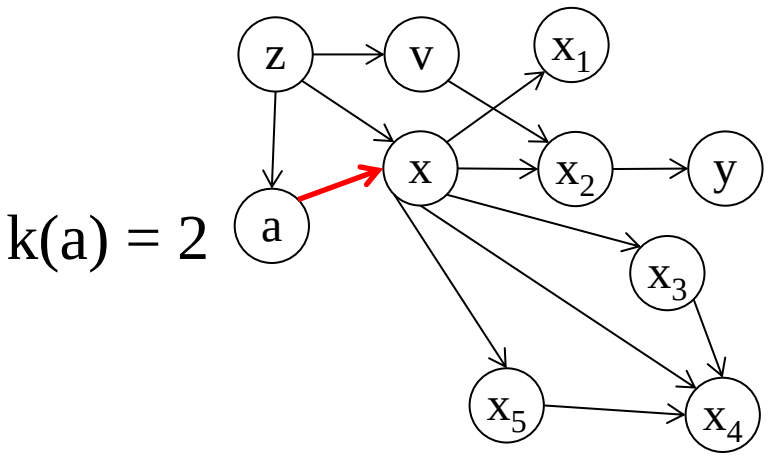



- At x we have $x_1, x_5, x_3, x_2, x_4$ in a heap with keys:

$k(x_1) = 2$, $k(x_5) = 3$, $k(x_3) = 3$, $k(x_2) = 4$, $k(x_4) = 4$

# "Pay" for search by increasing k() values

- k(x) increases, say from 2 to 3

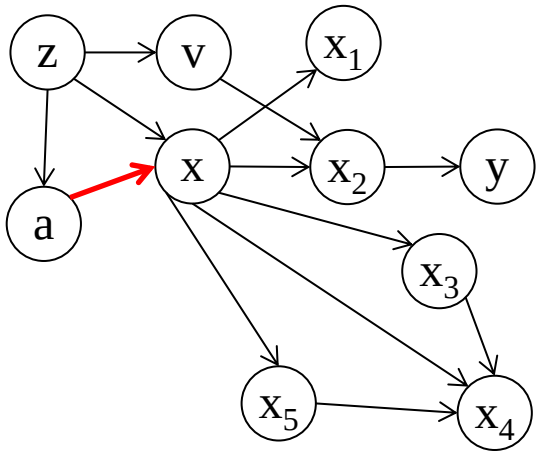



$k(x_1) = 2 \rightarrow 4$
$k(x_5) = 3 \rightarrow 4$
$k(x_3) = 3 \rightarrow 4$
$k(x_2) = 4$
$k(x_4) = 4$

- Need to update $k(x_1), k(x_5), k(x_3)$ to $1+k(x)$...

# "Pay" for search by increasing k() values

- k(x) increases, say from 2 to 3

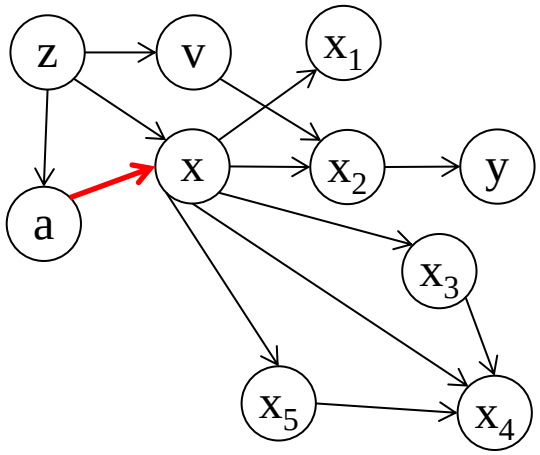



$k(x_1) = 2 \rightarrow 4$
$k(x_5) = 3 \rightarrow 4$
$k(x_3) = 3 \rightarrow 4$
$k(x_2) = 4$
$k(x_4) = 4$

- Using the heaps we avoid looking at neighbors that need not change and cannot be on a cycle

# Update backwards ?

- When k(x) increases, do we increase its key in the heaps of its in-neighbors ?

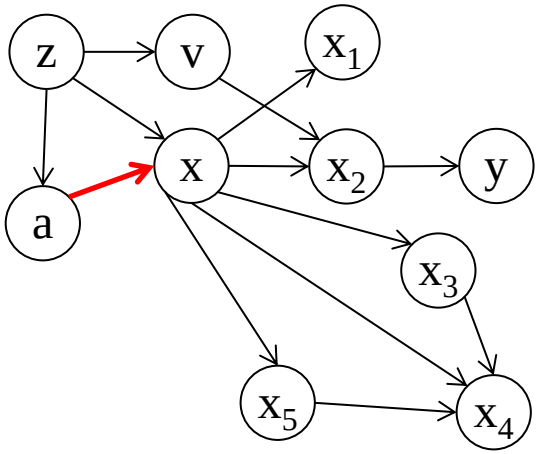



$k(x_1) = 2 \rightarrow 4$
$k(x_5) = 3 \rightarrow 4$
$k(x_3) = 3 \rightarrow 4$
$k(x_2) = 4$
$k(x_4) = 4$

- Here $x_1$, $x_3$, $x_5$ have no other in-neighbors, but

# Without backward updates

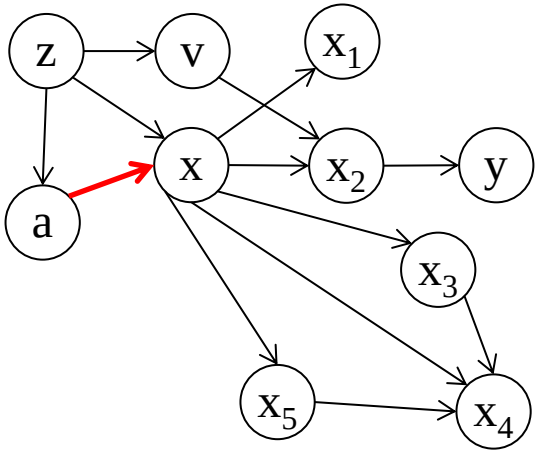
- When $x_5$ updates $k(x_4)$ to 5




$k(x_1) = 4$
$k(x_5) = 4$
$k(x_3) = 4$
$k(x_2) = 4$
$k(x_4) = 4 \rightarrow 5$

- We will not update the increase of the key of $x_4$ in the heaps of $x_3$ and x

# Be lazy don't update backwards



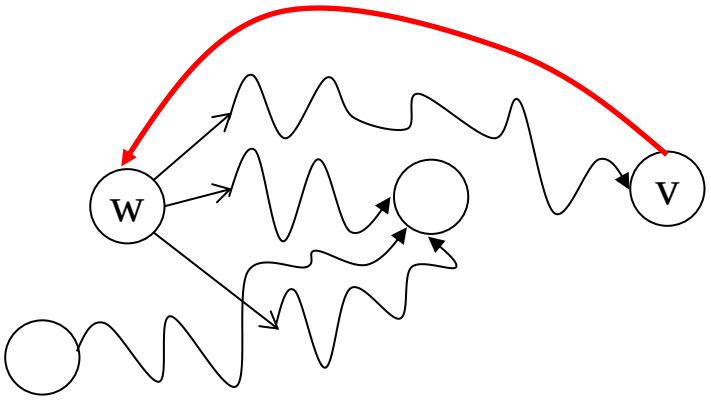

$k(x_1) = 4$
$k(x_5) = 4$
$k(x_3) = 4$
$k(x_2) = 4$
$k(x_4) = 4 \rightarrow 5$

- So $x_3$ thinks it has to update $x_4$ but it discovers that it does not

➔ futile forward updates

# Summary so far

- When we add an arc (v,w) where k(w) < k(v) we update k(w) to k(v) + 1
- Each vertex that changes propagates the change to successors he "thinks" are no larger than its new value
- If we reach v then a cycle is formed

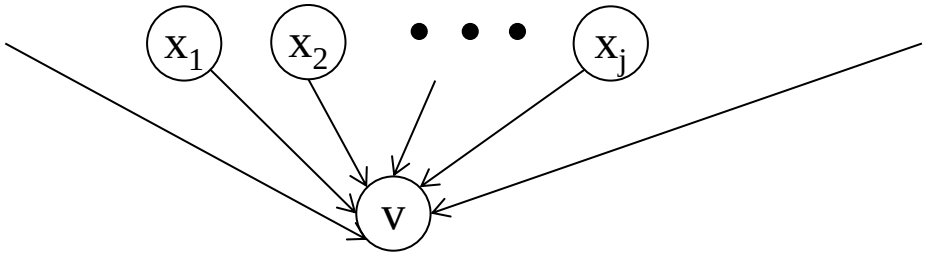


# Dealing with futile updates

# When v talks to w it wants to increase w as much as possible

- Maybe even above k(v) + 1..

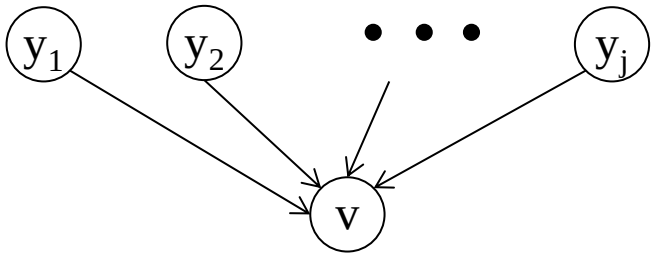
Lemma: If size($x_i$) ≥ s, i=1,…,j, then size(v) ≥ s + j




proof:

# How do we use it ?

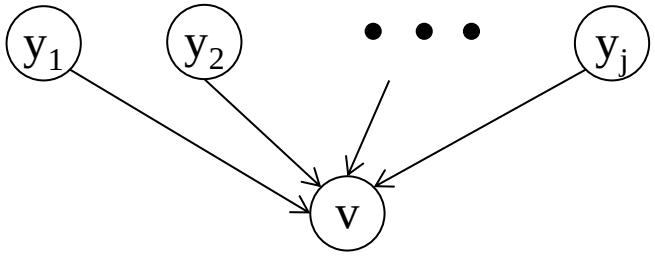
$k(y_1) \geq k(y_2) \geq \ldots\ldots \geq k(y_i)$




v maintains the nodes $y_i$, with an edge $(y_i,v)$ in a data structure sorted by $k(y_i)$ (at the time $y_i$ last updated v)

# How do we use it ?

$k(y_1) \geq k(y_2) \geq \ldots\ldots \geq k(y_i)$



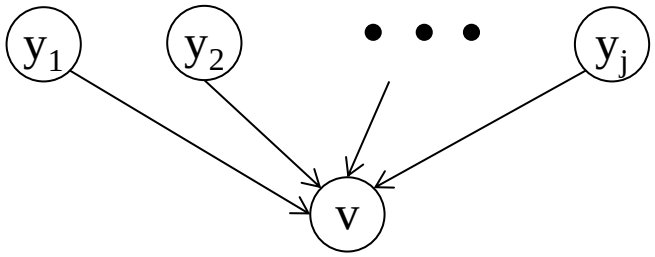

By the Lemma:

For any j, size(v) ≥ $k(y_j)$ + j

➔ So we can set $k(v) = \max_j\{k(y_j) + j\}$

# More precisely

$k(y_1) \geq k(y_2) \geq \ldots\ldots \geq k(y_i)$

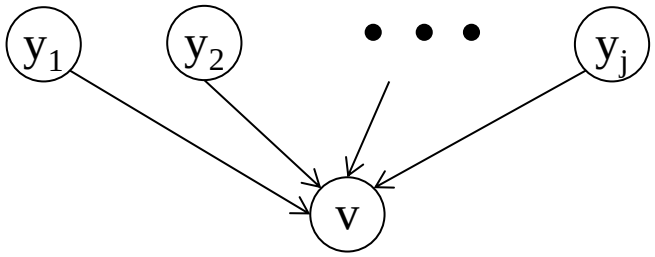



Each time some y tried to update v we update the list above and then set

$k(v) = \max_j\{k(v), k(y_j) + j\}$

# More precisely

$k(y_1) \geq k(y_2) \geq \ldots\ldots \geq k(y_i)$




$k(v) = \max_j\{k(v), k(y_j) + j\}$

Also, as before, y updates the new k(v) in its heap of outgoing arcs

# Remaining questions

- Which data structure should we use to maintain incoming arcs sorted by k(y) at v ?
- How many updates could there be in the worst case ?

# How many updates ?

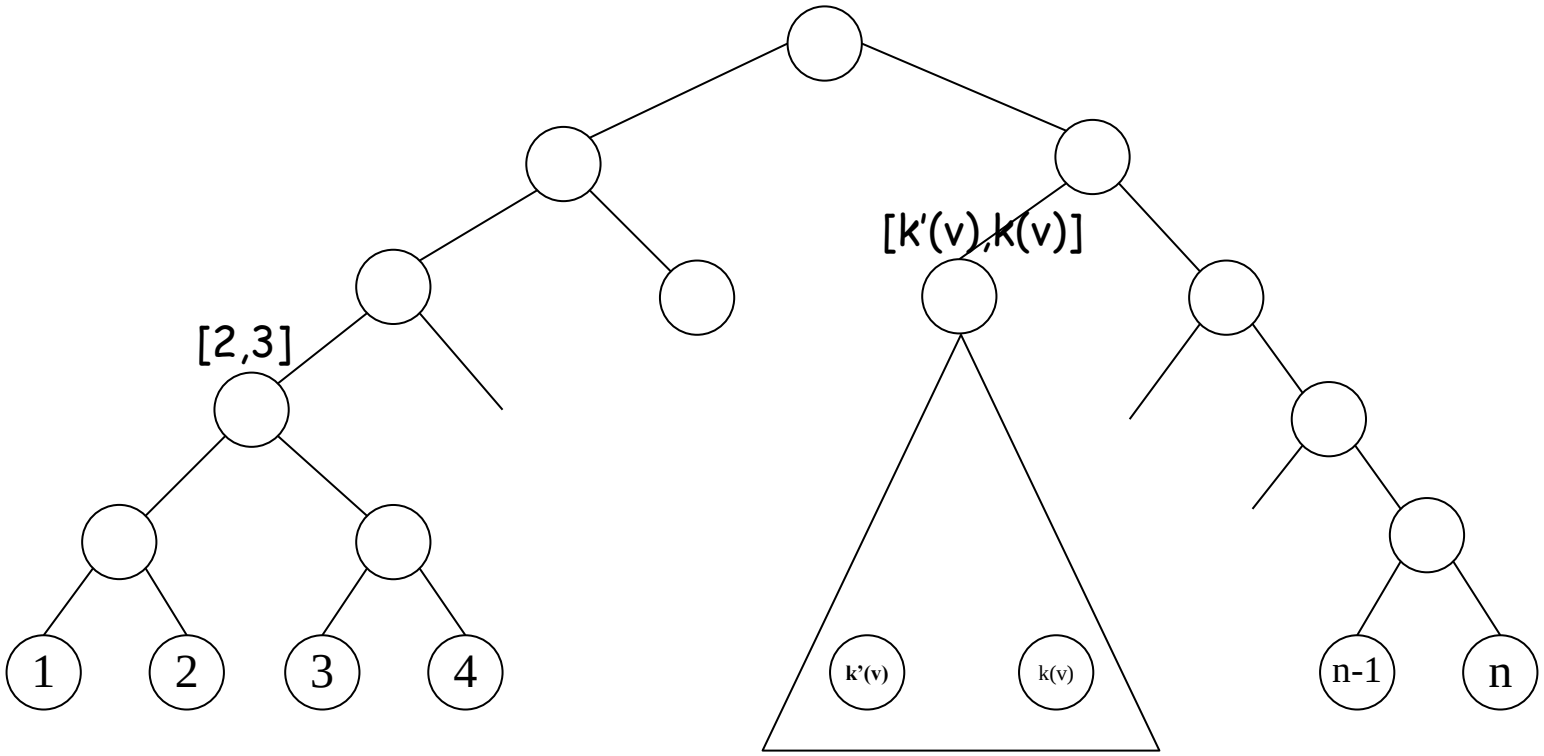
- There are at most n updates that really increase k(v) ➔ $O(n^2)$ total

- How many futile update are there ? (y thinks k(v) is no larger than k(y) but discovers that it is in fact larger and k(v) does not change)

# Futile updates to v

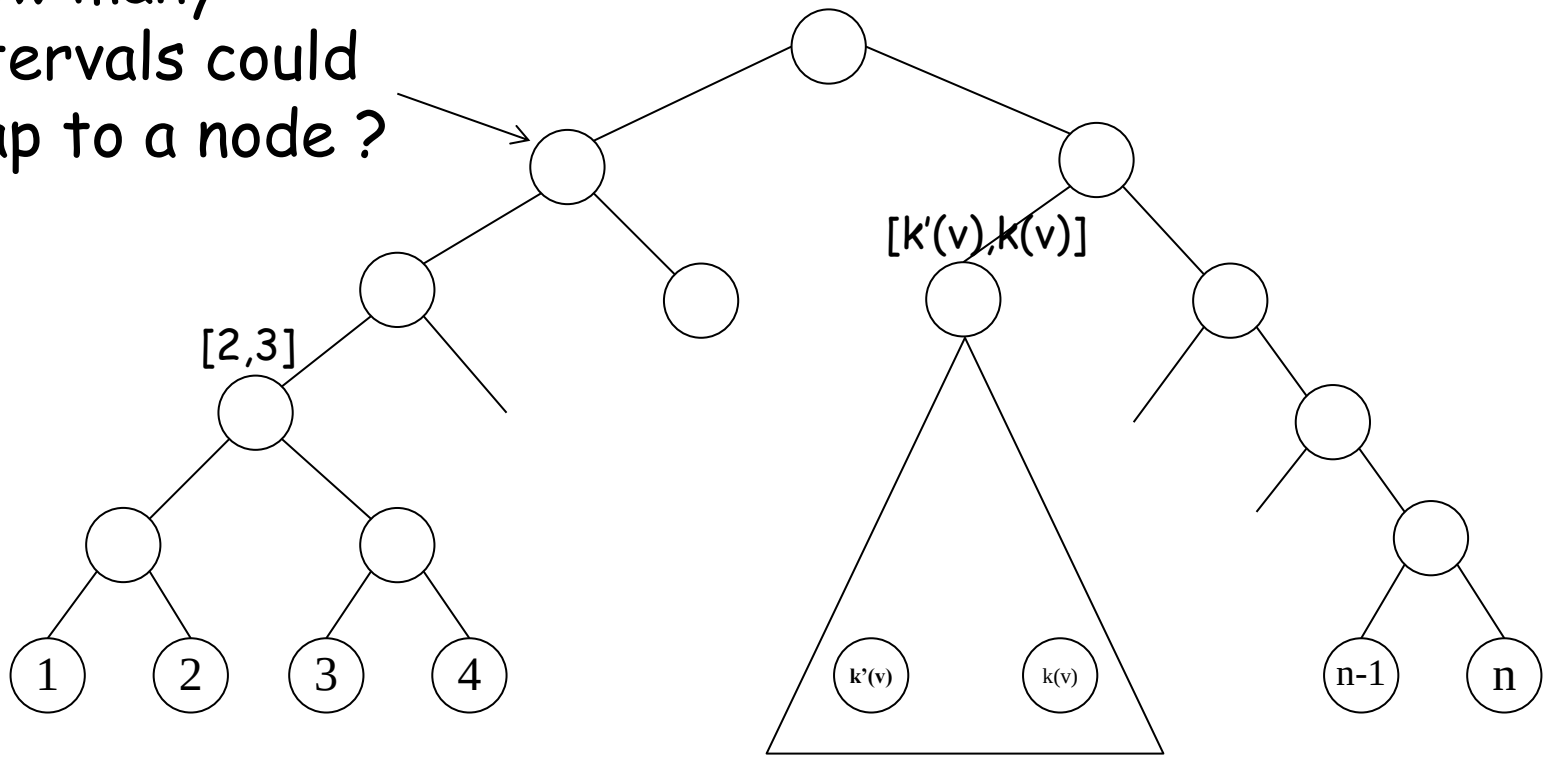
- Let $k'(v) \leq k(y)$ be the value that y thinks v has, and let $k(v) > k(y)$ the value that v really has
- Associate the interval $[k'(v),k(v)]$ with this futile update
- Map this interval to $LCA(k'(v),k(v))$

# A balanced binary tree over the values of k(v)



# A balanced binary tree over the values of k(v)

How many intervals could map to a node ?



# Number of updates

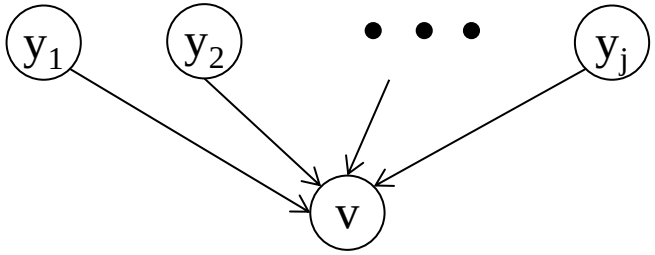
- Look at a node with $2^i$ leaf descendants
- At most one interval from a particular node y
- At most $2^i$ different nodes: After $2^i$ nodes y had tried to update v, with k(y) > k'(v) then k(v) should not be in this subtree anymore

# Conclusion

- $O(n\log(n))$ futile updates per node
- $O(n^2\log(n))$ updates overall

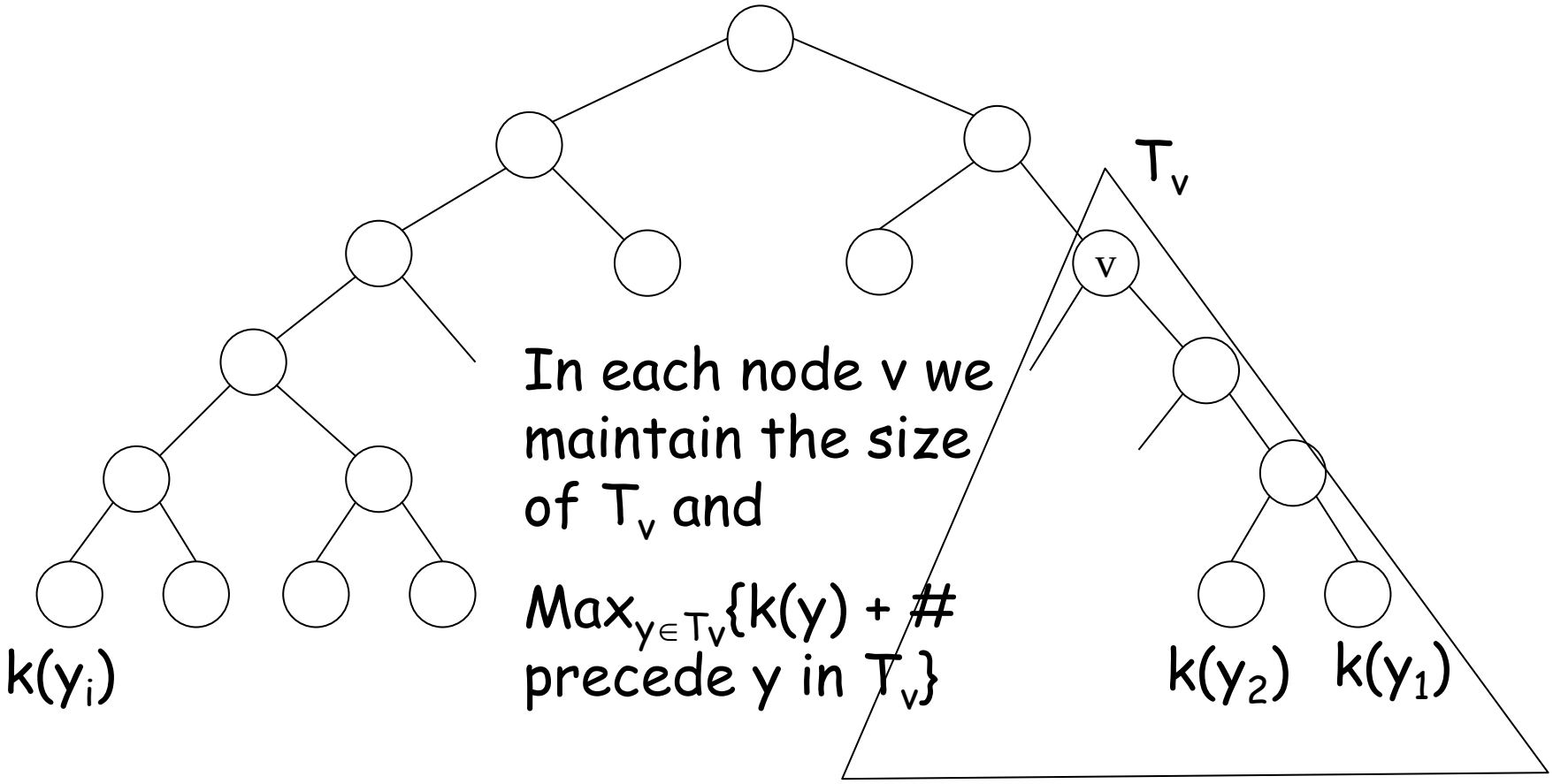
# A data structure to update k(v)

$k(y_1) \geq k(y_2) \geq \ldots\ldots \geq k(y_i)$




A dynamic balanced binary search tree over $y_1, y_2, \ldots, y_i$

# A dynamic balanced binary tree over the values of $k(y_j)$



# Summary

- At the root we have the value we need
- It takes $O(\log(n))$ time to update the tree
- $O(n^2 \log^2 n)$ total time