ECS32B

Introduction to Data Structures

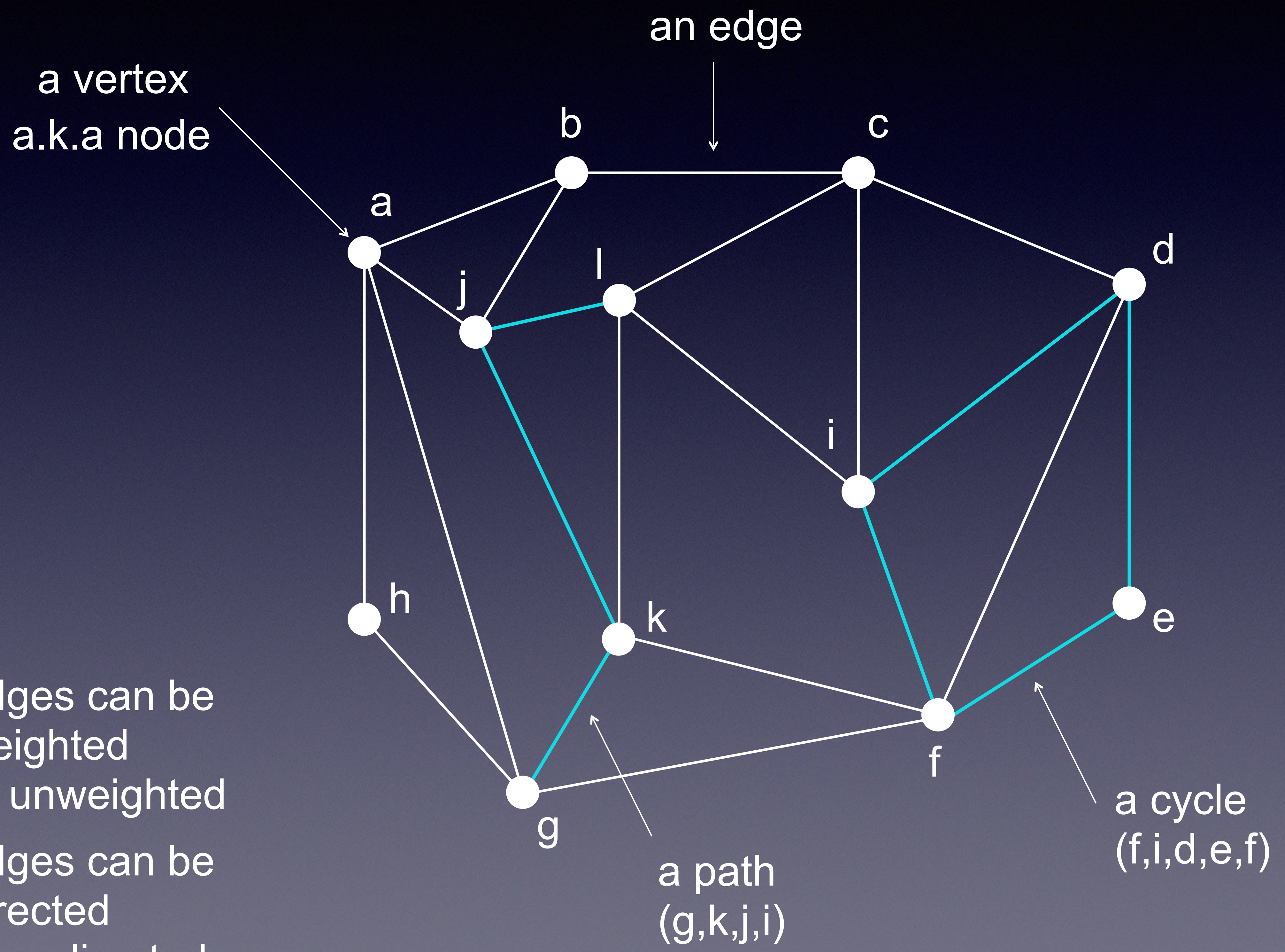
Graphs

Lecture 26

Announcements

- SLAC on Monday will cover HW6 and the Makeup assignment.
- A sample final will be posted on Monday.
- Last lecture will mostly be final exam preparation.

Graph anatomy review



There are |**V**| vertices. Often **n** is used to indicate the number of vertices.

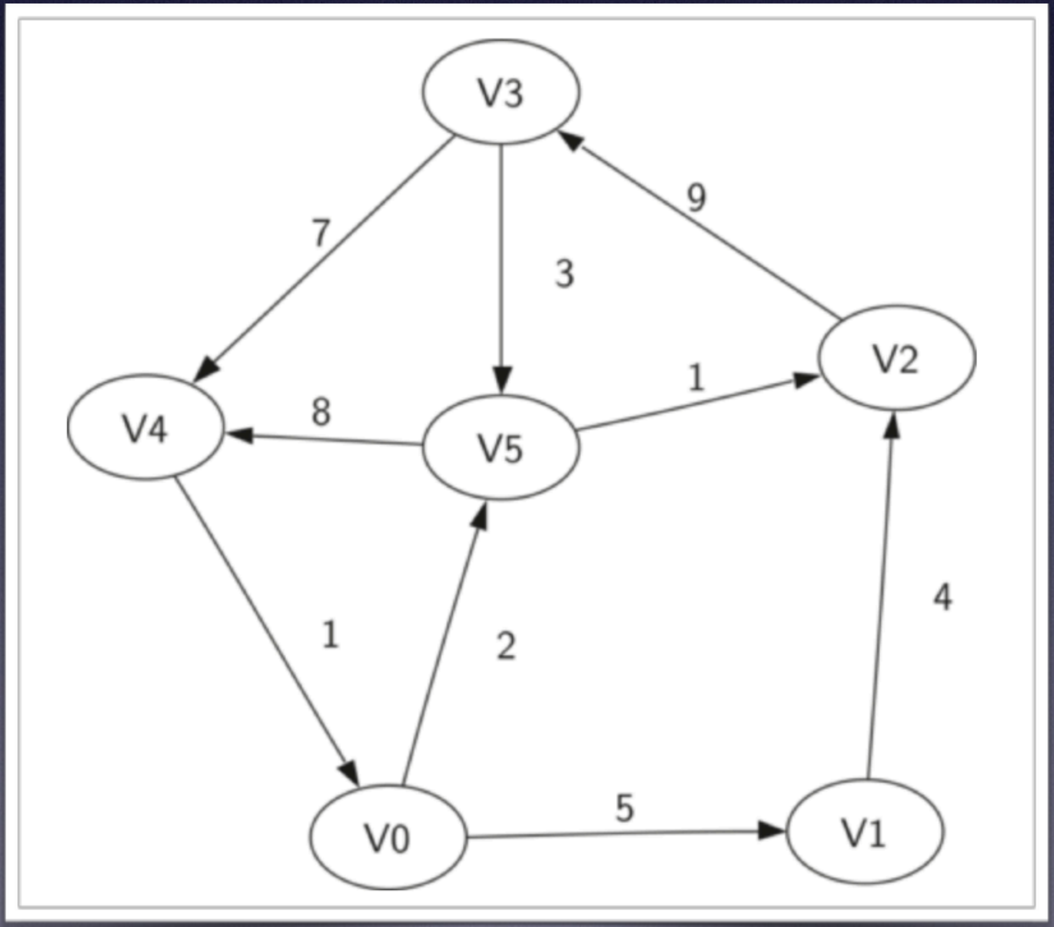
There are |**E**| edges. Often **m** is used to indicate the number of edges. The number of edges is $O(n^2)$

edges can be weighted or unweighted

edges can be directed or undirected

Concrete representation of a graph

How might you represent a graph in Python using nodes and pointers/references in a linked data structure?



Concrete representations of graphs

Two methods of representing a graph are popular:

- **Adjacency Matrix** (edge based)

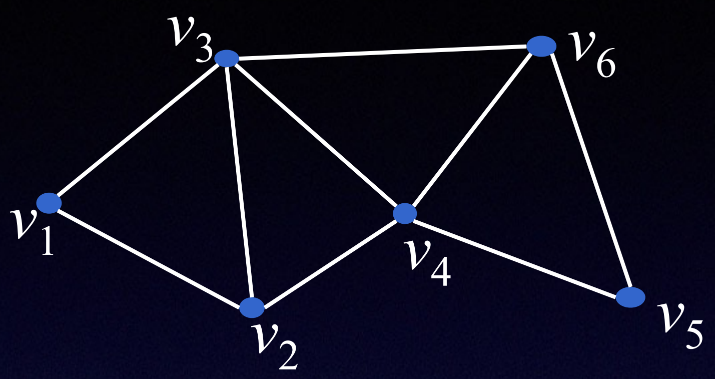
  A $n \times n$ matrix with 1 (or the weight) (or true) for an edge and 0 (or infinity) (or false) for not-an-edge.

- **Adjacency List** (vertex based)

  A vector (array) of lists, one for each vertex. Each list has the adjacent vertices.

Adjacency Matrix

E.g. A graph with n vertices
v_1, v_2, … v_n.



The entry in row i and column j of **A** tells us something about the edge (v_i, v_j) in the graph.

The **adjacency matrix** contains 0's and 1's indicating if an edge is present or not. You could also use True or False, or you could use weights in the case of a weighted graph.

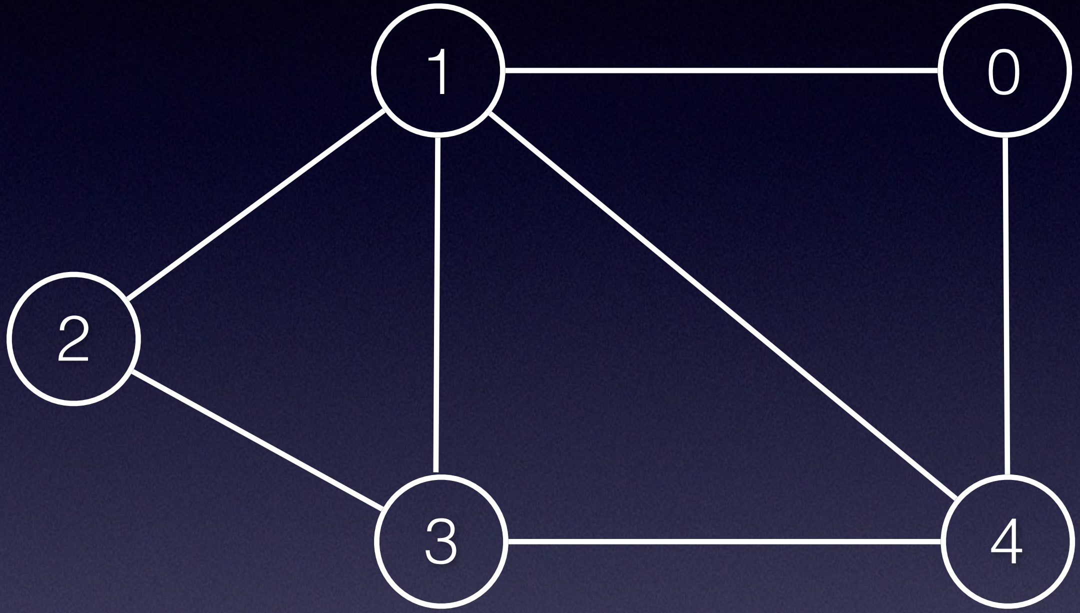
The adjacency matrix of an undirected graph is symmetric.

For graphs with few edges, the adjacency matrix is sparse (i.e., contains mostly zeros).

	v_1	v_2	v_3	v_4	v_5	v_6
v_1	0	1	1	0	0	0
v_2	1	0	1	1	0	0
v_3	1	1	0	1	0	1
v_4	0	1	1	0	1	1
v_5	0	0	0	1	0	1
v_6	0	0	1	1	1	0

Adjacency Matrix A

Another Adjacency Matrix

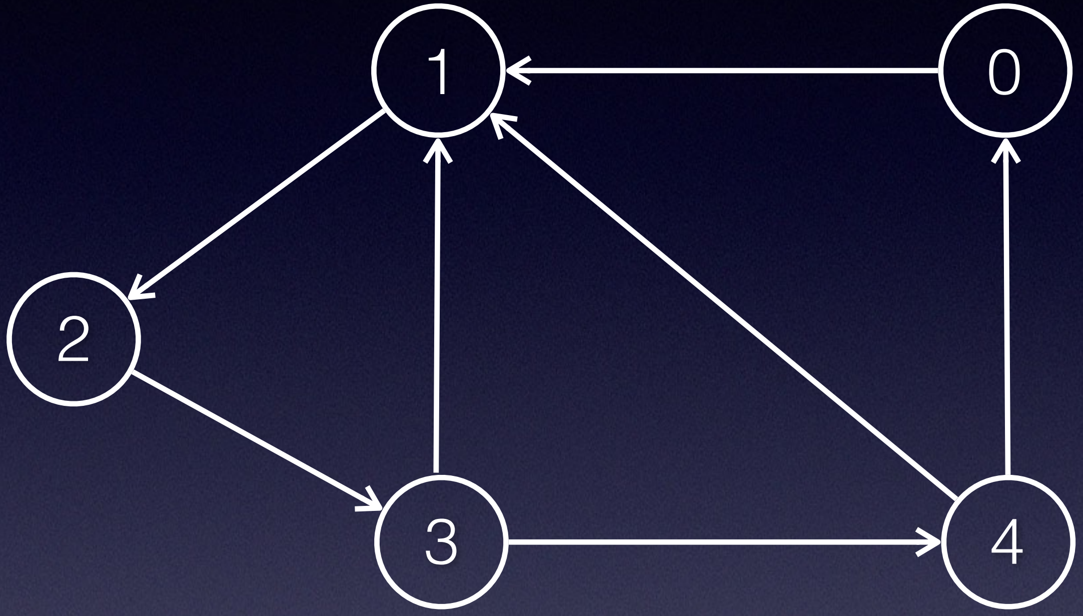


	0	*1*	*2*	*3*	*4*
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency Matrix

Notice the symmetry.

Another Adjacency Matrix (Digraph)



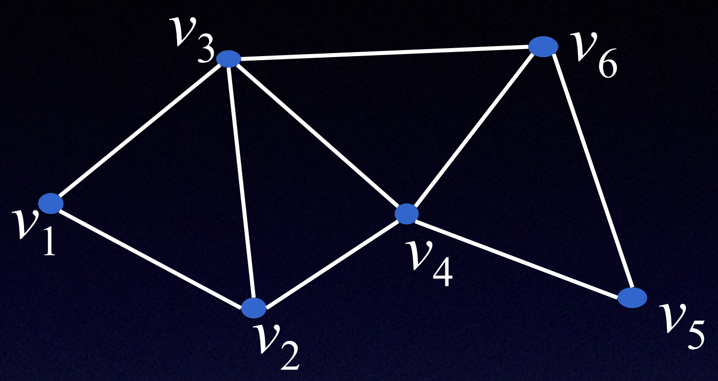
	0	*1*	*2*	*3*	*4*
0	0	1	0	0	0
1	0	0	1	1	0
2	0	0	0	1	0
3	0	0	0	0	1
4	1	1	0	0	0

Adjacency Matrix

No longer symmetric

Adjacency list

E.g. A graph with n vertices
$v_1, v_2, \ldots v_n$.



The adjacency list is never sparse (even for a graph with few edges). The space is well-used

Notice that an undirected edge is listed at both of the vertices it connects

v_1	v_2, v_3
v_2	v_1, v_3, v_4
v_3	v_1, v_2, v_4, v_6
v_4	v_2, v_3, v_5, v_6
v_5	v_4, v_6
v_6	v_3, v_4, v_5

Adjacency List

Concrete representation of a graph

We've been looking at abstract representations; now we'll see how we might implement them as an ADT

Operations on a graph (things we might want to do):

- Create a graph of n vertices.
- Insert a vertex
- Insert an edge
- Query the existence of an edge
- Iterate over the vertices adjacent to a given vertex.

Operations defined for the graph ADT

Graph()
returns an empty graph

addVertex(vert)
adds a vertex to a graph

addEdge(fromVert, toVert, weight=0)
adds an optionally weighted directed edge to a graph and any necessary vertices.

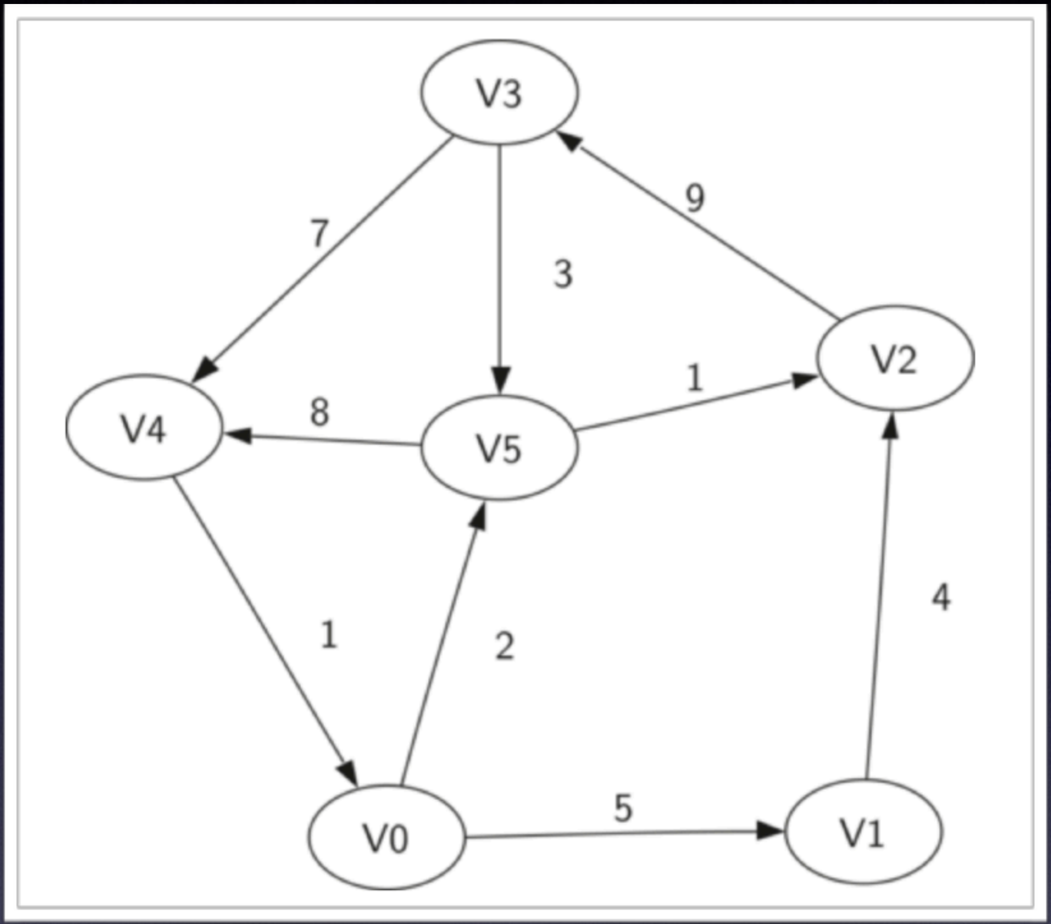
getVertex(vertKey)
returns vertex objected by name

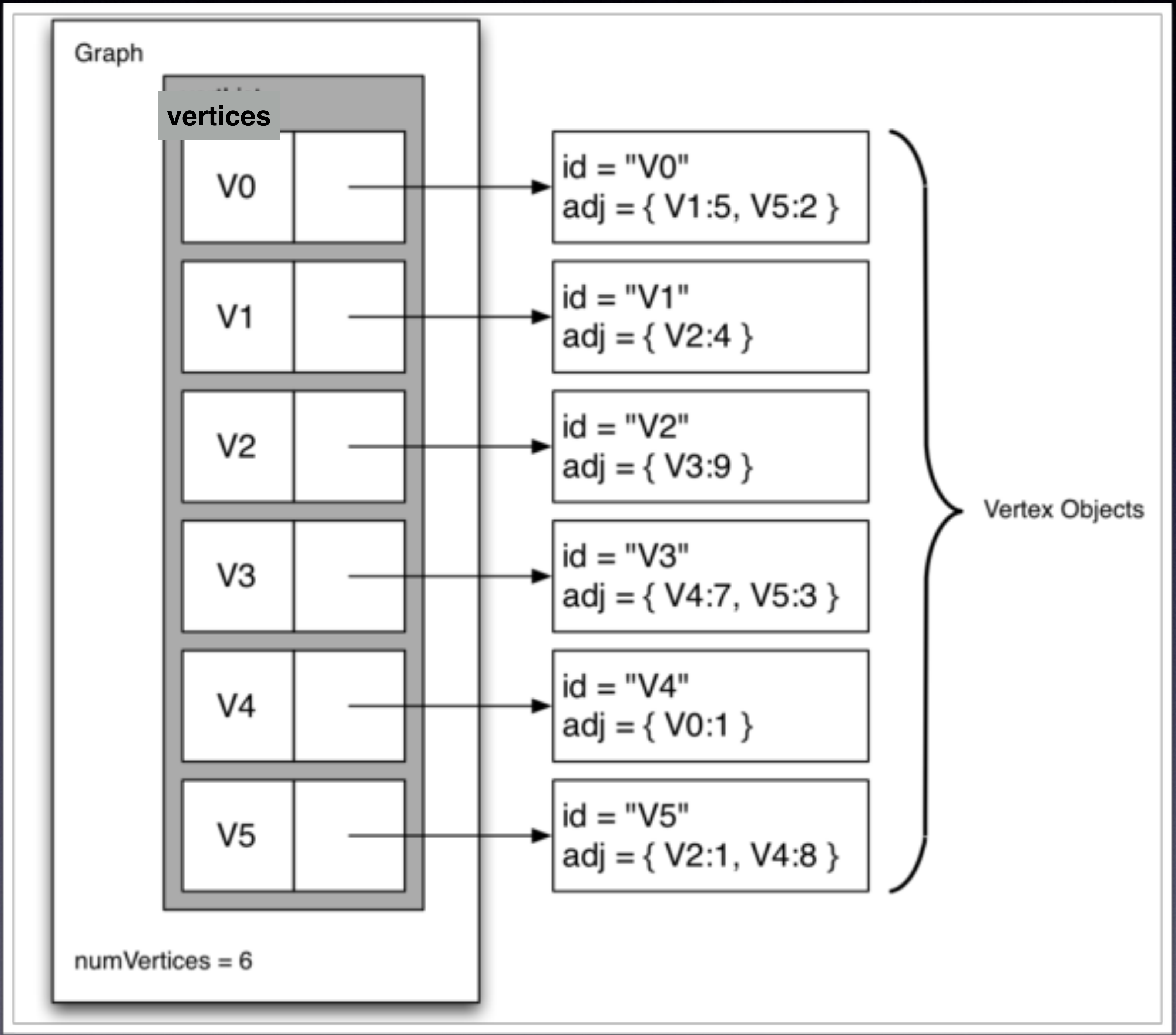
getVertices()
returns list of all vertices

in (is vertex in graph?)

The only two methods
you really need to create
a graph are **Graph**() and **addEdge**()

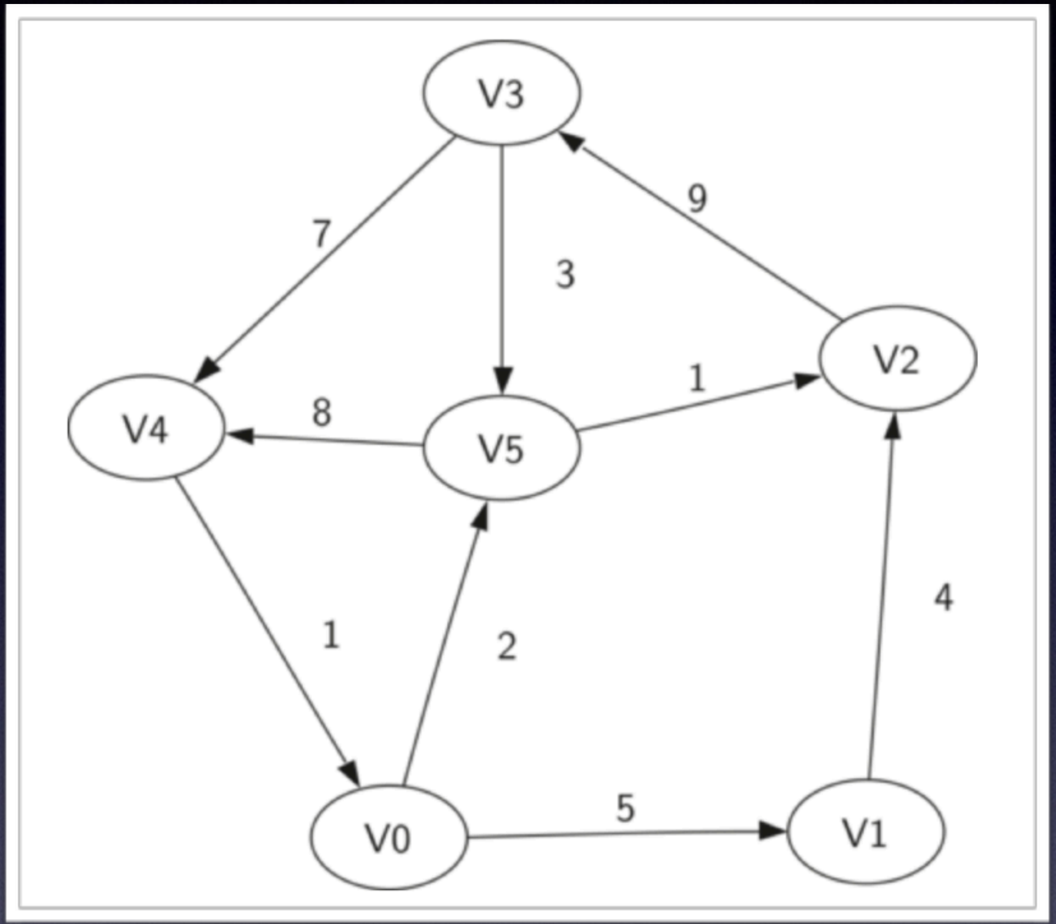
Graph object adjacency list representation

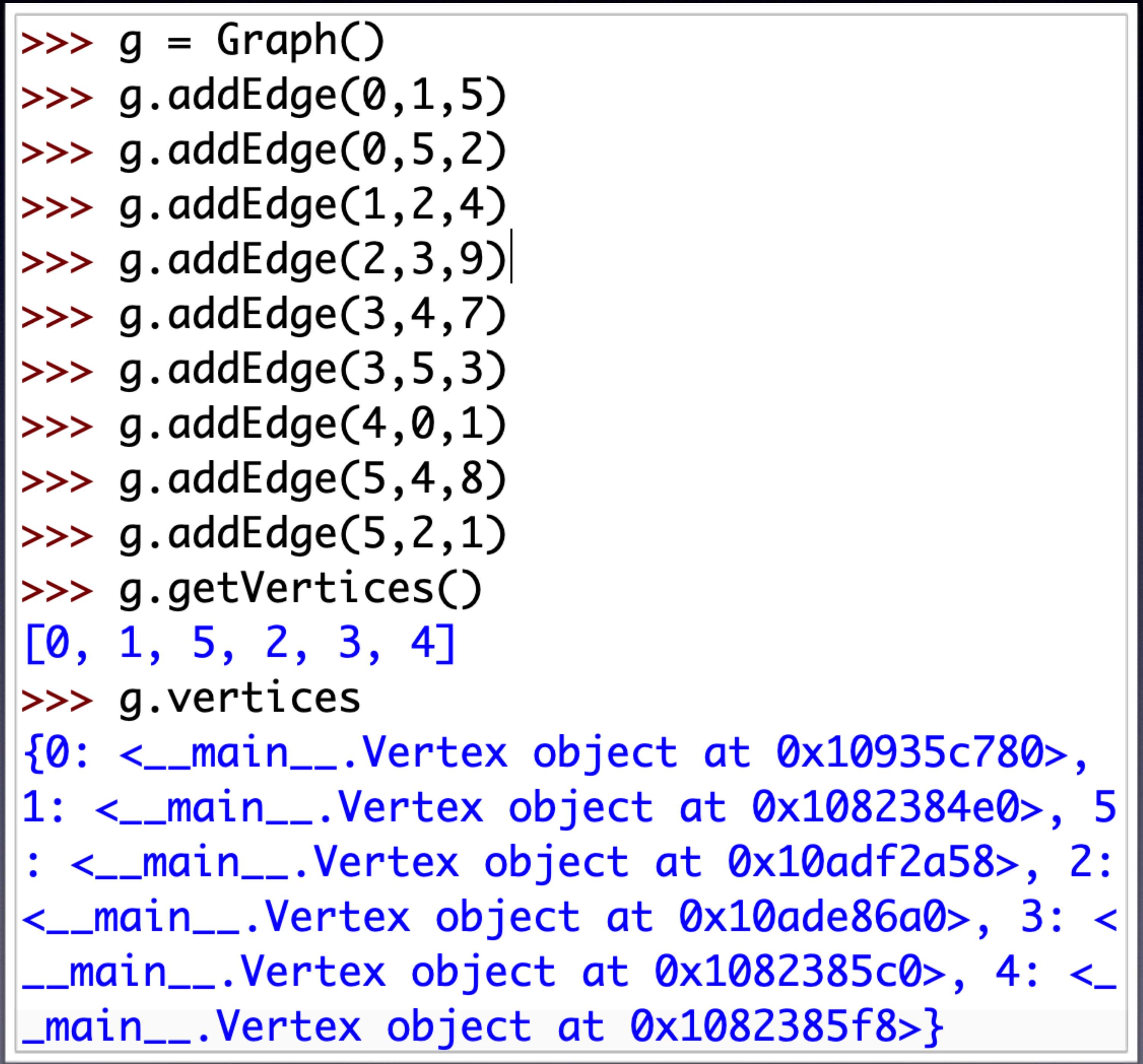




Out textbook uses an adjacency list in each Vertex object to represent the edges

Creating a graph





```
>>> g = Graph()
>>> g.addEdge(0,1,5)
>>> g.addEdge(0,5,2)
>>> g.addEdge(1,2,4)
>>> g.addEdge(2,3,9)
>>> g.addEdge(3,4,7)
>>> g.addEdge(3,5,3)
>>> g.addEdge(4,0,1)
>>> g.addEdge(5,4,8)
>>> g.addEdge(5,2,1)
>>> g.getVertices()
[0, 1, 5, 2, 3, 4]
>>> g.vertices
{0: <__main__.Vertex object at 0x10935c780>,
1: <__main__.Vertex object at 0x1082384e0>, 5
: <__main__.Vertex object at 0x10adf2a58>, 2:
<__main__.Vertex object at 0x10ade86a0>, 3: <
__main__.Vertex object at 0x1082385c0>, 4: <_
_main__.Vertex object at 0x1082385f8>}
```

g = Graph() creates a new graph **g**.
g.addEdge() also creates the vertices
g.getVertices() returns a list of ids while **g.vertices** is actually a dictionary mapping ids to Vertex objects

Graph traversal

- Many graph algorithms involve visiting each vertex in a systematic order.
- The two most common traversal algorithms are **breadth-first** and **depth-first**.
- Although these are traversals, the traversals are usually employed to find something in the graph (e.g., a vertex/node, or more likely a path to that vertex/node), so they're more commonly called **breadth-first search** and **depth-first search**.

Breadth-first concept

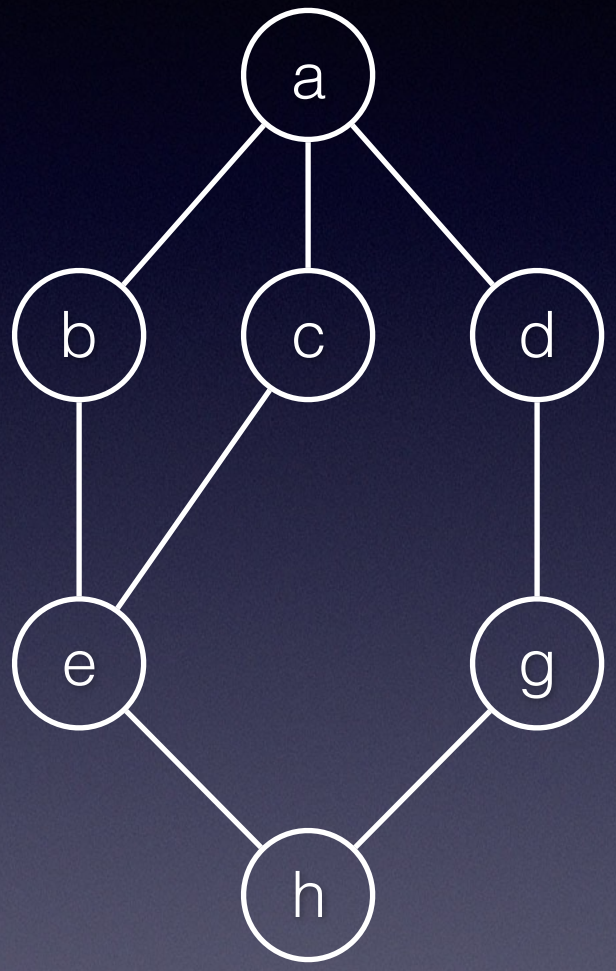
- Starting at a source vertex **s**
- Systematically explore the edges to discover every vertex reachable from **s**.
- Produces a “breadth-first tree”
  - Root of s
  - Contains all vertices reachable from s
  - **Path from s to v has the shortest number of edges**

Breadth-first algorithm

```
Take a start (or source) vertex, mark it identified
(color it grey), and place it into a queue.

While the queue is not empty:
   Take a vertex, u, out of the queue (Begin processing u)
   For all vertices v, adjacent to u,
      If v is still white
         Mark it identified (color it grey)
         Place it into the queue
   We are now done processing u (color it black)
```

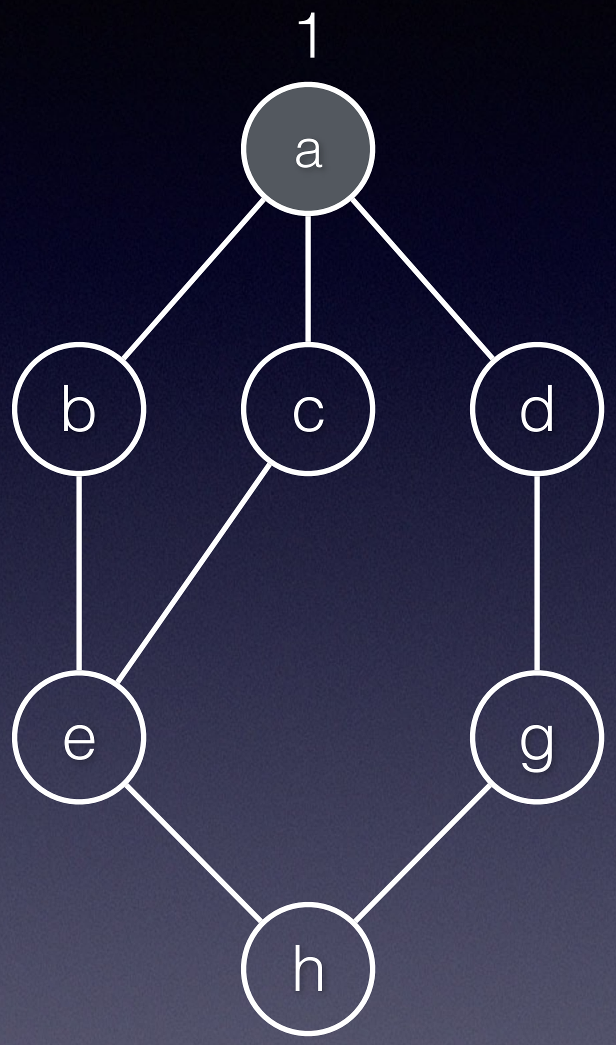
Breadth-first concept



Queue: [ ] (right is front of queue)

Processing:

Breadth-first concept

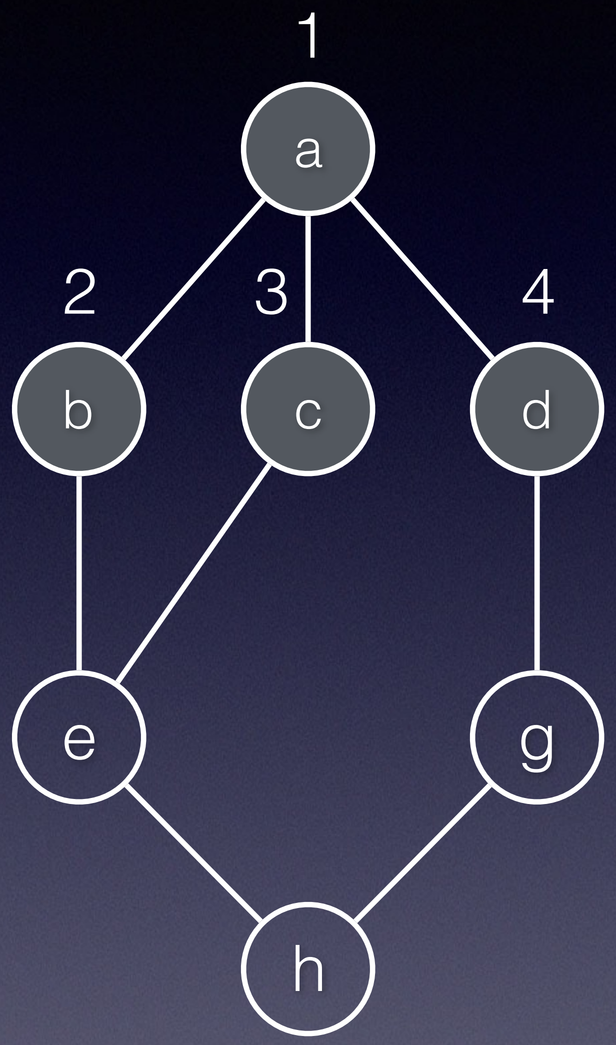


Queue: [a] (right is front of queue)

Processing:

We will number the vertices in the order they are identified/ processed/visited. This is not essential to implementing the algorithm

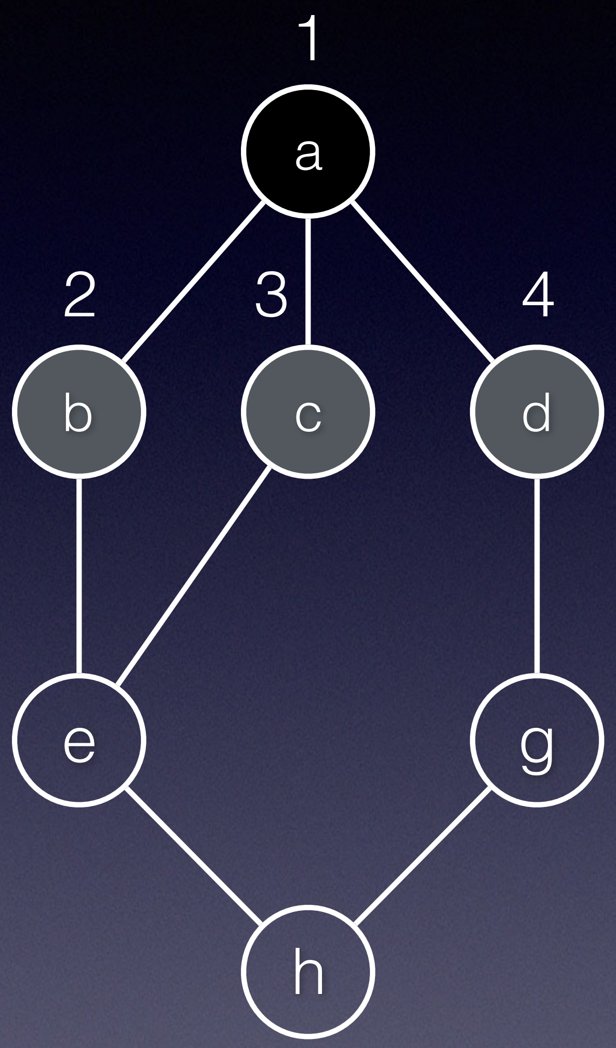
Breadth-first concept



Queue: [d,c,b]

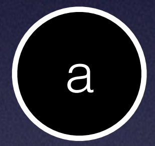
Processing: a

Breadth-first concept

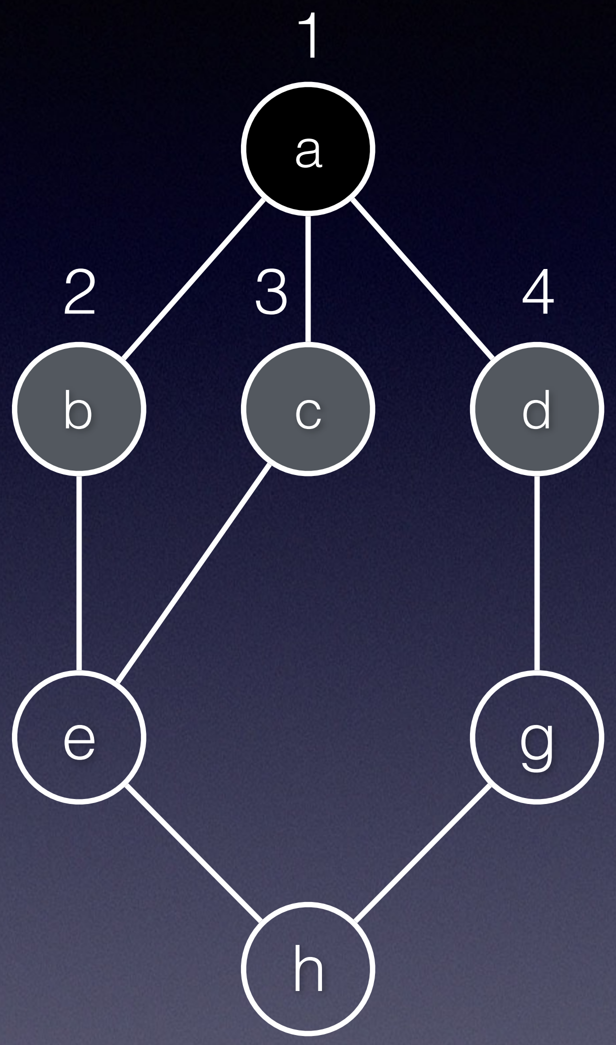


Queue: [d,c,b]

Processing:



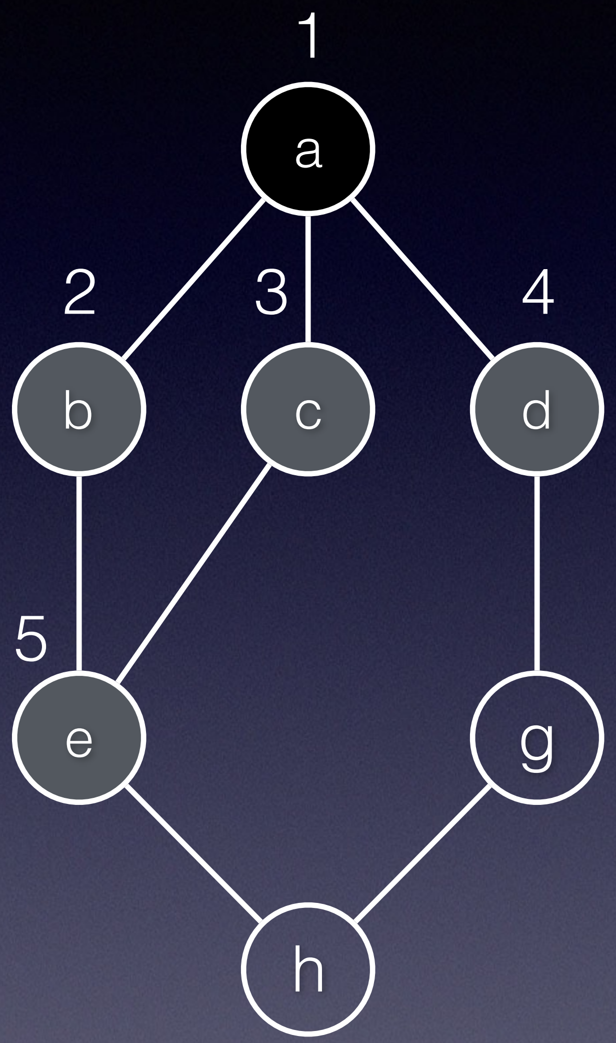
Breadth-first concept



Queue: [d,c]

Processing: b

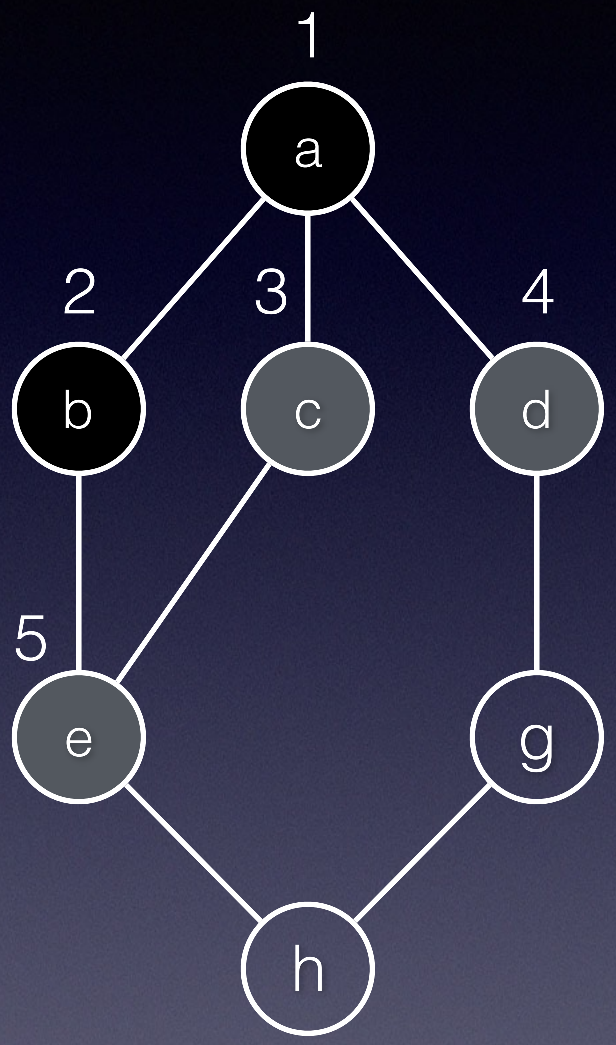
Breadth-first concept



Queue: [e,d,c]

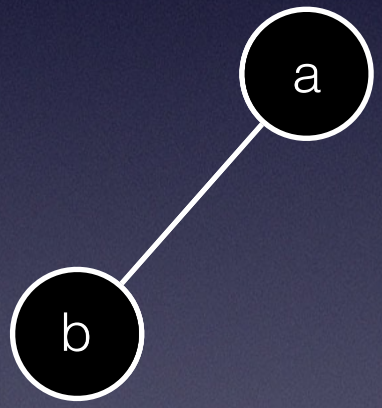
Processing: b

Breadth-first concept

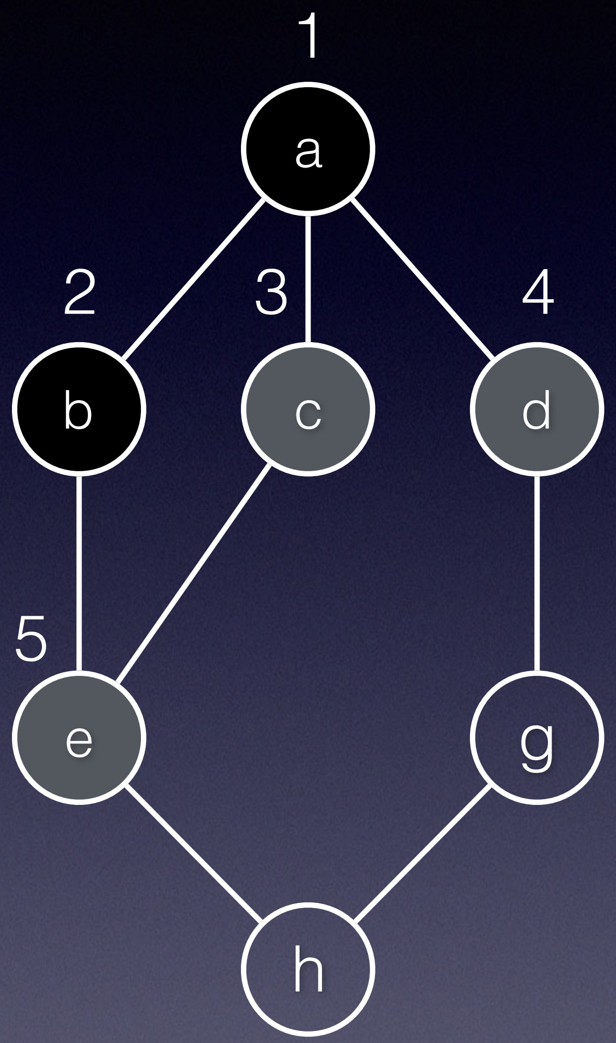


Queue: [e,d,c]

Processing:

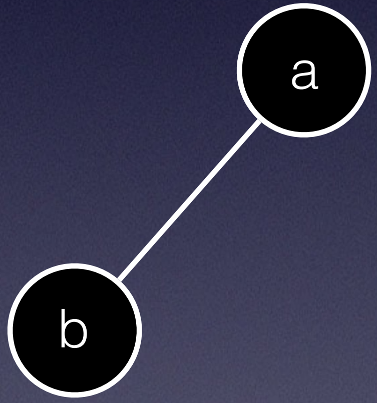


Breadth-first concept

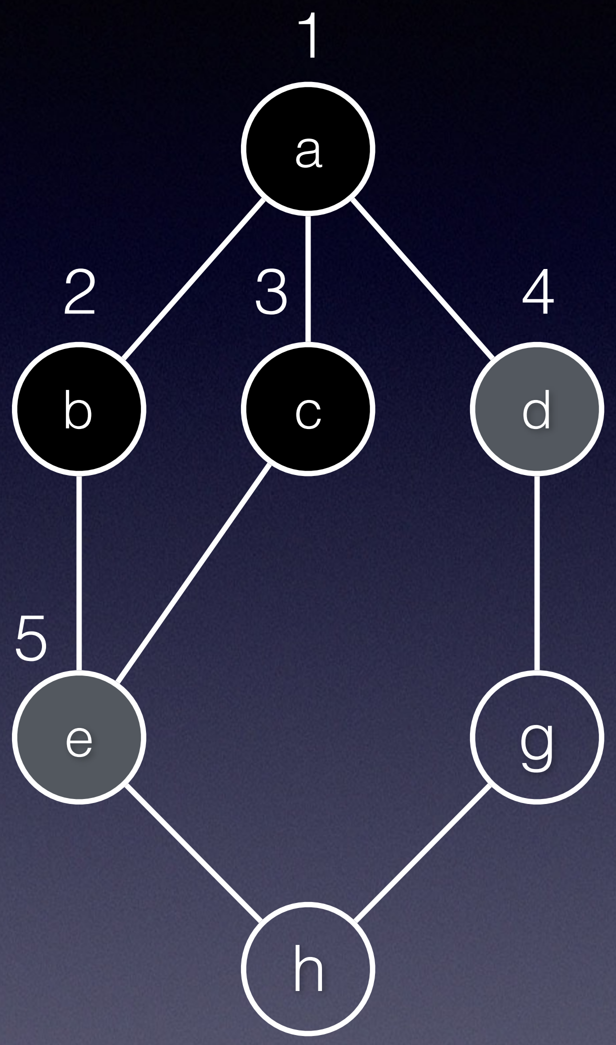


Queue: [e,d]

Processing: c

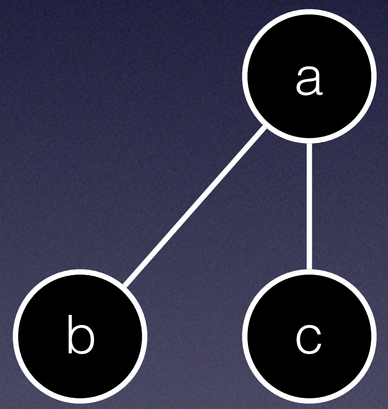


Breadth-first concept

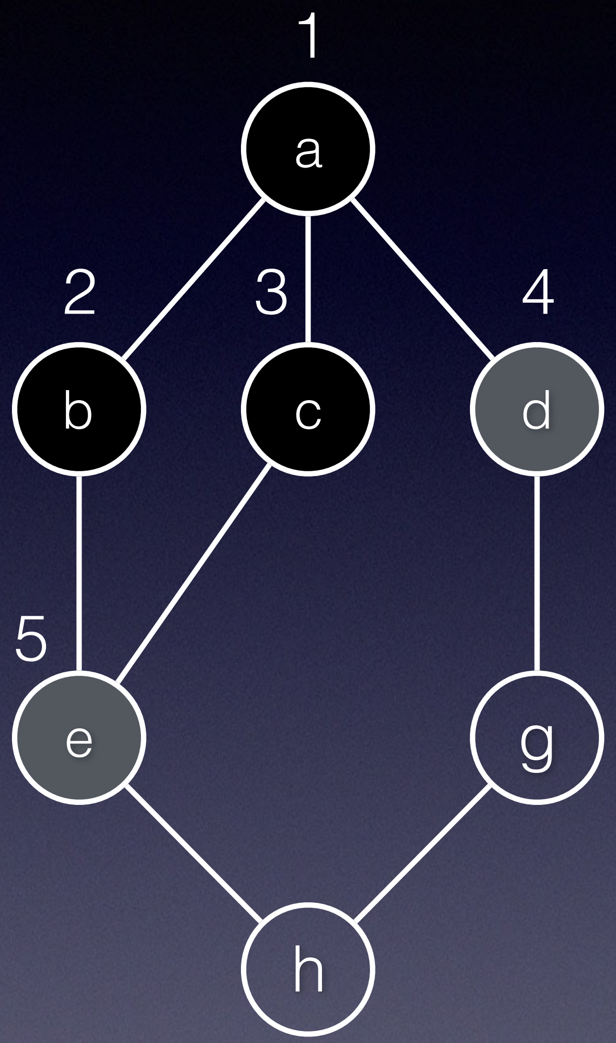


Queue: [e,d]

Processing:

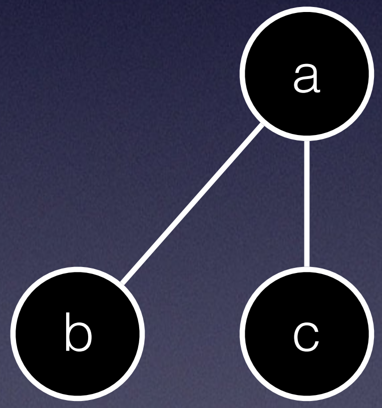


Breadth-first concept

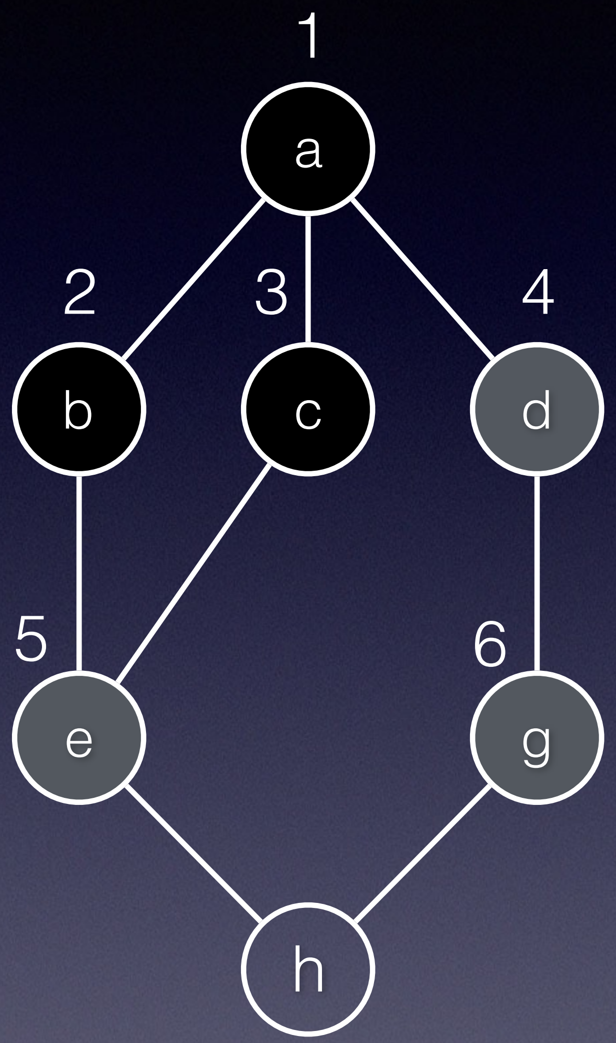


Queue: [e]

Processing: d

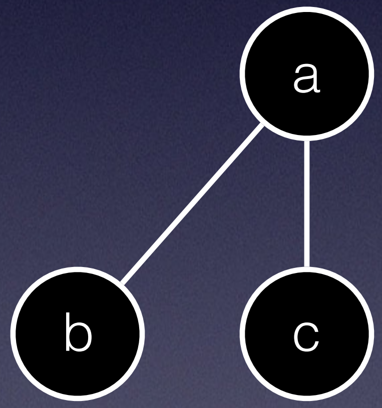


Breadth-first concept

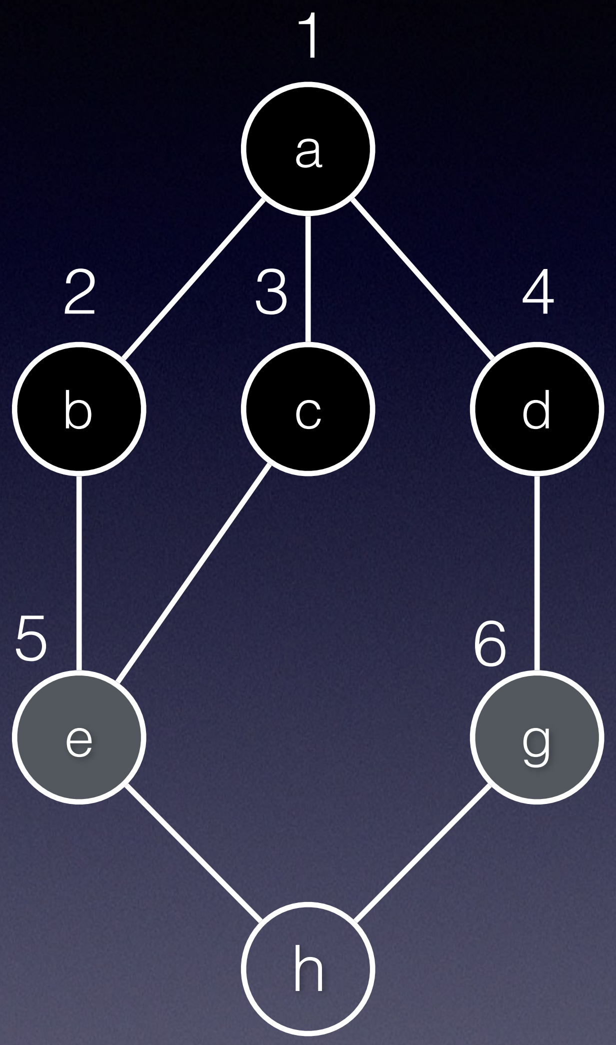


Queue: [g,e]

Processing: d

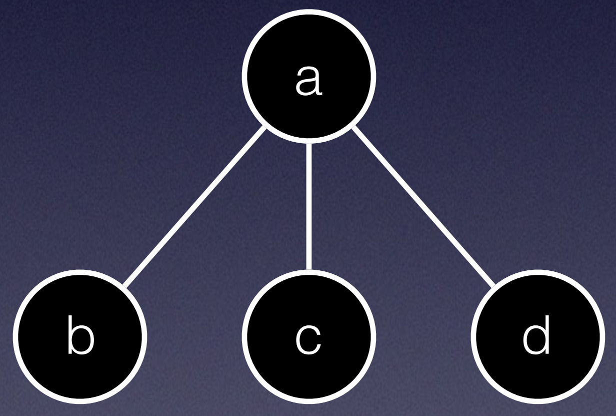


Breadth-first concept

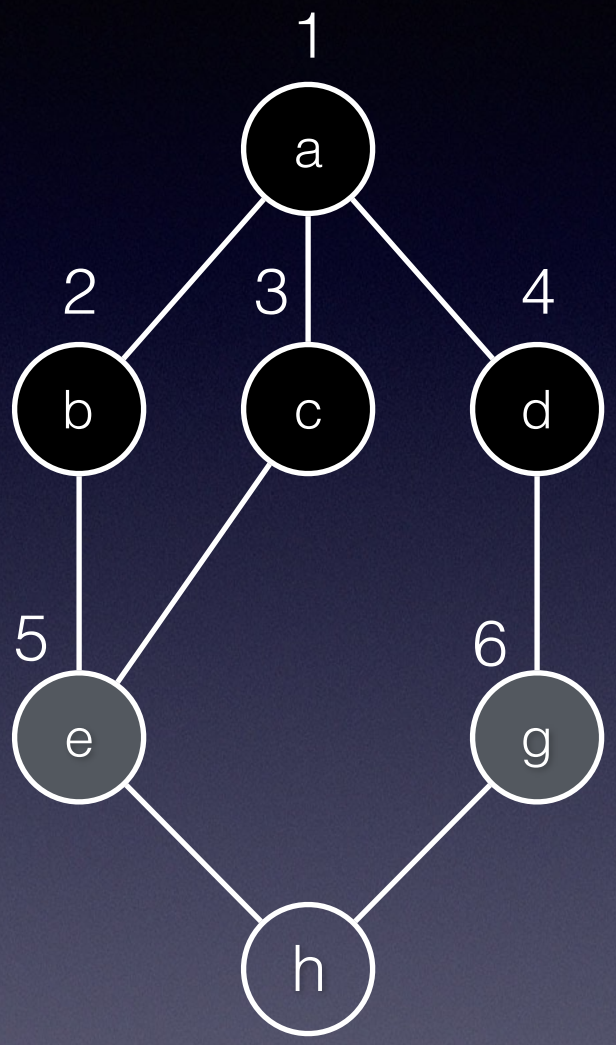


Queue: [g,e]

Processing:

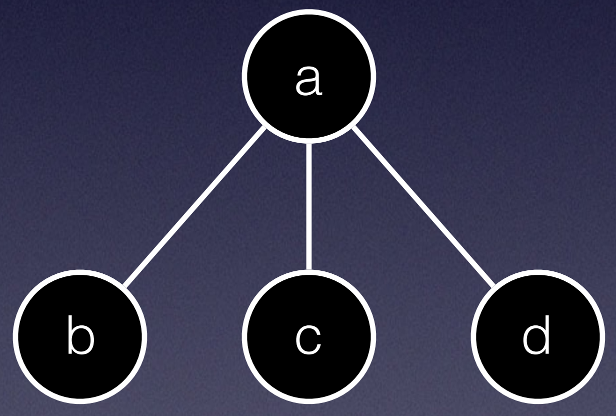


Breadth-first concept

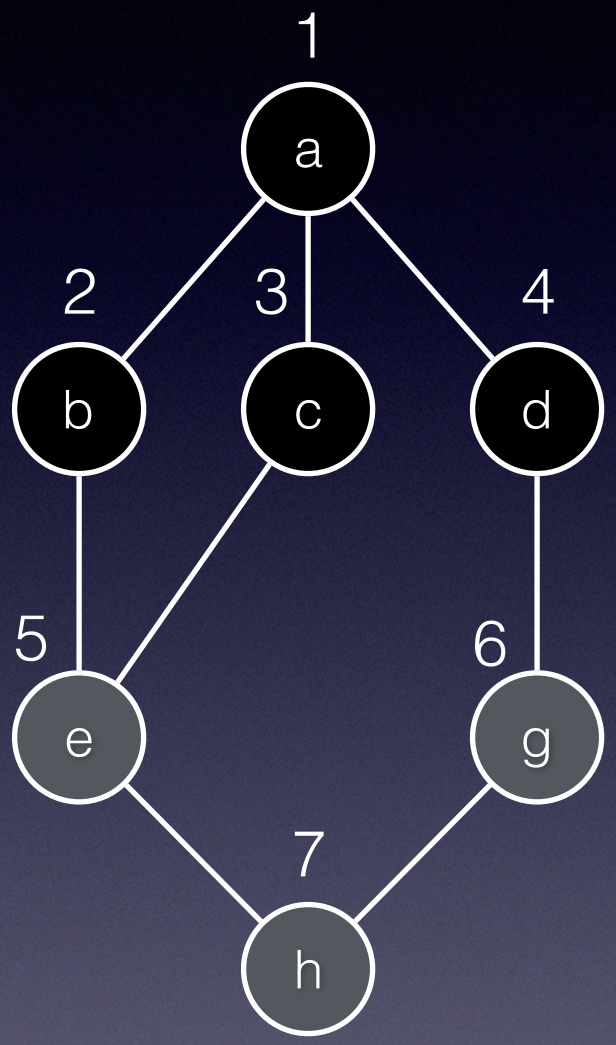


Queue: [g]

Processing: e

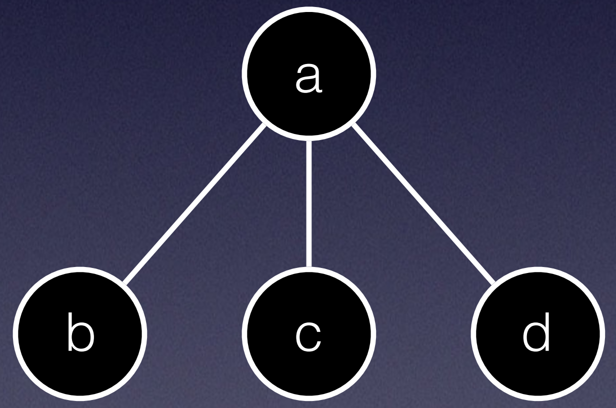


Breadth-first concept

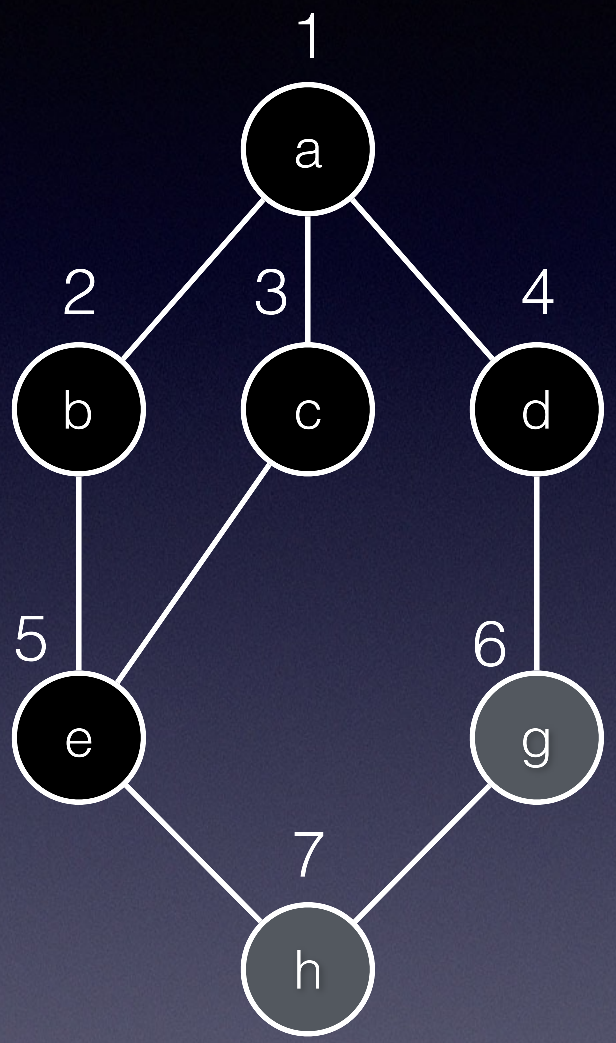


Queue: [h,g]

Processing: e

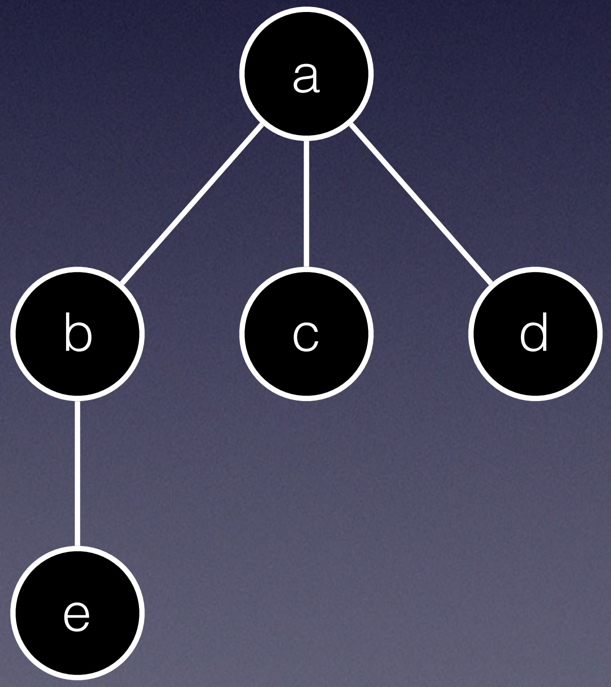


Breadth-first concept

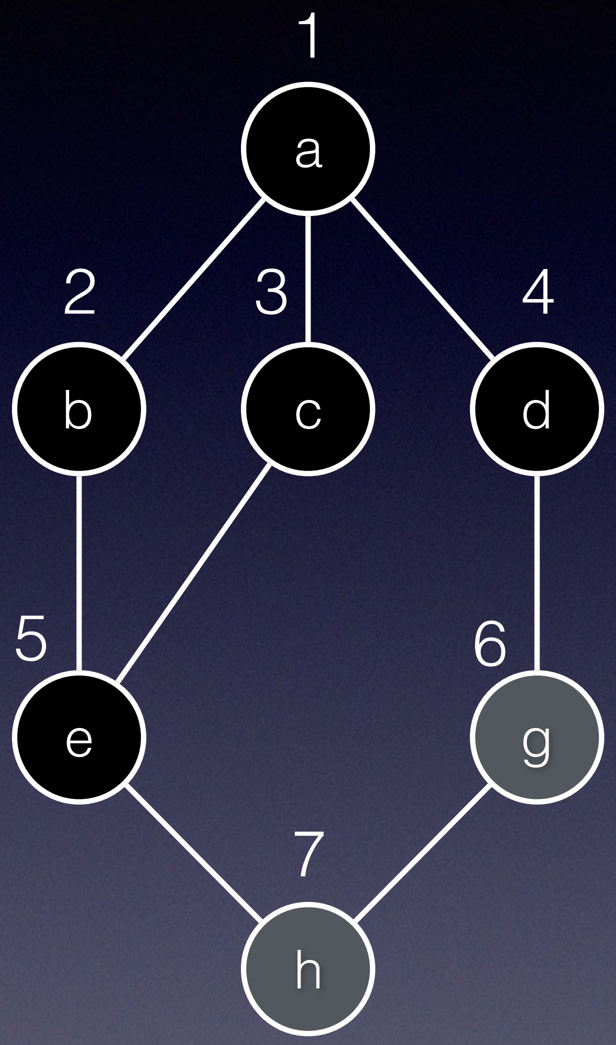


Queue: [h,g]

Processing:

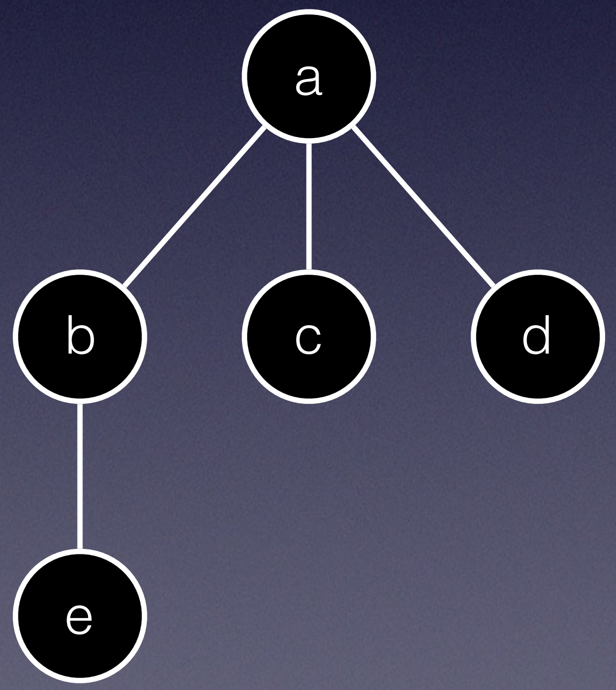


Breadth-first concept

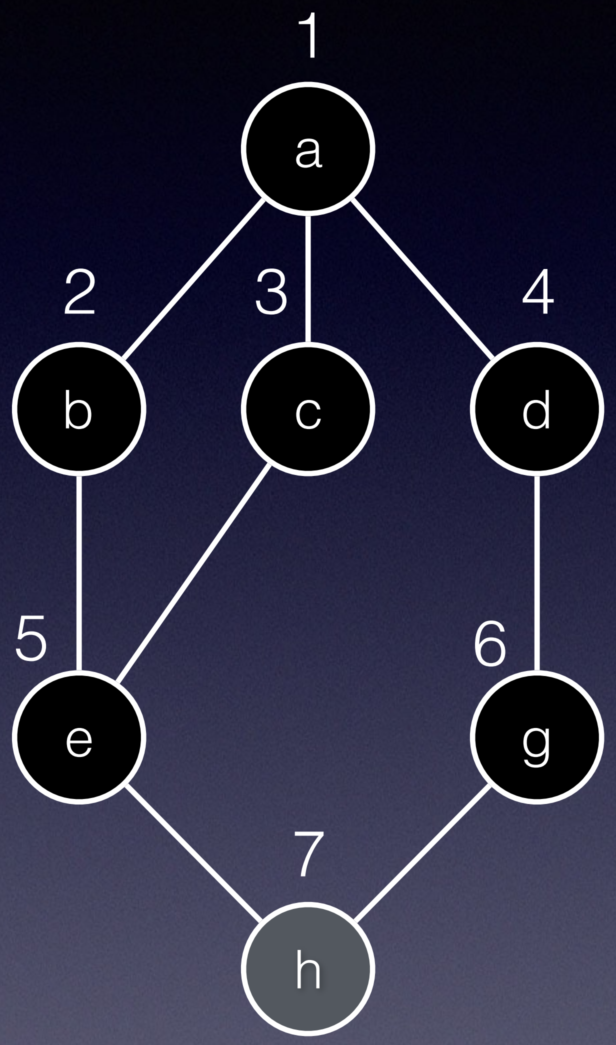


Queue: [h]

Processing: g

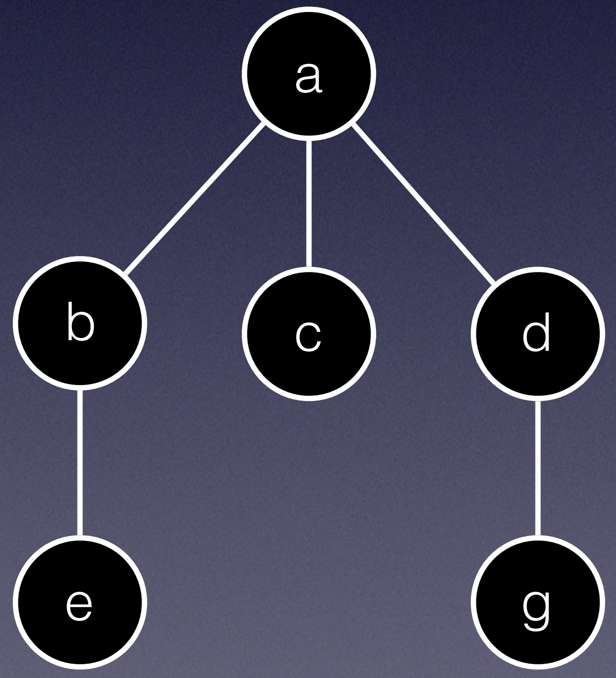


Breadth-first concept

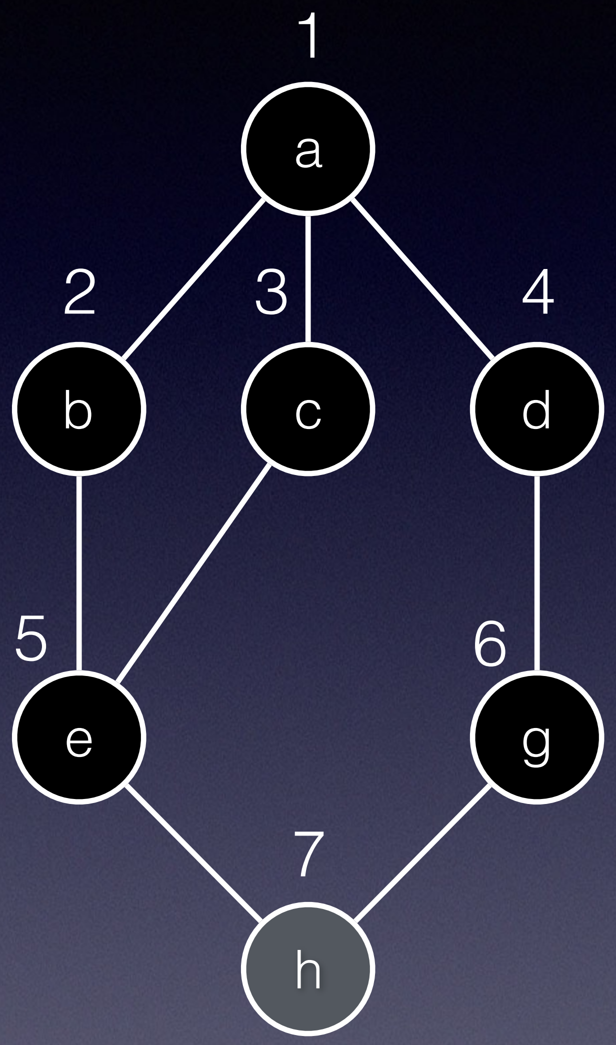


Queue: [h]

Processing:

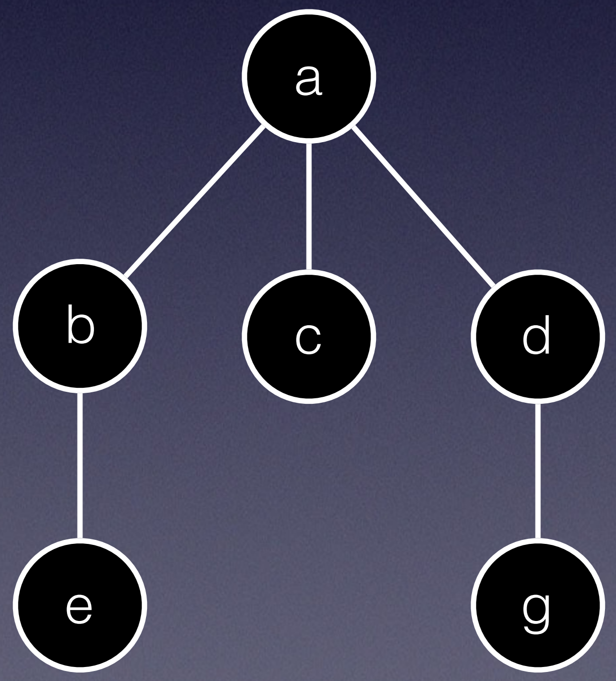


Breadth-first concept

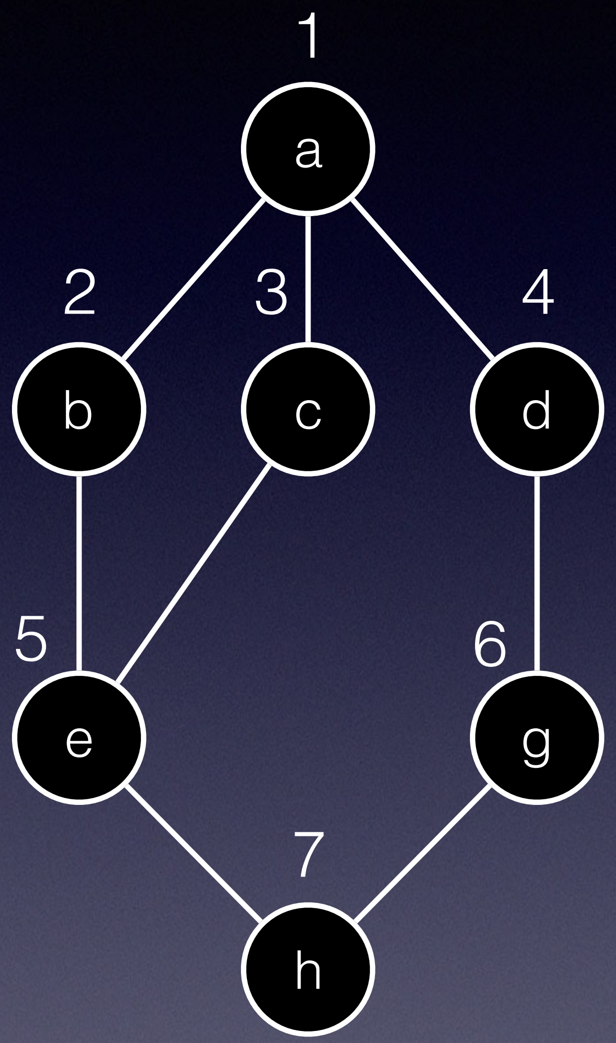


Queue: []

Processing: h

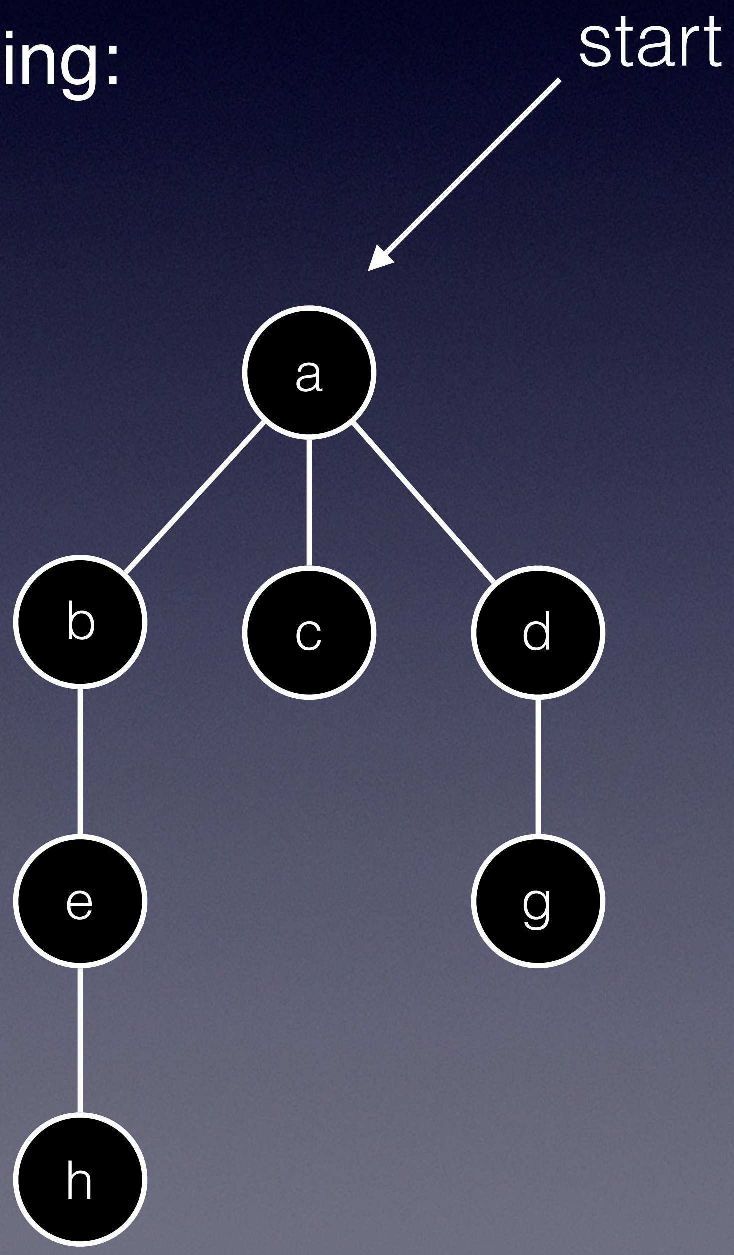


Breadth-first concept



Queue: []

Processing:



Assuming all edges have equal cost (e.g., distance), breadth-first will always find shortest path from start to goal.

Lewis Caroll’s Word Ladders

```
FOOL
POOL
POLL
POLE
PALE
SALE
SAGE
```

A game invented on Christmas day 1877. Player is asked to transform a start word into an end word, using single letter substitutions to create existing words at each step.

Lewis Caroll’s Word Ladders

BLACK	WHEAT	FIND	LOVE	WARM
CLACK	CHEAT	FINE	LIVE	WORM
CRACK	CHEAP	LINE	GIVE	WORD
TRACK	CHEEP	LIVE	GAVE	CORD
TRICK	CREEP	LOVE	LATE	COLD
TRICE	CREED	LOSE	HATE	
TRITE	BREED			
WRITE	BREAD			
WHITE				

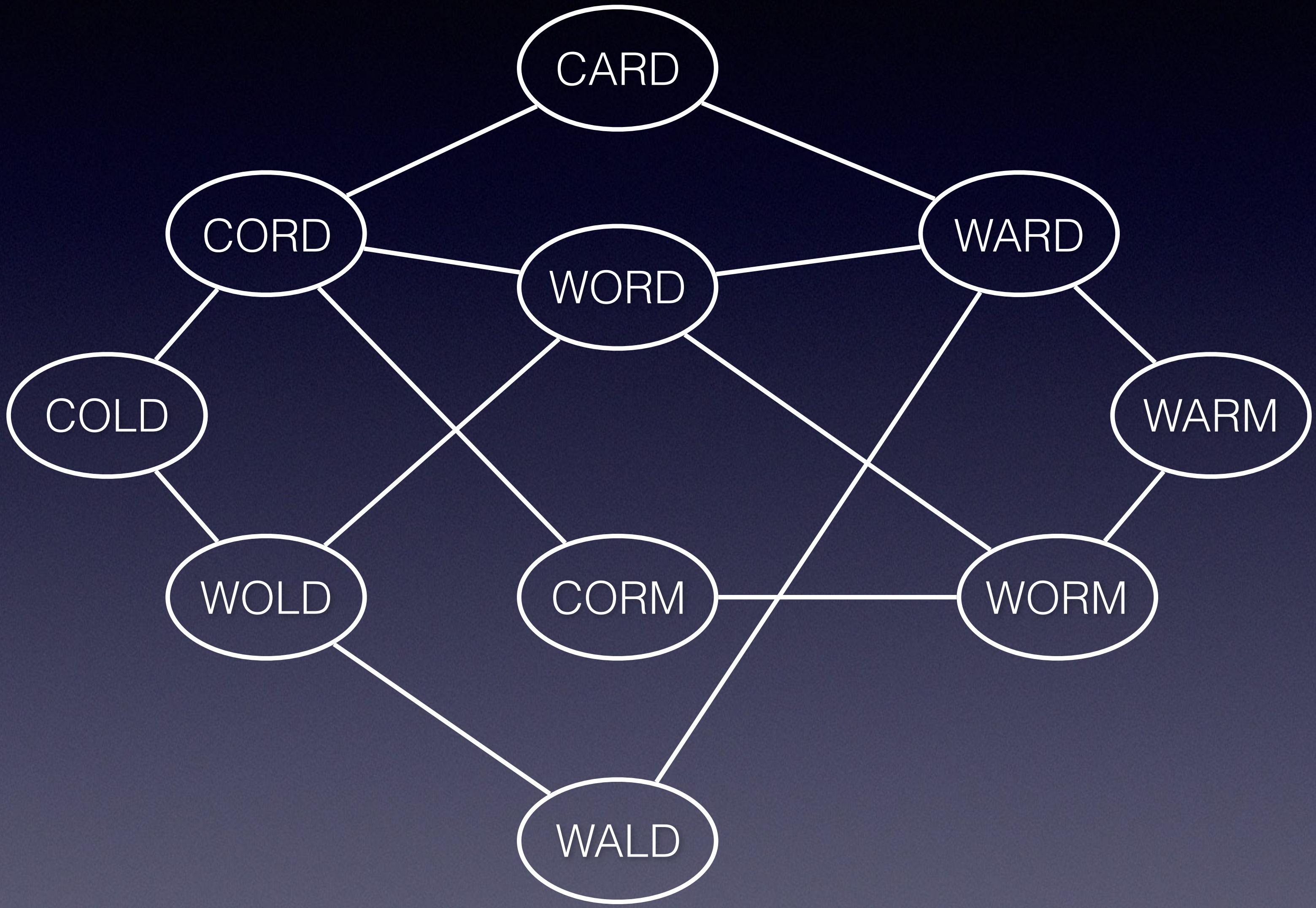
He used clever start and end word pairs

Word Ladders

```
COLD → CORD → CARD → WARD → WARM
COLD → CORD → CORM → WORM → WARM
COLD → CORD → WORD → WARD → WARM
COLD → CORD → WORD → WORM → WARM
COLD → WOLD → WALD → WARD → WARM
COLD → WOLD → WORD → WARD → WARM
COLD → WOLD → WORD → WORM → WARM
```

As wikipedia knows, there are many ways to transform COLD to WARM

Word Ladders

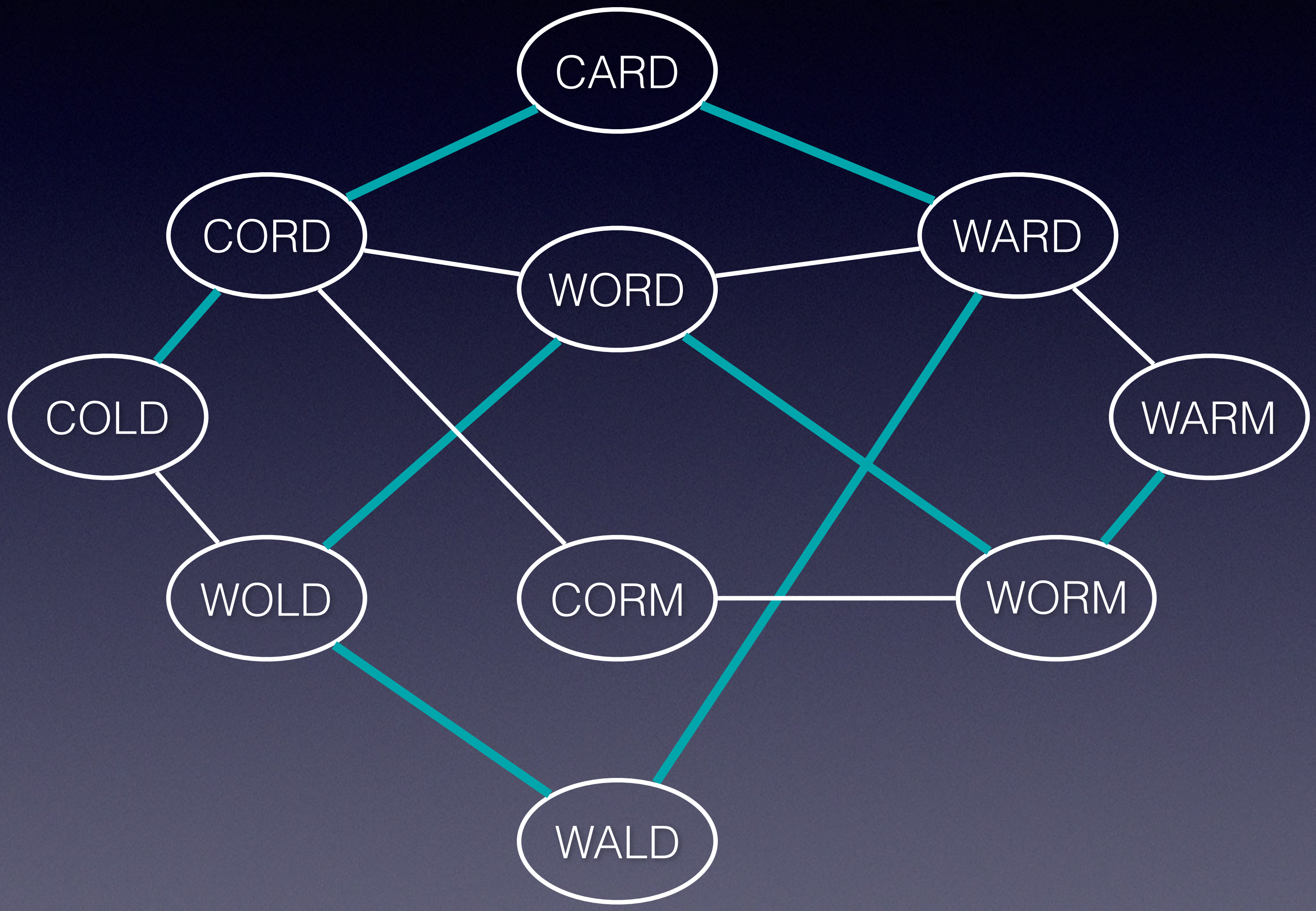


The vertices represent words

What defines an edge?

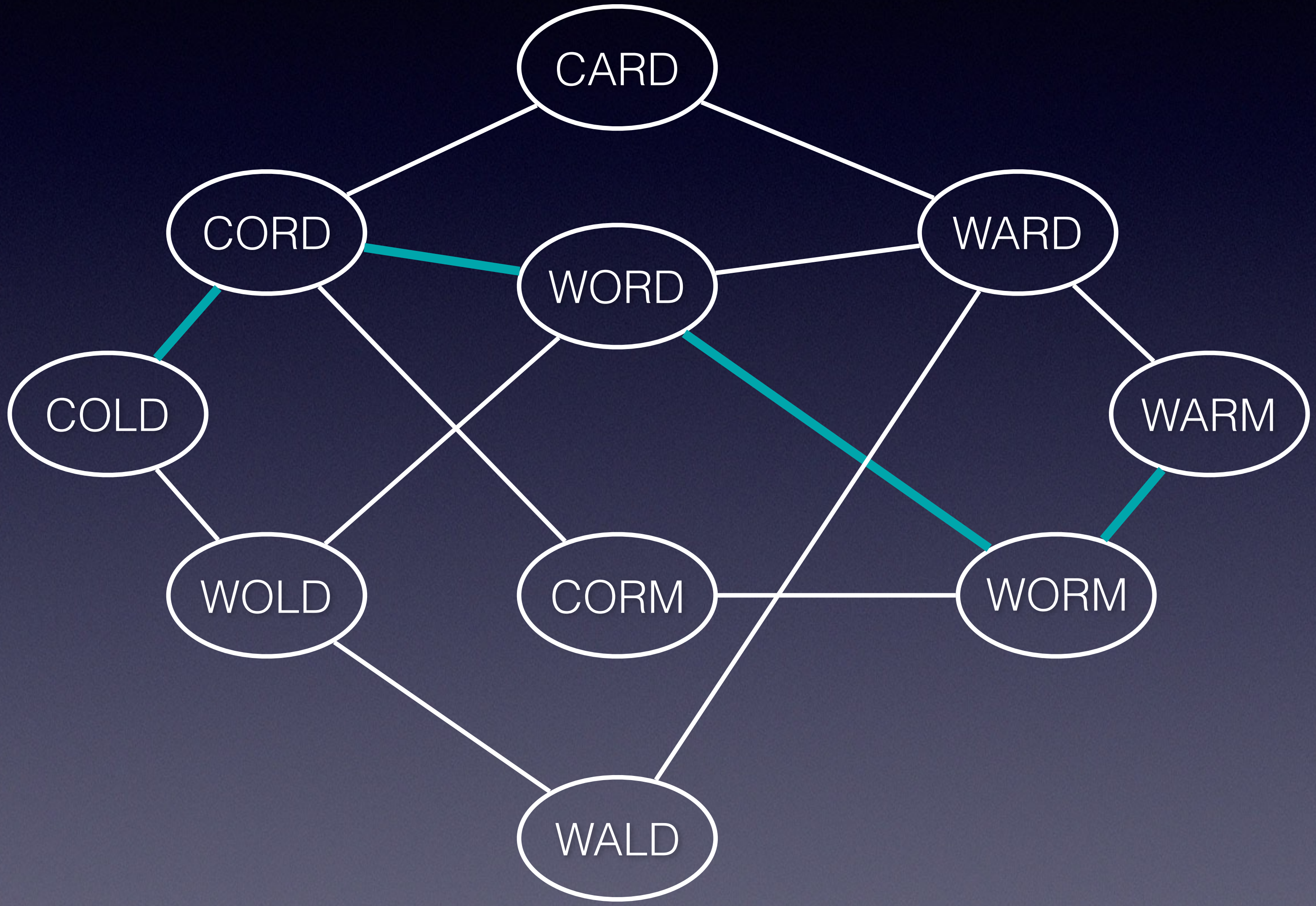
What is a path?

Word Ladders



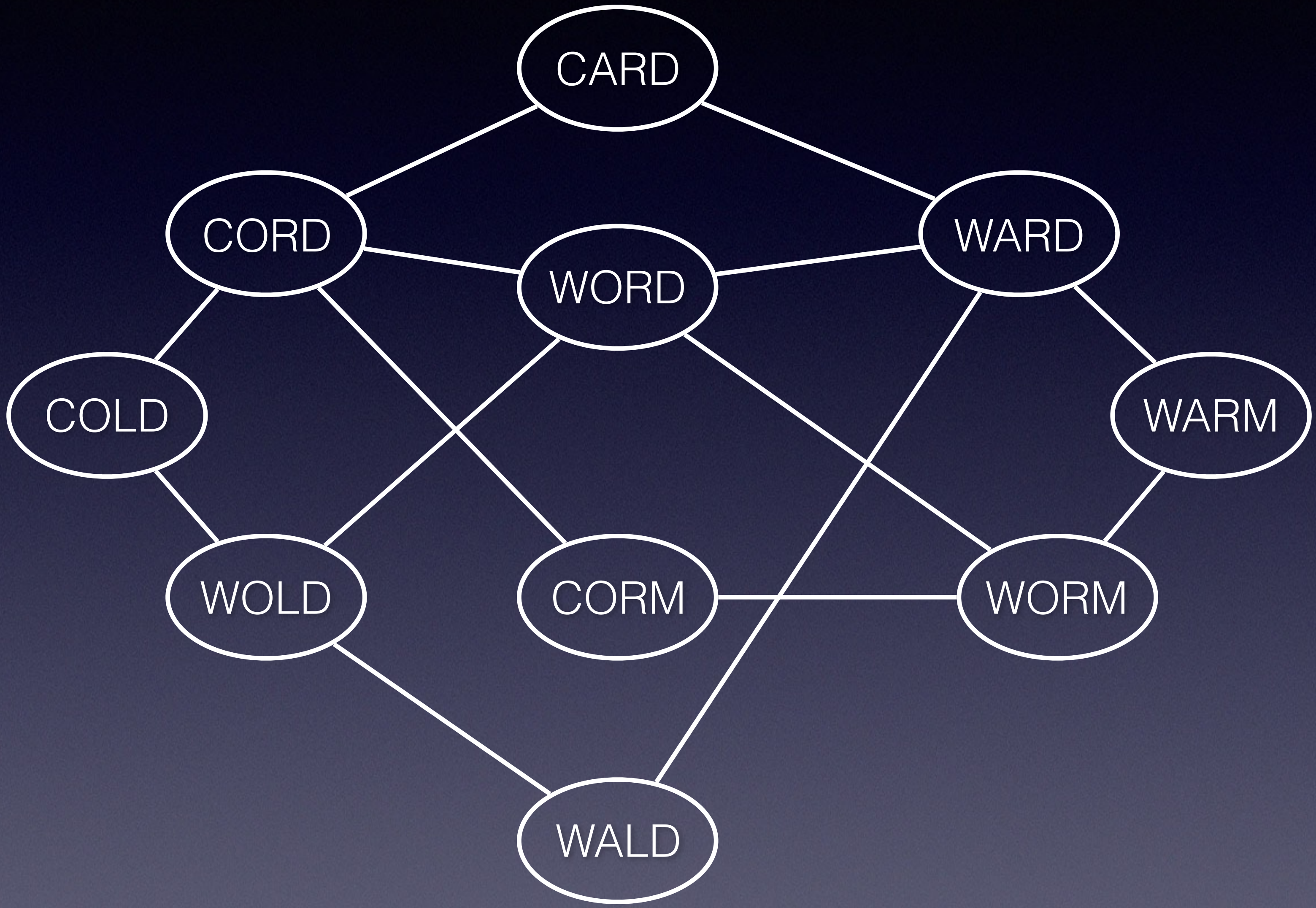
COLD
CORD
CARD
WARD
WALD
WOLD
WORD
WORM
WARM

Word Ladders



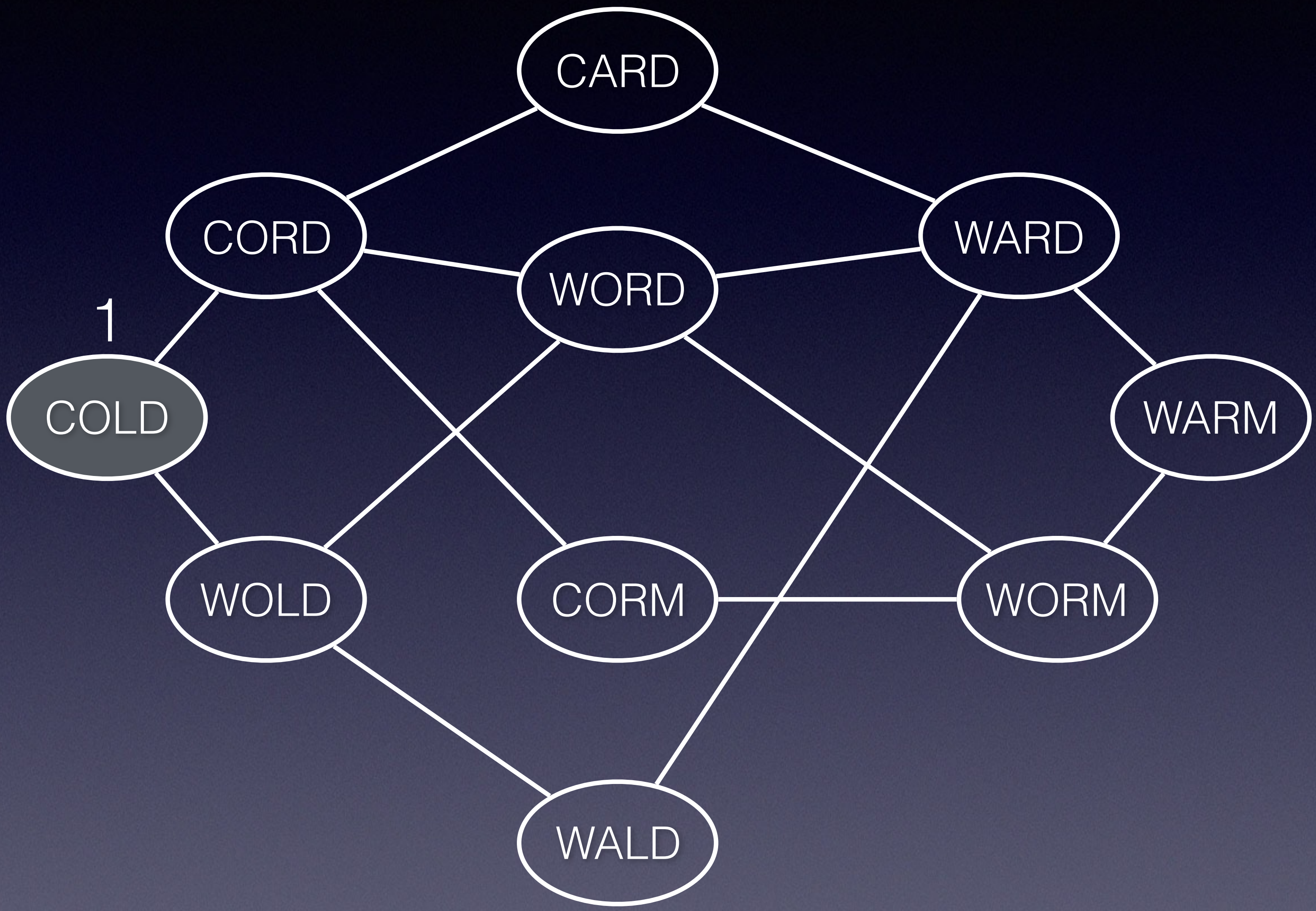
```
COLD
CORD
WORD
WORM
WARM
```

Use breadth first search to find a shortest path



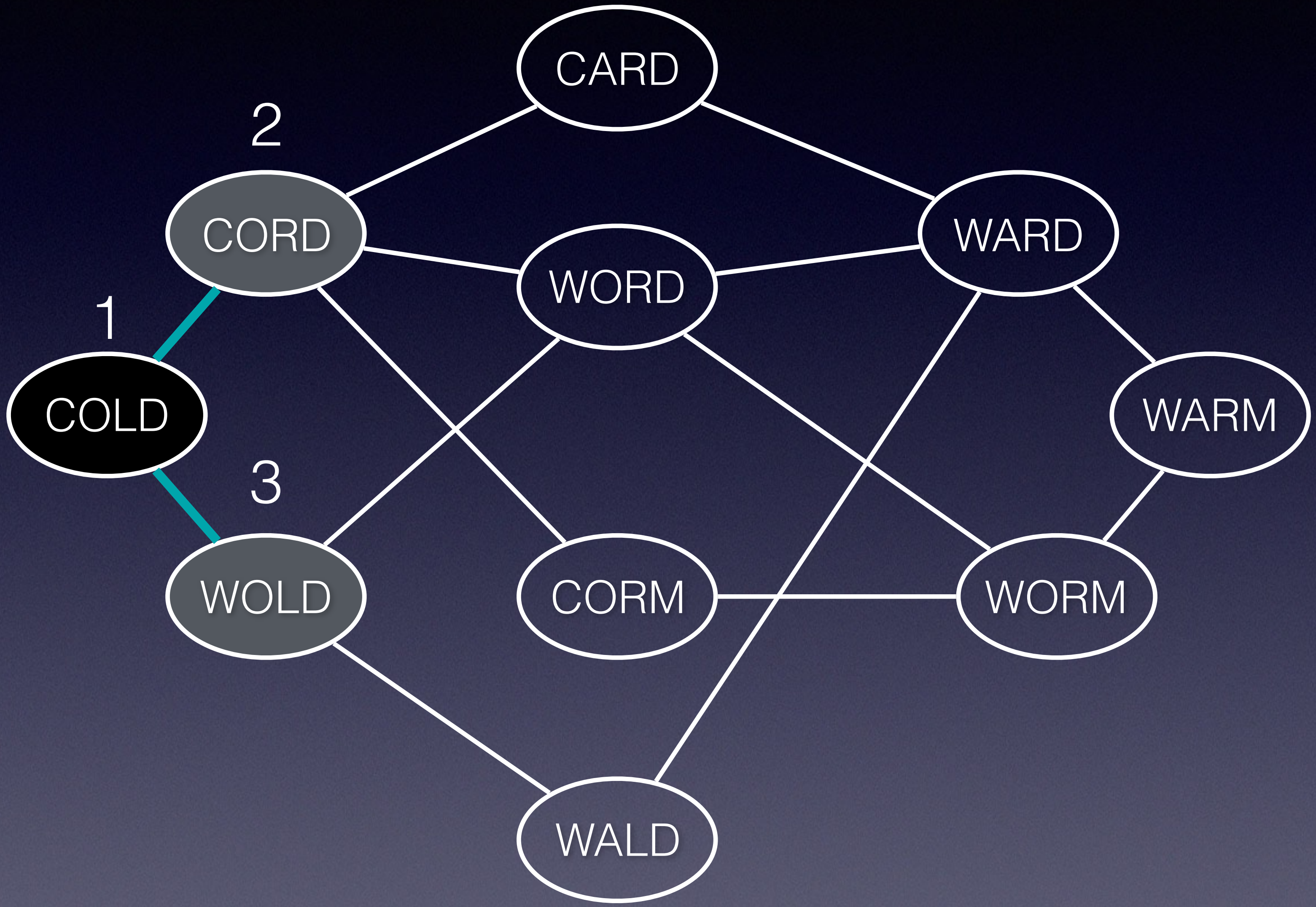
```
Queue: []
Processing:
```

Word Ladders



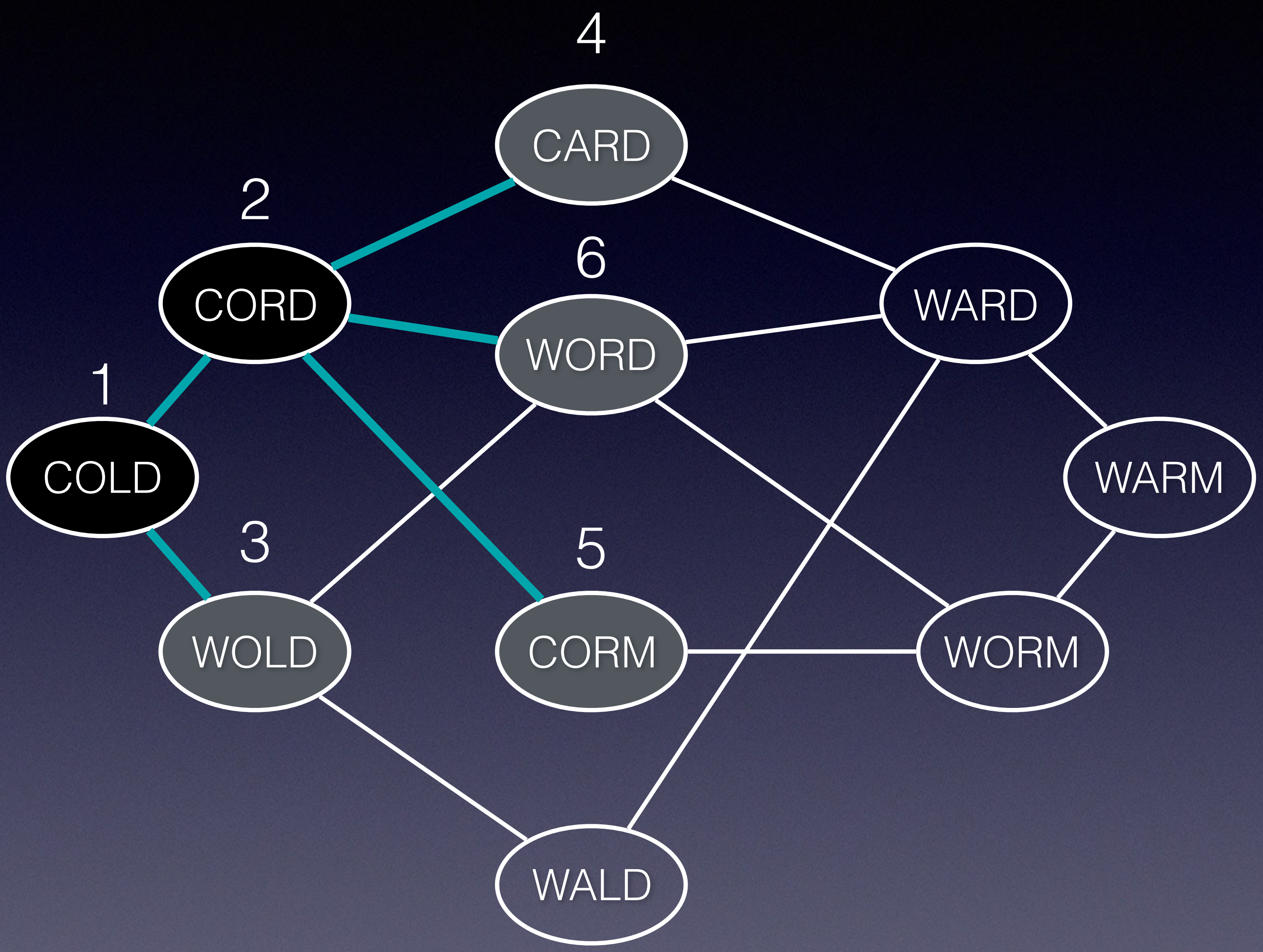
```
Queue: [COLD]
Processing:
```

Word Ladders



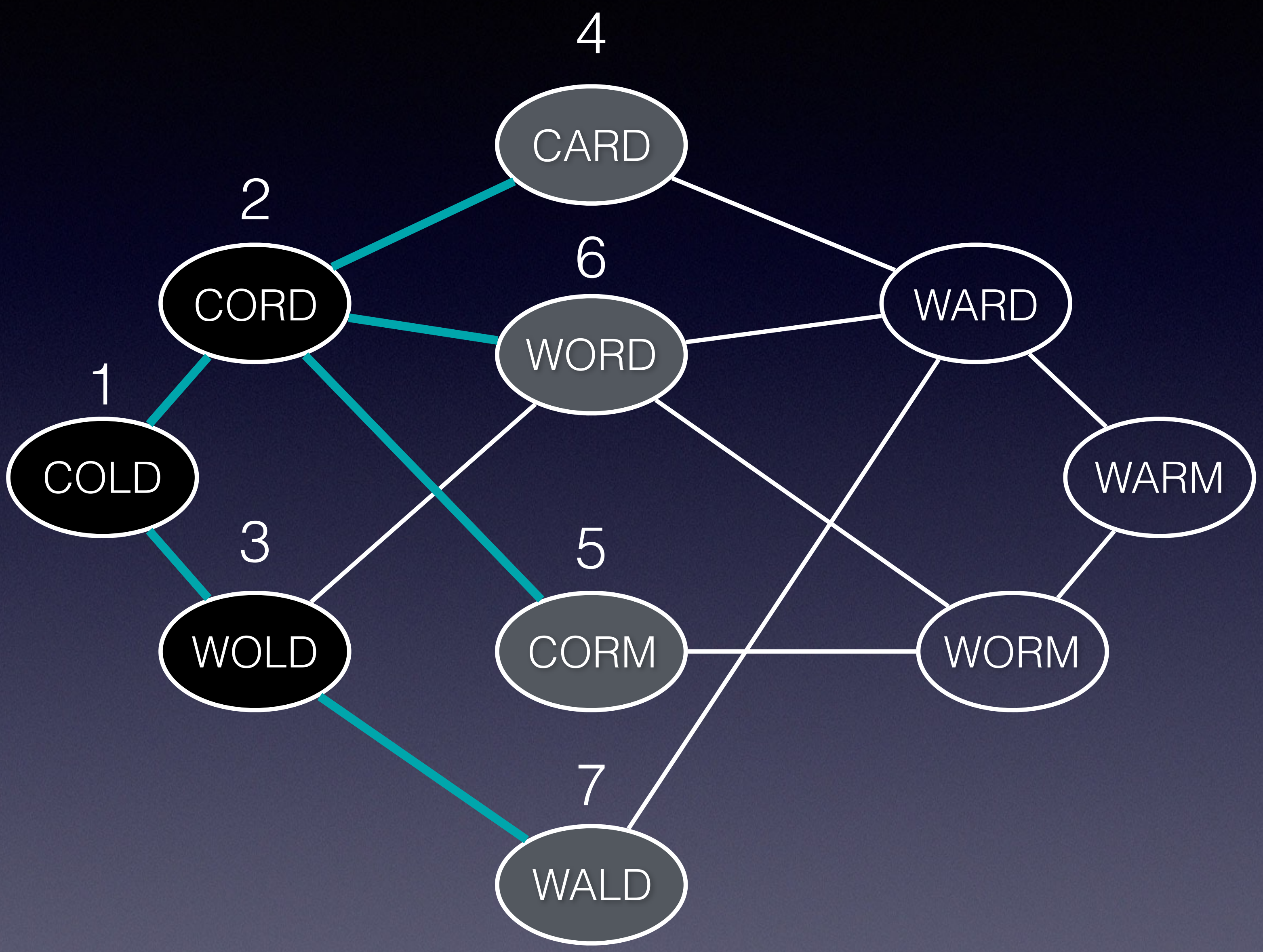
```
Queue: [WOLD,CORD]
Processing: COLD
```

Word Ladders



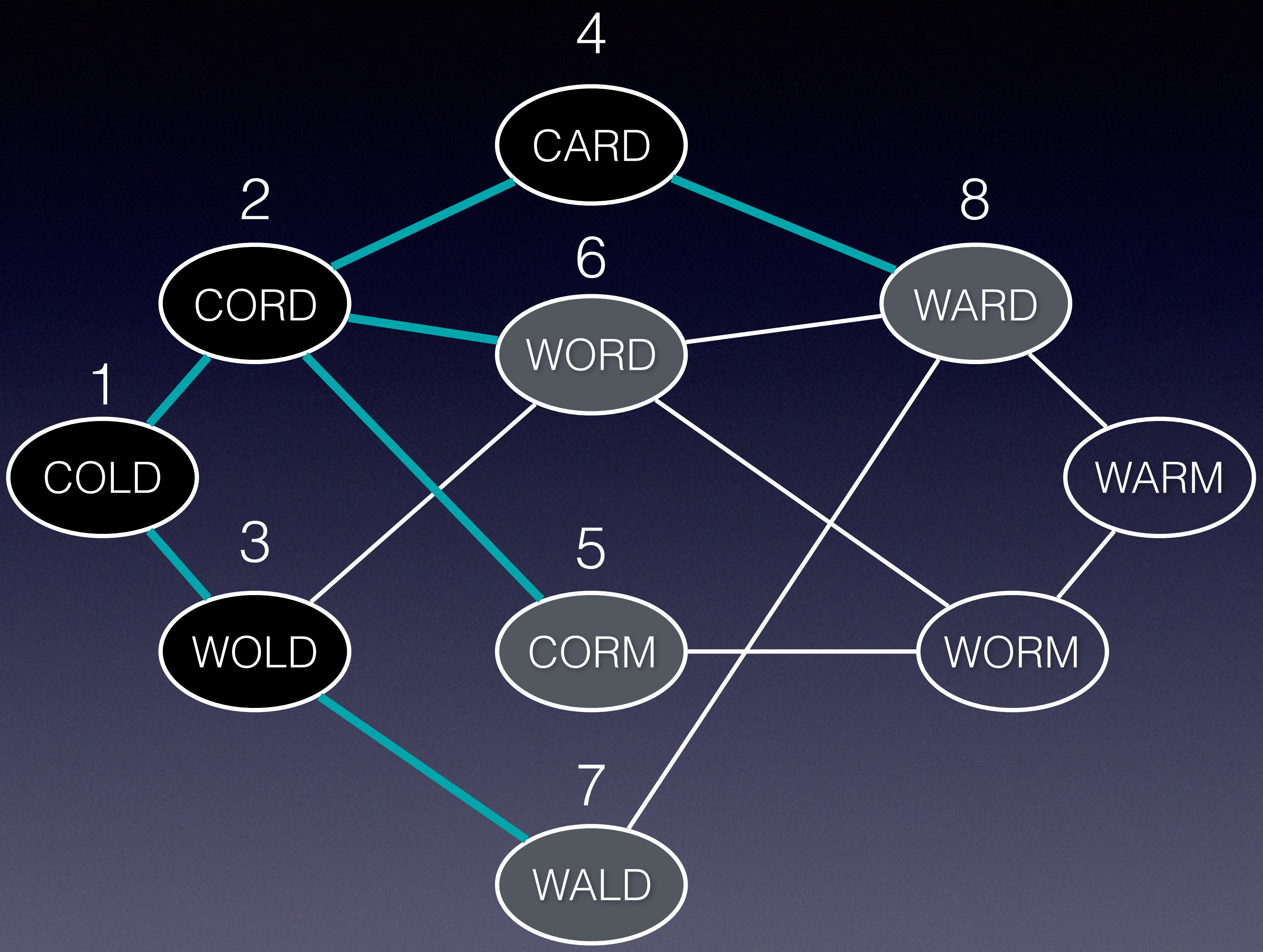
```
Queue: [WORD,CORM,CARD,WOLD]
Processing: CORD
```

Word Ladders



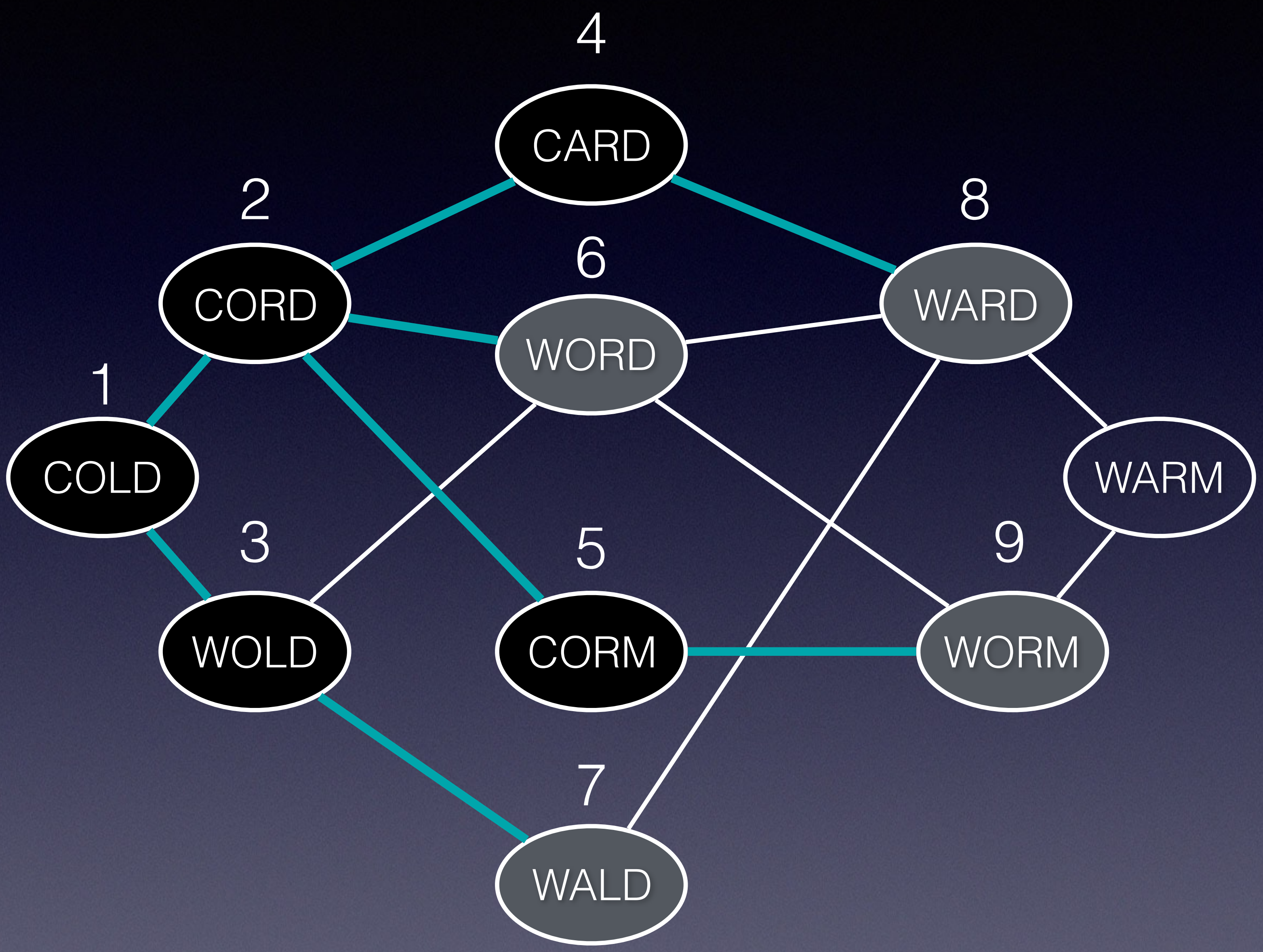
```
Queue: [WALD,WORD,CORM,CARD]
Processing: WOLD
```

Word Ladders



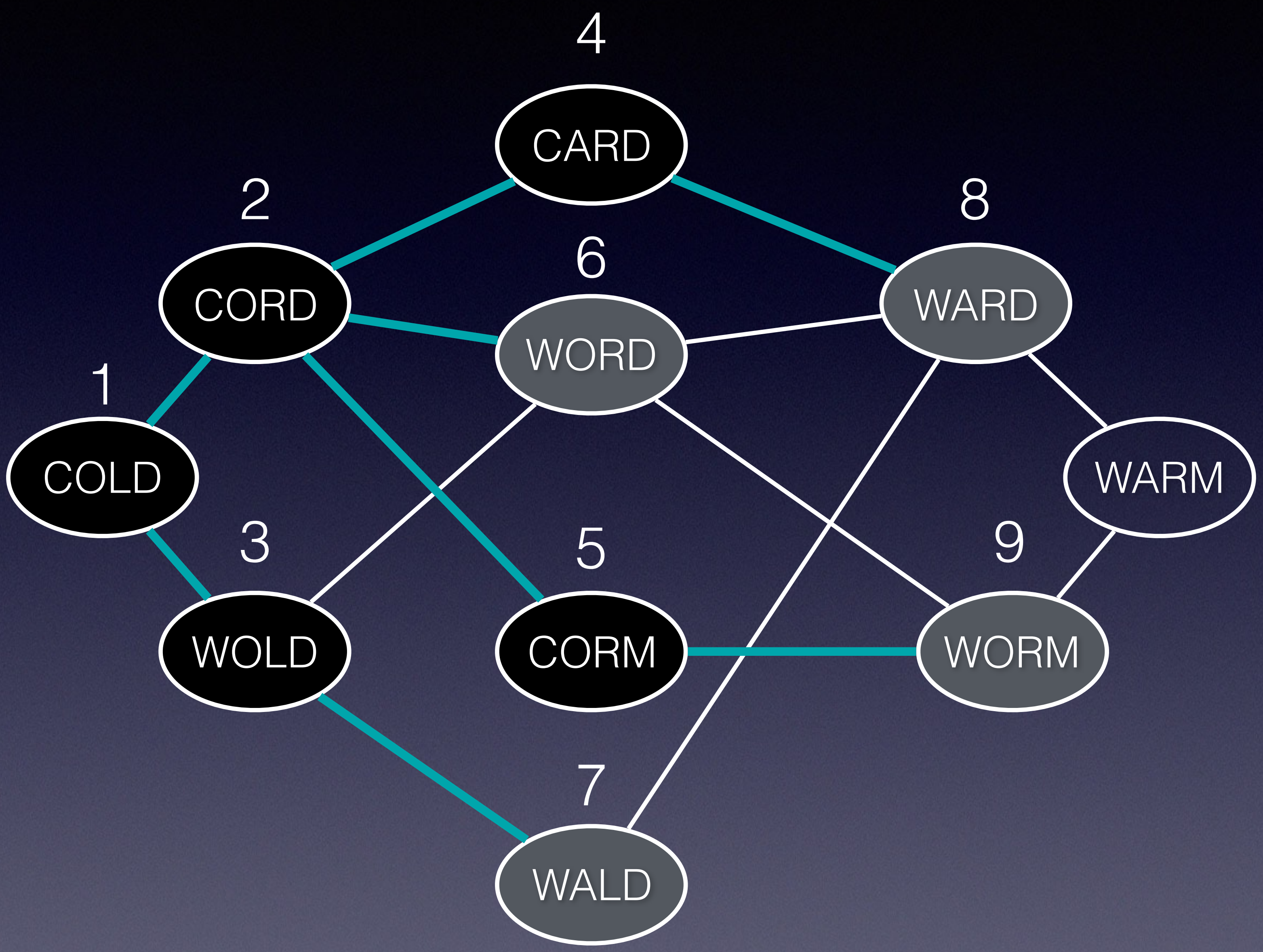
```
Queue: [WARD,WALD,WORD,CORM]
Processing: CARD
```

Word Ladders



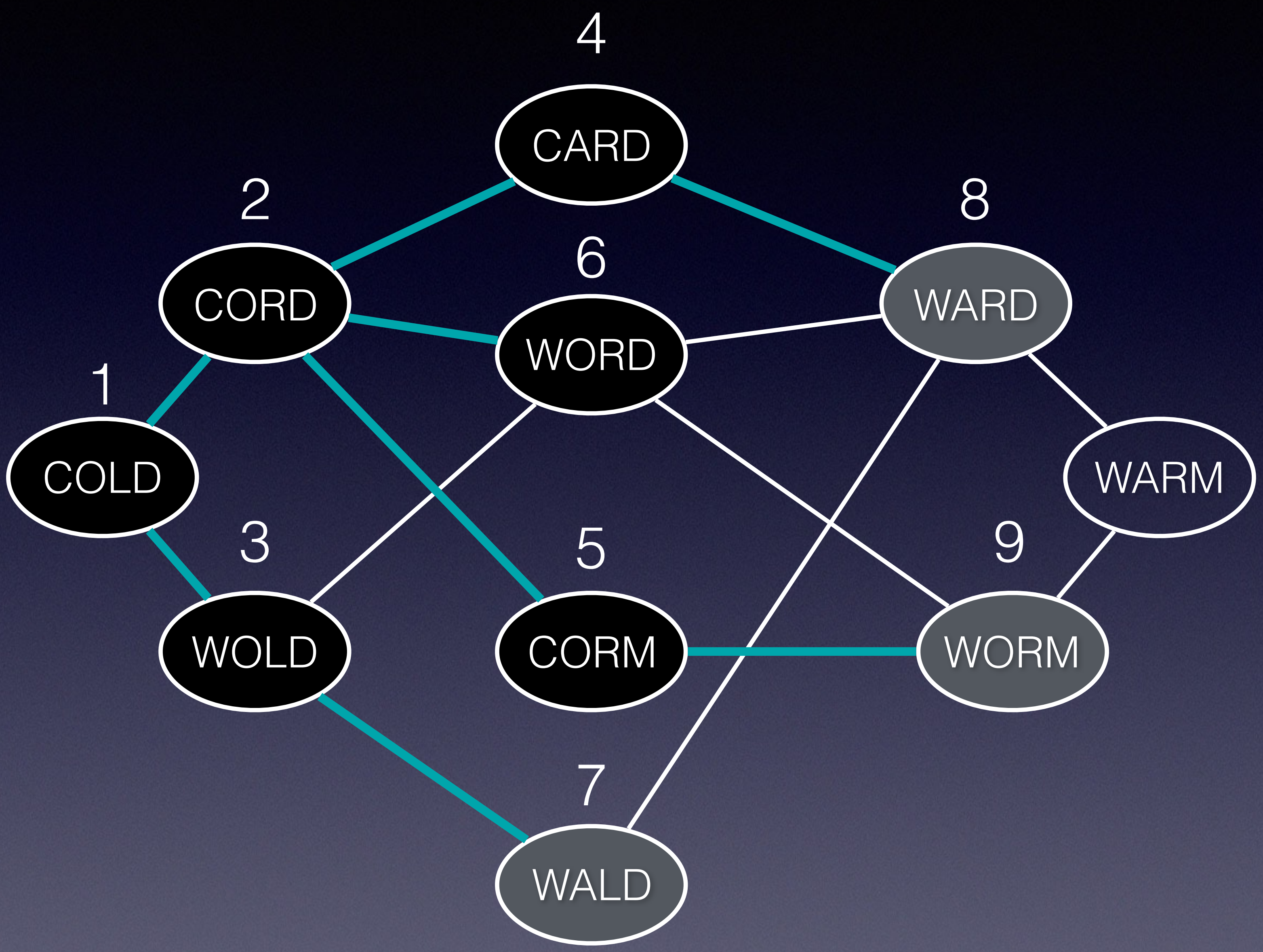
```
Queue: [WORM,WARD,WALD,WORD]
Processing: CORM
```

Word Ladders



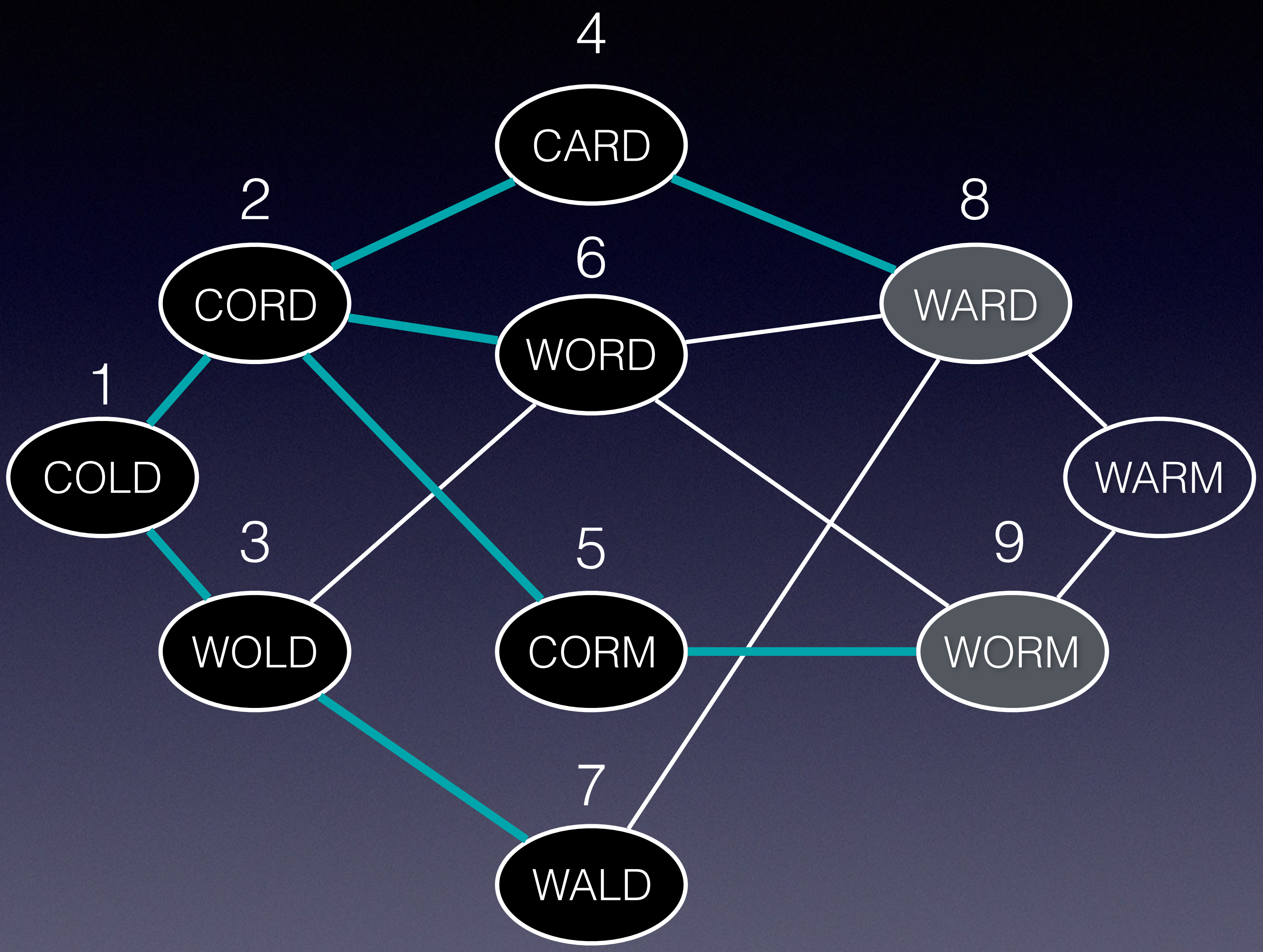
At this point we know the order in which the nodes will be processed. We get the last node for free. But let's continue

Word Ladders



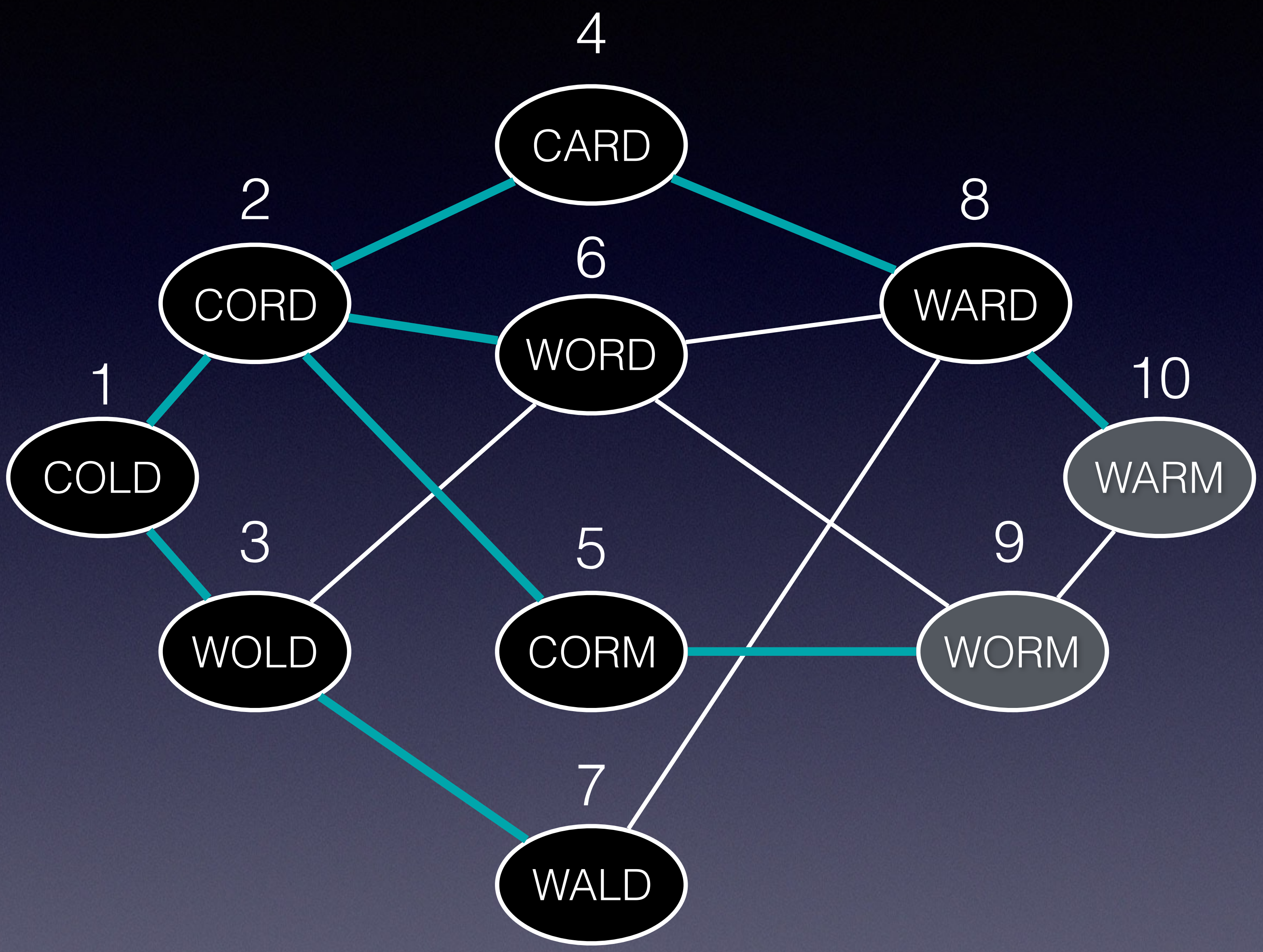
```
Queue: [WORM,WARD,WALD]
Processing: WORD
```

Word Ladders



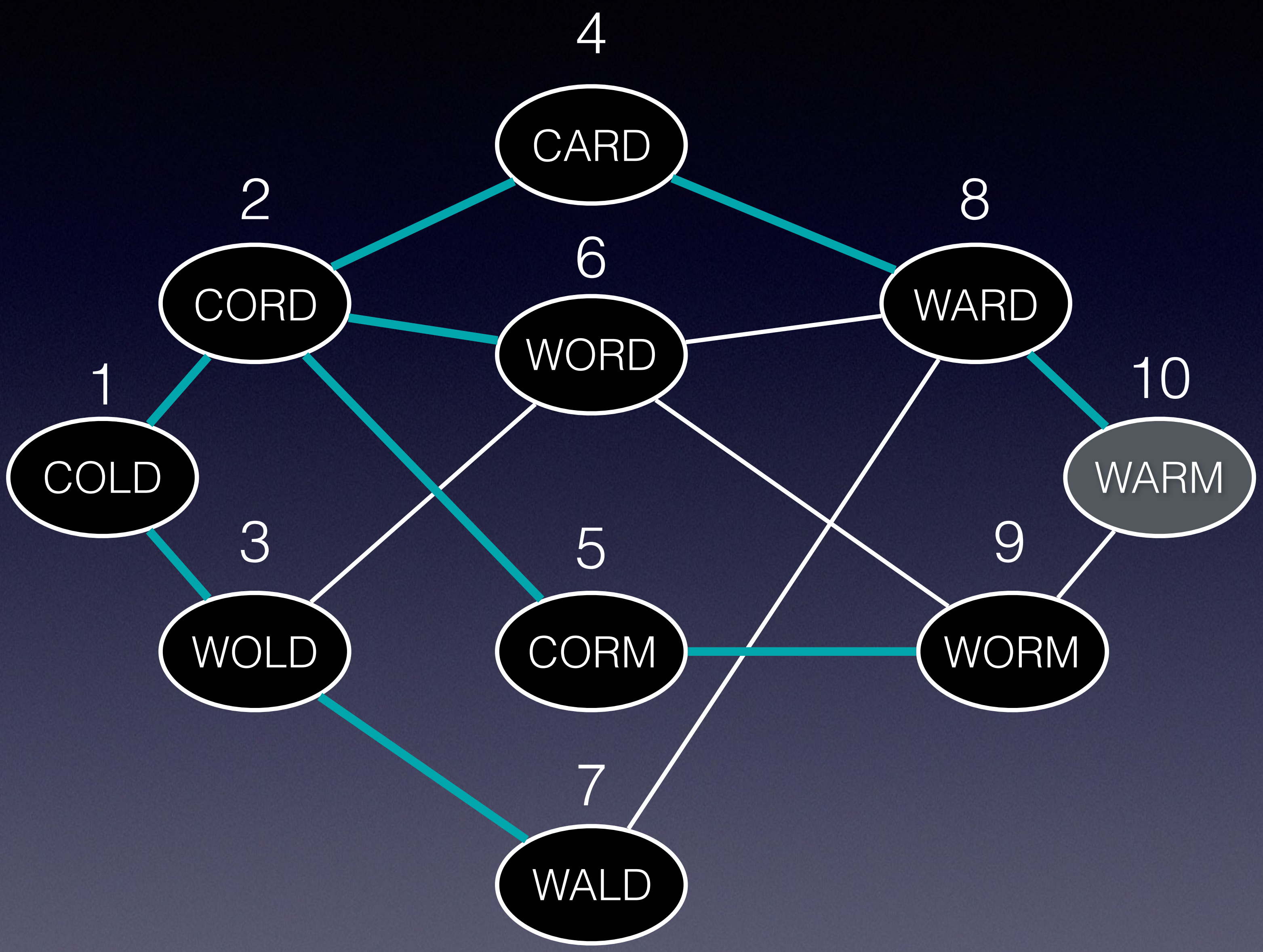
```
Queue: [WORM,WARD]
Processing: WALD
```

Word Ladders



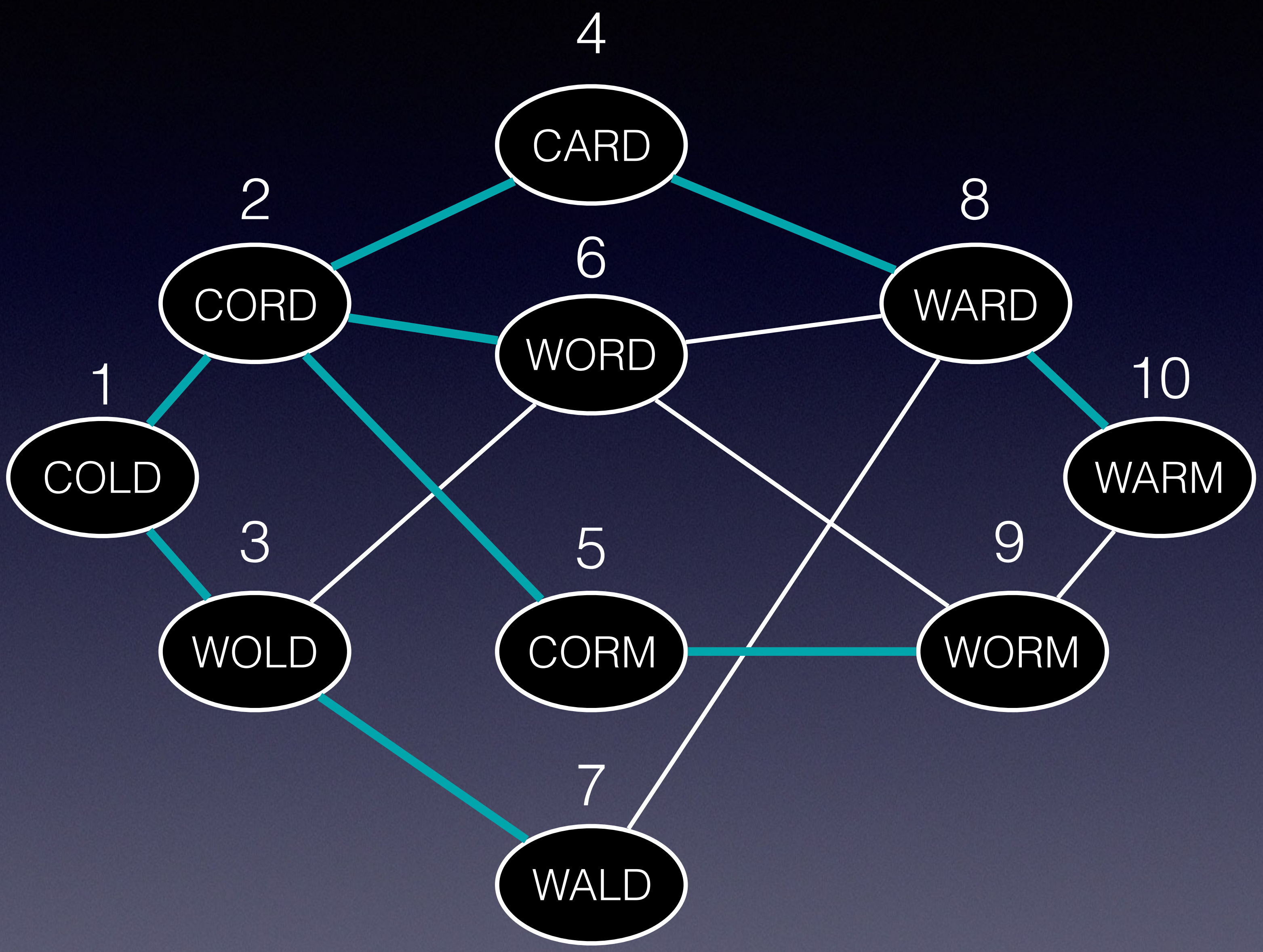
```
Queue: [WARM,WORM]
Processing: WARD
```

Word Ladders



```
Queue: [WARM]
Processing: WORM
```

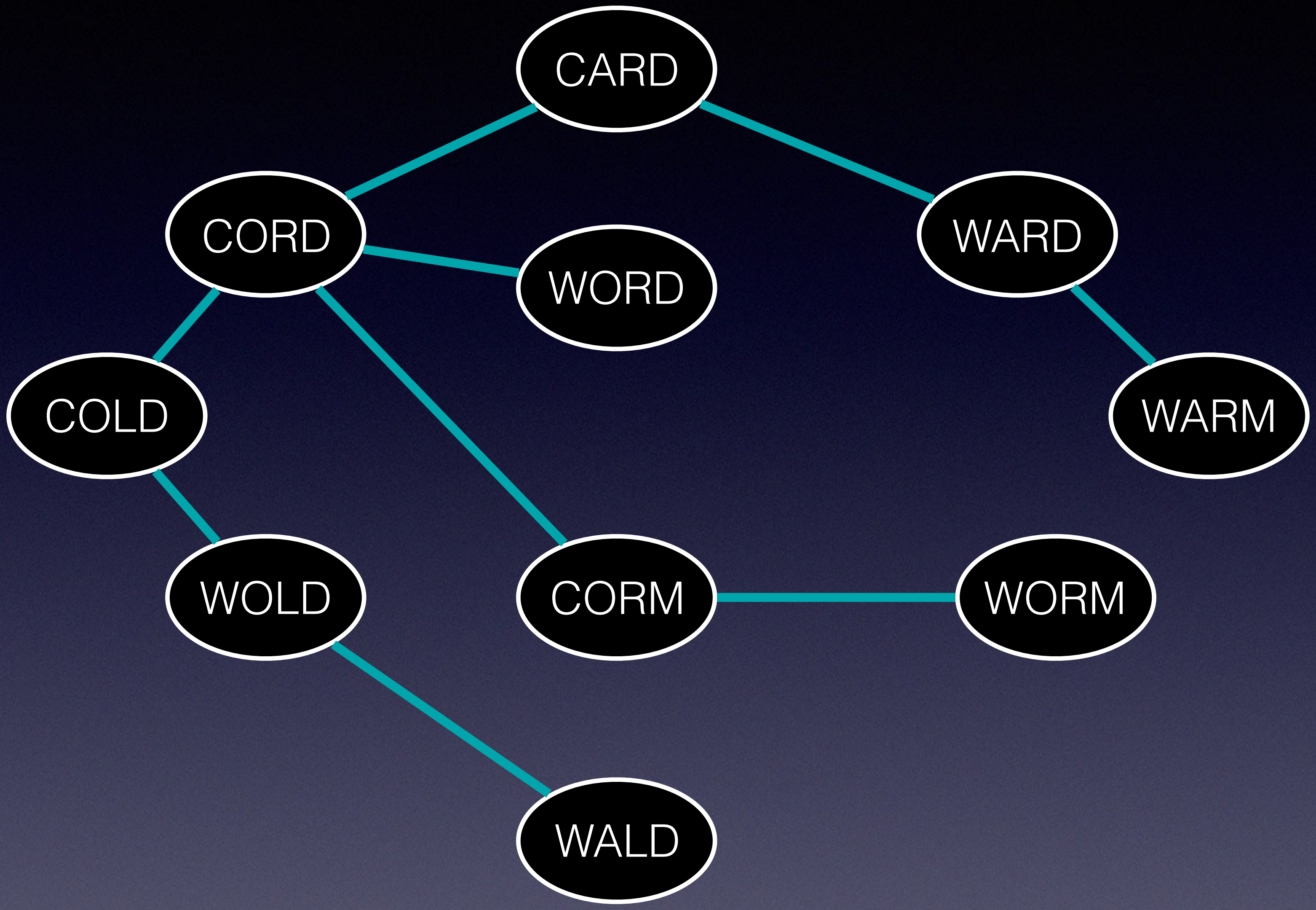
Word Ladders



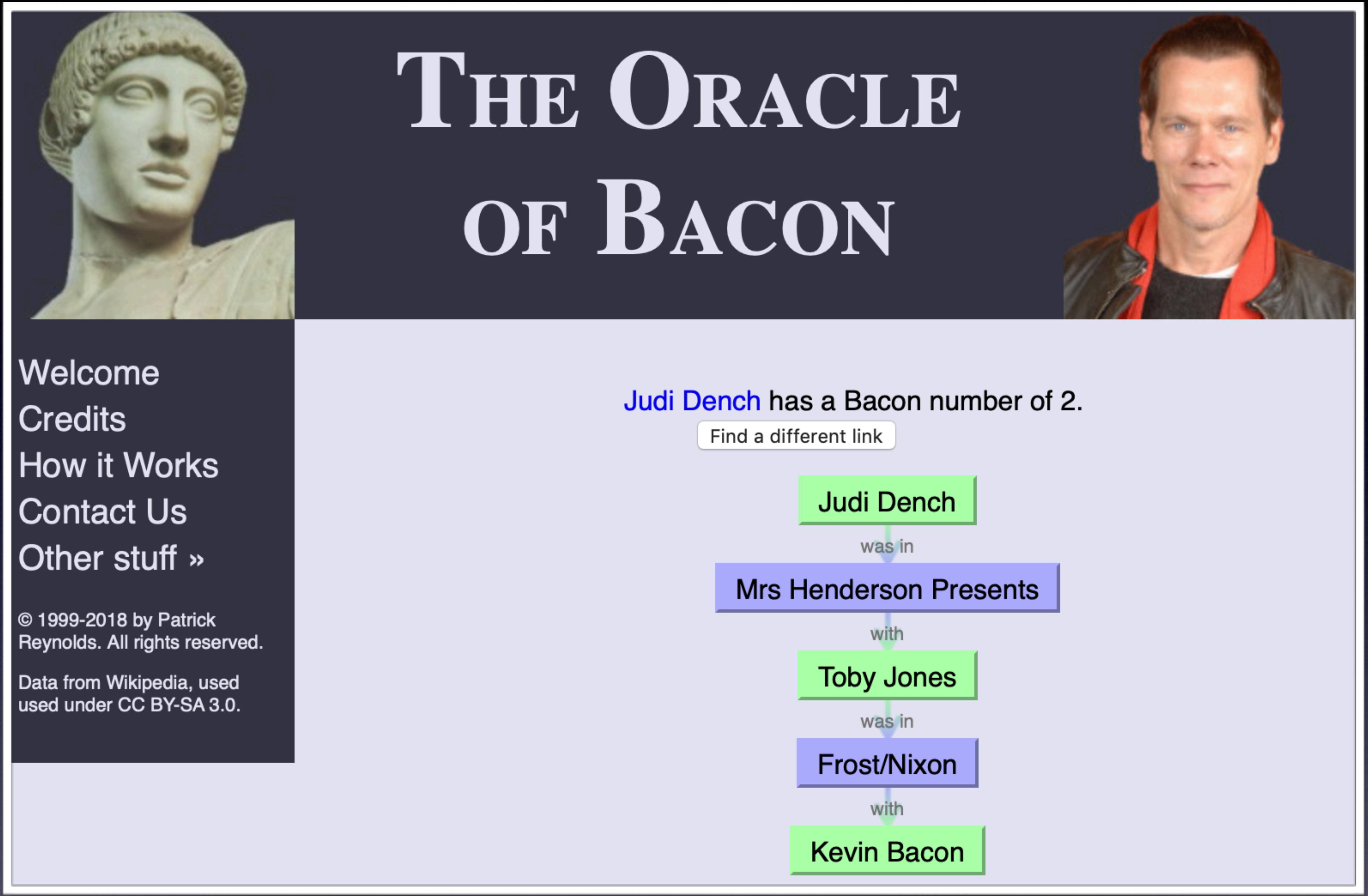
```
Queue: []
Processing: WARM
```

Done processing nodes

Word Ladders



Notice that we have built a tree. Each path from the root to a leaf is a shortest word ladder from the root word to the leaf word. Though as we showed earlier, there may be others equally short


THE ORACLE OF BACON

Welcome
Credits
How it Works
Contact Us
Other stuff »

© 1999-2018 by Patrick Reynolds. All rights reserved.

Data from Wikipedia, used used under CC BY-SA 3.0.

Judi Dench has a Bacon number of 2.

Find a different link

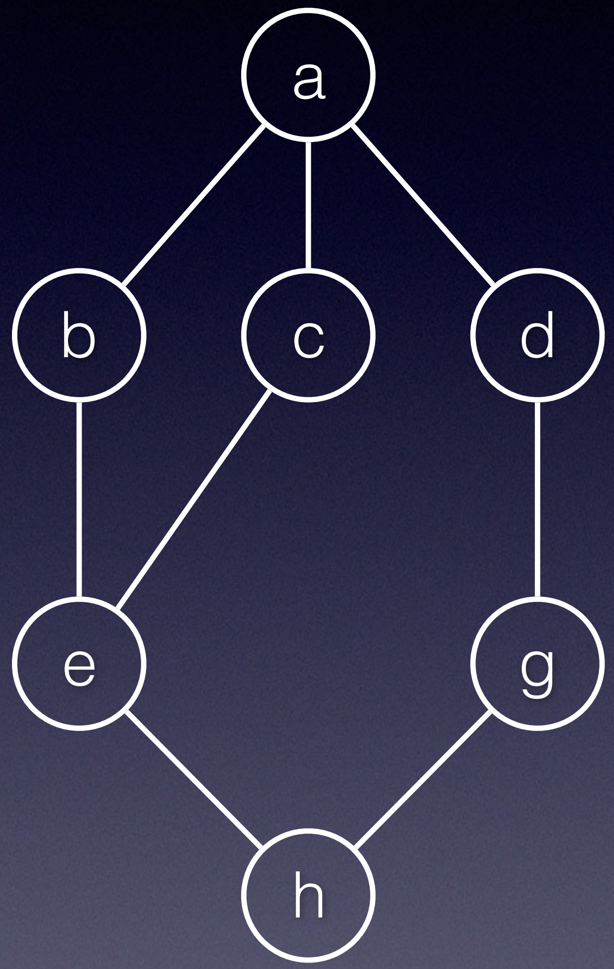
Judi Dench
was in
Mrs Henderson Presents
with
Toby Jones
was in
Frost/Nixon
with
Kevin Bacon

Do you know about the six degrees of separation?
How about the six degrees of Kevin Bacon?
How would you use a graph here?

Depth-first concept

- Starting at a source vertex s
- Follow a simple path discovering new vertices until you cannot find a new vertex
- Back-up until you can start finding new vertices
- Creates a "depth first tree"

Depth-first search



What's the order in which these vertices will be visited?

Assume that when we have a choice of which edge to follow, we always follow the edge connected to the alphabetically smallest vertex first.

Iterative depth-first algorithm

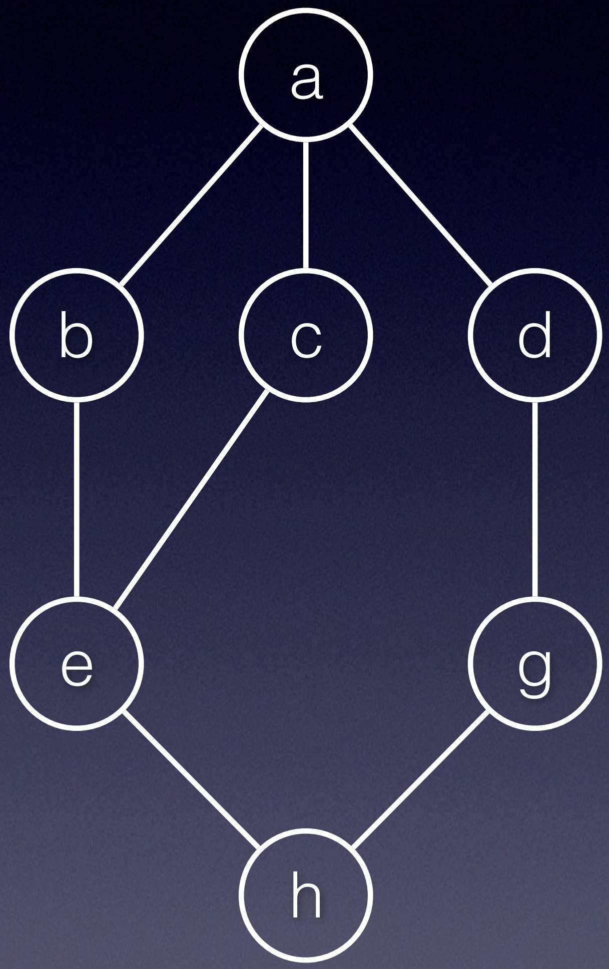
```
Take a start vertex and push it on a stack.

While the stack is not empty
    Pop a vertex, u, from the stack (Begin visiting u)
    If not visited, mark it visited (color it **grey**)
     For all vertices v, adjacent to u: (ordered z-a)*
        If v is still **white**
            Push it on the stack
     We are now done visiting u (color it **black**)
```

This is the same as the breadth-first algorithm, except (1) it uses a **stack** instead of a **queue**, and (2) it marks a vertex as identified after it is popped from the stack, not before.

* For a iterative algorithm, to visit nodes from a-z, we need to reverse their order before pushing them onto the stack.

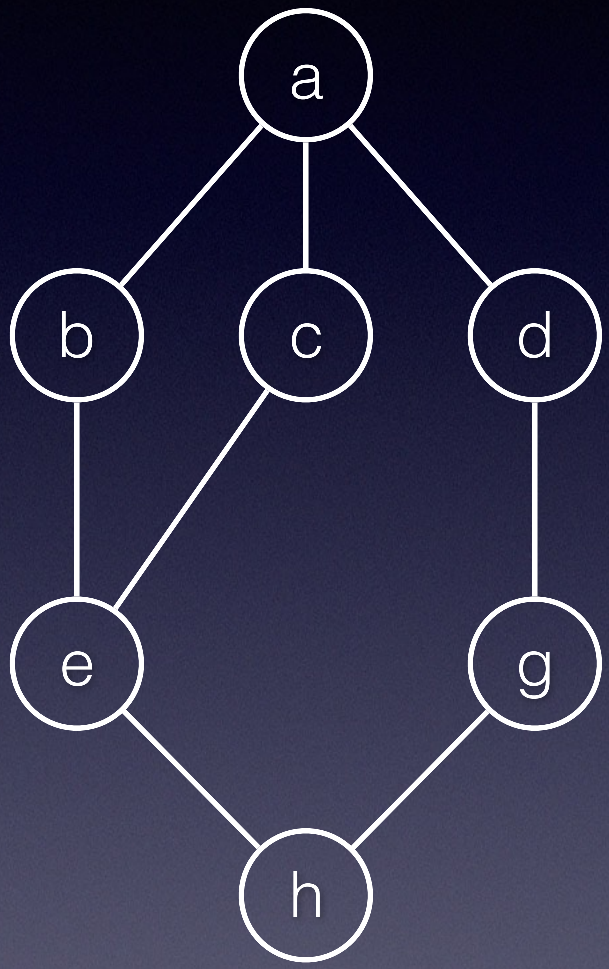
Depth-first search



Stack: []   (right is top of stack)

Visiting:

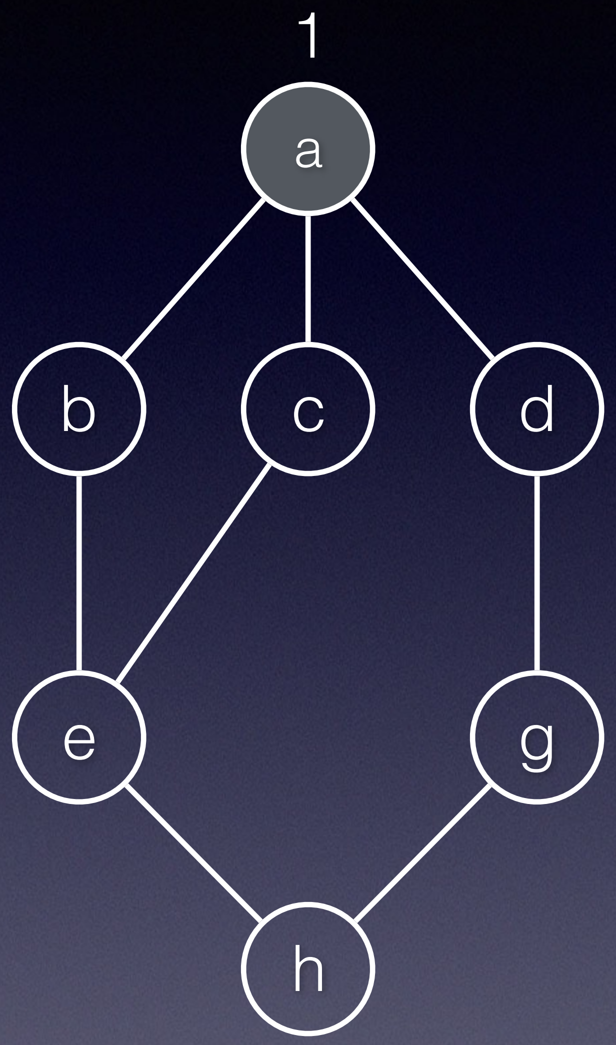
Depth-first search



Stack: [a]   (right is top of stack)

Visiting:

Depth-first search

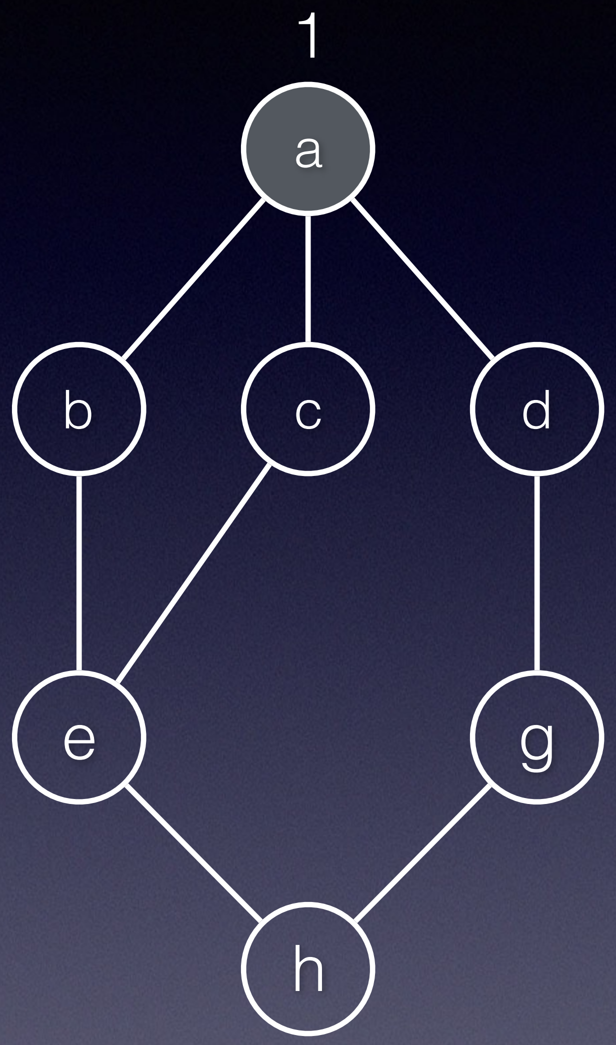


Stack: []

Visiting:a

We will number the vertices in the order they are first visited. This is not essential to implementing the algorithm

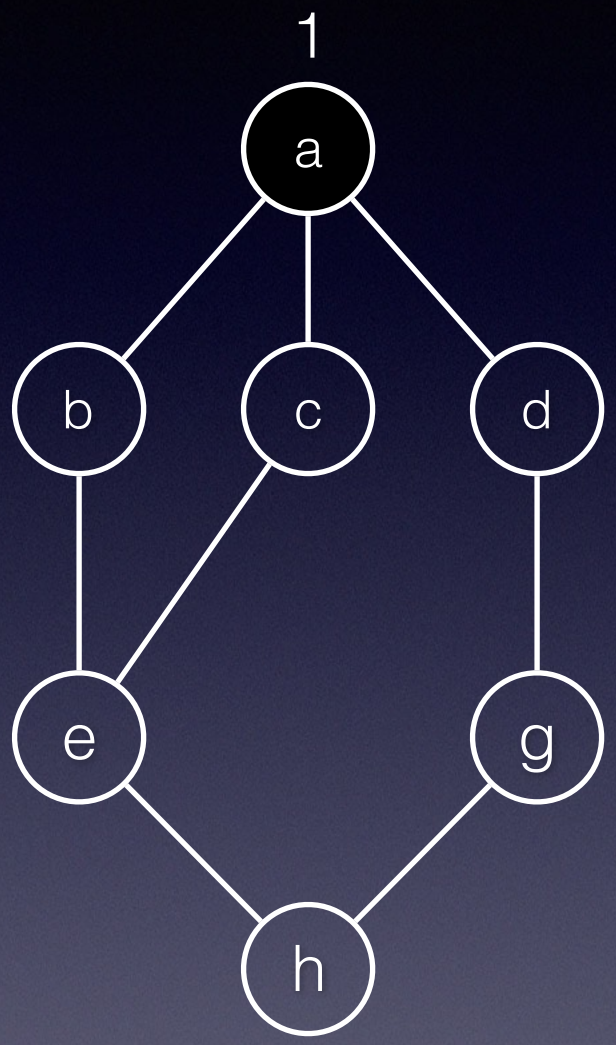
Depth-first search



Stack: [d,c,b]

Visiting:a

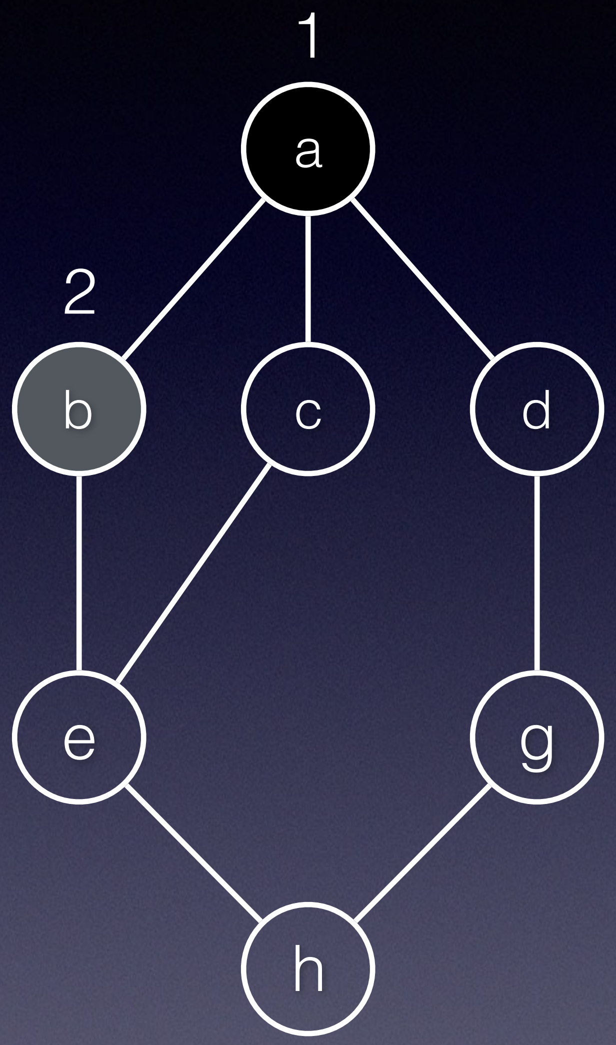
Depth-first search



Stack: [d,c,b]

Visiting:

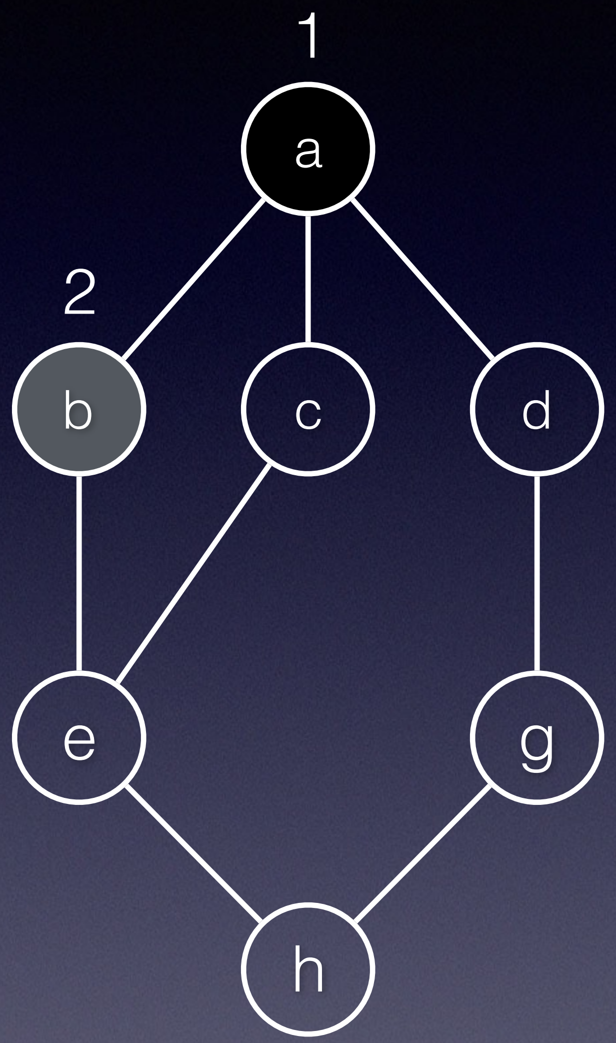
Depth-first search



Stack: [d,c]

Visiting: b

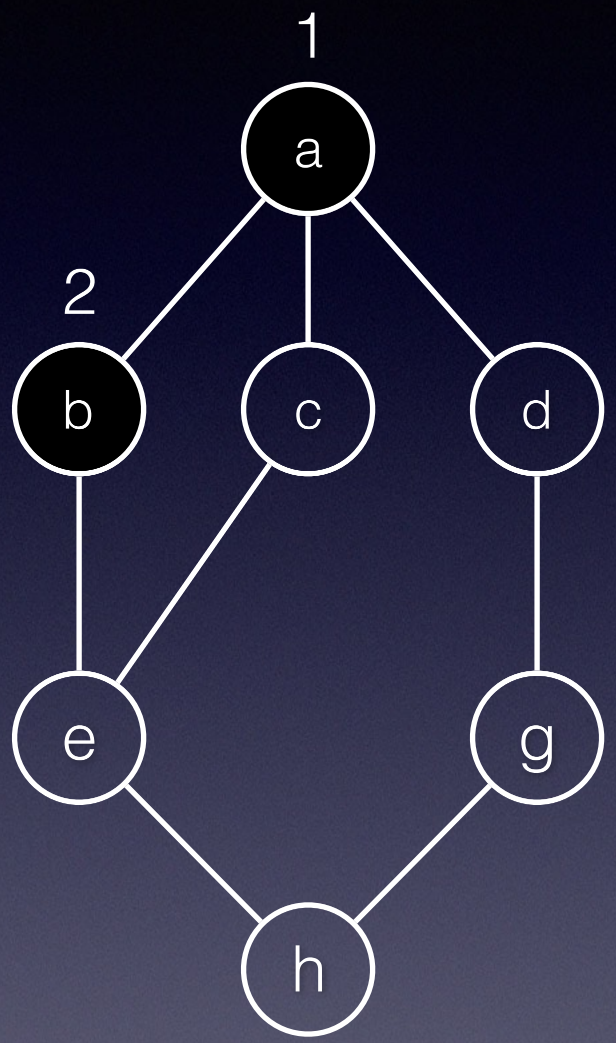
Depth-first search



Stack: [d,c,e]

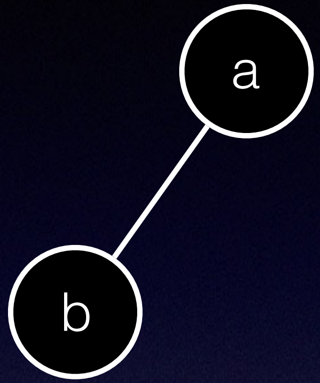
Visiting: b

Depth-first search

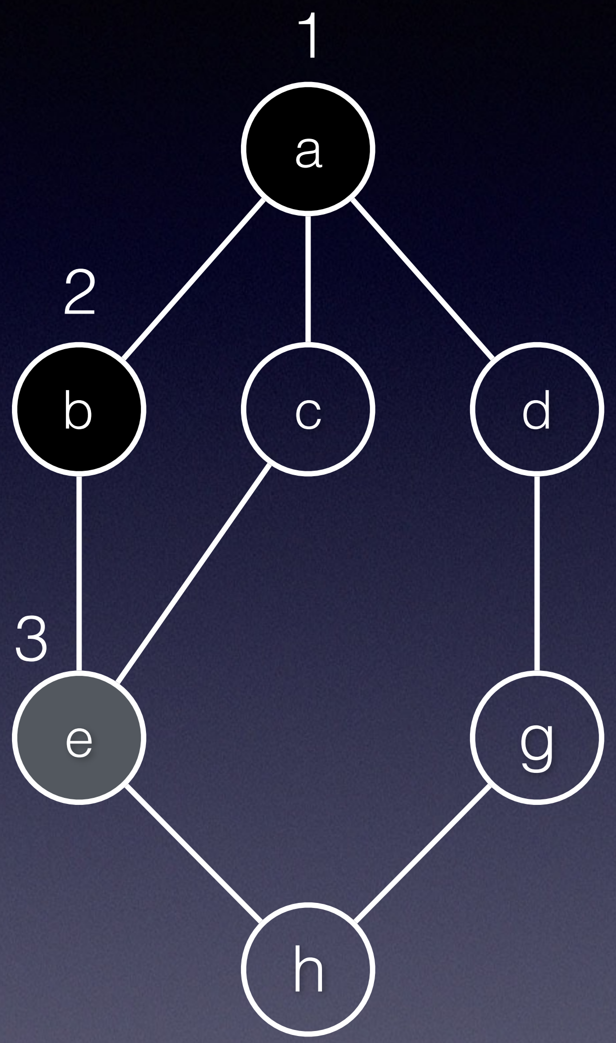


Stack: [d,c,e]

Visiting:

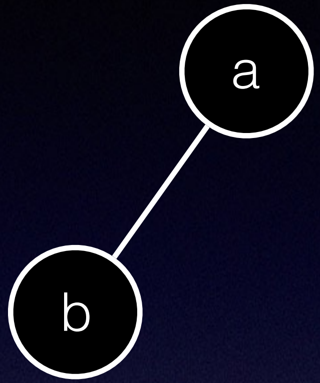


Depth-first search

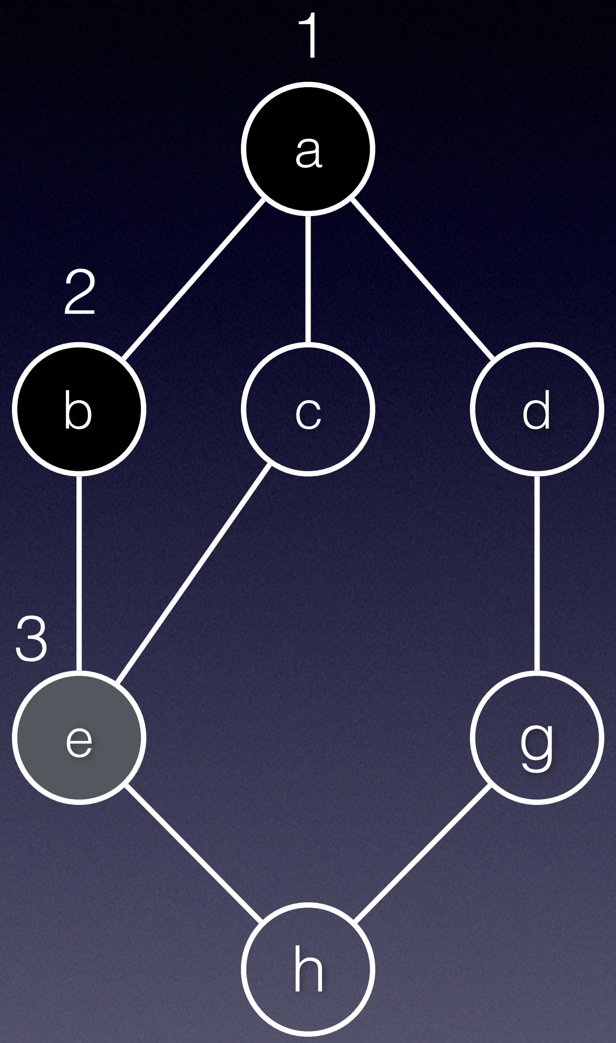


Stack: [d,c]

Visiting: e

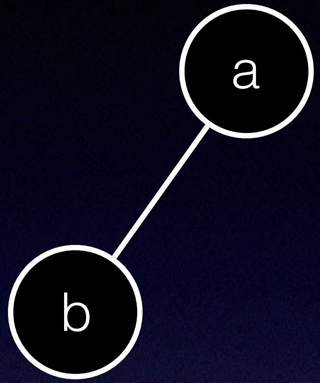


Depth-first search

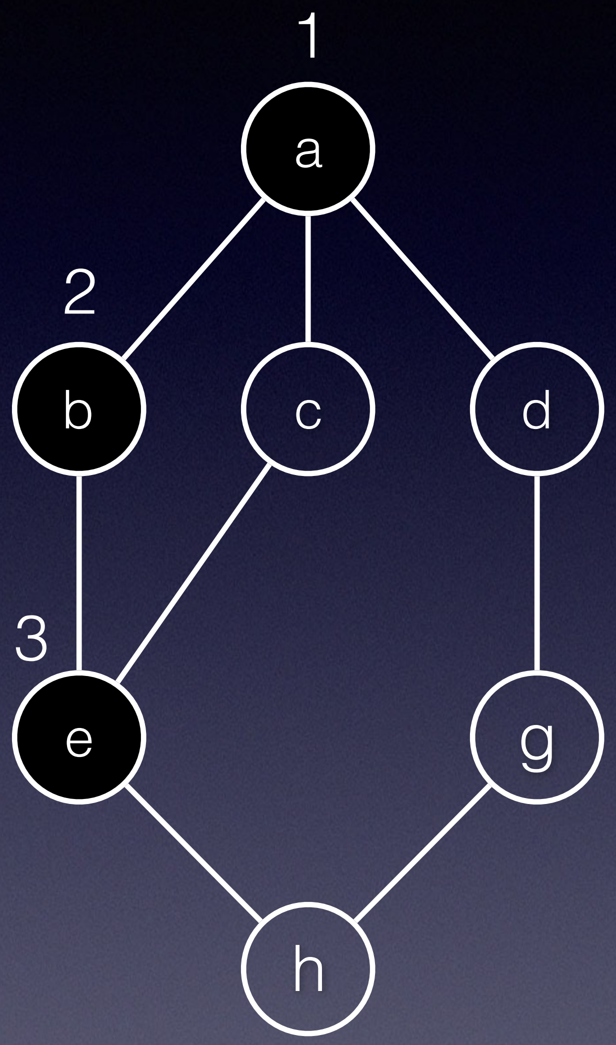


Stack: [d,c,h,c]

Visiting: e

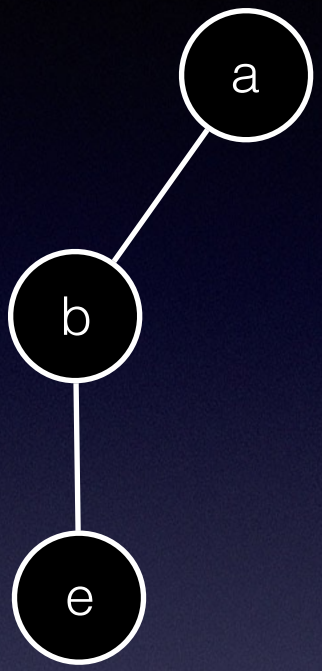


Depth-first search

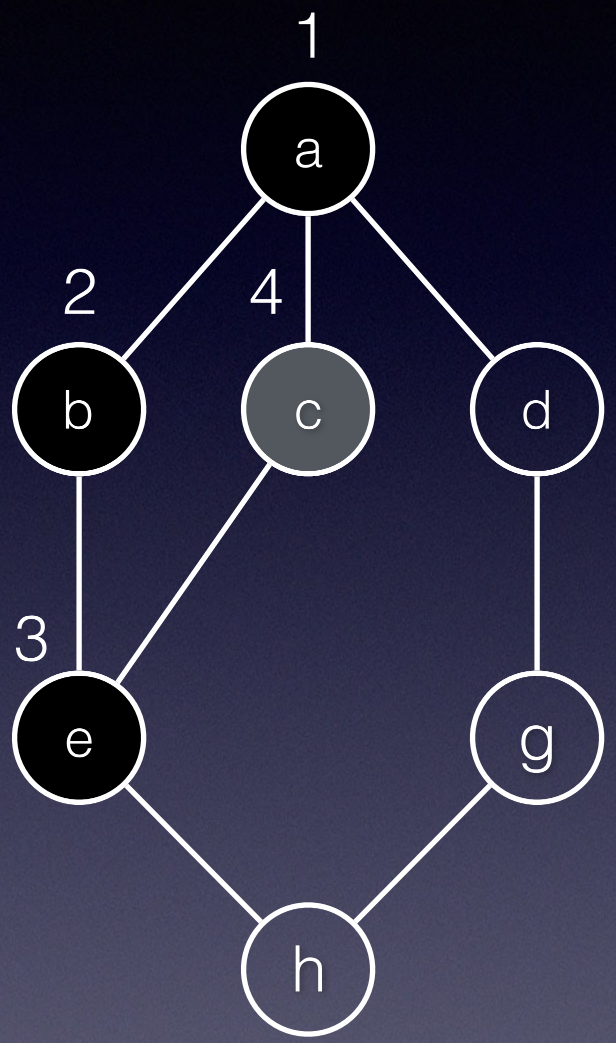


Stack: [d,c,h,c]

Visiting:

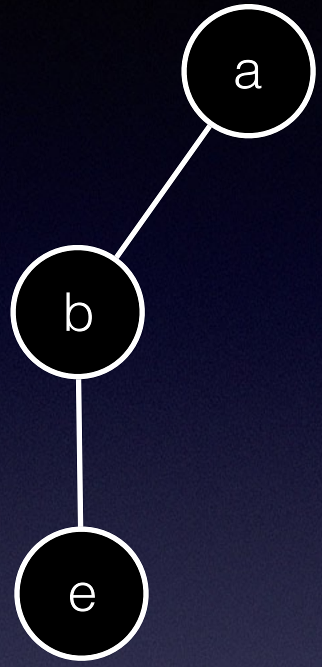


Depth-first search

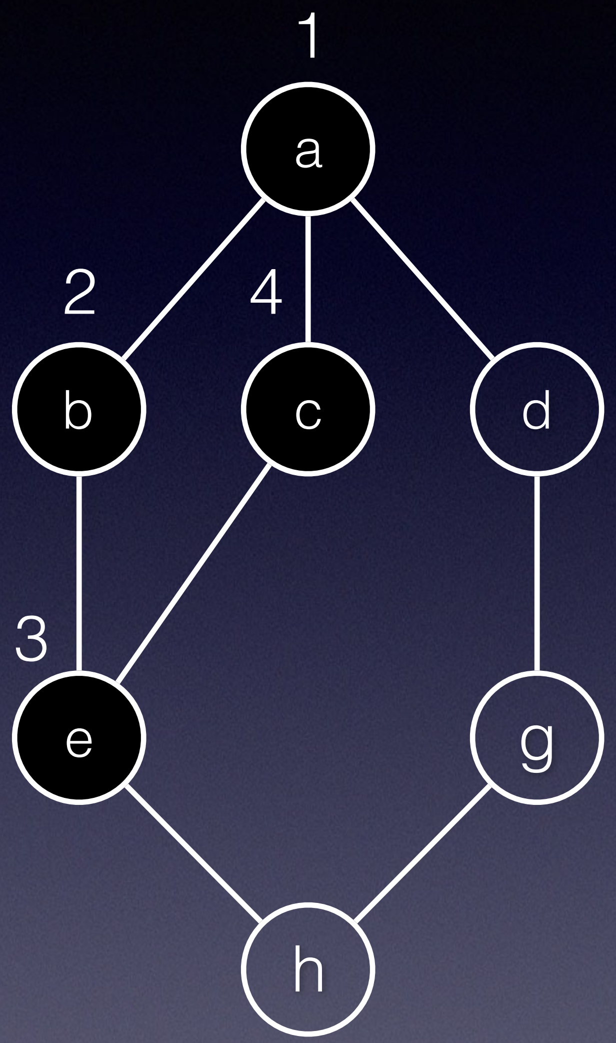


Stack: [d,c,h]

Visiting: c

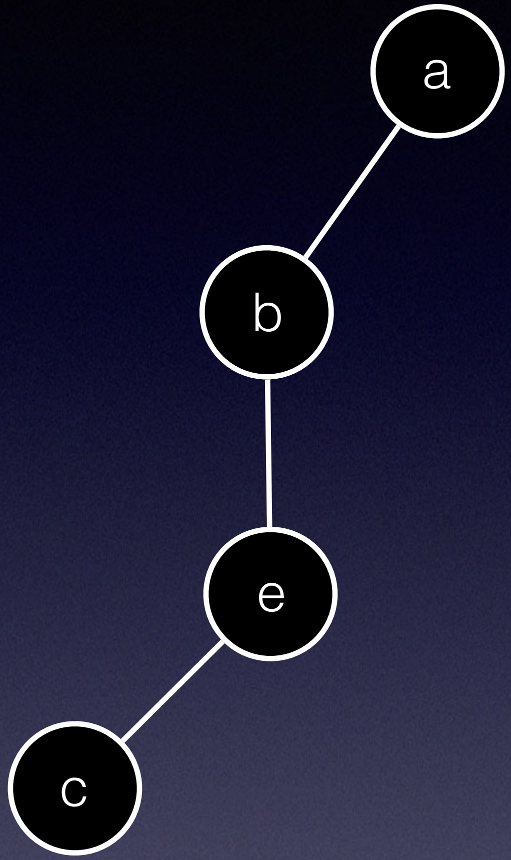


Depth-first search

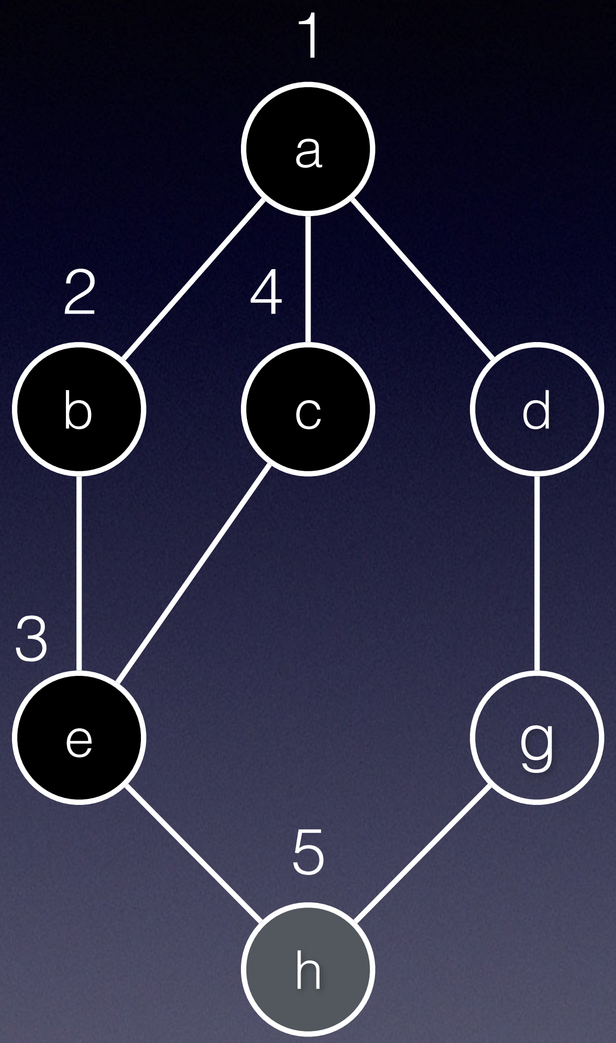


Stack: [d,c,h]

Visiting:

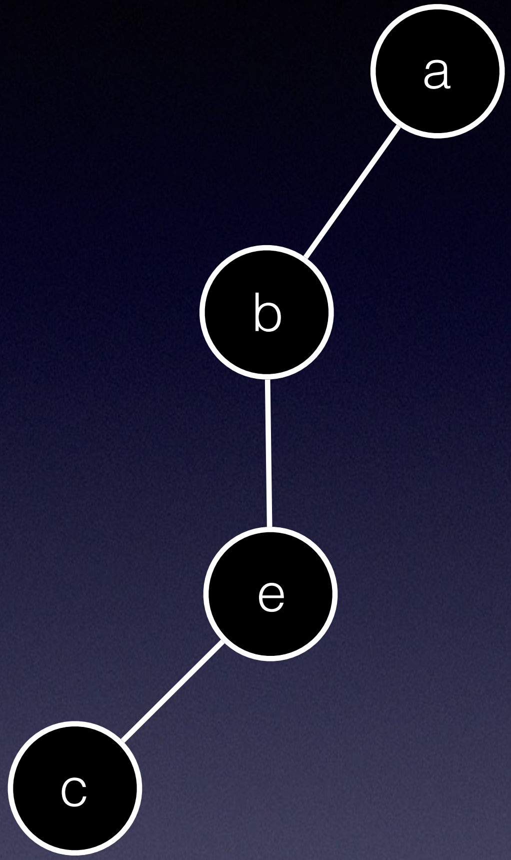


Depth-first search

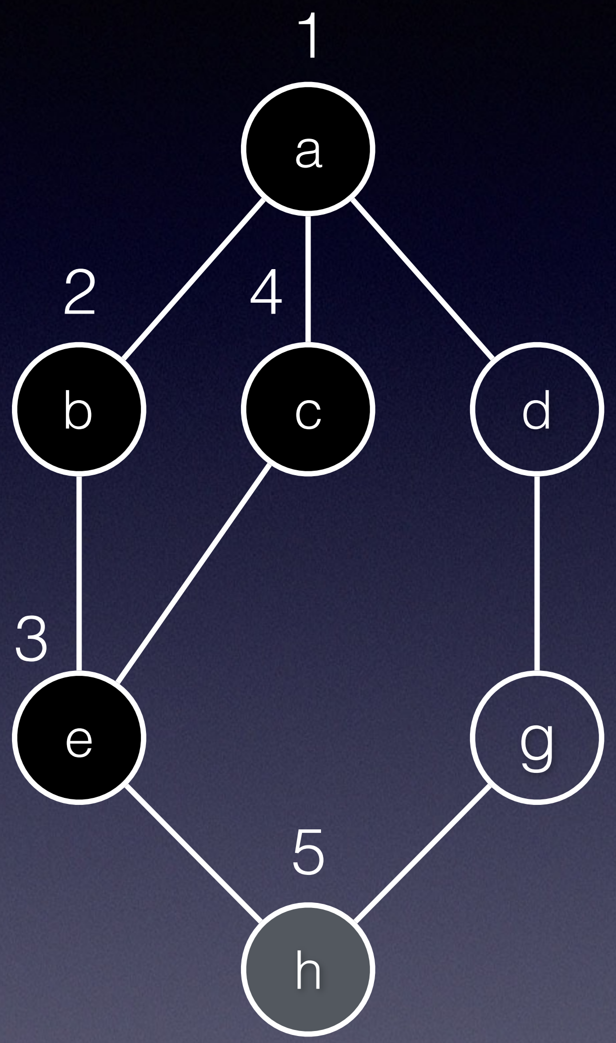


Stack: [d,c]

Visiting: h

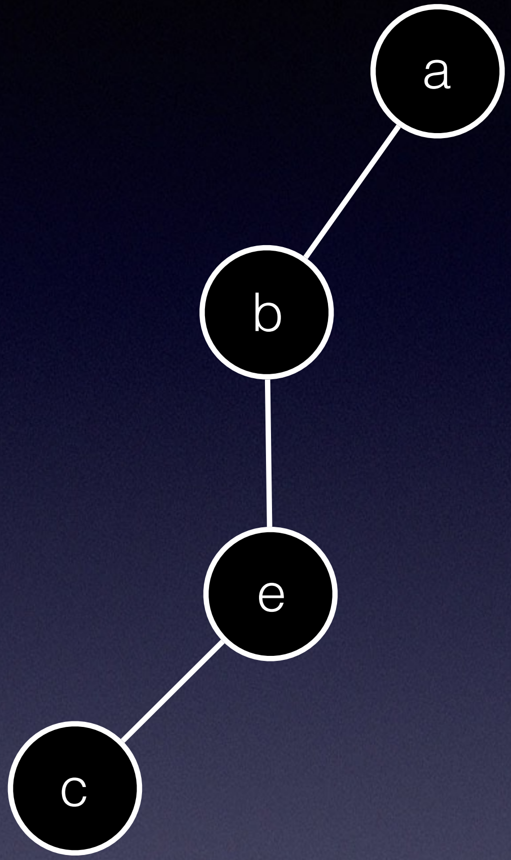


Depth-first search

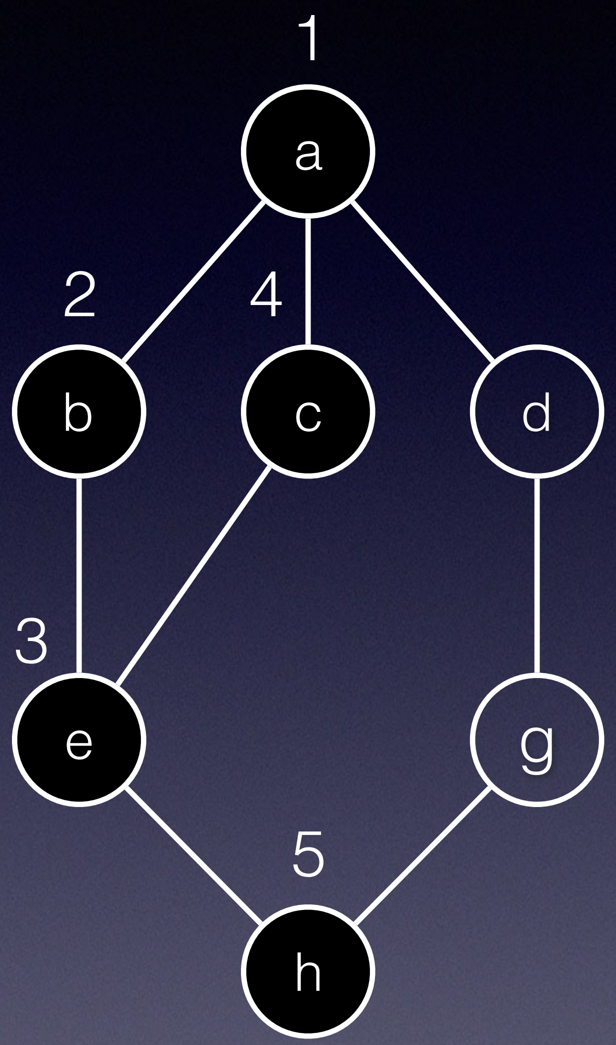


Stack: [d,c,g]

Visiting: h

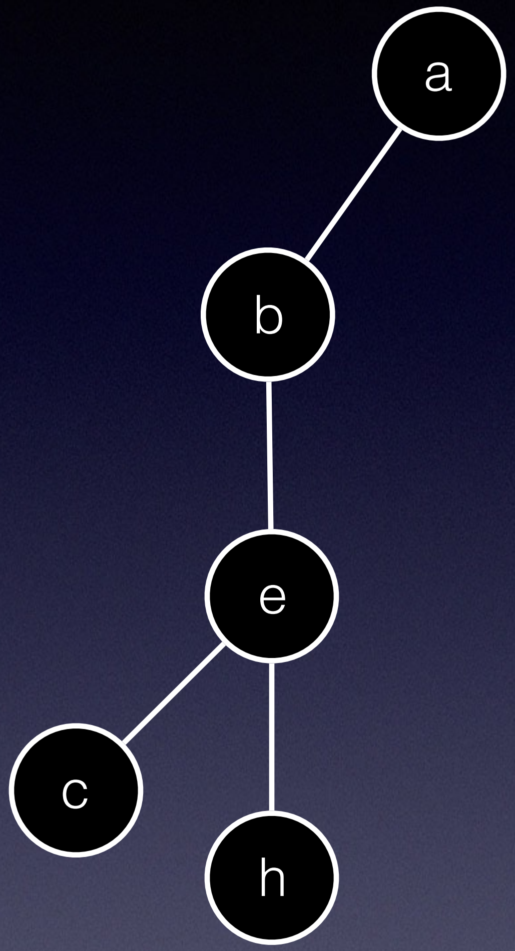


Depth-first search

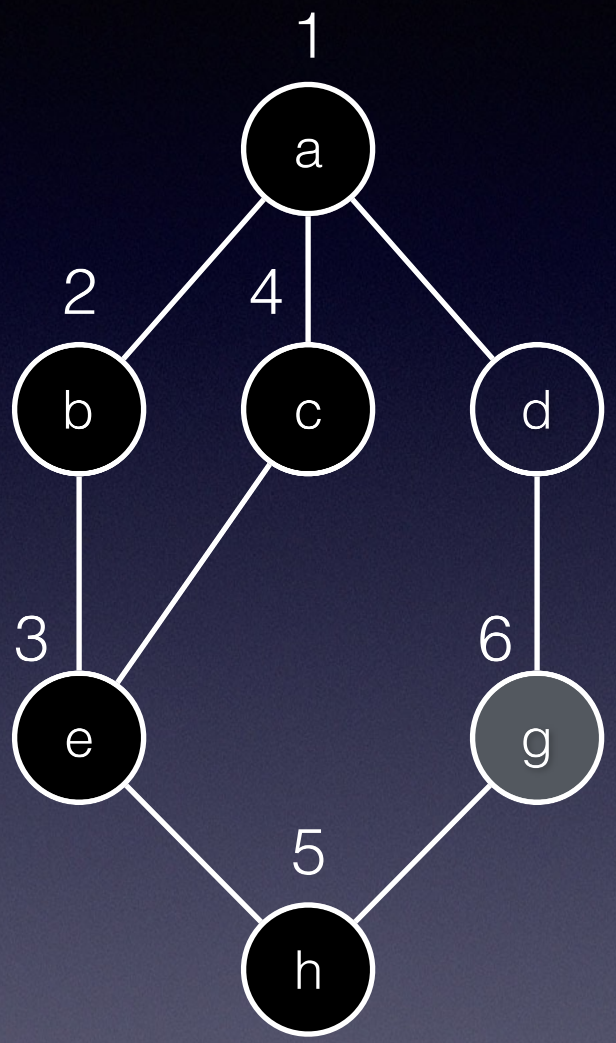


Stack: [d,c,g]

Visiting:

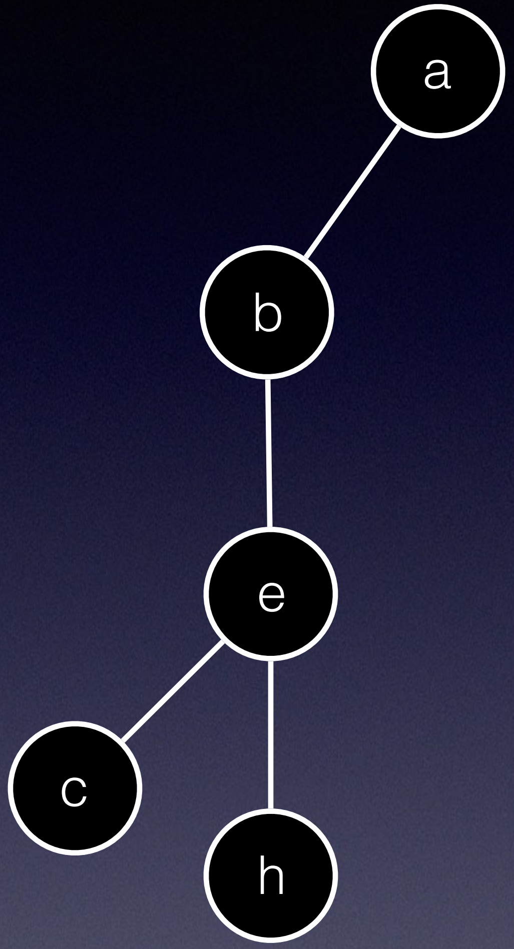


Depth-first search

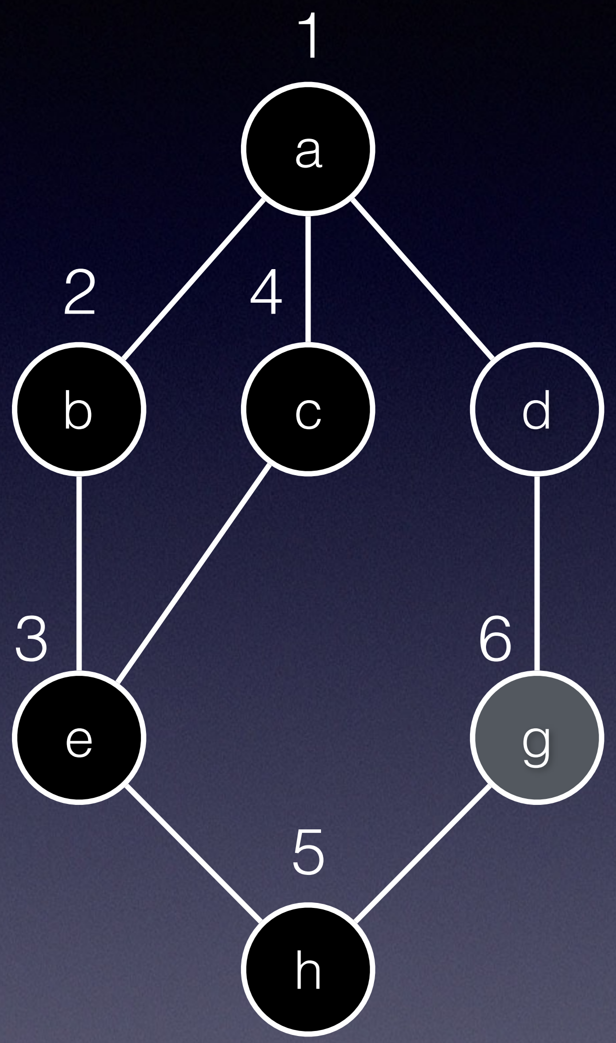


Stack: [d,c]

Visiting: g

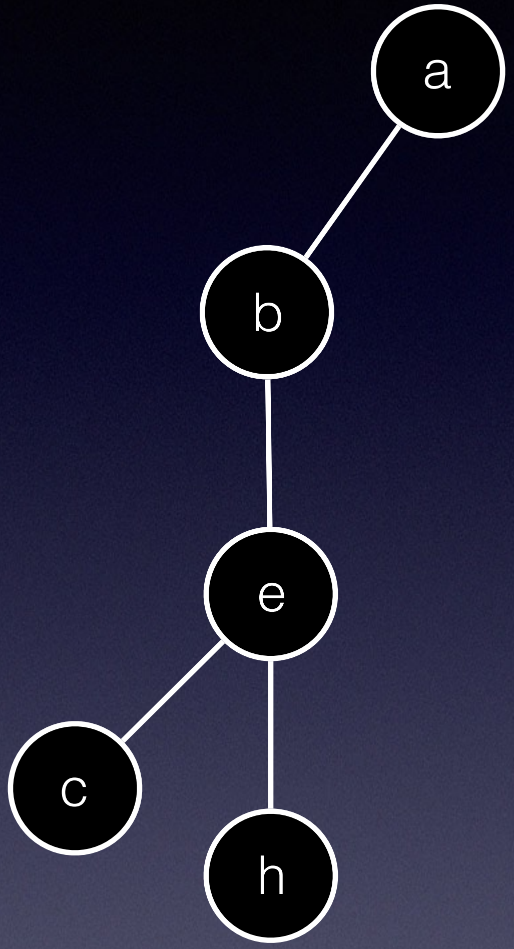


Depth-first search

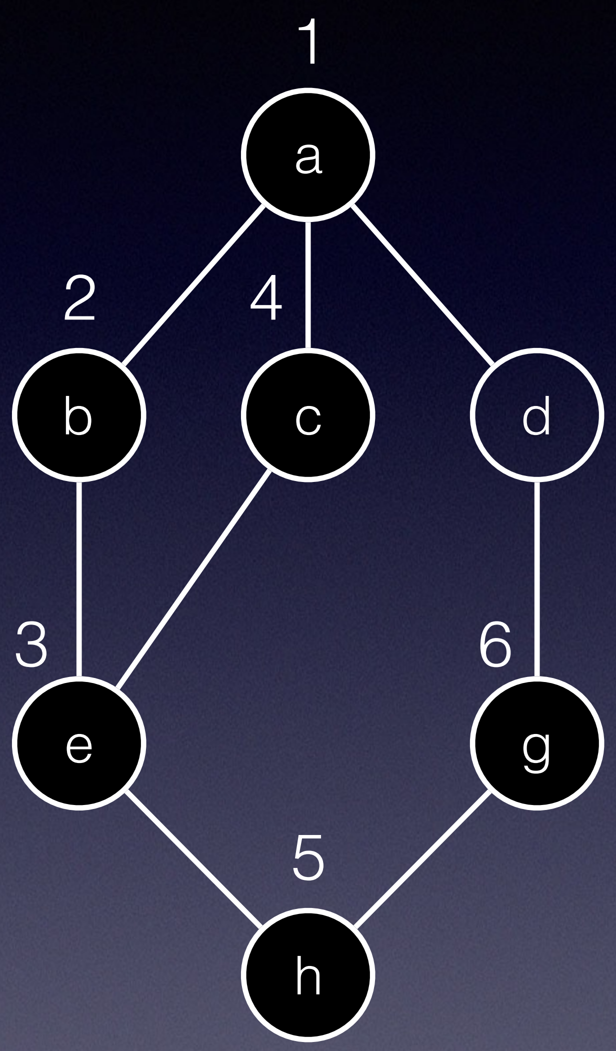


Stack: [d,c,d]

Visiting: g

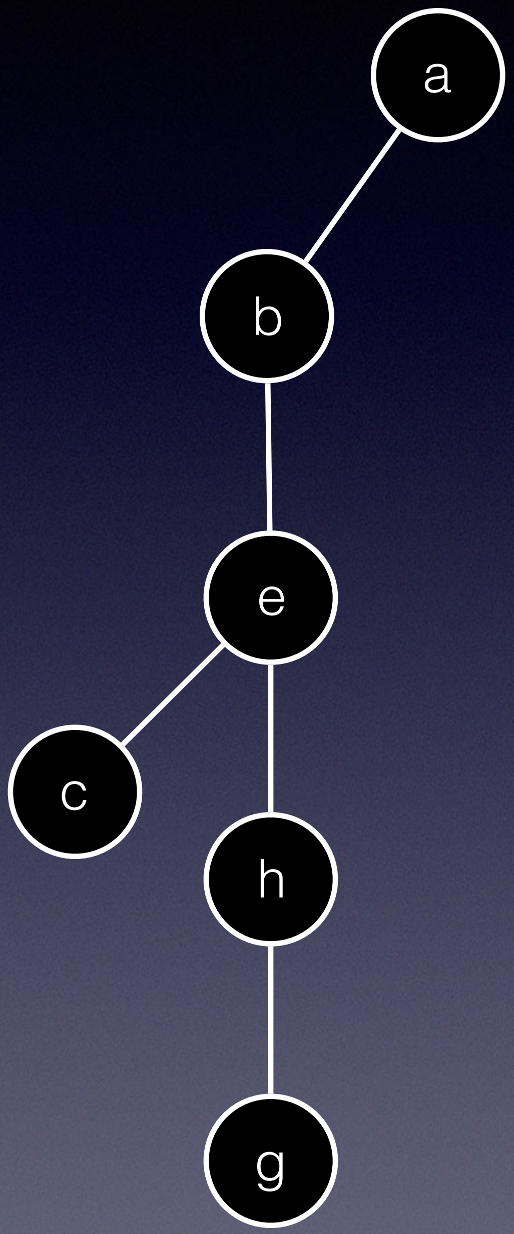


Depth-first search

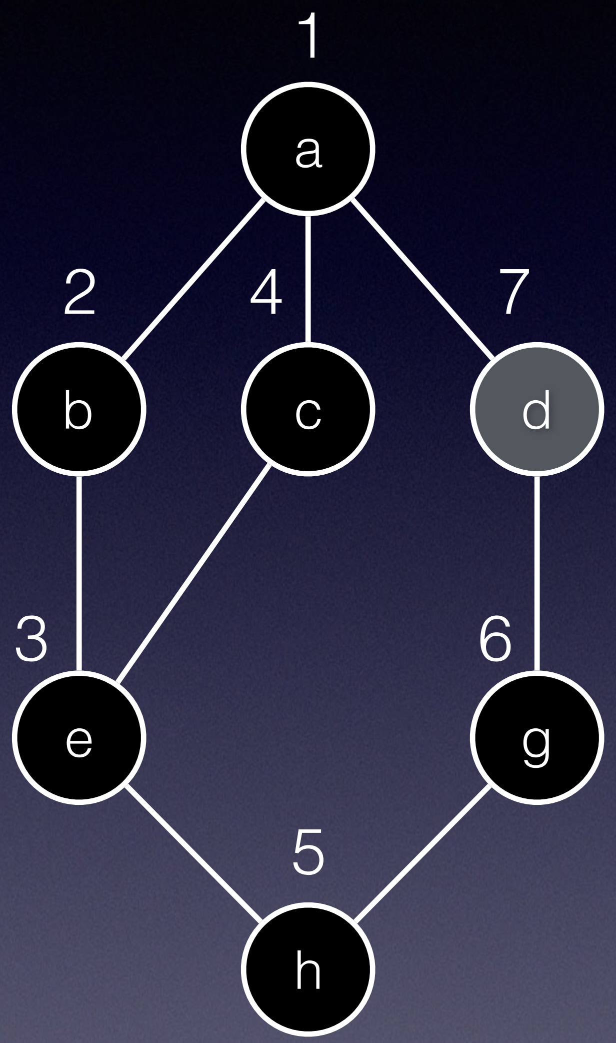


Stack: [d,c,d]

Visiting:

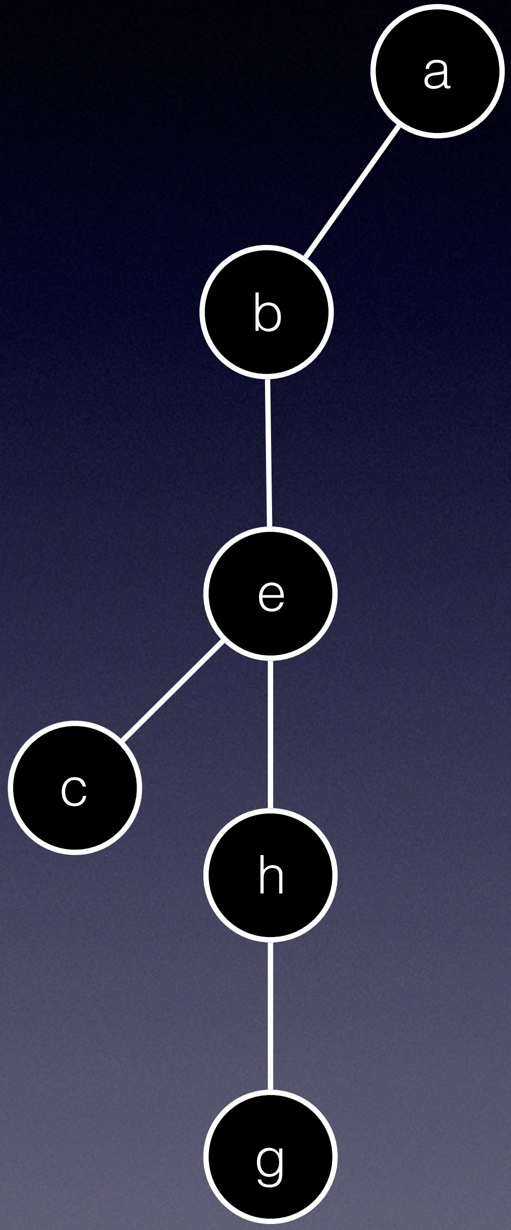


Depth-first search

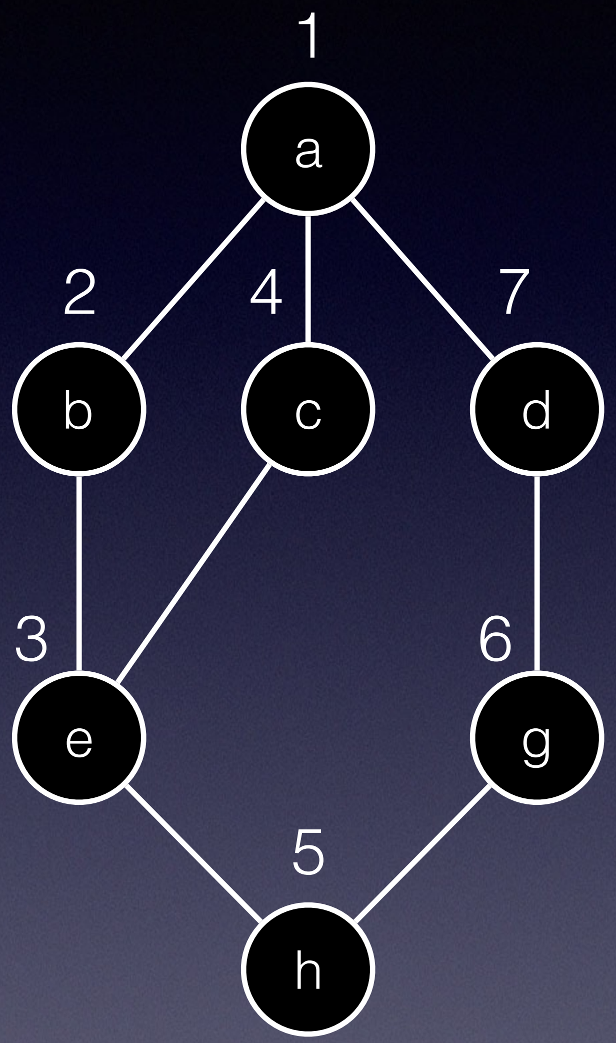


Stack: [d,c]

Visiting: d

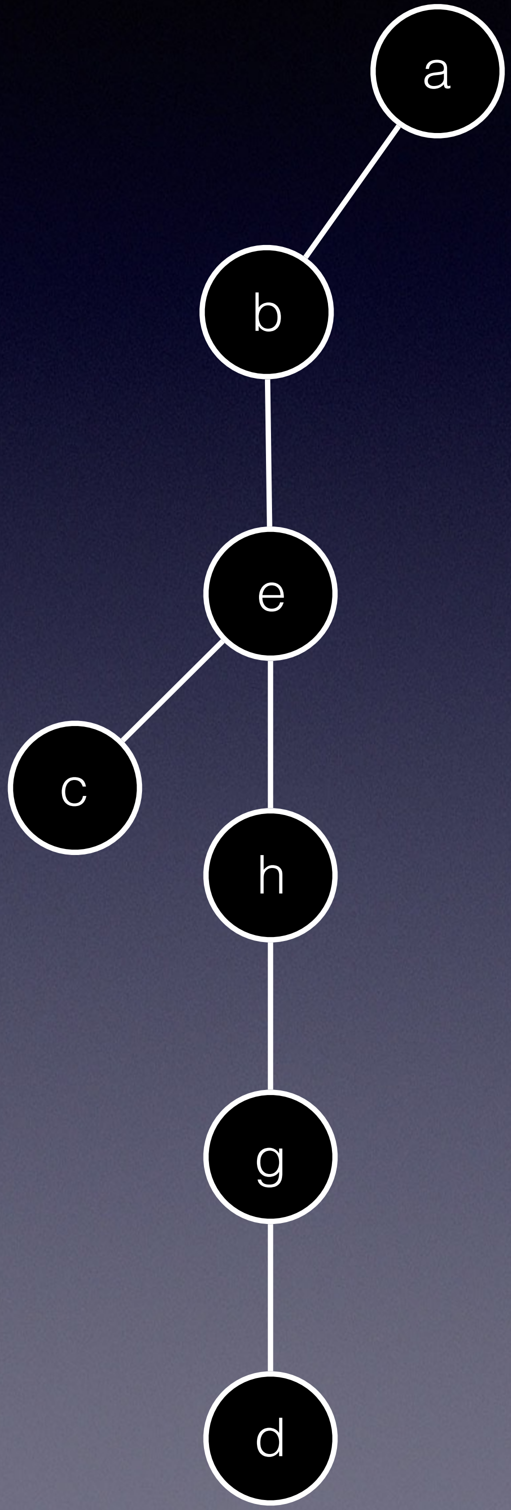


Depth-first search

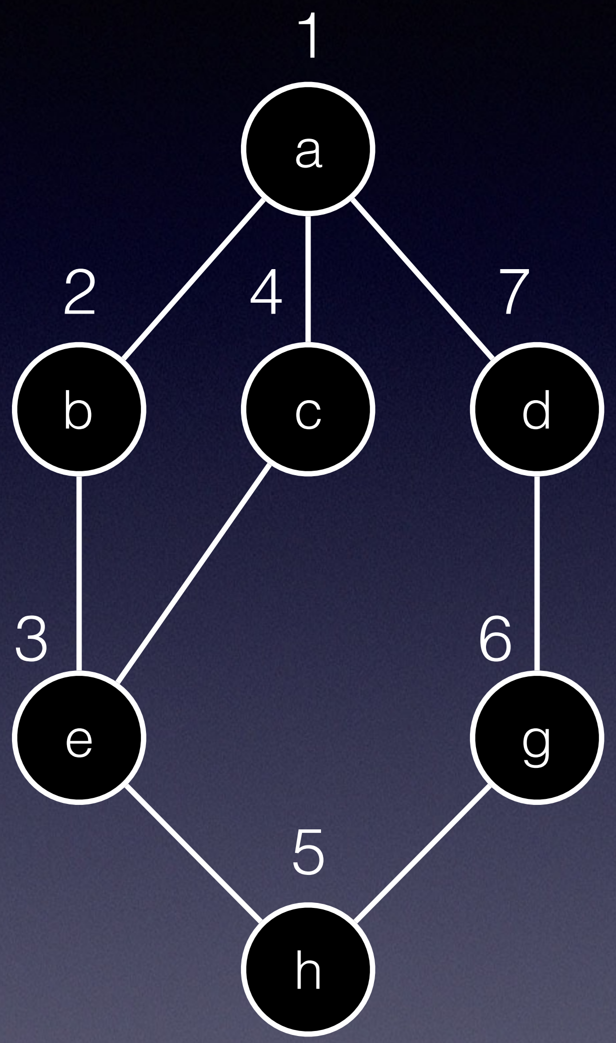


Stack: [d,c]

Visiting:



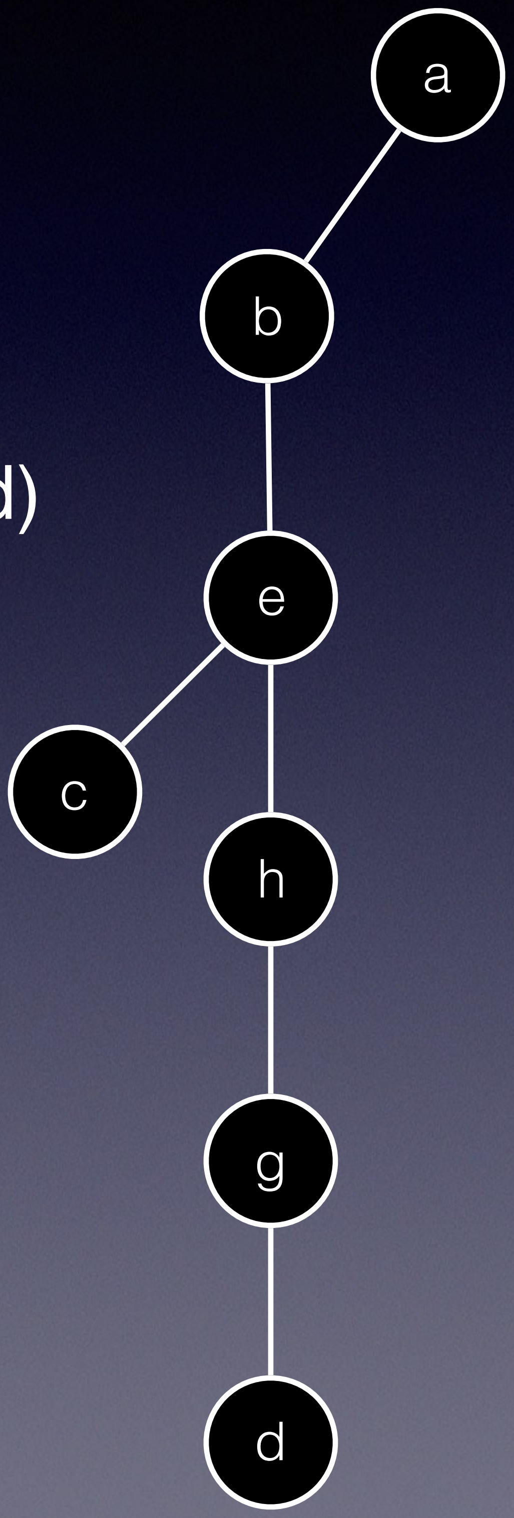
Depth-first search



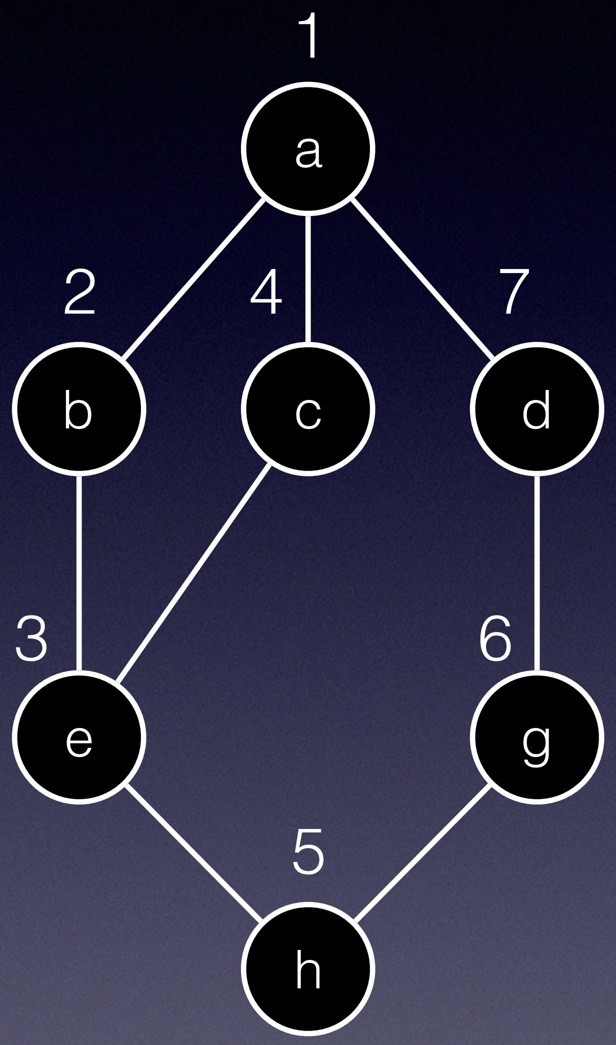
Stack: [d]

Visiting: c

(do nothing already visited)



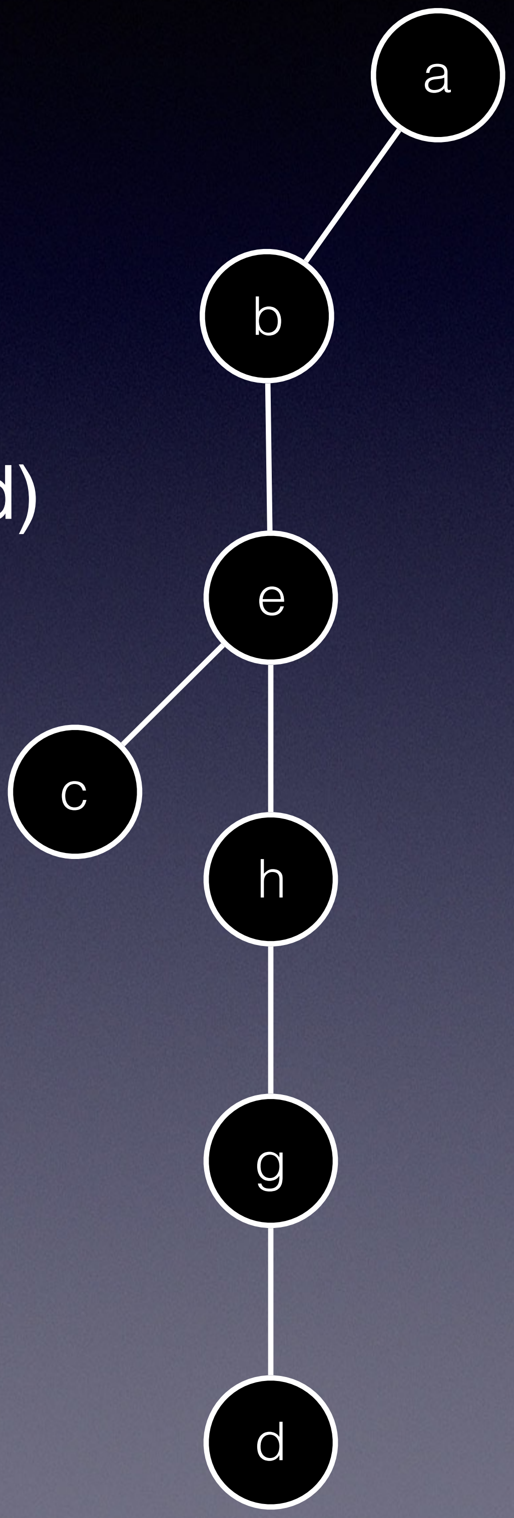
Depth-first search



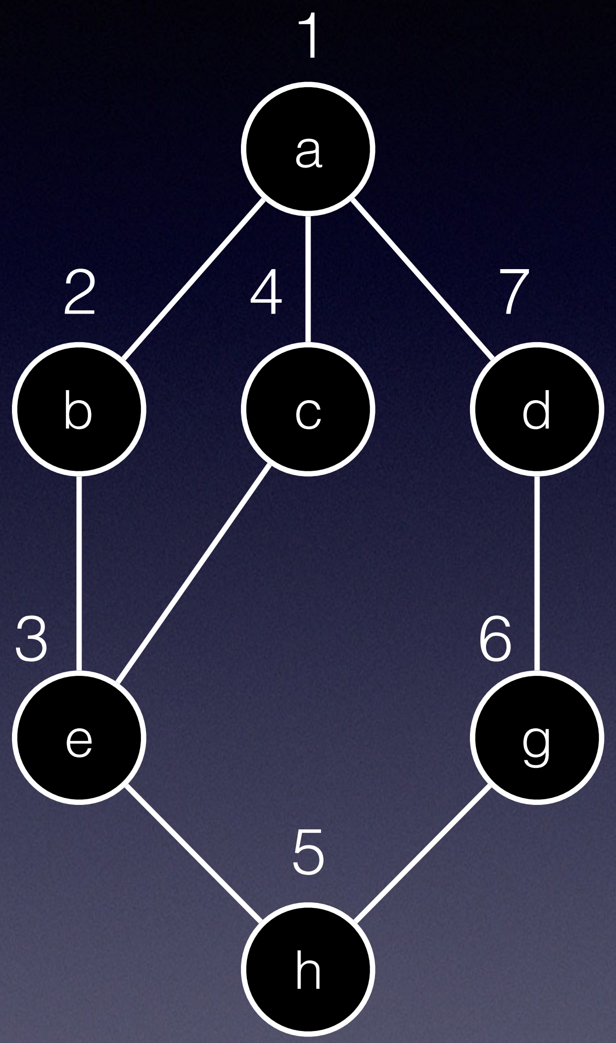
Stack: []

Visiting: d

(do nothing already visited)

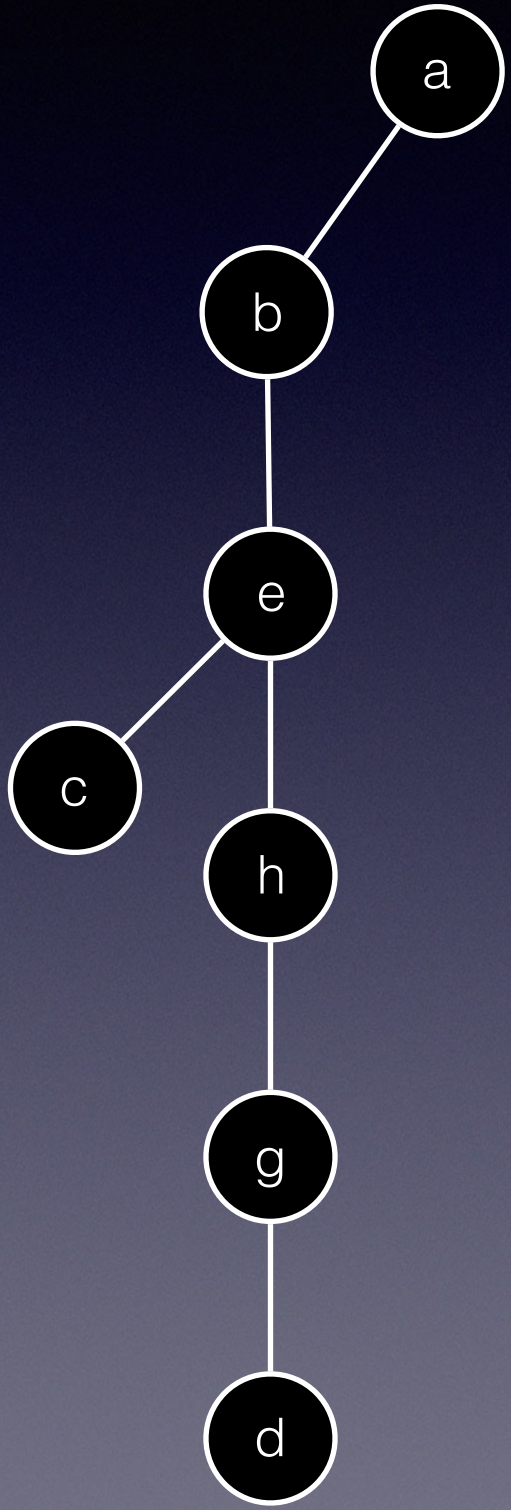


Depth-first search



Stack: []

Visiting:



Depth-first will find what it's looking for eventually, assuming the graph isn't infinitely large. If you write your DFS program recursively, as our book does, you don't need an explicit stack because recursion's use of the call stack takes care of those details for us.