ECS32B

Introduction to Data Structures

Graphs

Lecture 27

Announcements

- SLAC on tonight at 6:30 in Hutchison 73 will cover HW6 and the Makeup assignment.
- The Makeup assignment will be due next tuesday. There will be a SLAC next monday 6:30 in Hutchison 73 that will specifically cover the Makeup assignment
- A sample final will be posted tonight.
- Wednesday's lecture will focus on review and final exam preparation.

Recursive depth-first algorithm

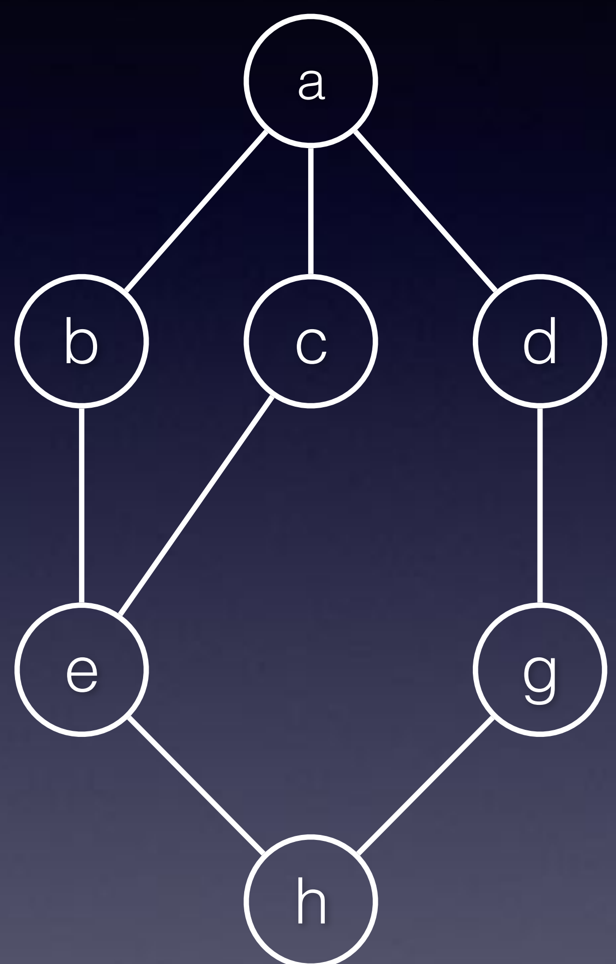
```
Call dfsvisit on a white start vertex.


dfsvisit(u):
    Mark u visited (color it grey)
    For all vertices v, adjacent to u: (in a-z order)
        If v is still white:
            dfsvisit(v)
    Mark u done (color it black)
```

This visits vertices in the same order as iterative depth first search, except unlike the two iterative algotithms vertices are marked done (**black**) in a different order than they are marked (**grey**).

Recursive depth-first search

currently visiting:

recursion tree. the node labeled v is shorthand for dfsvisit(v)



unexamined edge

tree edge

examined and not in tree

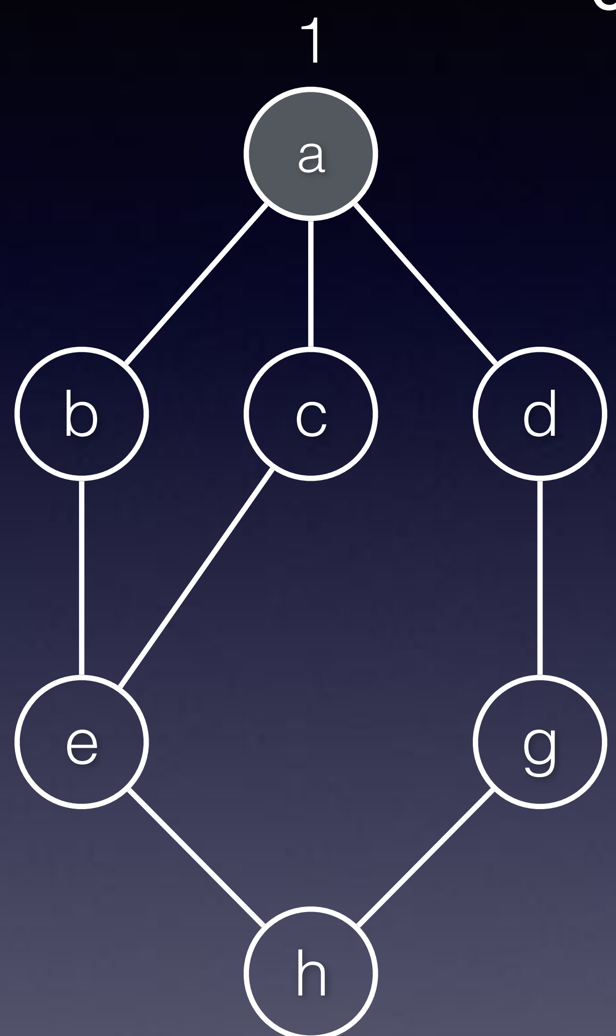
v not visited

v visited

v done visiting

Recursive depth-first search

currently visiting: a



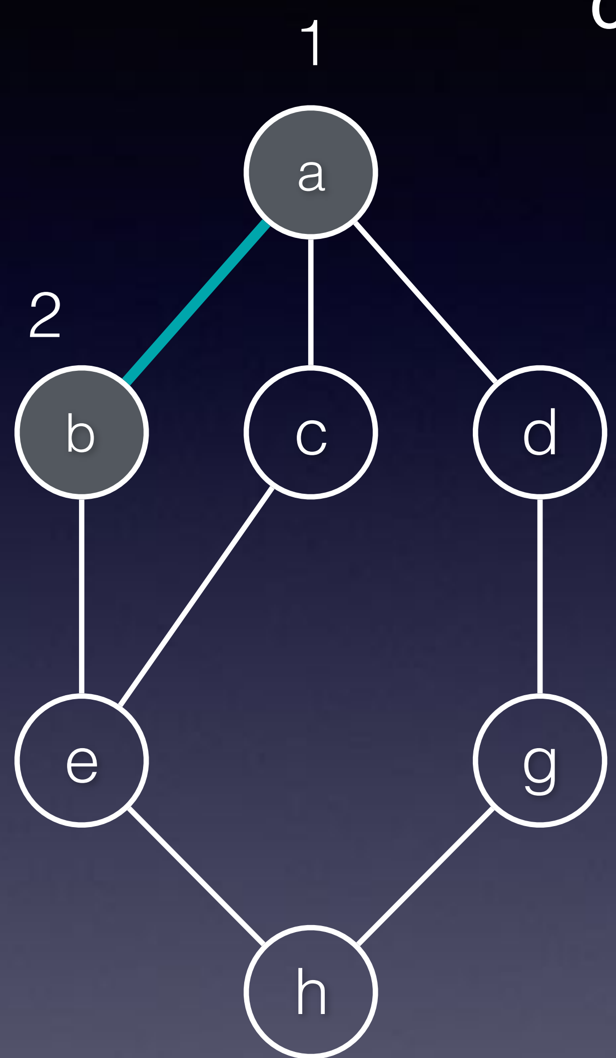
recursion tree. the node labeled v is shorthand for dfsvisit(v)

unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

Recursive depth-first search

currently visiting: a

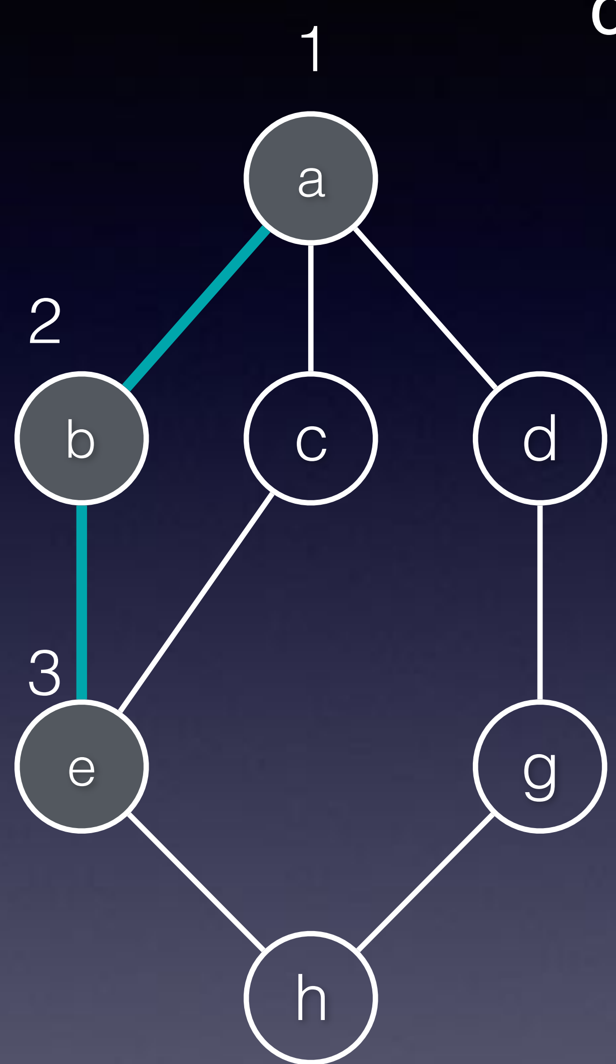


recursion tree. the node labeled v is shorthand for dfsvisit(v)

For recursive DFS, an edge from the current vertex to the vertex used for the recursive call can be added to the tree when the recursive call is made. The DFS tree is the same as the recursion tree.

unexamined edge
tree edge
examined and not in tree

Recursive depth-first search



currently visiting: e

recursion tree. the node labeled v is shorthand for dfsvisit(v)

unexamined edge

tree edge

examined and not in tree

not visited

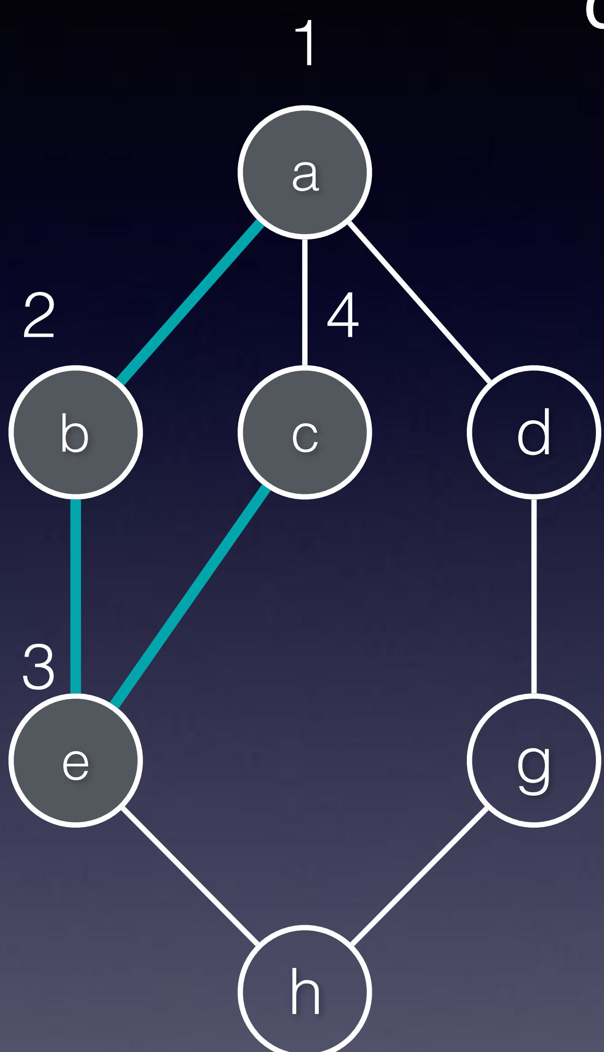
visited

done visiting

Recursive depth-first search

currently visiting: e

recursion tree. the node labeled v is shorthand for dfsvisit(v)

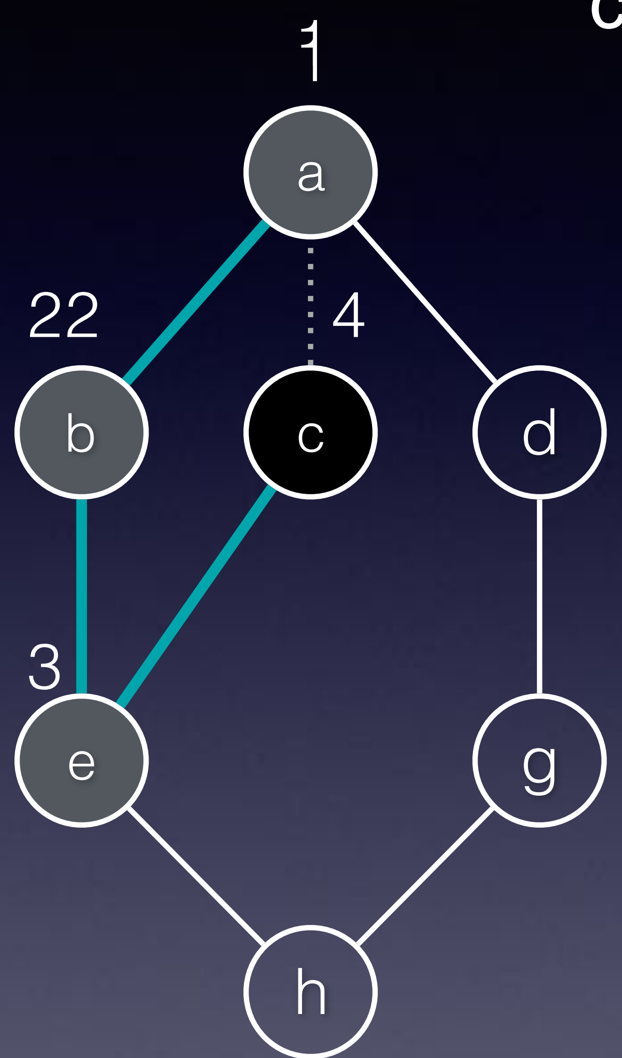


unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

Recursive depth-first search

currently visiting: e



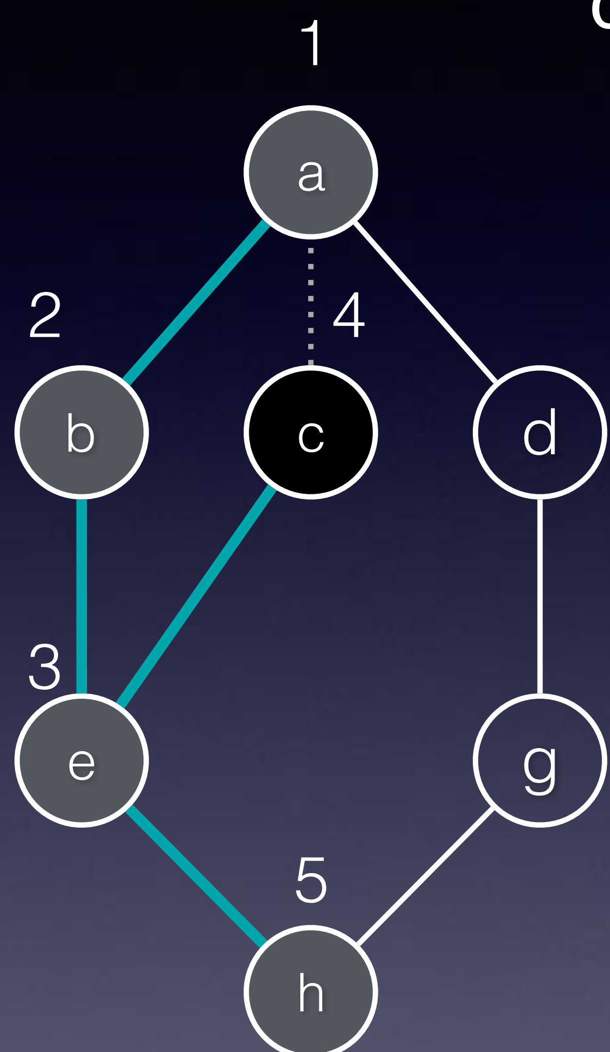
recursion tree. the node labeled v is shorthand for dfsvisit(v)

unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

Recursive depth-first search

currently visiting: h



recursion tree. the node labeled v is shorthand for dfsvisit(v)

unexamined edge

tree edge

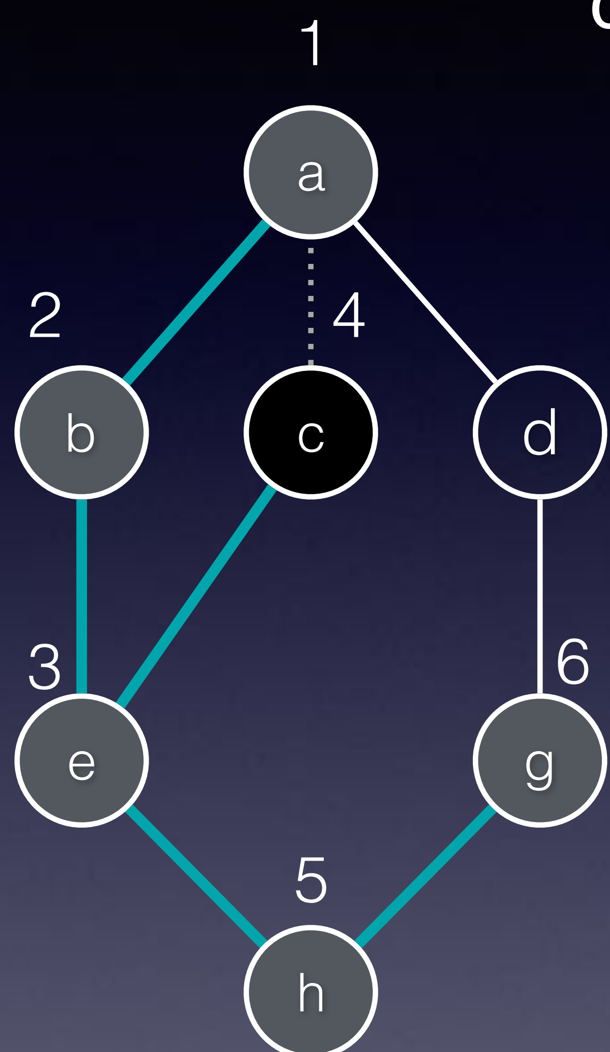
examined and not in tree

v not visited

v visited

v done visiting

Recursive depth-first search



currently visiting: g

recursion tree. the node labeled v is shorthand for dfsvisit(v)

unexamined edge

tree edge

examined and not in tree

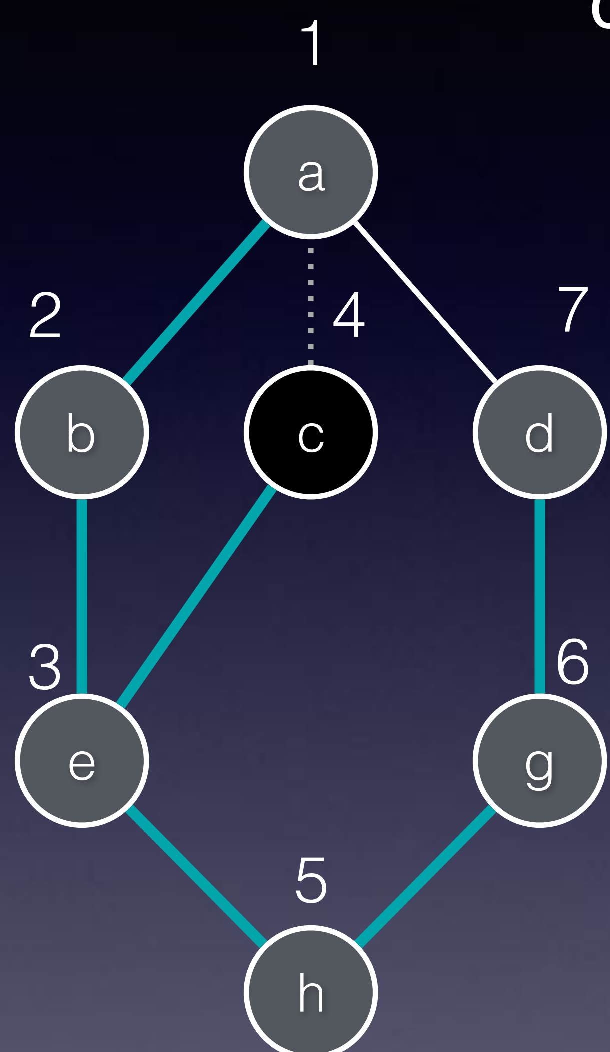
v not visited

v visited

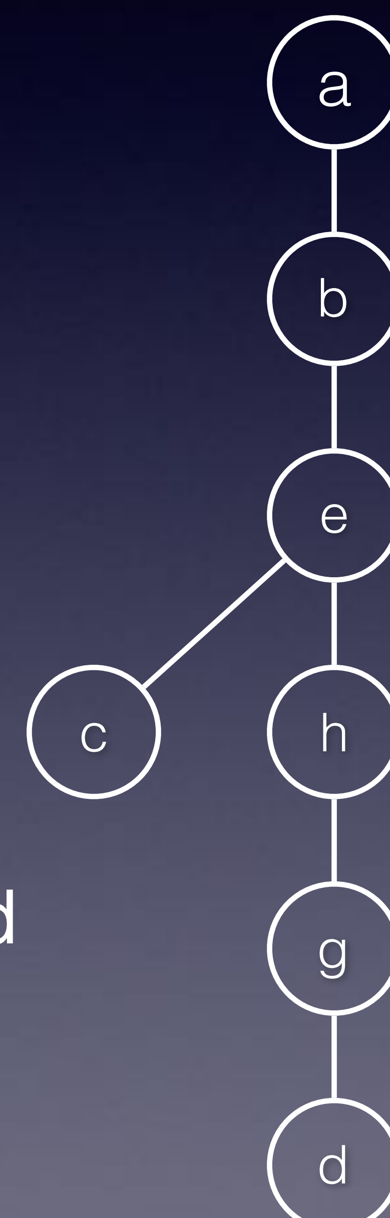
v done visiting

Recursive depth-first search

currently visiting: d



recursion tree. the node labeled v is shorthand for dfsvisit(v)

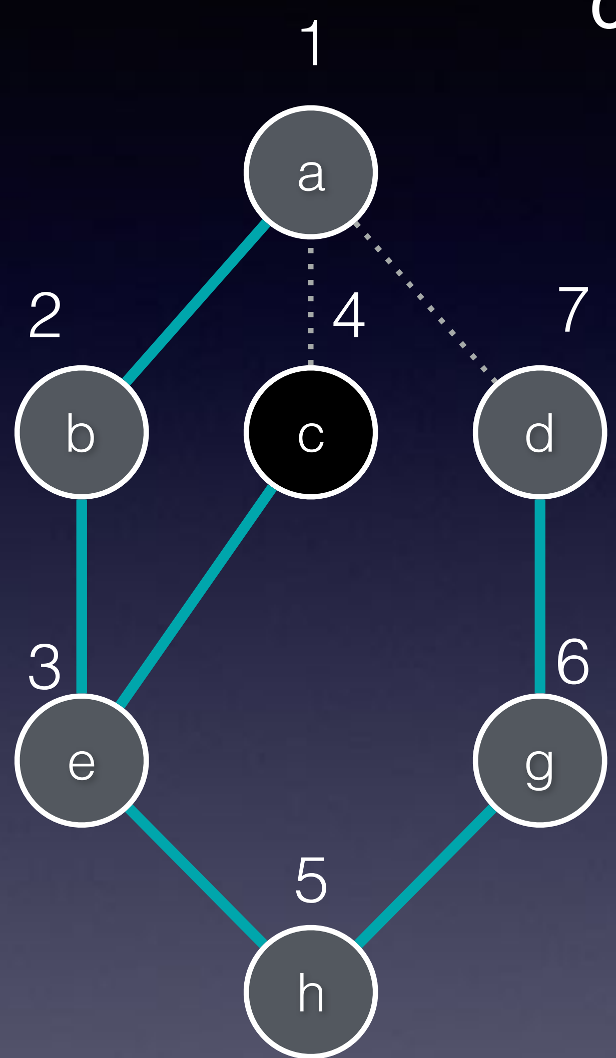


unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

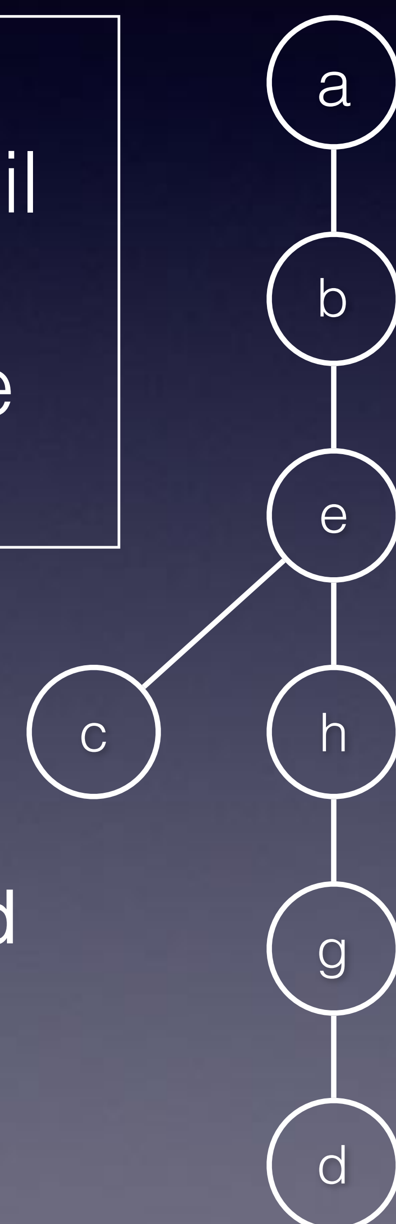
Recursive depth-first search

currently visiting: d



recursion tree. the node labeled v is shorthand for dfsvisit(v)

We are not done visiting the nodes until they are all marked done as the recursive calls return.

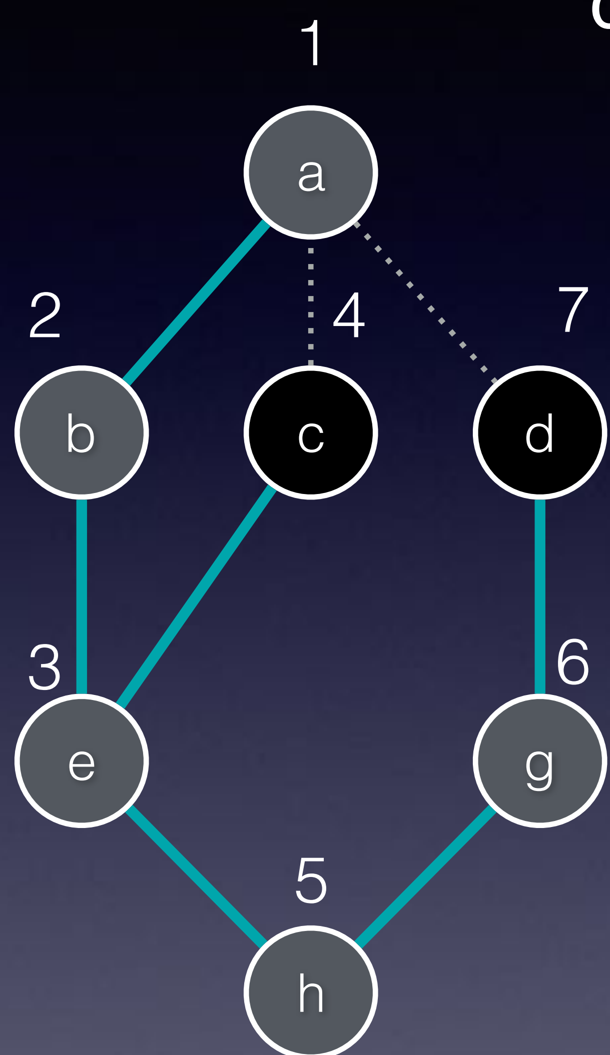


unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

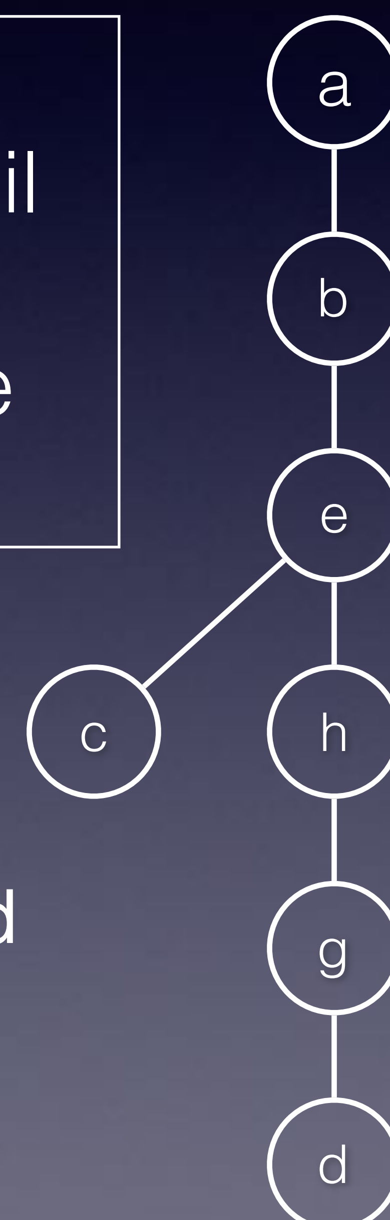
Recursive depth-first search

currently visiting: d



We are not done visiting the nodes until they are all marked done as the recursive calls return.

recursion tree. the node labeled v is shorthand for dfsvisit(v)

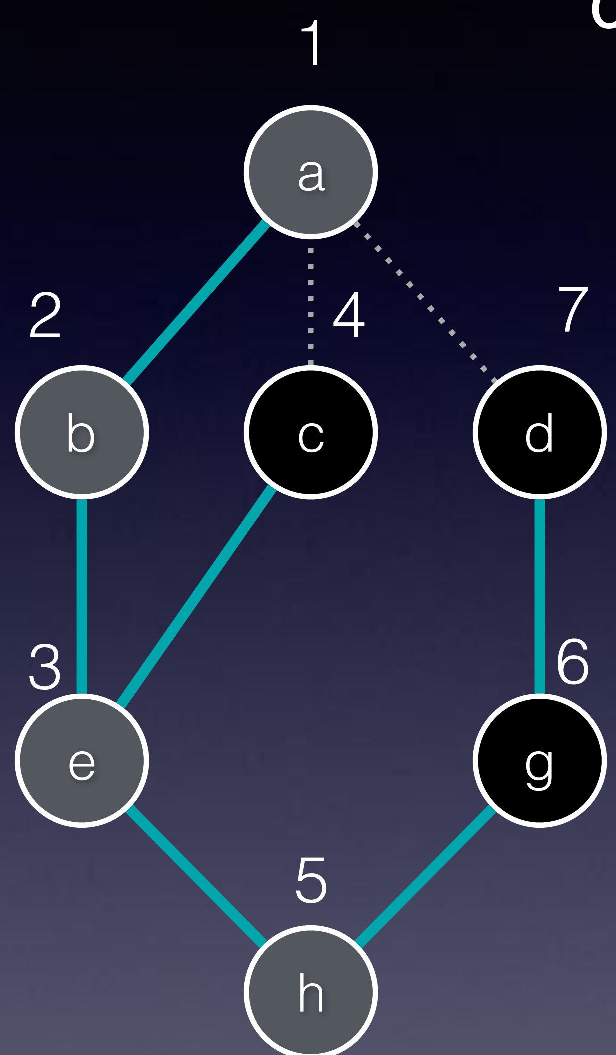


unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

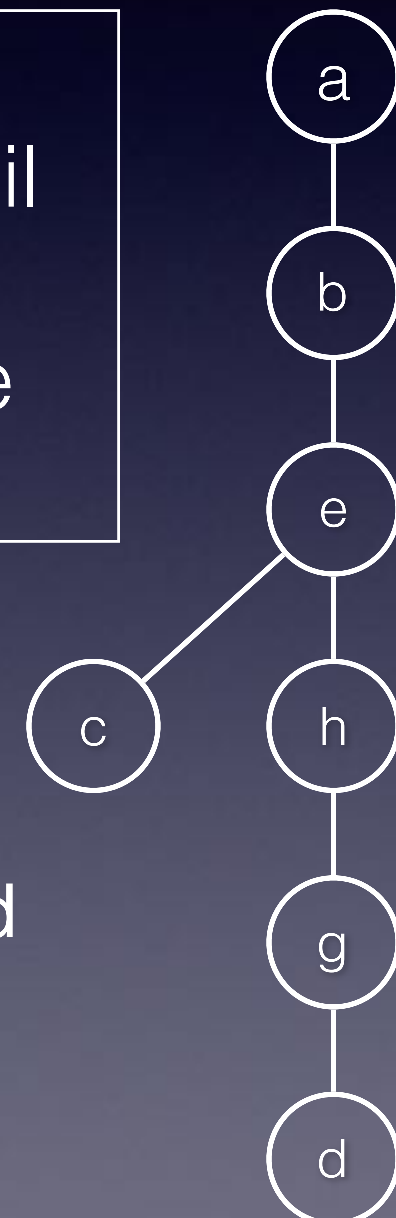
Recursive depth-first search

currently visiting: d



We are not done visiting the nodes until they are all marked done as the recursive calls return.

recursion tree. the node labeled v is shorthand for dfsvisit(v)



unexamined edge

tree edge

examined and not in tree

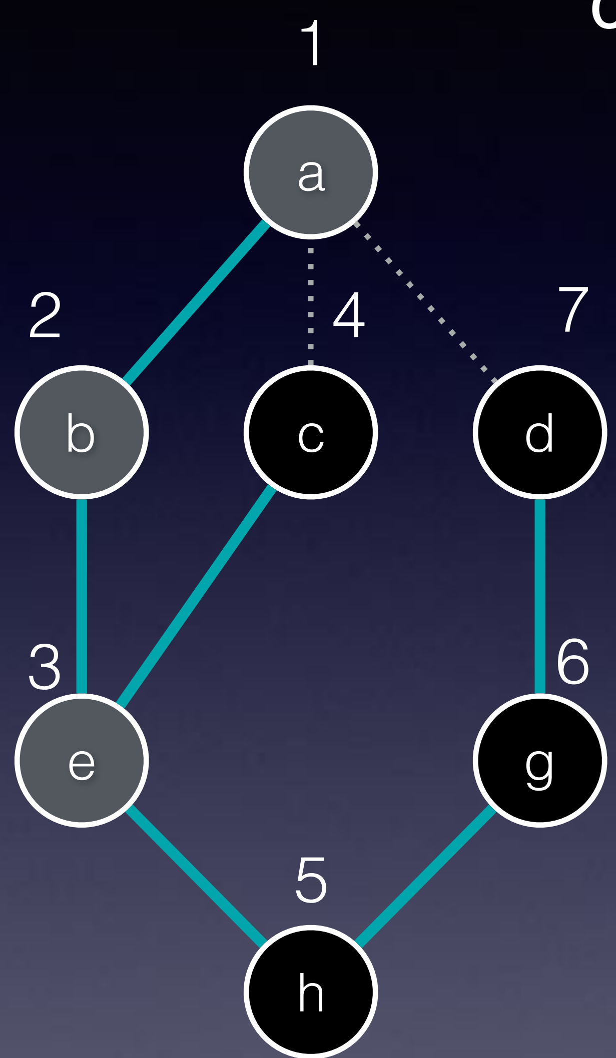
v not visited

v visited

v done visiting

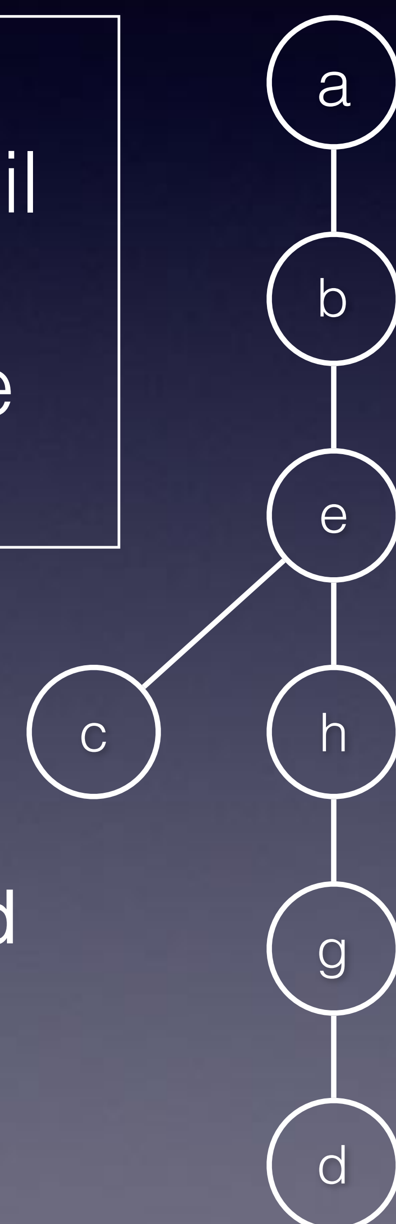
Recursive depth-first search

currently visiting: d



We are not done visiting the nodes until they are all marked done as the recursive calls return.

recursion tree. the node labeled v is shorthand for dfsvisit(v)



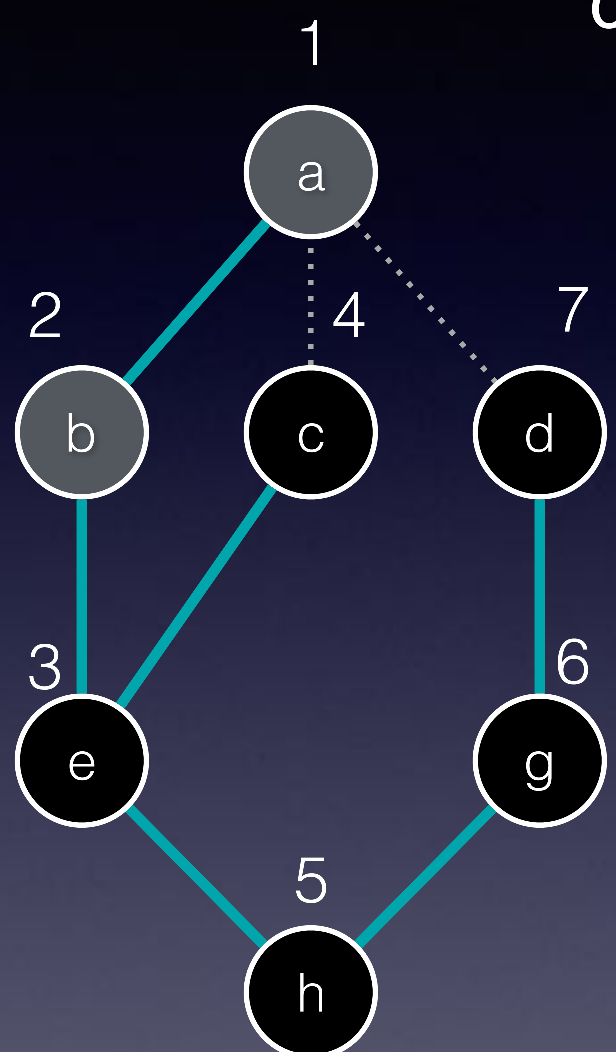
unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

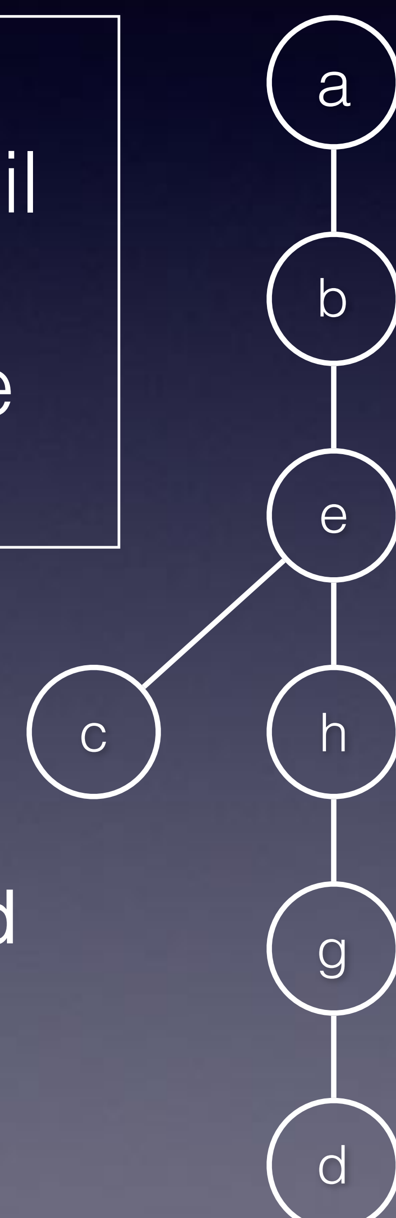
Recursive depth-first search

currently visiting: d

recursion tree. the node labeled v is shorthand for dfsvisit(v)



We are not done visiting the nodes until they are all marked done as the recursive calls return.



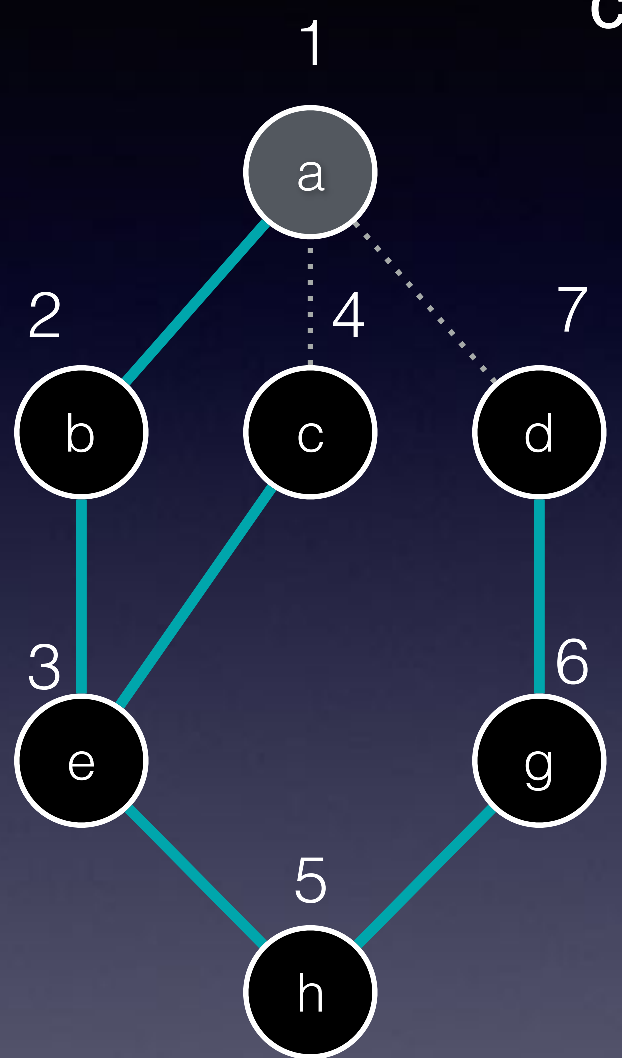
unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

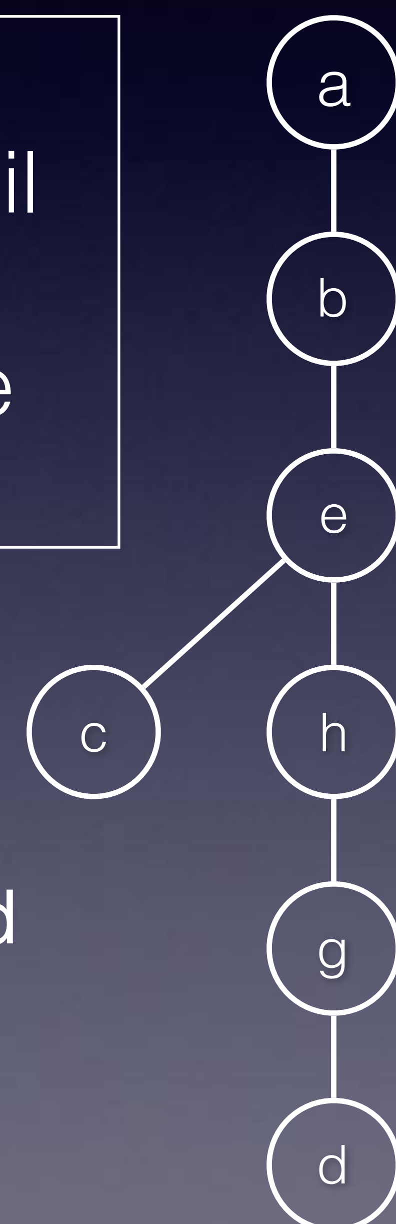
Recursive depth-first search

currently visiting: d

recursion tree. the node labeled v is shorthand for dfsvisit(v)



We are not done visiting the nodes until they are all marked done as the recursive calls return.

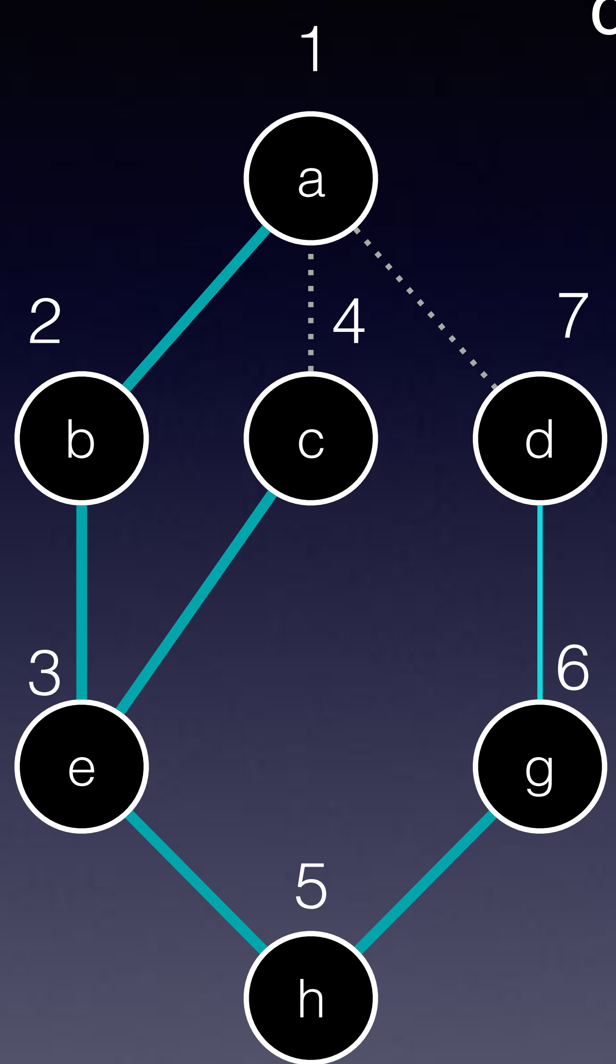


unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

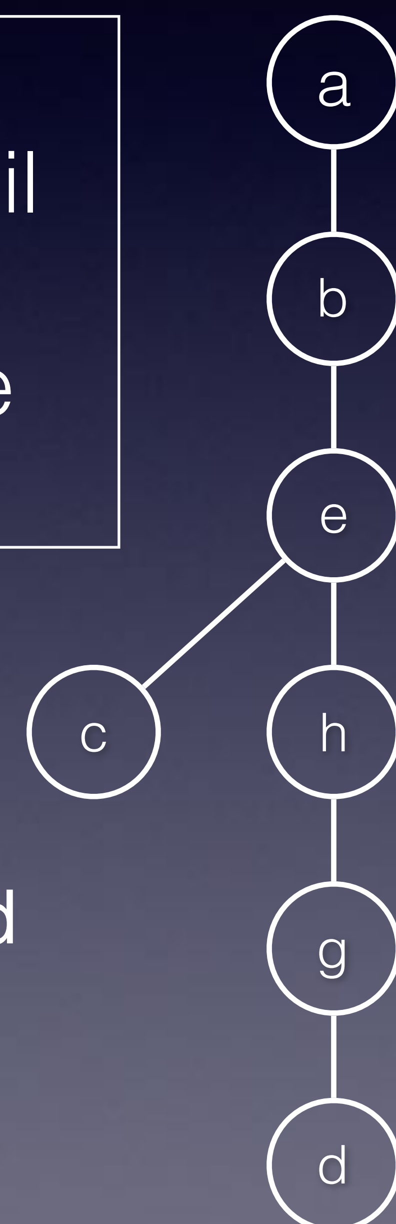
Recursive depth-first search

currently visiting: d



We are not done visiting the nodes until they are all marked done as the recursive calls return.

recursion tree. the node labeled v is shorthand for dfsvisit(v)

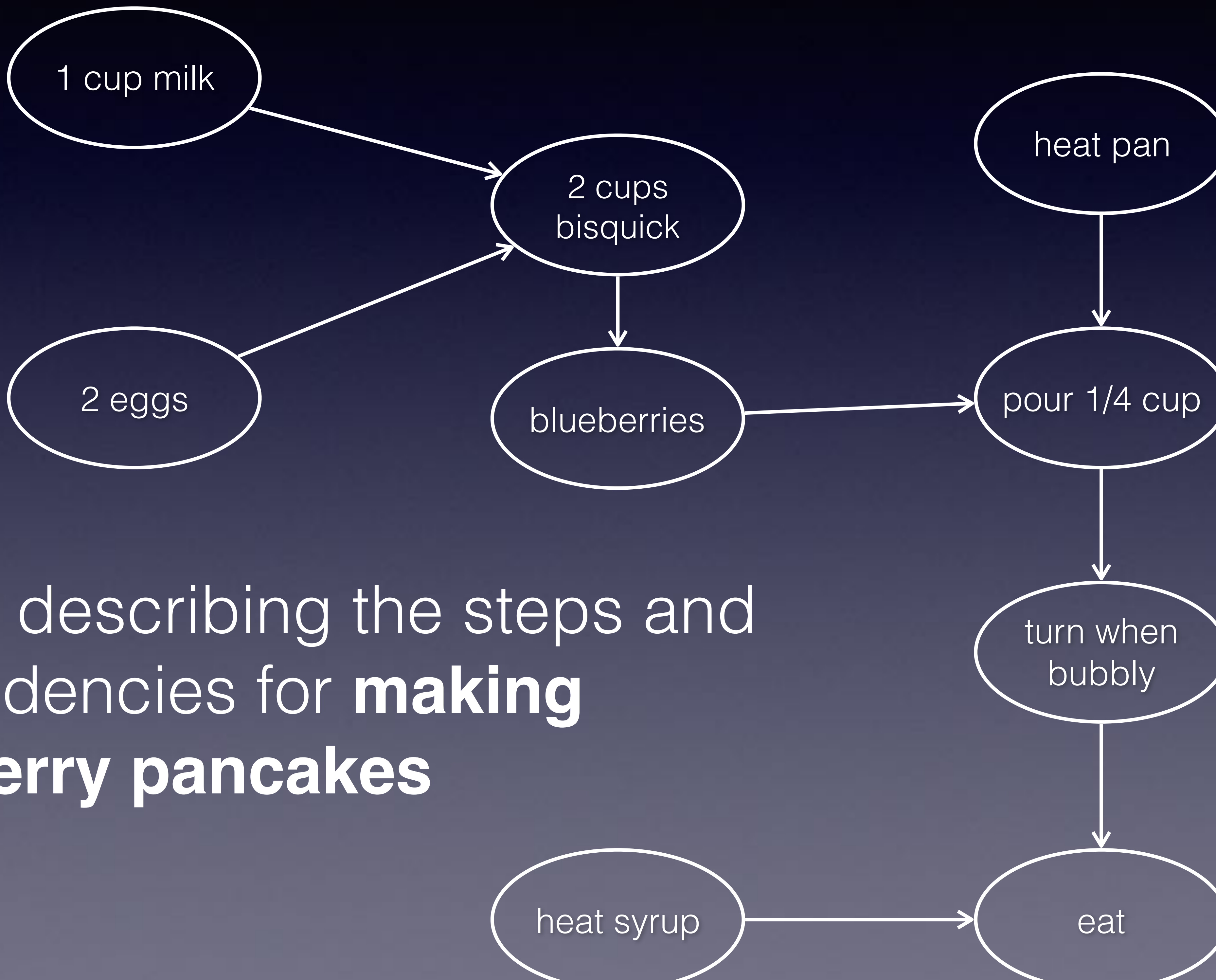


unexamined edge
tree edge
examined and not in tree

v not visited
v visited
v done visiting

Let’s make blueberry pancakes with a graph!

Topological Sorting



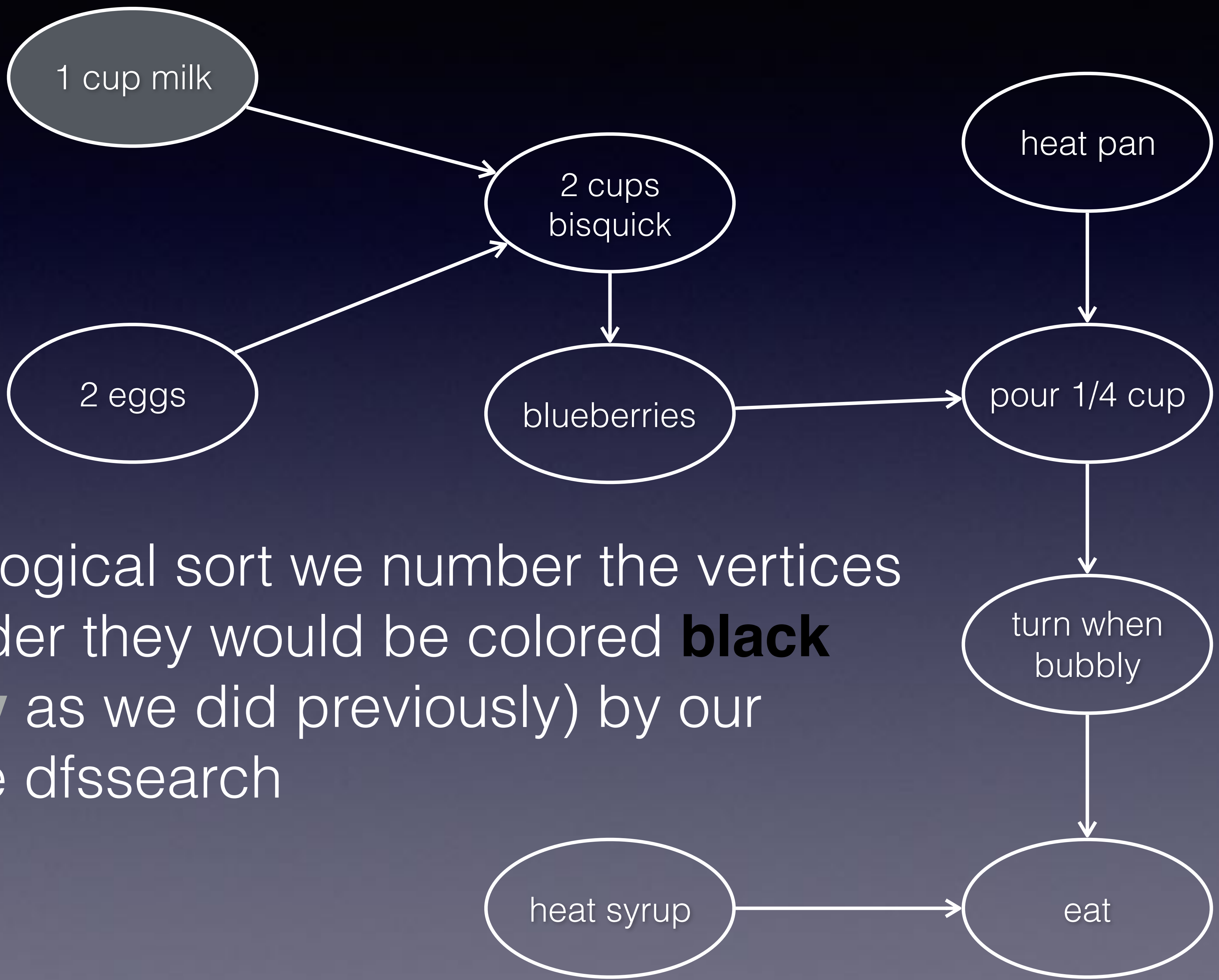
Graph describing the steps and dependencies for **making blueberry pancakes**

Topological Sorting

- A **topological sort** takes a directed acyclic graph (like the one on the previous slide) and produces a linear ordering of the vertices such that if there is an edge from u to v in the graph then u comes before v in the sorted order.

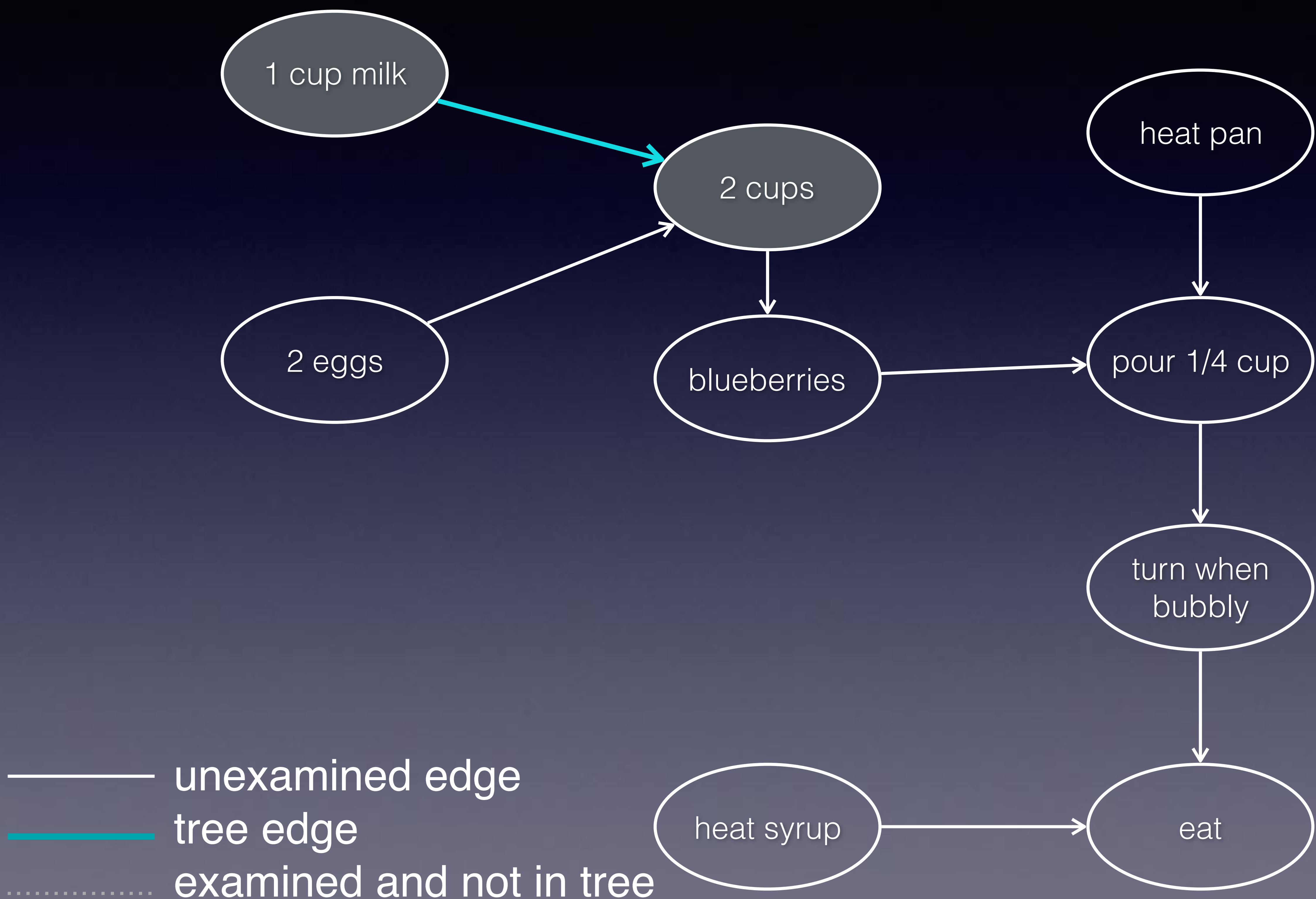
- In other words, given our blueberry pancake dependency graph, a topological sort gives us a **valid order to perform the steps**.

Making Blueberry Pancakes

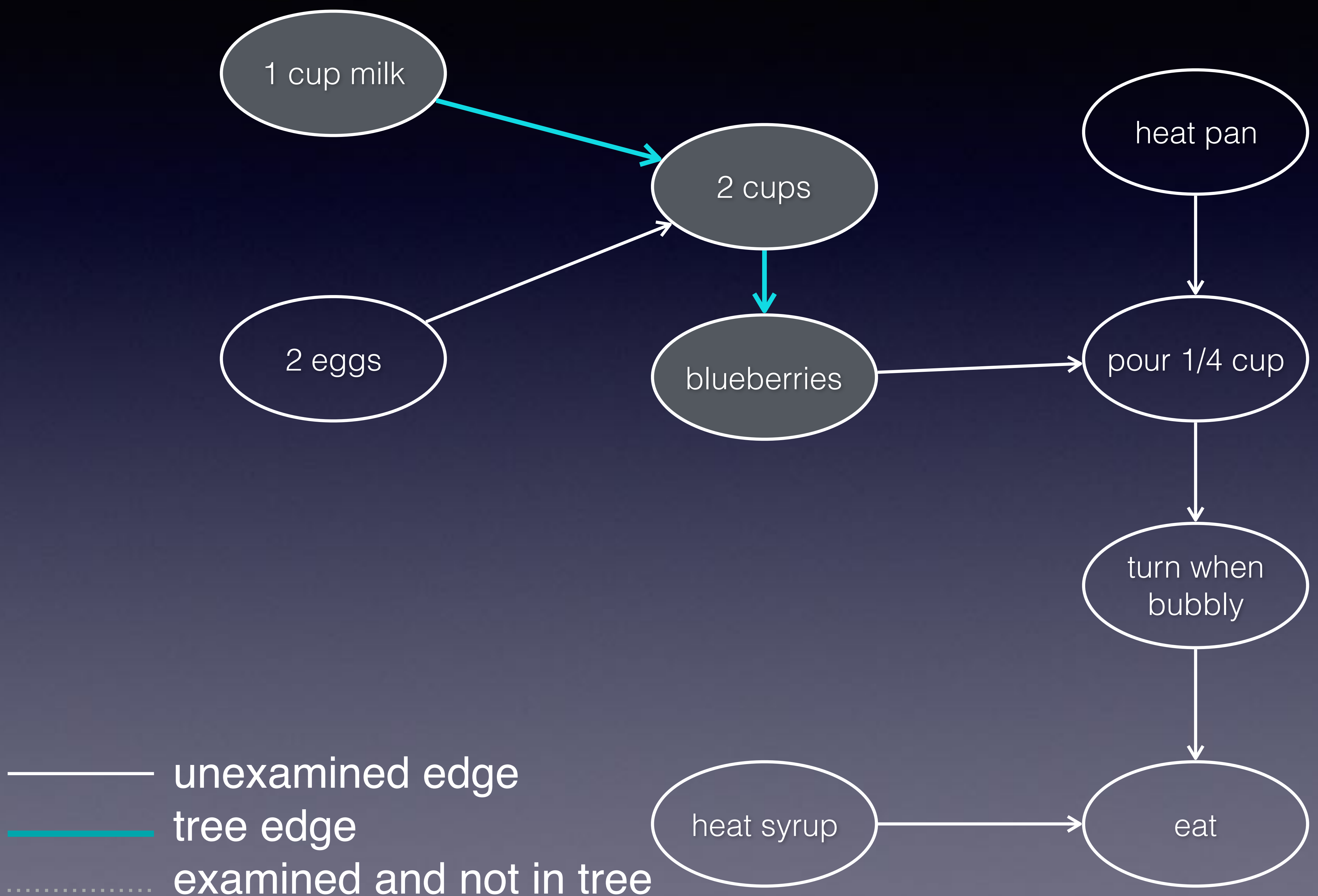


For topological sort we number the vertices in the order they would be colored **black** (not **grey** as we did previously) by our recursive dfssearch

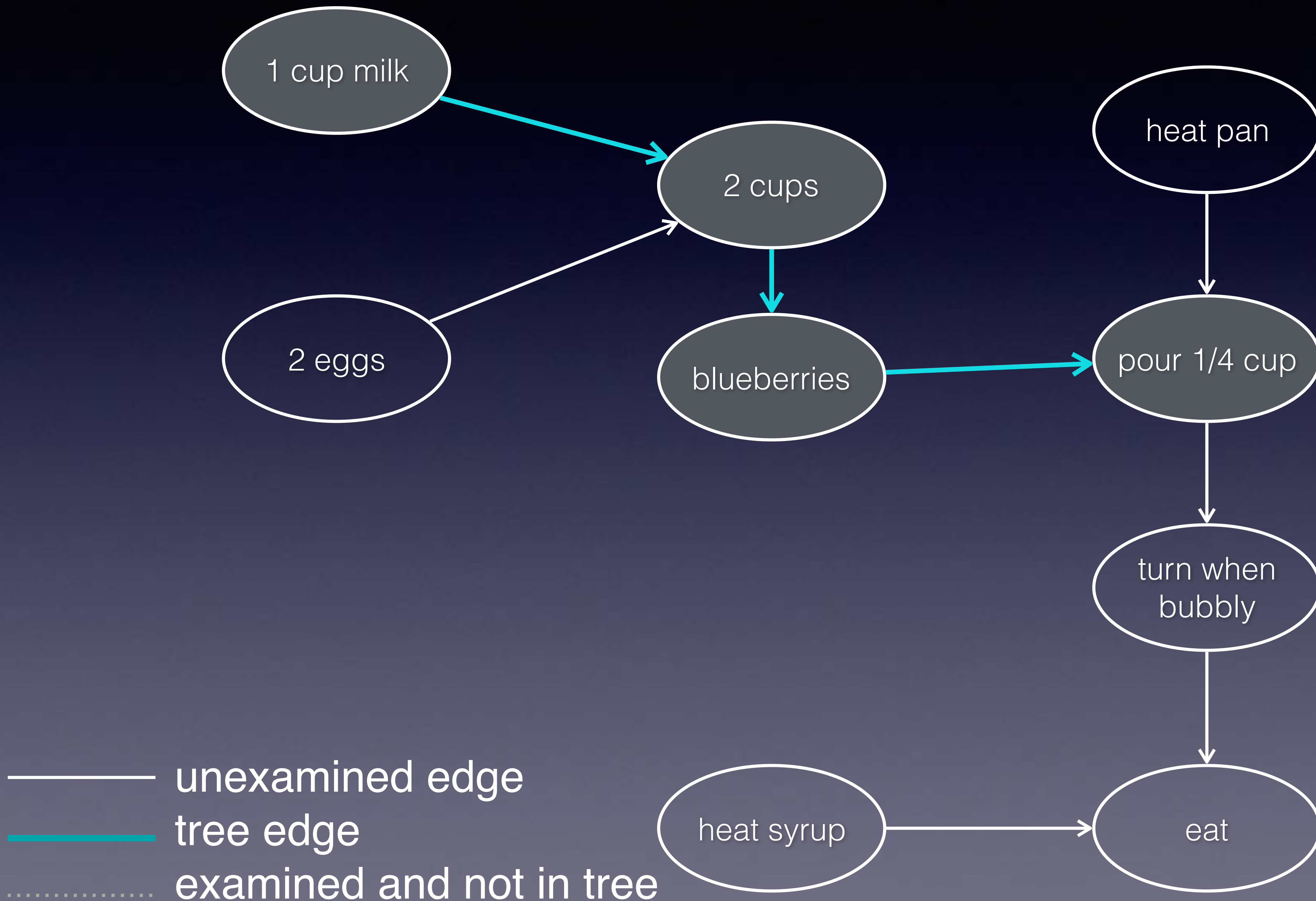
Making Blueberry Pancakes



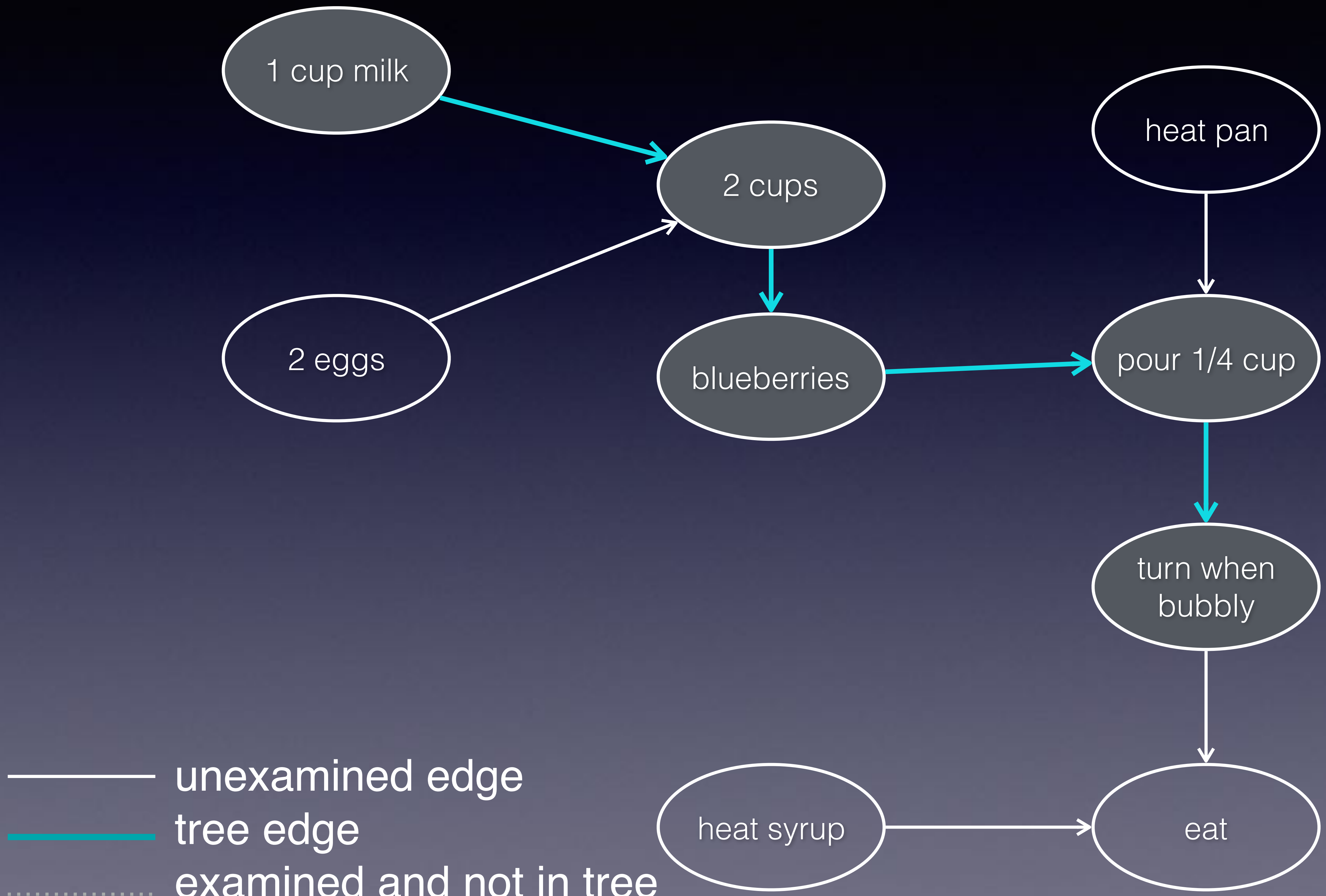
Making Blueberry Pancakes



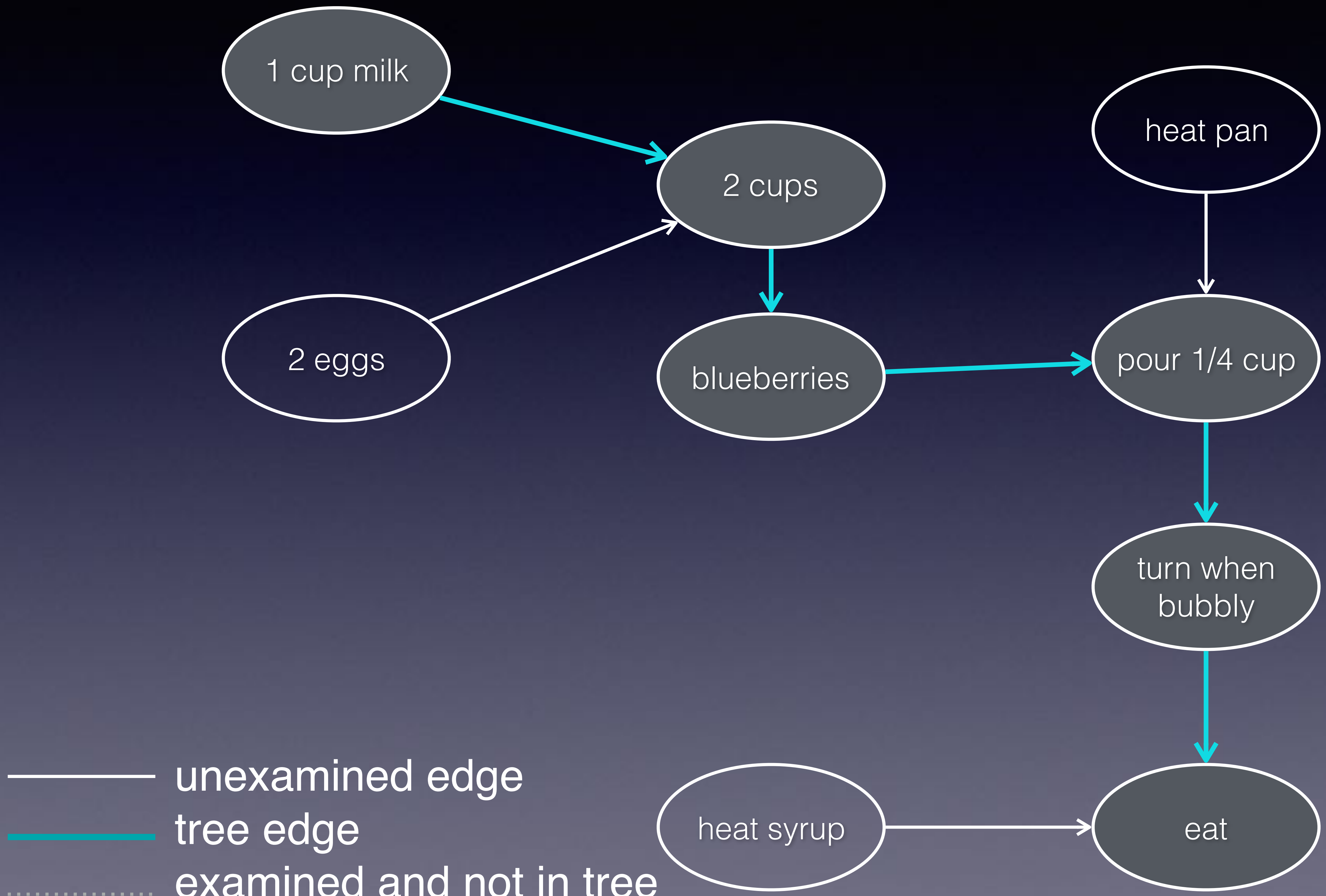
Making Blueberry Pancakes



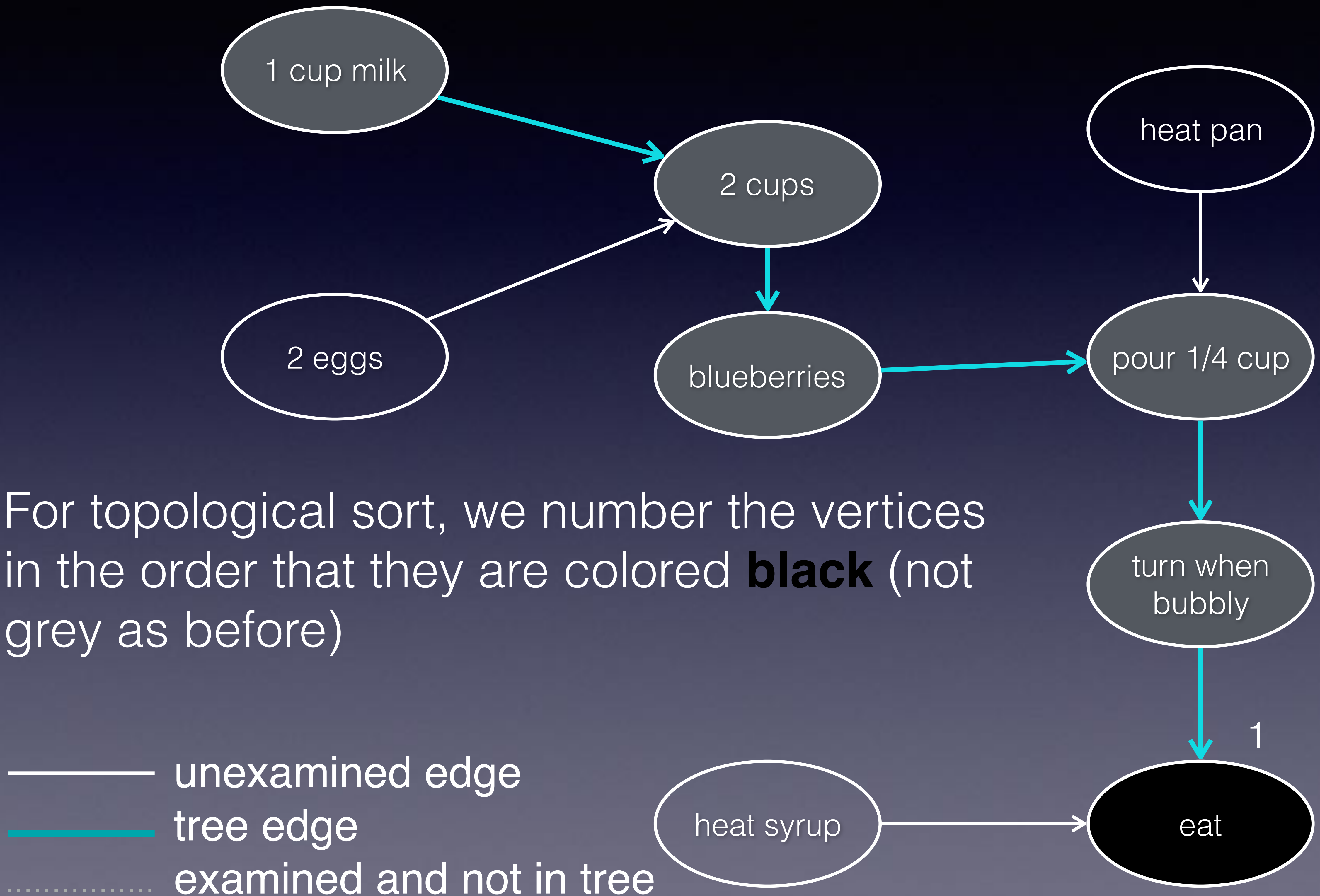
Making Blueberry Pancakes



Making Blueberry Pancakes

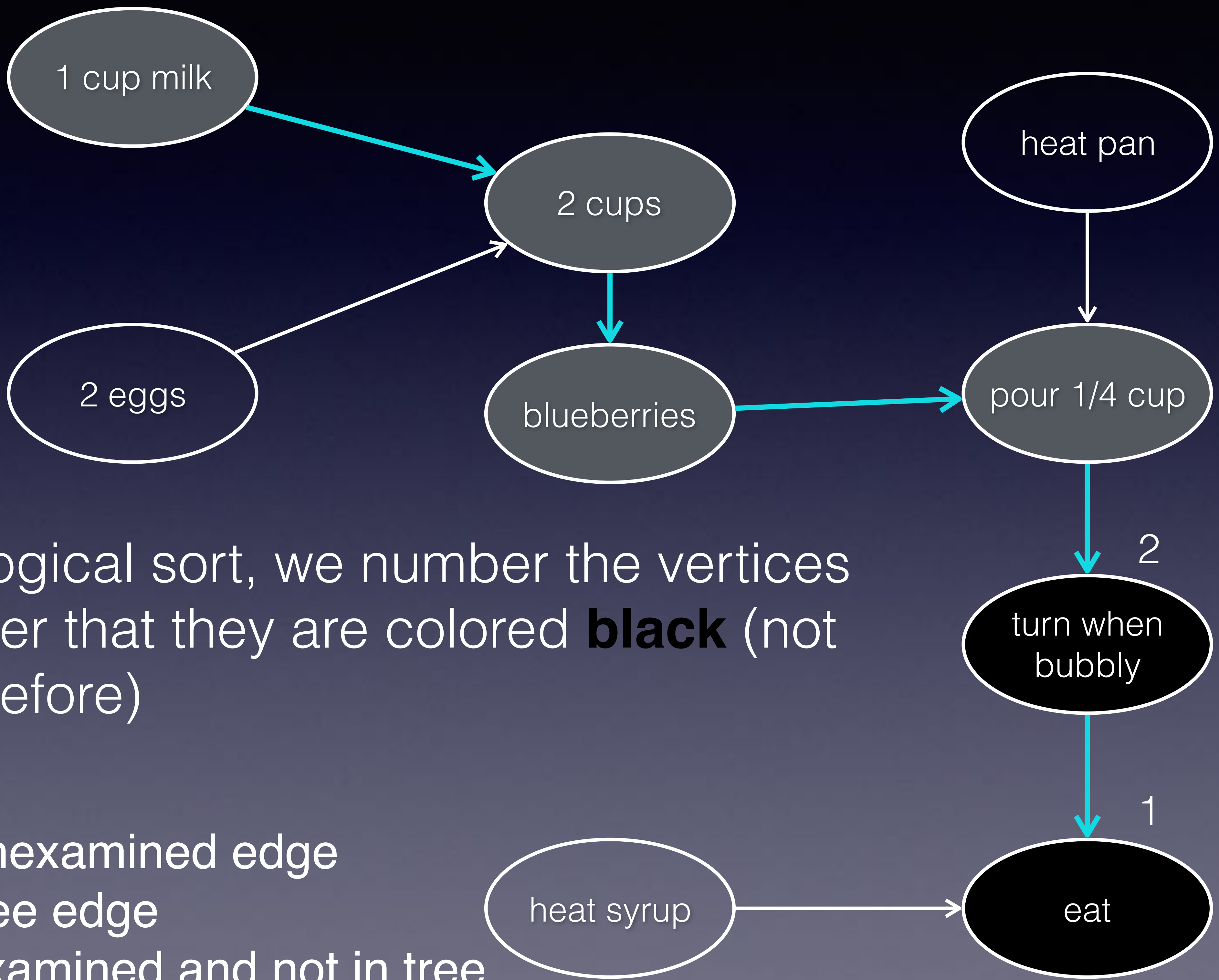


Making Blueberry Pancakes



For topological sort, we number the vertices in the order that they are colored **black** (not grey as before)

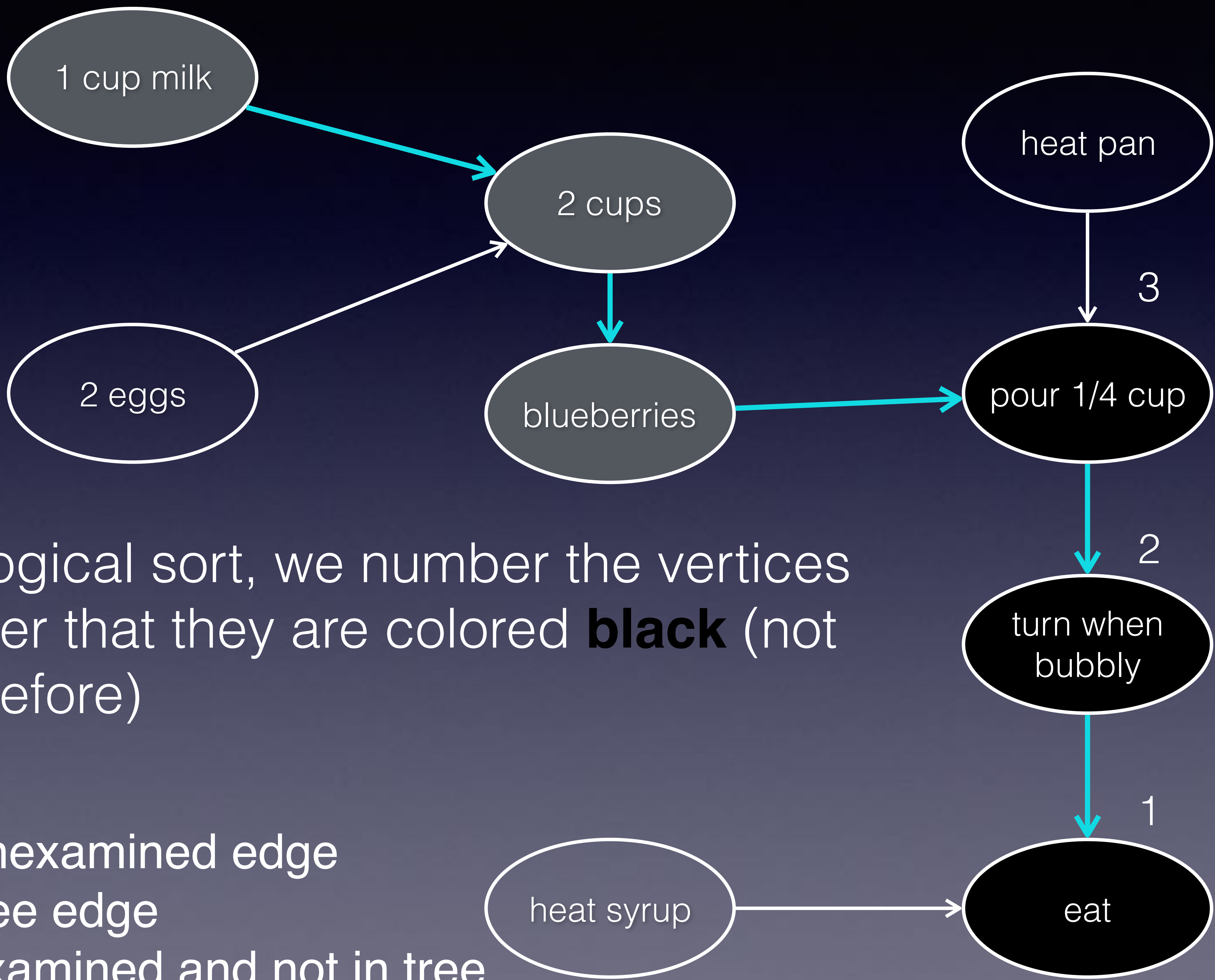
Making Blueberry Pancakes



For topological sort, we number the vertices in the order that they are colored **black** (not grey as before)

unexamined edge
tree edge
examined and not in tree

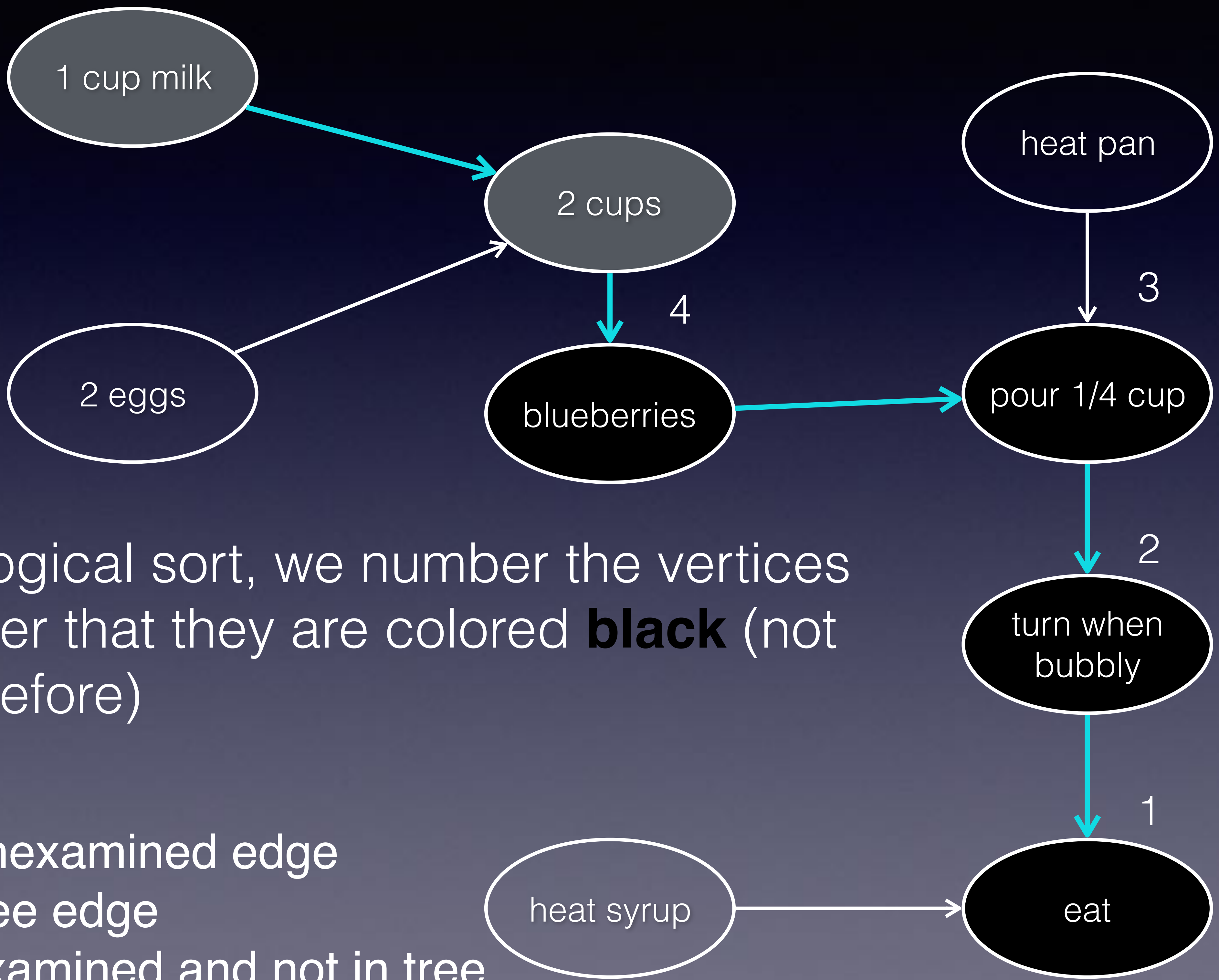
Making Blueberry Pancakes



For topological sort, we number the vertices in the order that they are colored **black** (not grey as before)

unexamined edge
tree edge
examined and not in tree

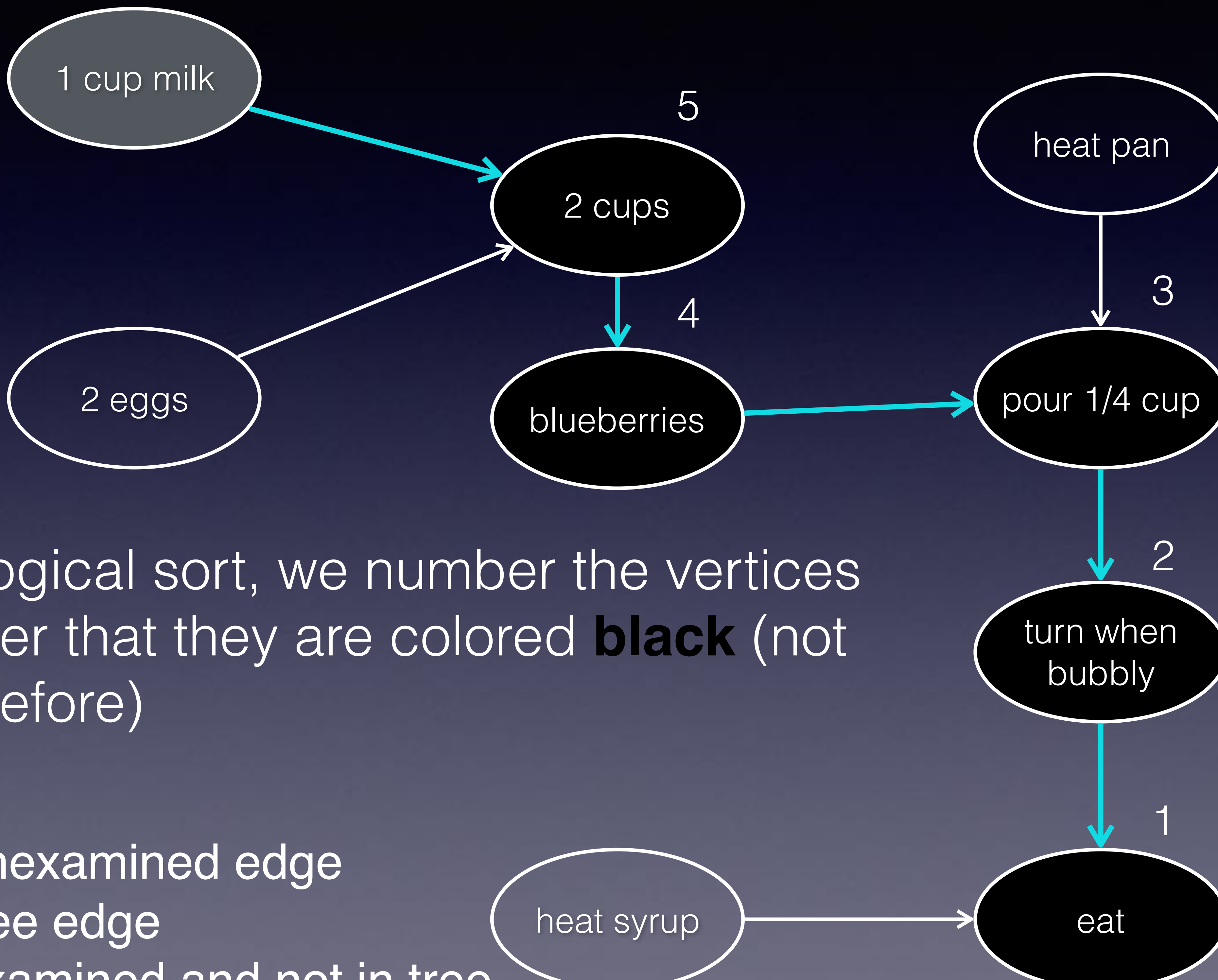
Making Blueberry Pancakes



For topological sort, we number the vertices in the order that they are colored **black** (not grey as before)

unexamined edge
tree edge
examined and not in tree

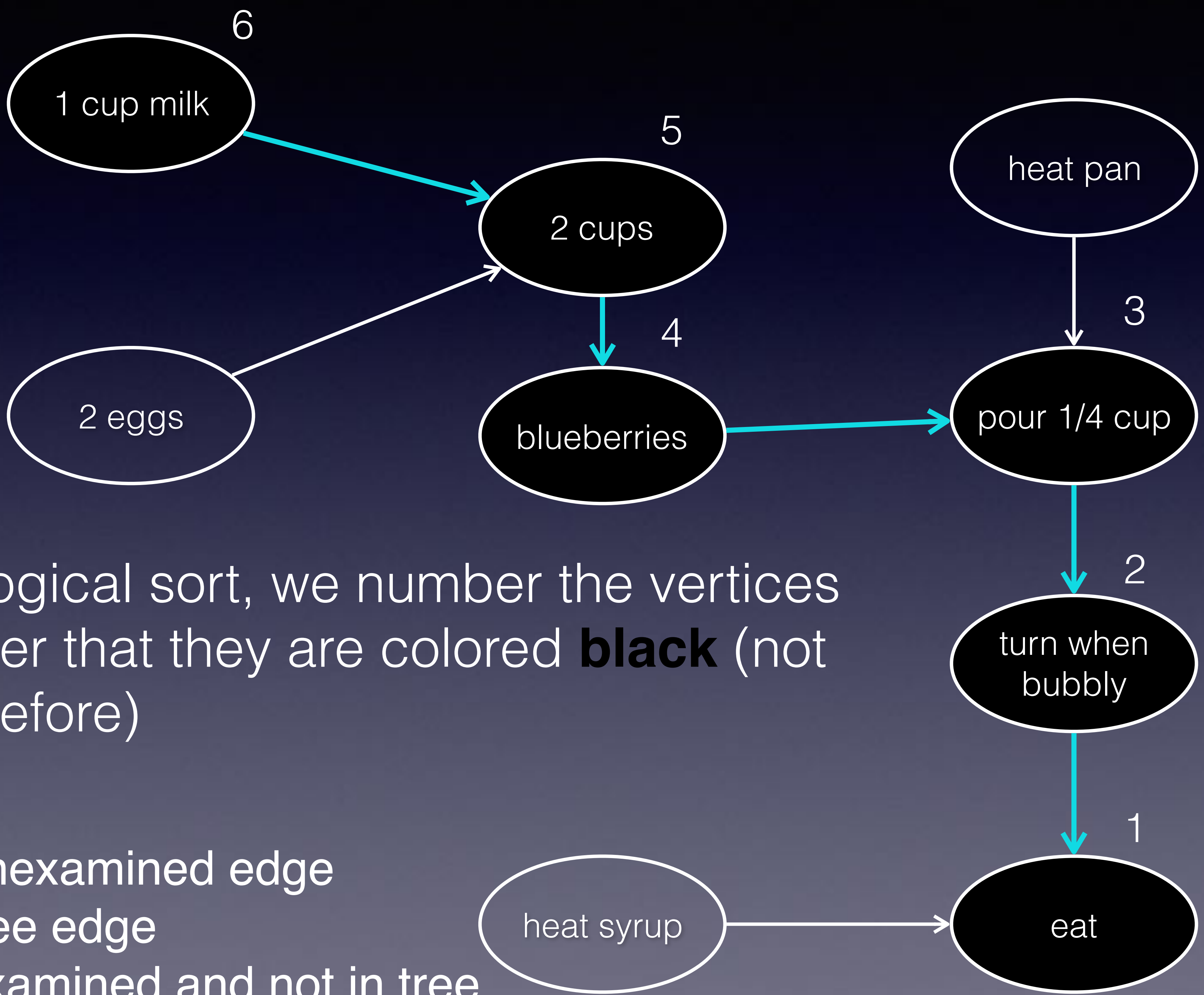
Making Blueberry Pancakes



For topological sort, we number the vertices in the order that they are colored **black** (not grey as before)

unexamined edge
tree edge
examined and not in tree

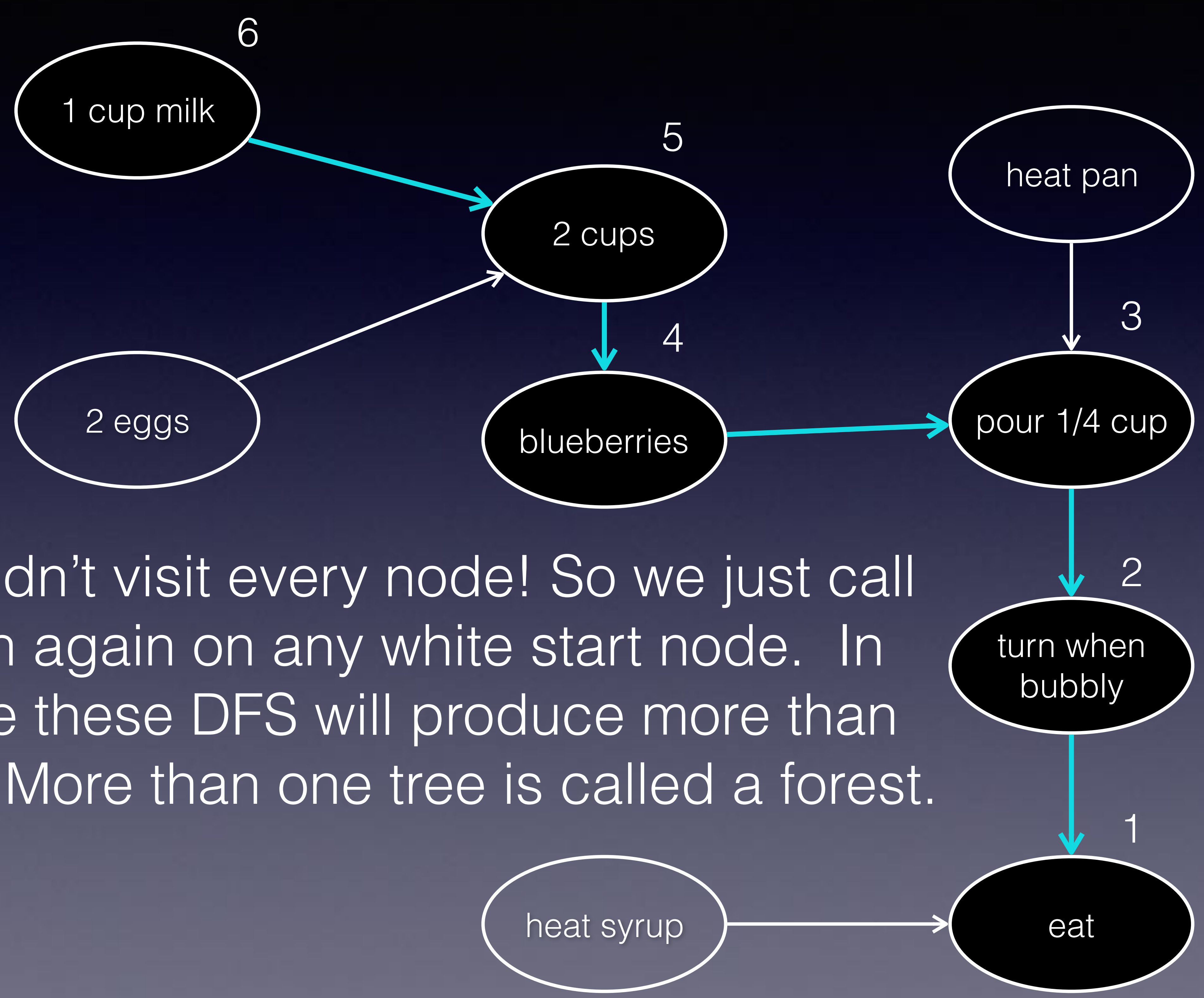
Making Blueberry Pancakes



For topological sort, we number the vertices in the order that they are colored **black** (not grey as before)

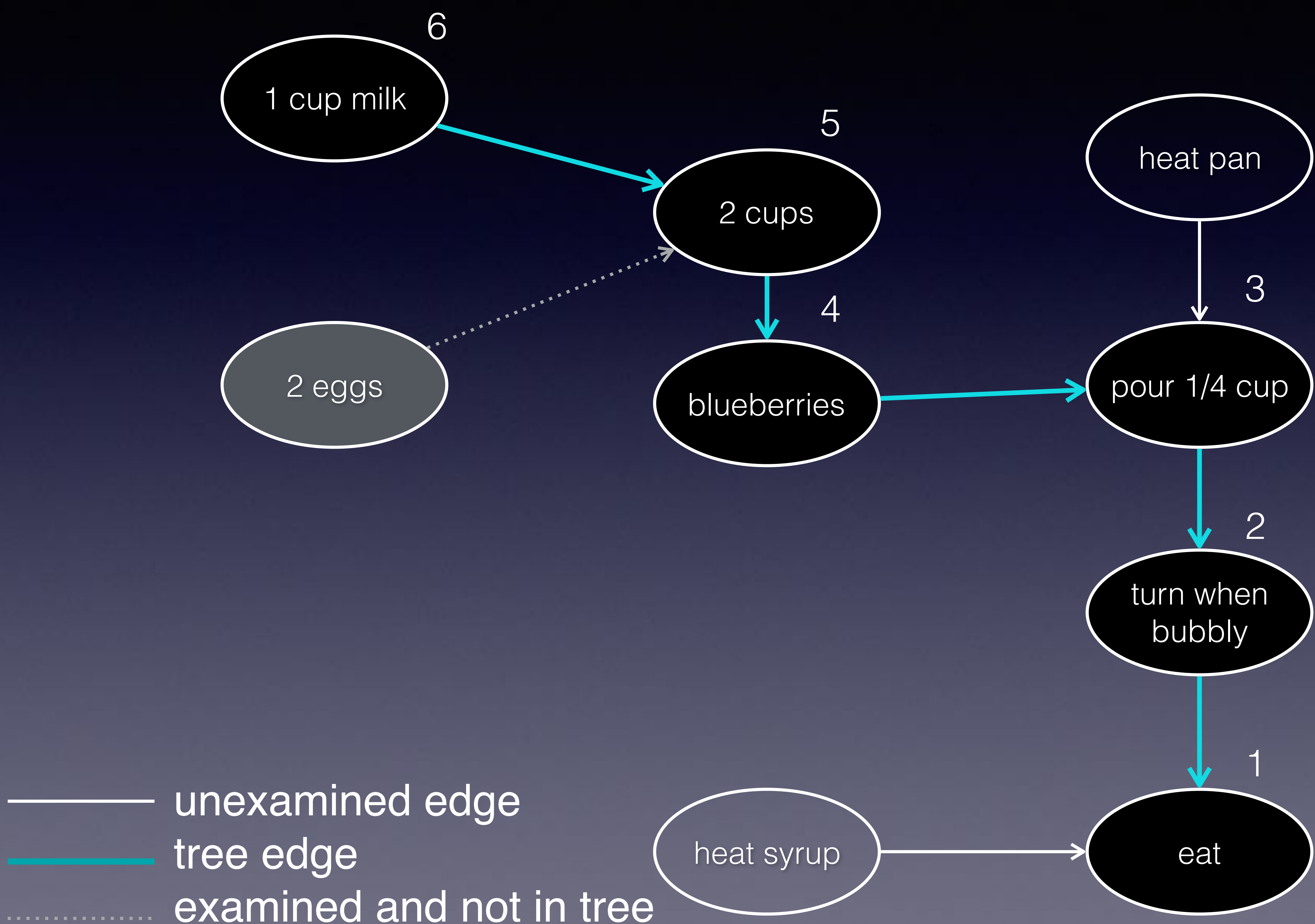
unexamined edge
tree edge
examined and not in tree

Making Blueberry Pancakes

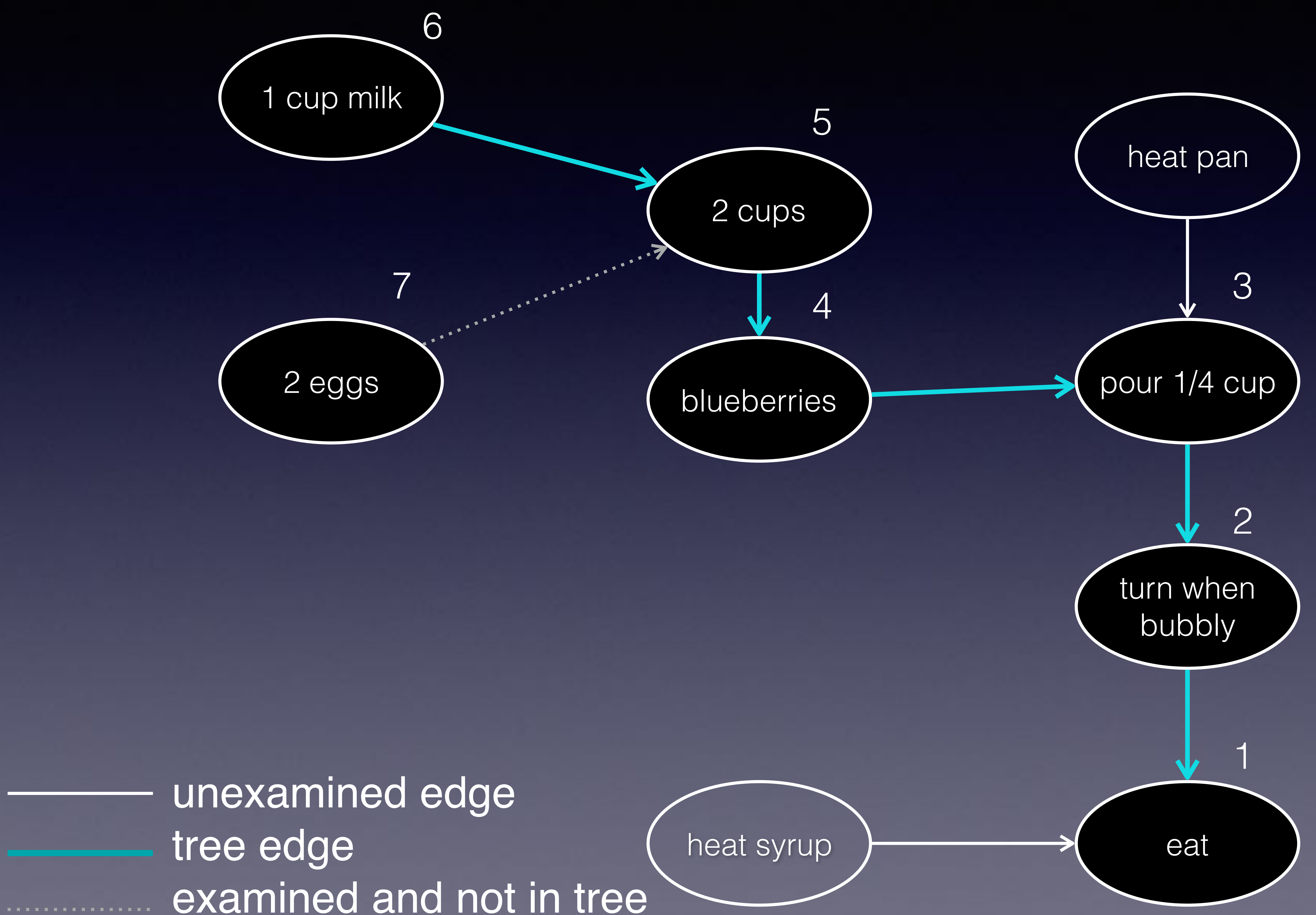


But we didn't visit every node! So we just call dfssearch again on any white start node. In cases like these DFS will produce more than one tree. More than one tree is called a forest.

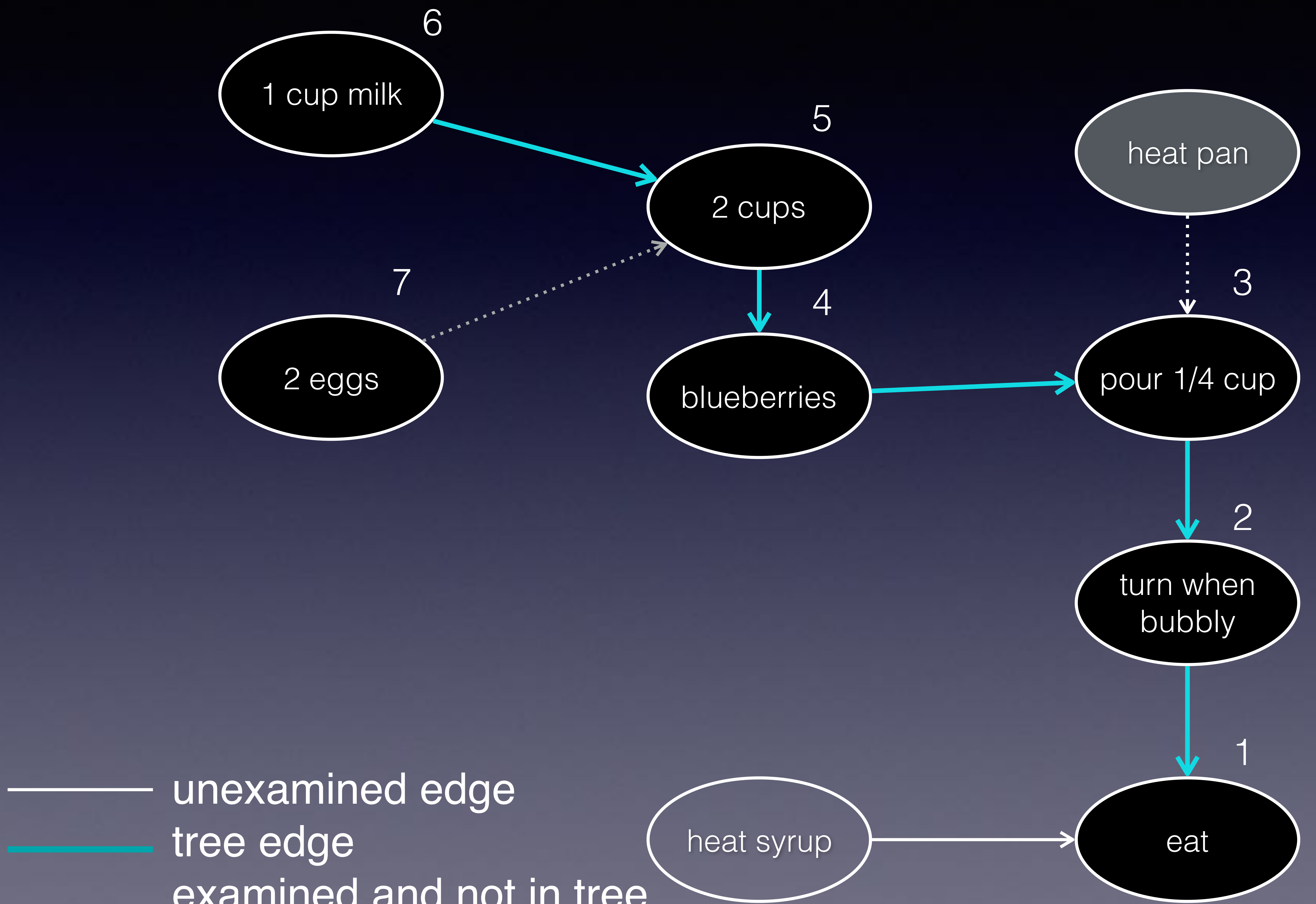
Making Blueberry Pancakes



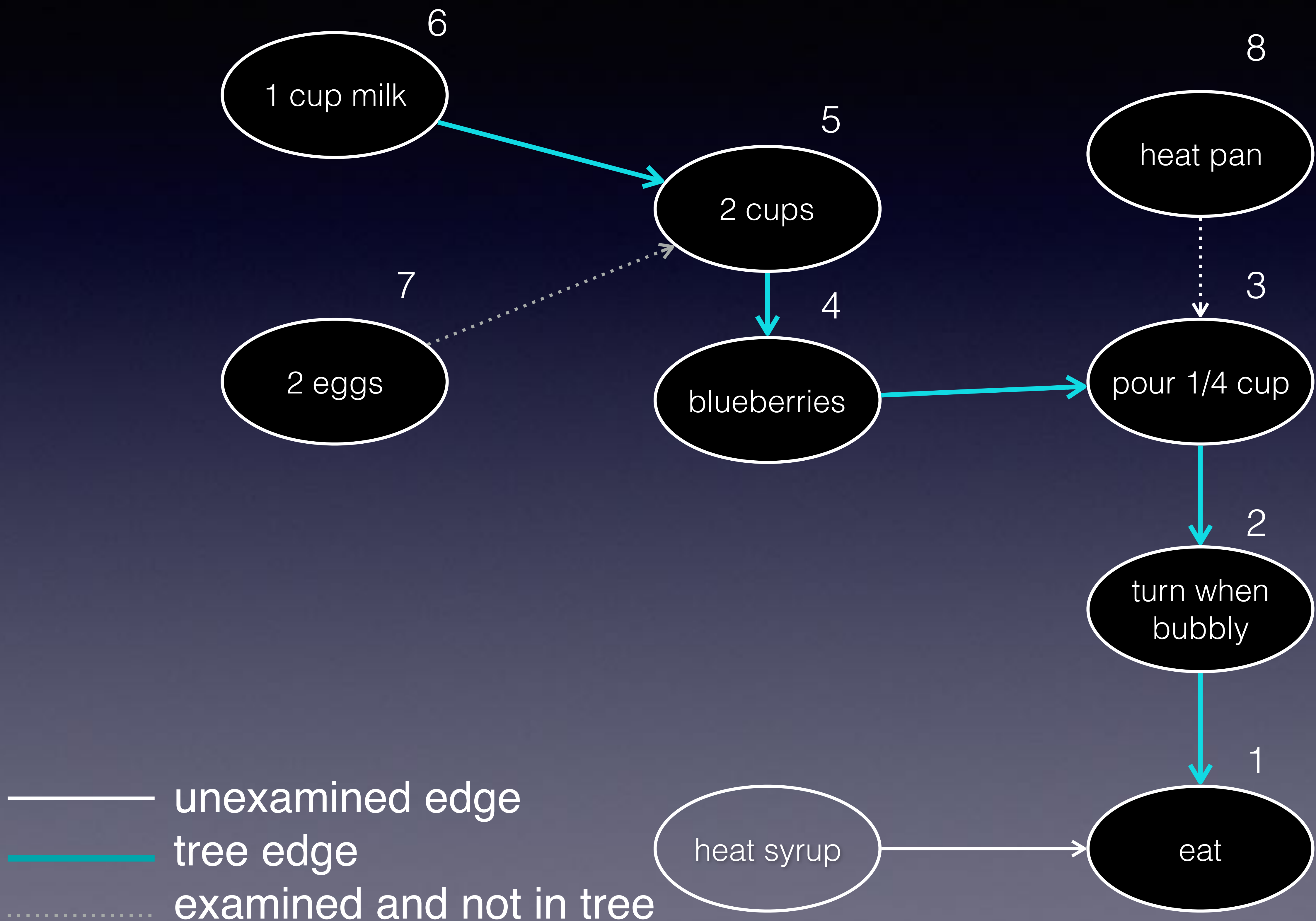
Making Blueberry Pancakes



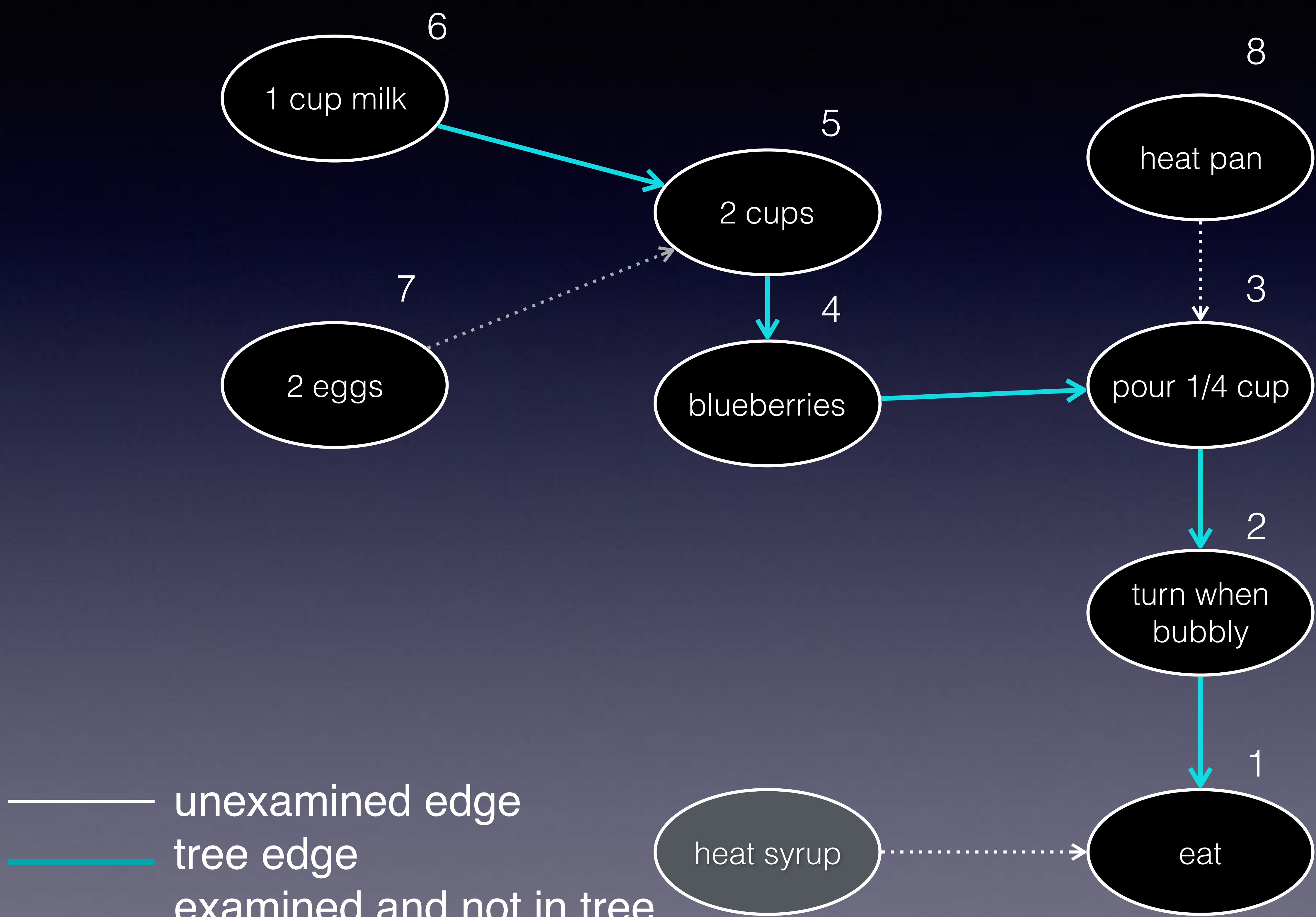
Making Blueberry Pancakes



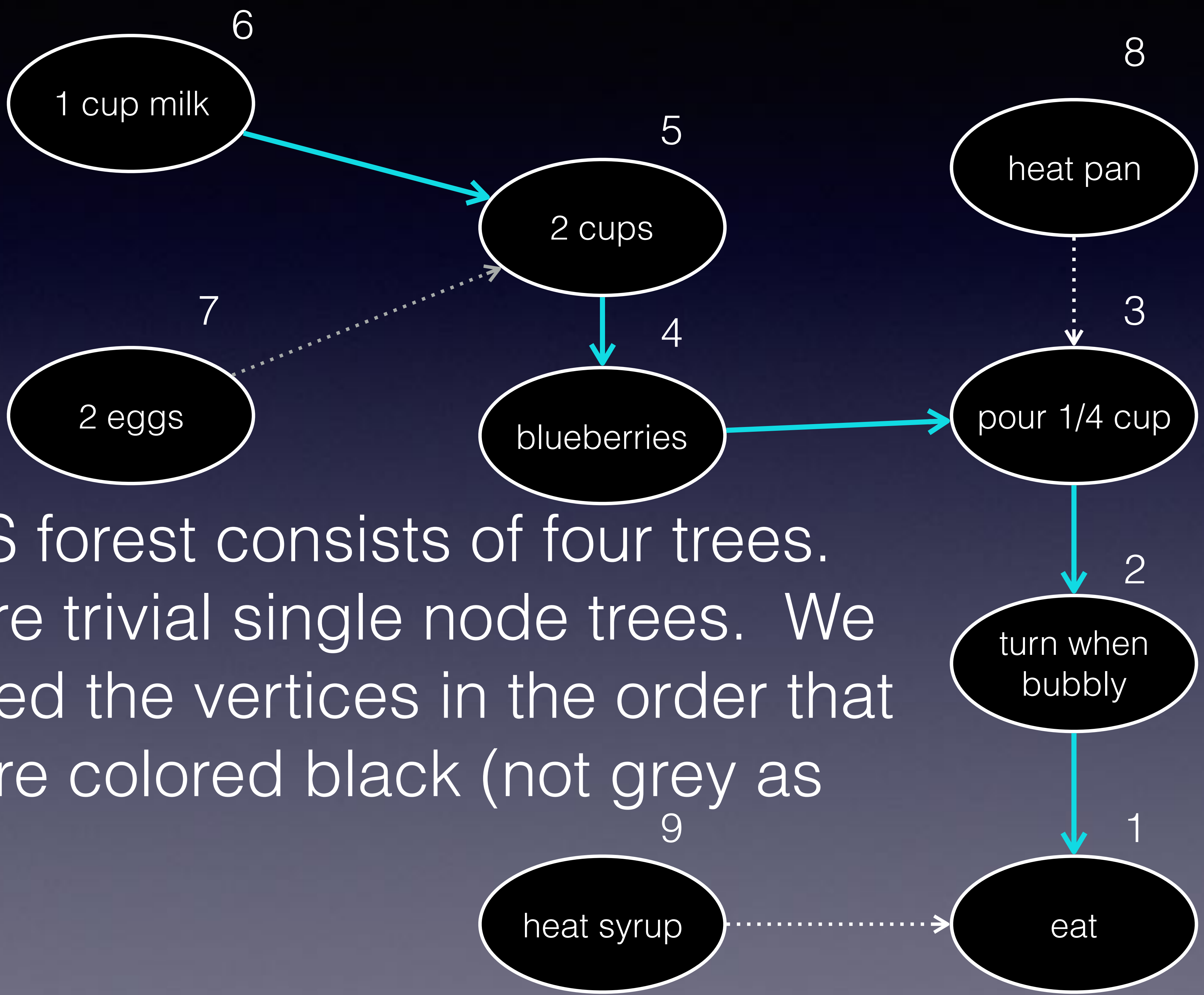
Making Blueberry Pancakes



Making Blueberry Pancakes

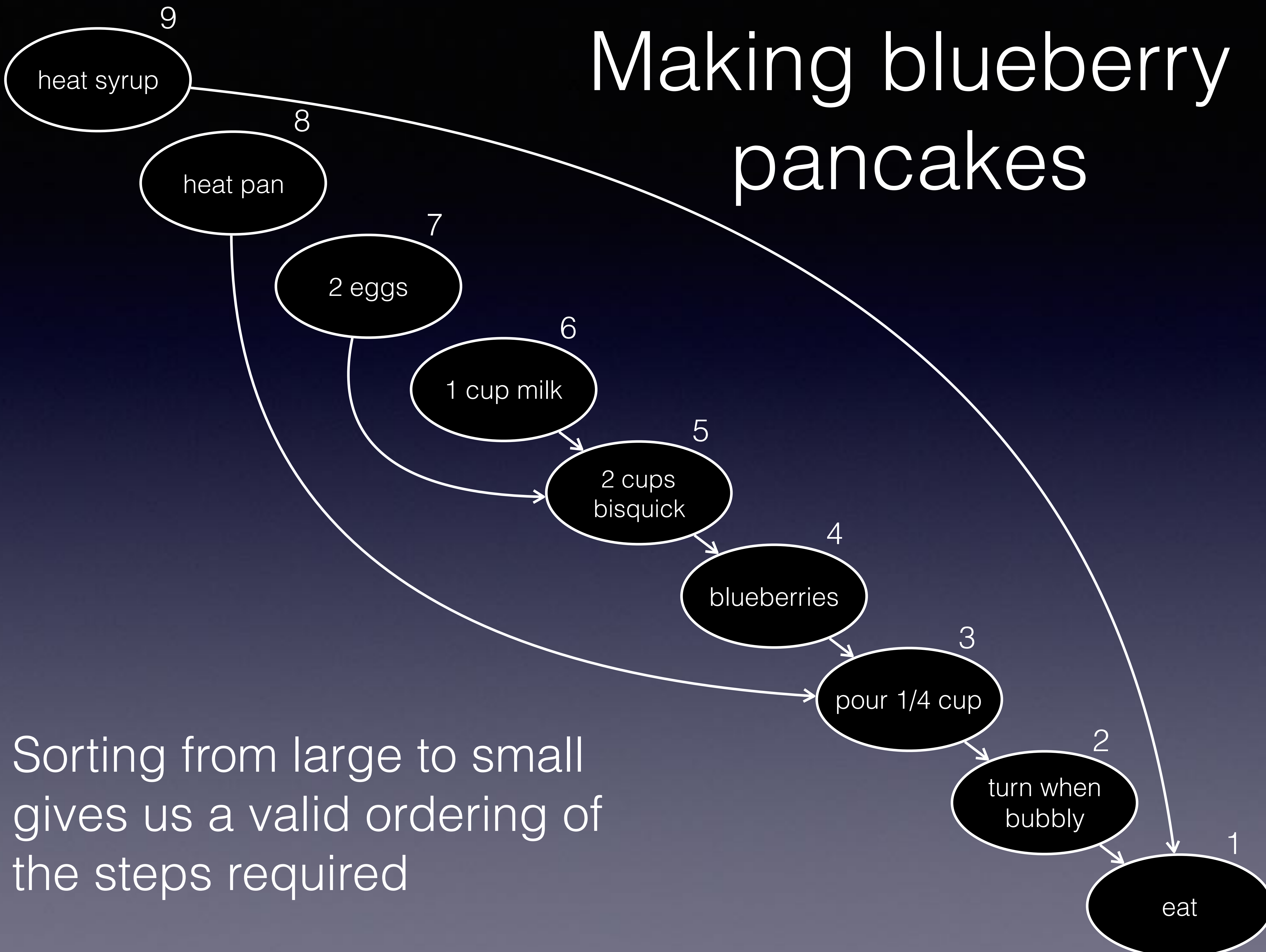


Making Blueberry Pancakes



Our DFS forest consists of four trees. Three are trivial single node trees. We numbered the vertices in the order that they were colored black (not grey as before)

Making blueberry pancakes

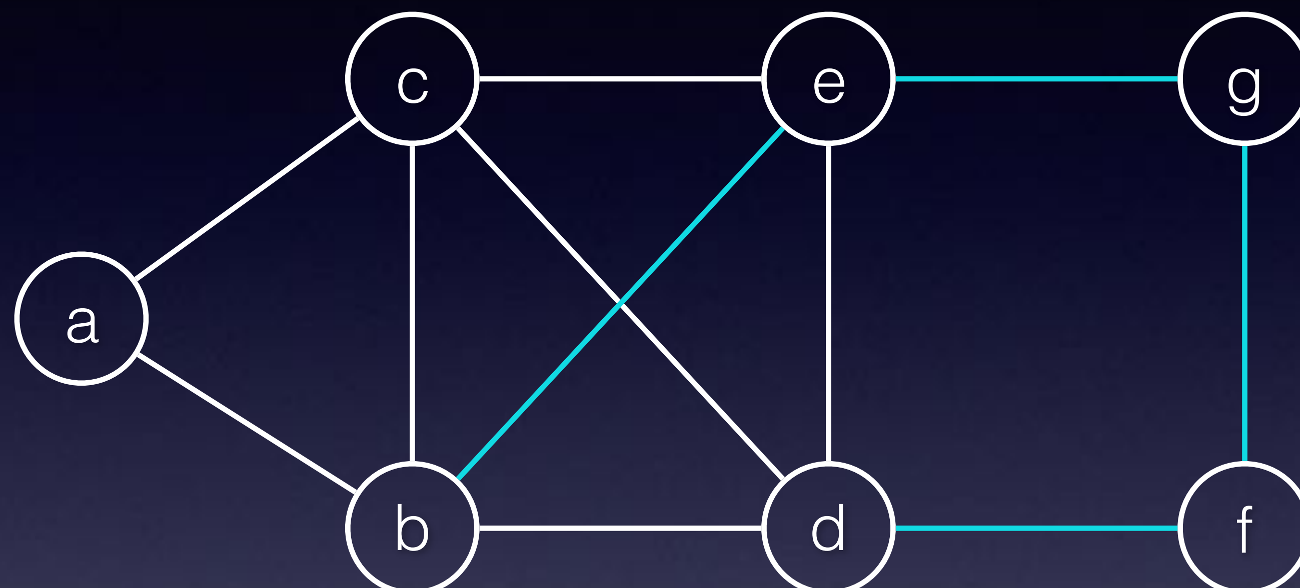


Sorting from large to small gives us a valid ordering of the steps required

Shortest Path

- Breadth-first search will find the shortest path from one vertex to another if all edges have the same weight.
- What if the edges have different weights?

Shortest Path

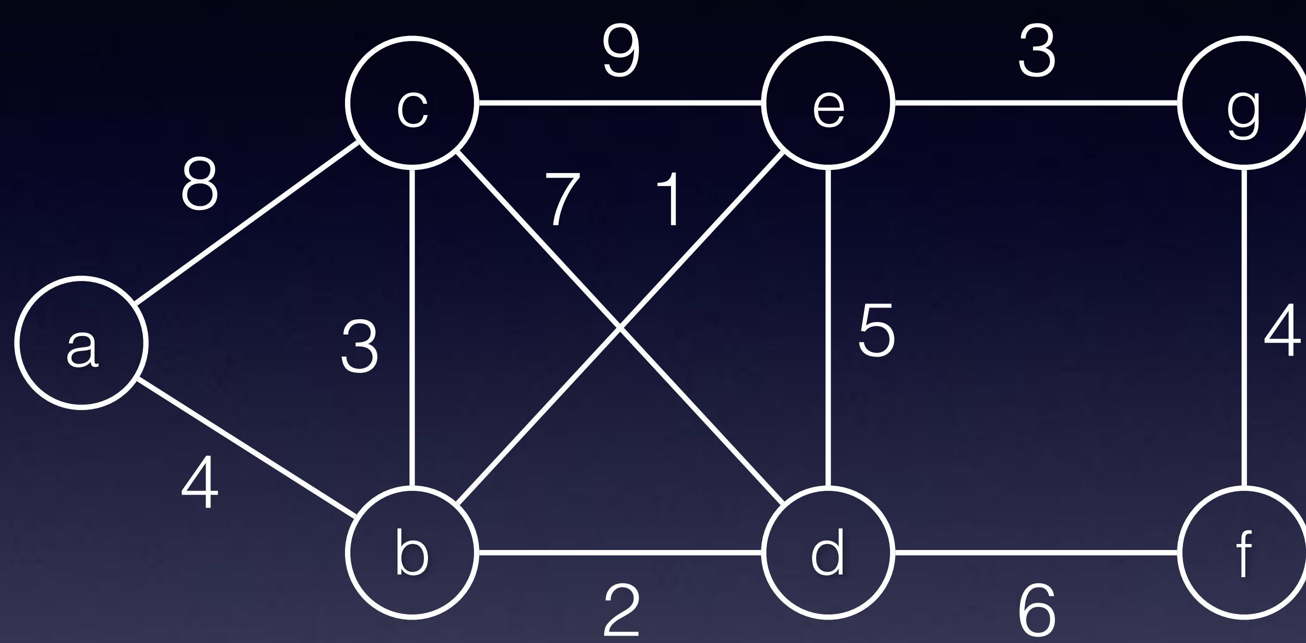


Path **P** = b,e,g,f,d

The length of **P** is the number of edges in the path

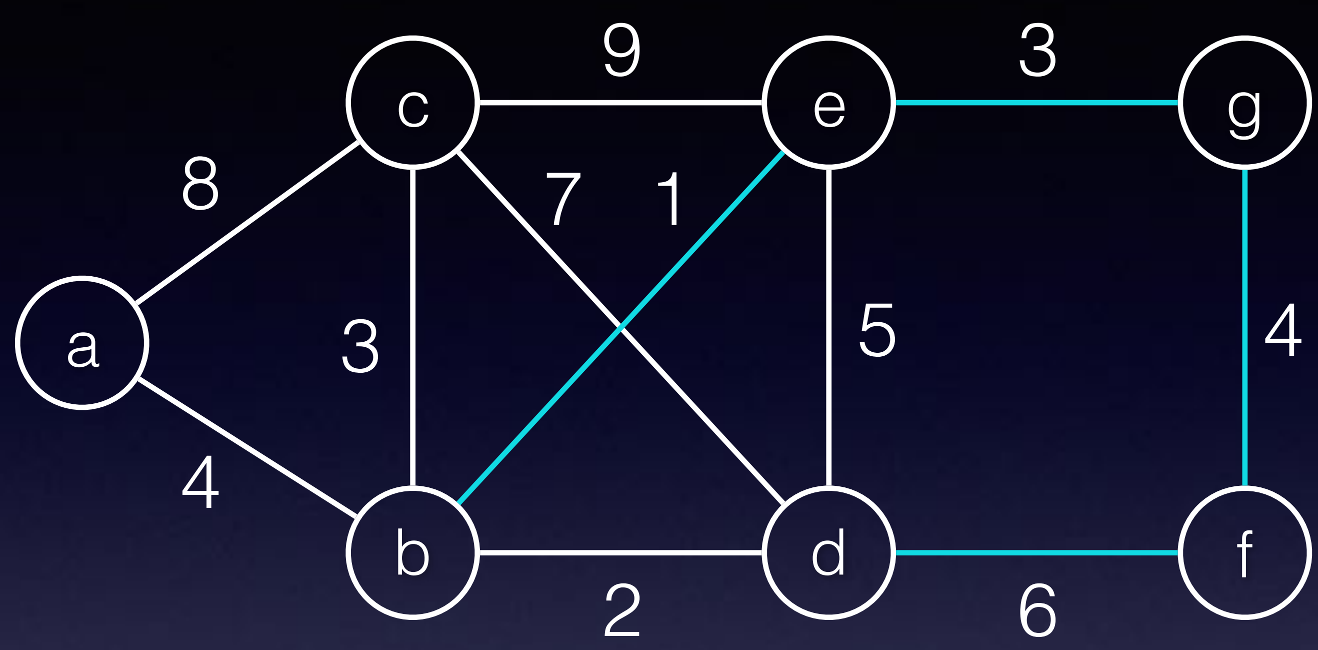
Here, the length of **P** is 4

Shortest Path



Now edges have a weight or cost, so path length not same for example, the road map (Google map, transportation network) edge is a road segment weight might be distance or expected time.

Shortest Path



weight is an edge property

e	a,b	a,c	b,c	b,d	b,e	c,d	c,e	d,e	d,f	e,g	f,g
weight[e]	4	8	3	2	1	7	9	5	6	3	4

length of **P** = sum of edge weights on path P = b,e,g,f,d

length of **P** = weight[b,e] + weight[e,g] + weight[g,f] + weight[f,d]

= 1 + 3 + 4 + 6 = 14

Shortest Path Problems

The algorithm we will demonstrate today is the classic shortest path algorithm called **Dijkstra's Algorithm**.

The algorithm finds the shortest path from a start vertex to all other vertices in the graph.

This is like a breadth-first search, but Dijkstra's Algorithm considers the weights on the graph

Shortest Path Problems

A path **P** from vertex **u** to **v** is a **shortest path** if there is no other path from **u** to **v** with a length that is shorter (smaller) than P.

Note:

If $\mathbf{P = w_1, w_2, \ldots, w_k}$ is a shortest path from $\mathbf{w_1}$ to $\mathbf{w_k}$, then $\mathbf{w_1, w_2, \ldots, w_{k-1}}$ is a shortest path from $\mathbf{w_1}$ to $\mathbf{w_{k-1}}$

Why? This is easily proven by contradiction. Since we can reach $\mathbf{w_k}$ from $\mathbf{w_{k-1}}$ if there was a shorter path from $\mathbf{w}$ to $\mathbf{w_{k-1}}$ then we could use it improve on **P**.

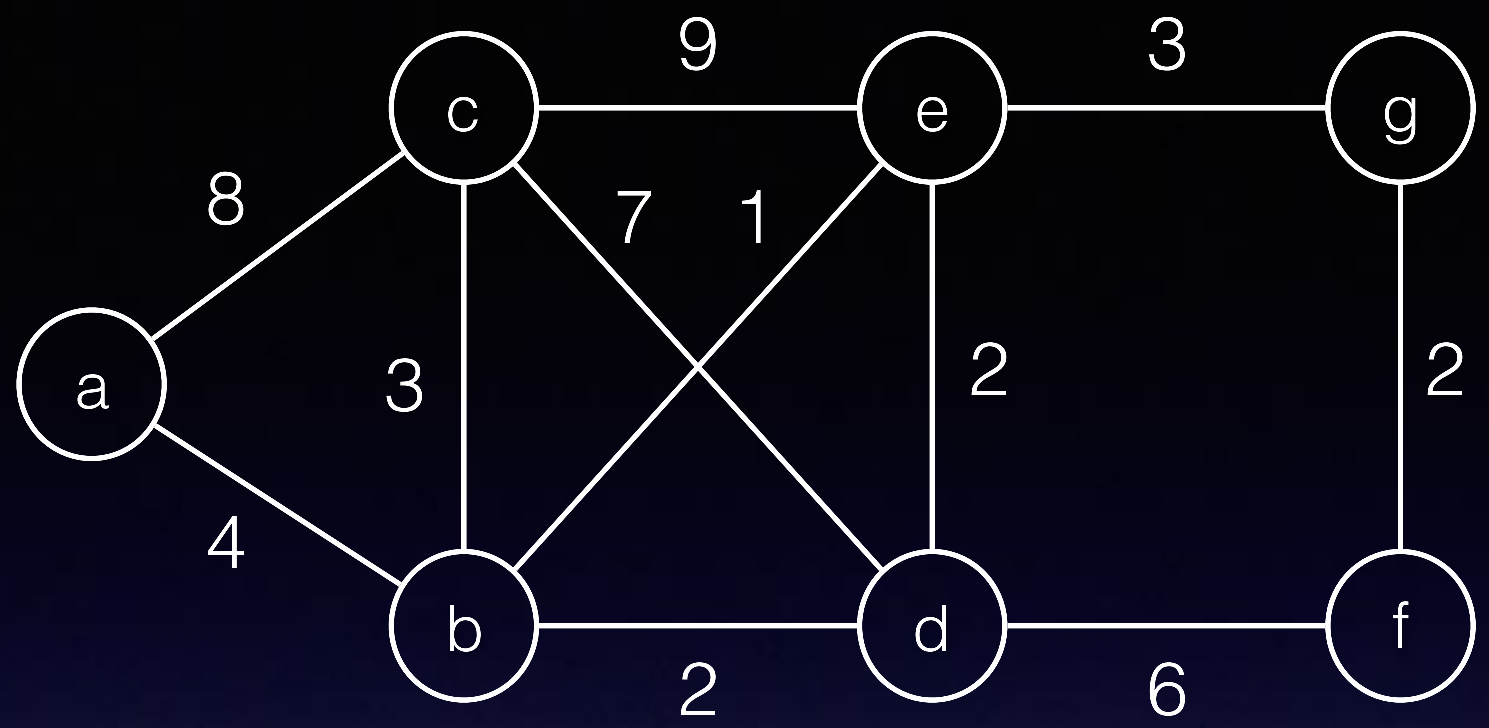
Expanding on this idea leads to Dijkstra's Algorithm...

Dijkstra’s Algorithm

To keep track of the total cost from the start vertex to each destination vertex, we associate a distance from start variable with each vertex.

That variable contains the current total weight/distance of the shortest path from the start vertex to the vertex in question.

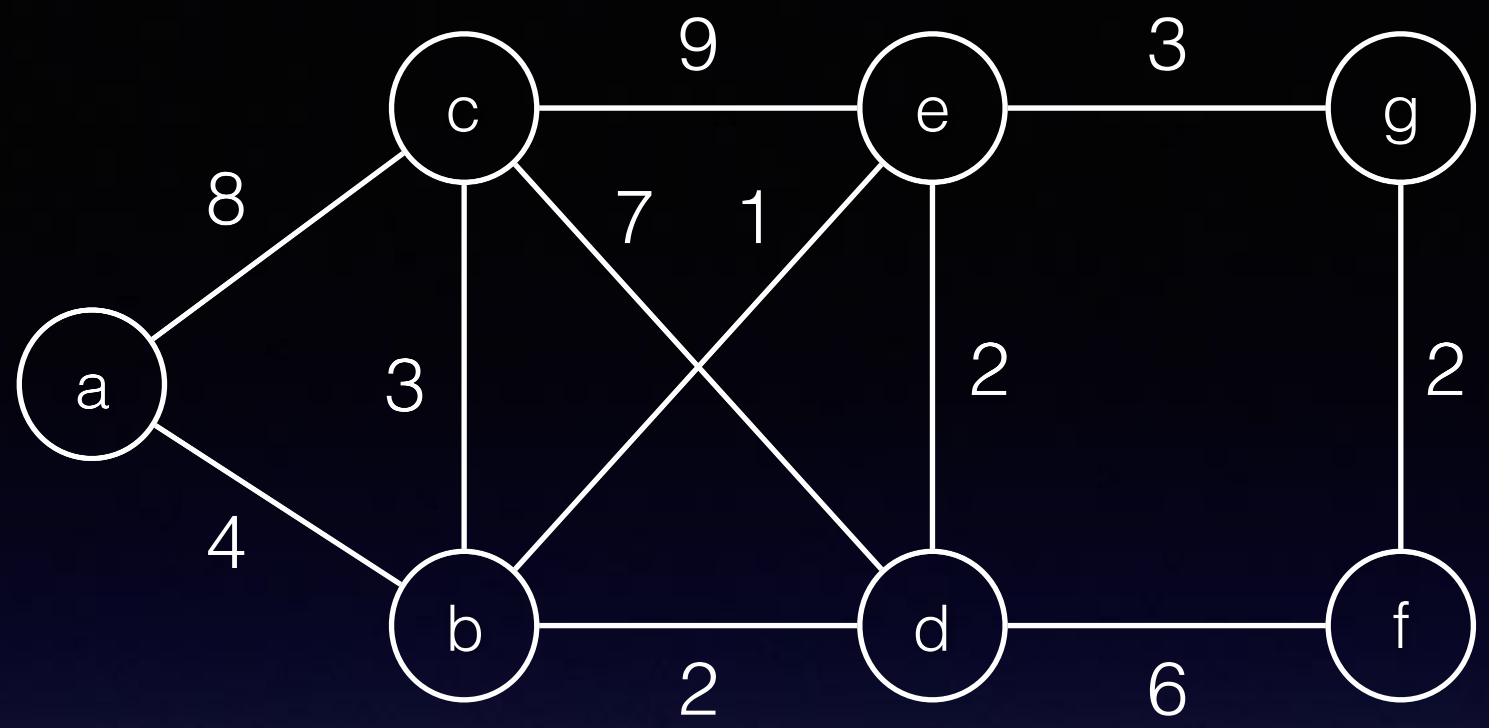
The algorithm iterates once for every vertex in the graph, but the order in which the vertices are processed is controlled by a priority queue. The distance from start determines the order of the vertices in the priority queue. The initial distance value for all the vertices is infinity.



vertex:	a	b	c	d	e	f	g
d(a,v):	∞	∞	∞	∞	∞	∞	∞
predecessor:	?	?	?	?	?	?	?

for each vertex **v** in the graph:

set distance from **a** to **v**, **d(a,v)** = infinity
set predecessor vertex to ?
enqueue the vertex on priority queue.

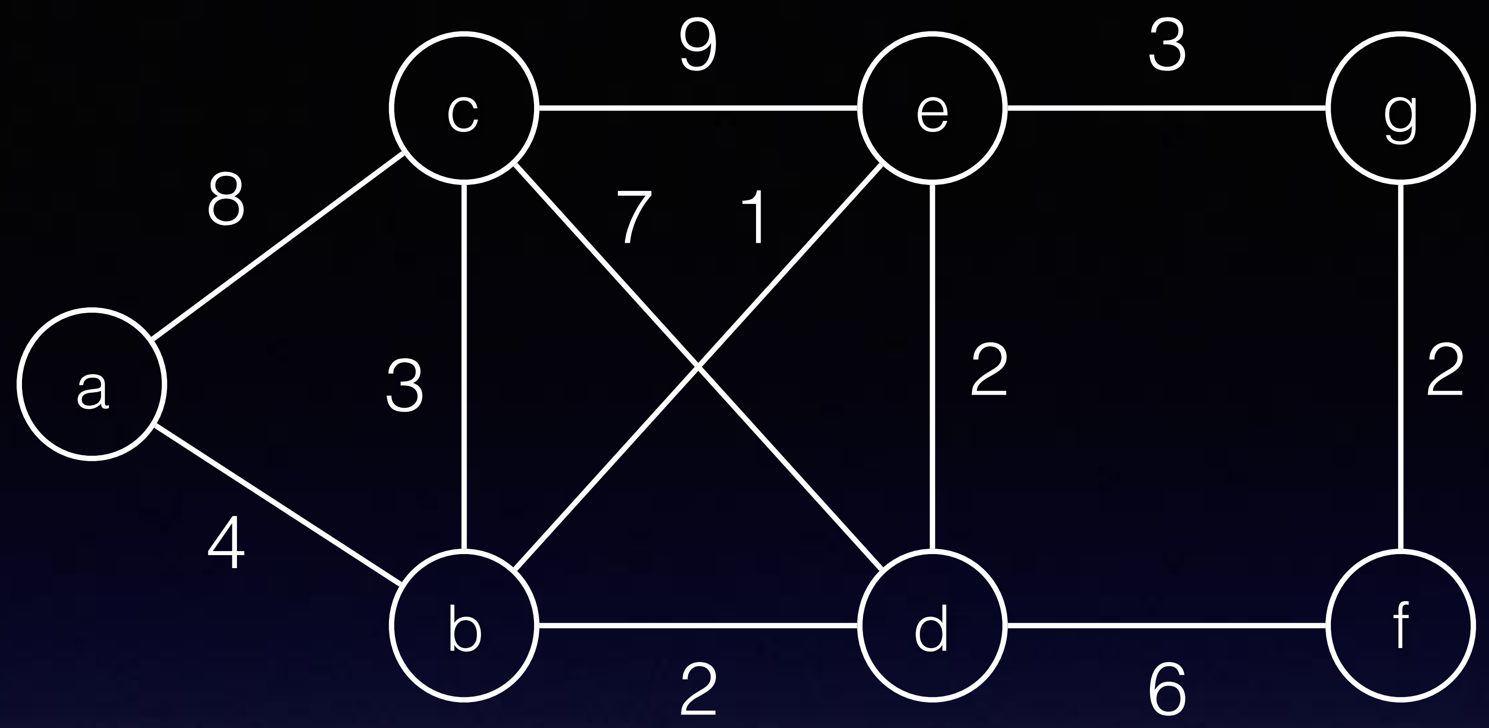


vertex:	a	b	c	d	e	f	g
d(a,v):	∞	∞	∞	∞	∞	∞	∞
predecessor:	?	?	?	?	?	?	?

At the end of Dijkstra's Algorithm:

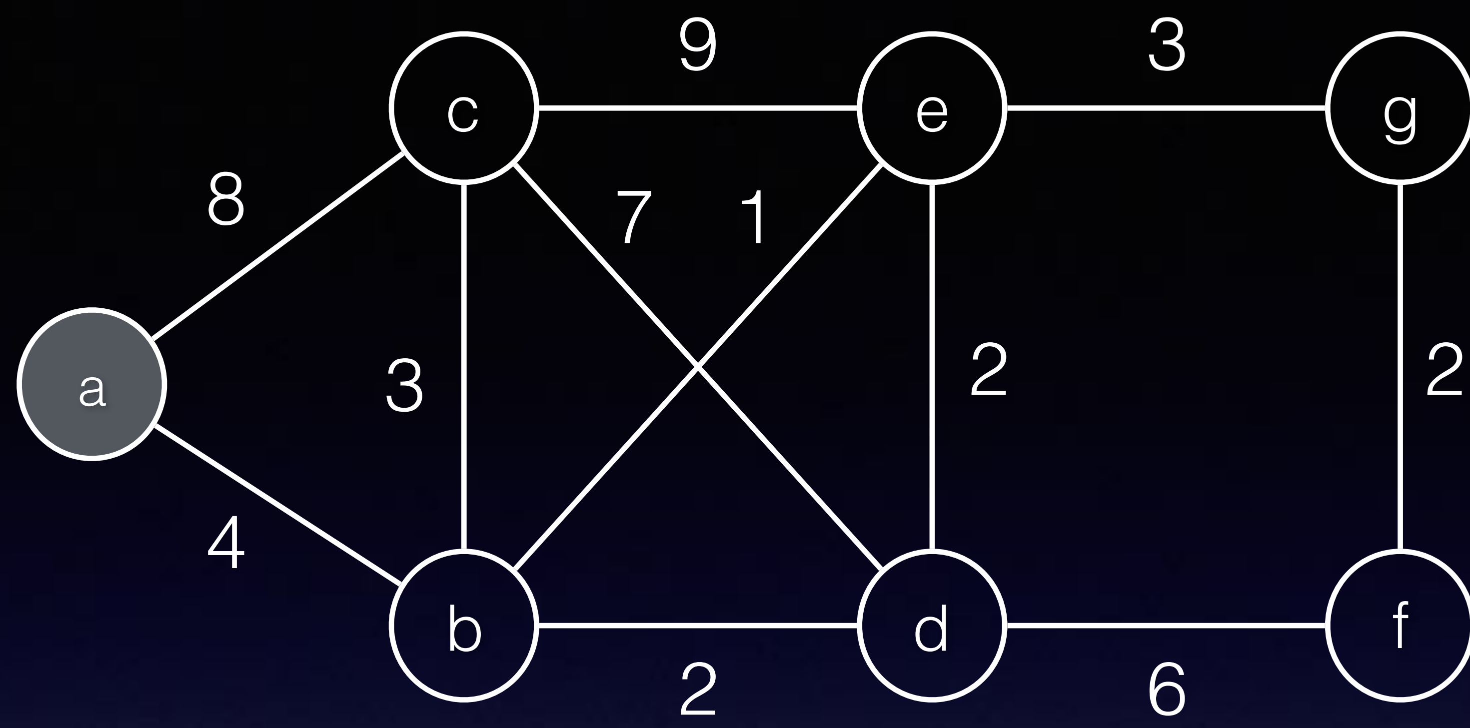
d(a,v) will contain the length of the shortest path from **a** to **v**.

predecessor will contain the links (edges) necessary to get from **v** back to **a**.



vertex:	a	b	c	d	e	f	g
d(a,v):	0	∞	∞	∞	∞	∞	∞
predecessor:	?	?	?	?	?	?	?

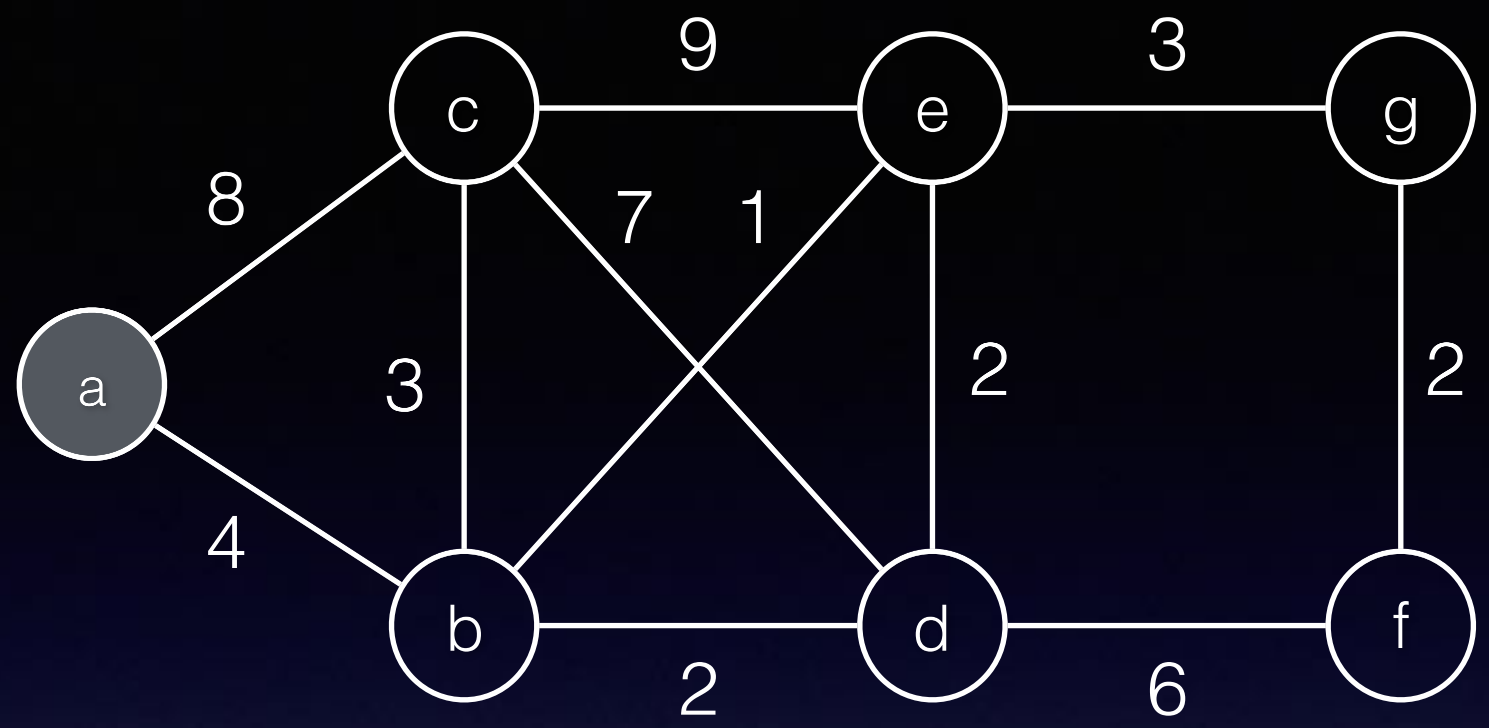
set **d(a,a)** = 0



vertex:	a	b	c	d	e	f	g
d(a,v):	0	∞	∞	∞	∞	∞	∞
predecessor:	?	?	?	?	?	?	?

dequeue unvisited vertex with shortest distance to **a**:

current vertex: **a**

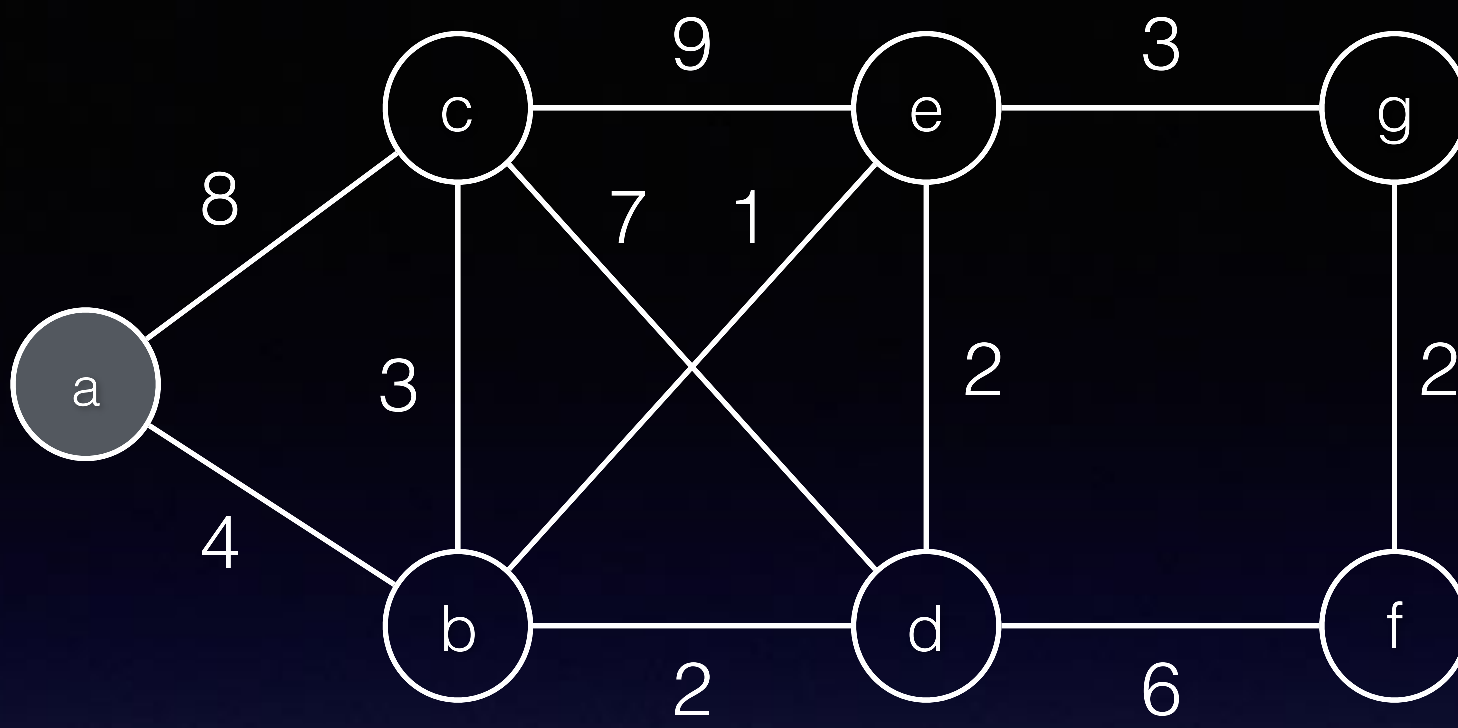


vertex:	a	b	c	d	e	f	g
d(a,v):	0	∞	∞	∞	∞	∞	∞
predecessor:	?	?	?	?	?	?	?

for each unvisited vertex adjacent to the current vertex:

compute distance from start to current and continue on to adjacent vertex

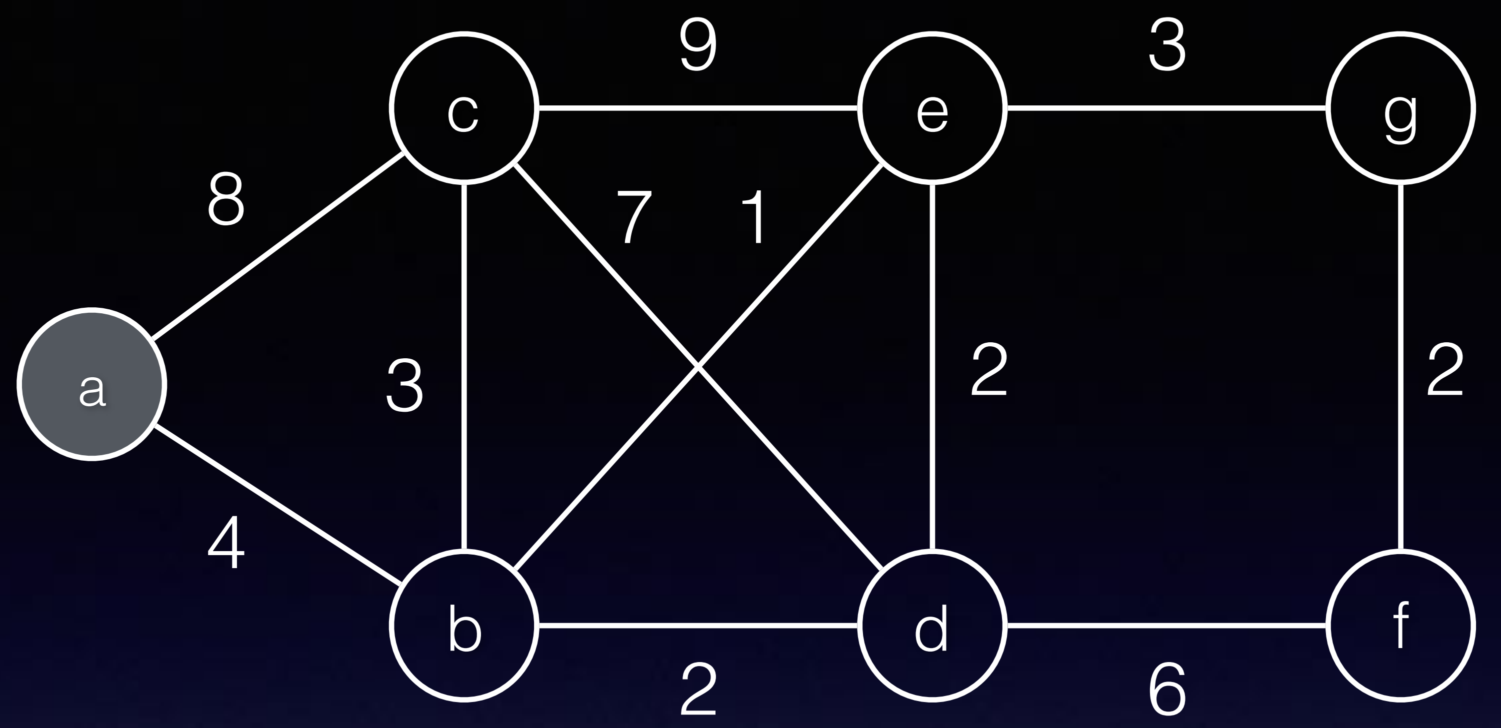
if that path's distance is **shorter** than the adjacent vertex's current distance, **update** the adjacent vertex's **distance** and **predecessor**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	8	∞	∞	∞	∞
predecessor:	?	a	a	?	?	?	?

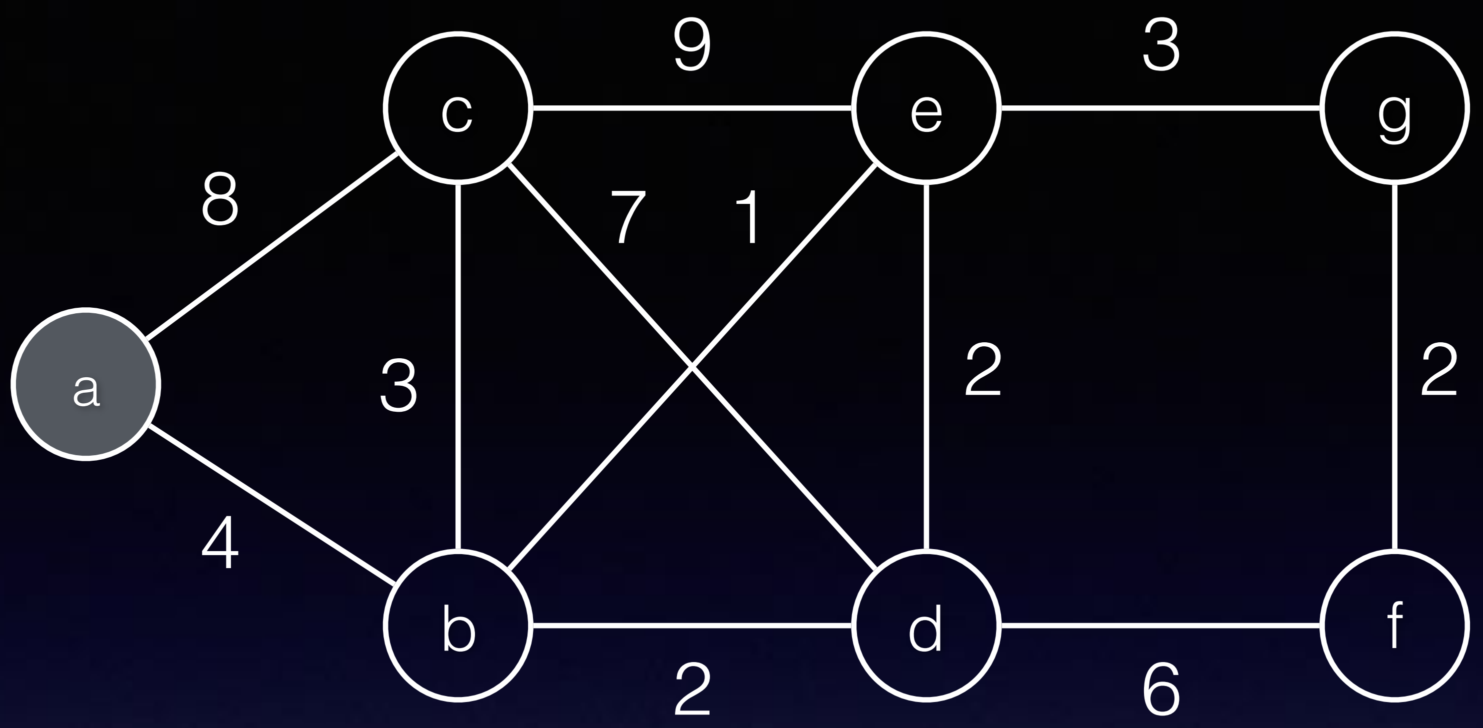
for each unvisited vertex adjacent to the current vertex:

- compute distance from start to current and continue on to adjacent vertex
- if that path's distance is **shorter** than the adjacent vertex's current distance, **update** the adjacent vertex's **distance** and **predecessor**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	8	∞	∞	∞	∞
predecessor:	?	a	a	?	?	?	?

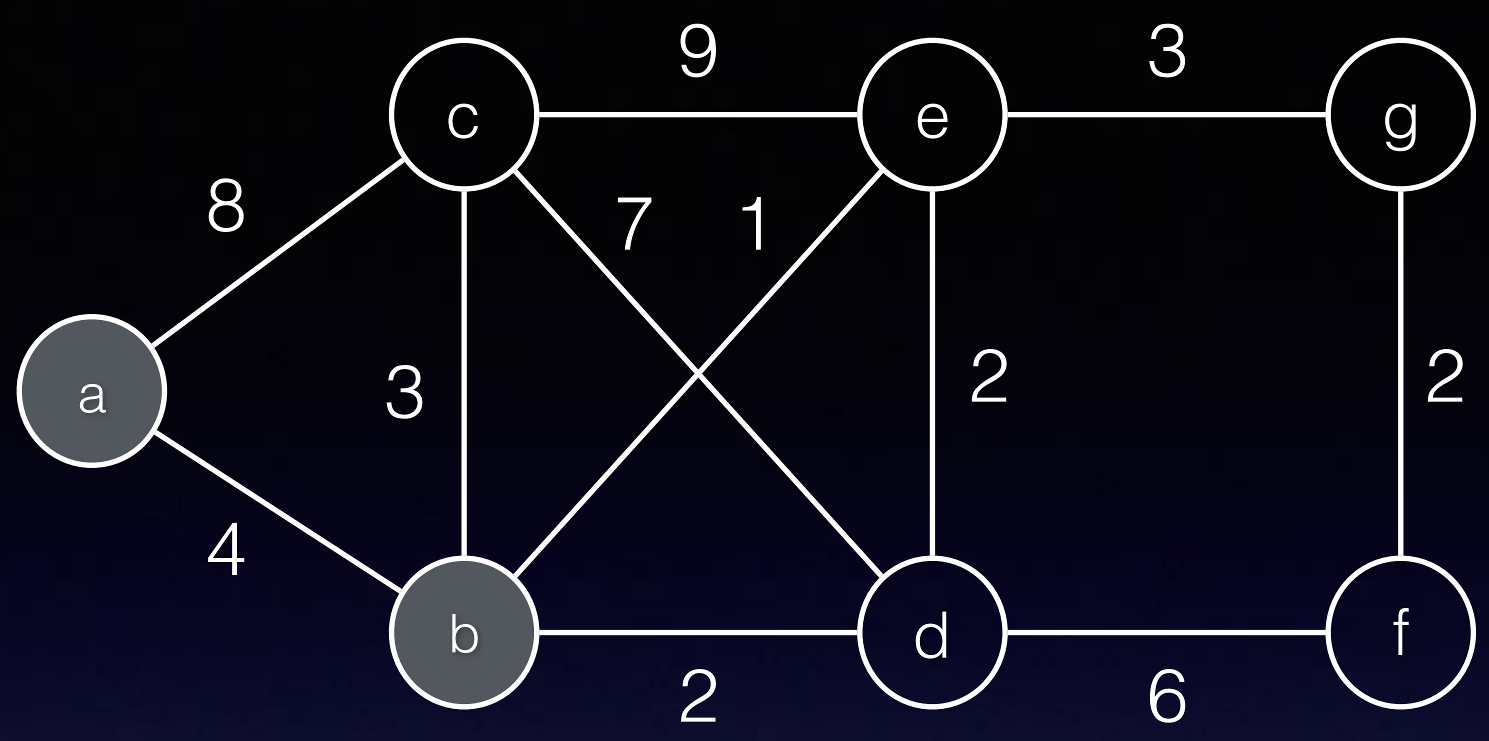
Once we are done visiting a vertex v, then Dijkstra's Algorithm guarentees **d(a,v)** will contain the shortest path from **a** to **v**.



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	8	∞	∞	∞	∞
predecessor:	?	a	a	?	?	?	?

What do we know about the final **shortest path d(a,b)** from **a** to **b** or **d(a,c)** from **a** to **c**?

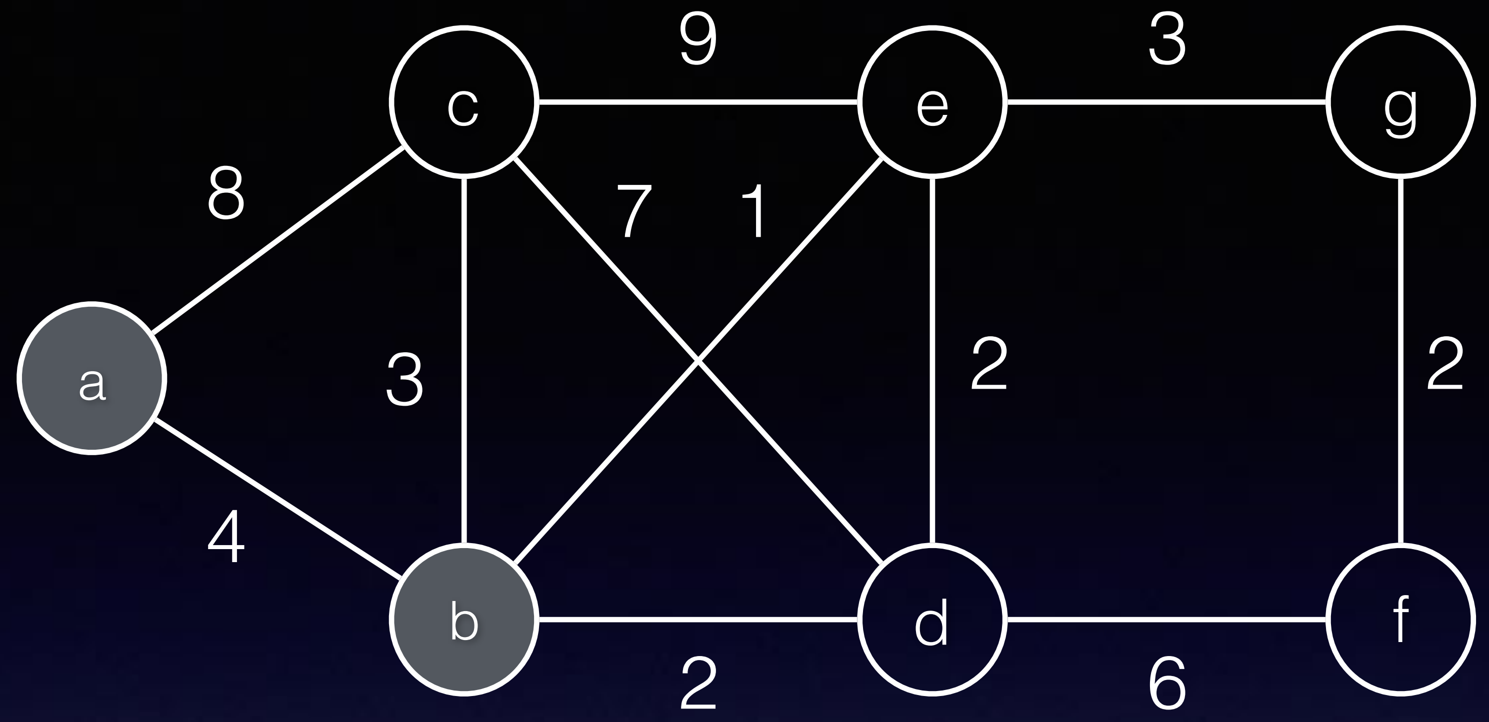
- We know **d(a,b)** will be **less than or equal to 4**
- We know **d(a,c)** will be **less than or equal to 8**
- We might find a shorter path later on



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	8	∞	∞	∞	∞
predecessor:	?	a	a	?	?	?	?

dequeue unvisited vertex with shortest distance to **a**

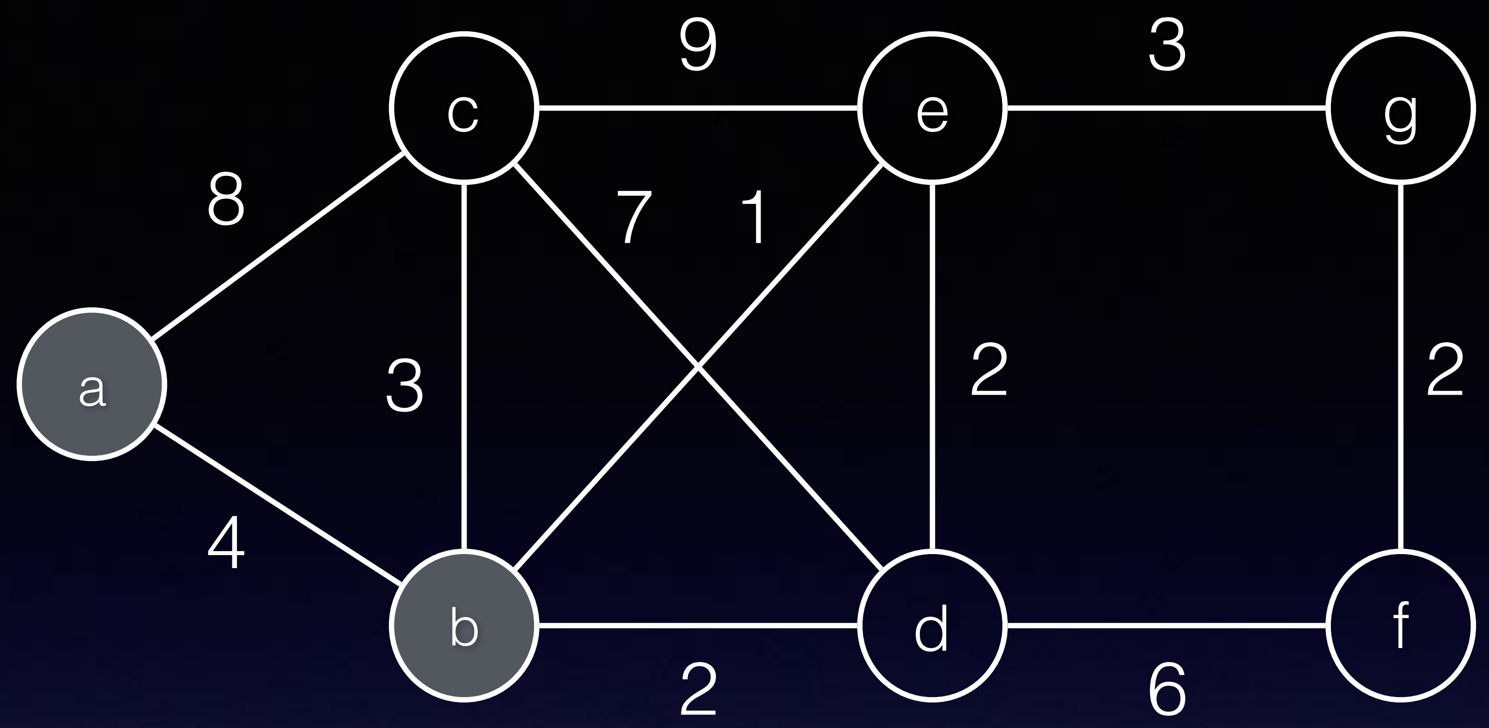
current vertex is **b**
adjacent vertices are **c e d**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	8	∞	∞	∞	∞
predecessor:	?	a	a	?	?	?	?

for adjacent unvisited vertices **c e d**

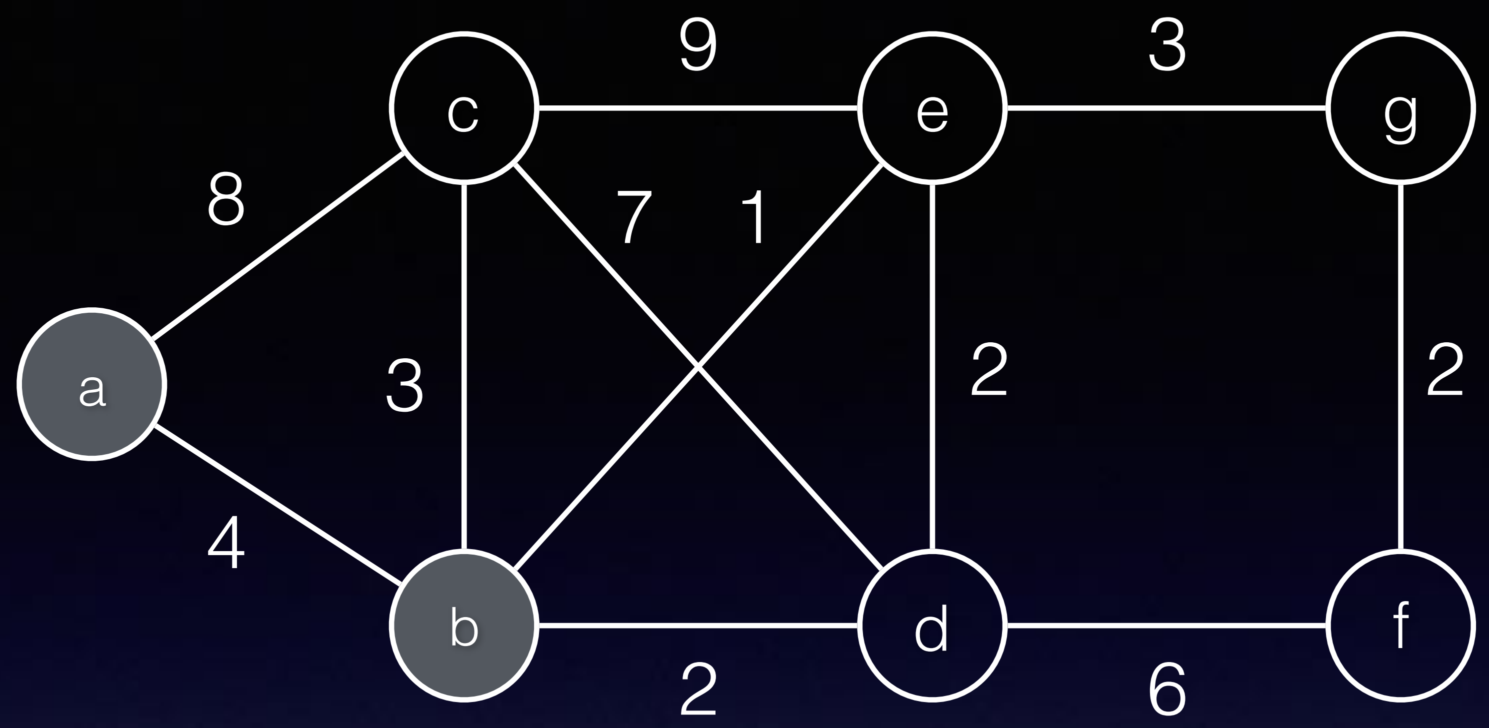
- compute distance from start to current and continue on to adjacent vertex
- if that path's distance is **shorter** than the adjacent vertex's current distance, **update** the adjacent vertex's **distance** and **predecessor**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	8	∞	∞	∞	∞
predecessor:	?	a	a	?	?	?	?

current vertex is **b** adjacent unvisited vertices are **c e d**

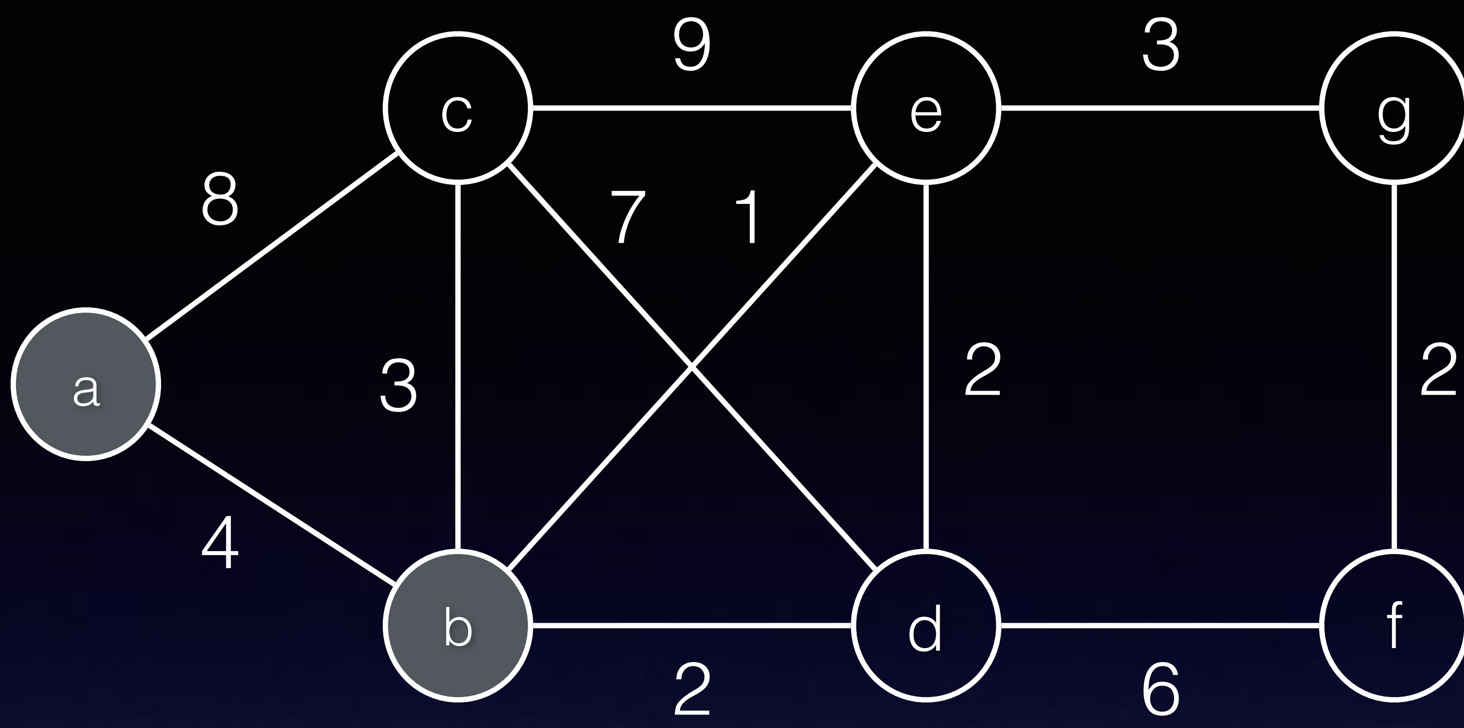
distance from **a** to **b** then to **c** is 7,
7 is less than 8 so update **d(a,c)** and change c's predecessor to current vertex **b**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	∞	∞	∞	∞
predecessor:	?	a	b	?	?	?	?

current vertex is **b** adjacent unvisited vertices are **c e d**

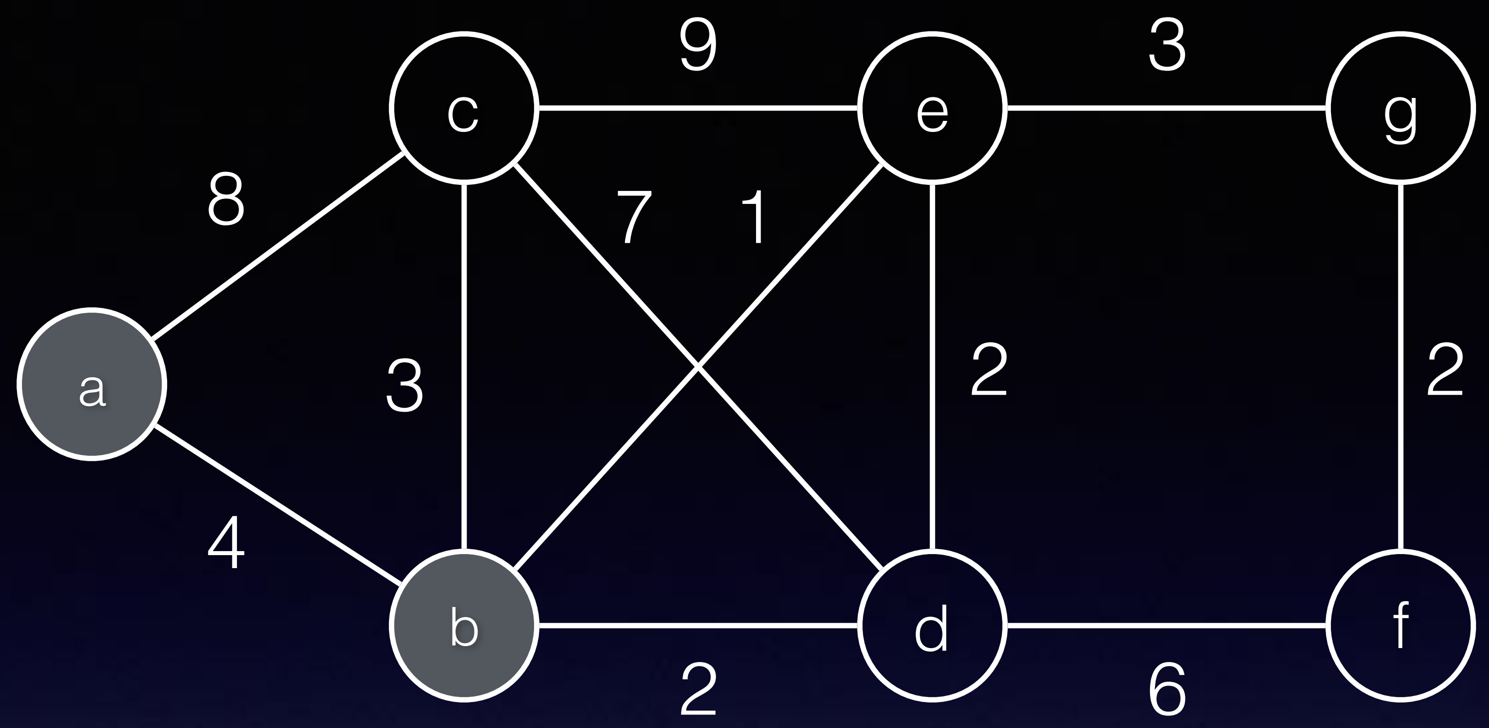
distance from **a** to **b** then to **e** is 5,
5 is less than ∞ so update **d(a,e)** and change **e**'s predecessor to current vertex **b**



vertex:	a	**b**	c	d	e	f	g
d(a,v):	0	4	7	∞	5	∞	∞
predecessor:	?	a	b	?	b	?	?

current vertex is **b** adjacent unvisited vertices are **c e d**

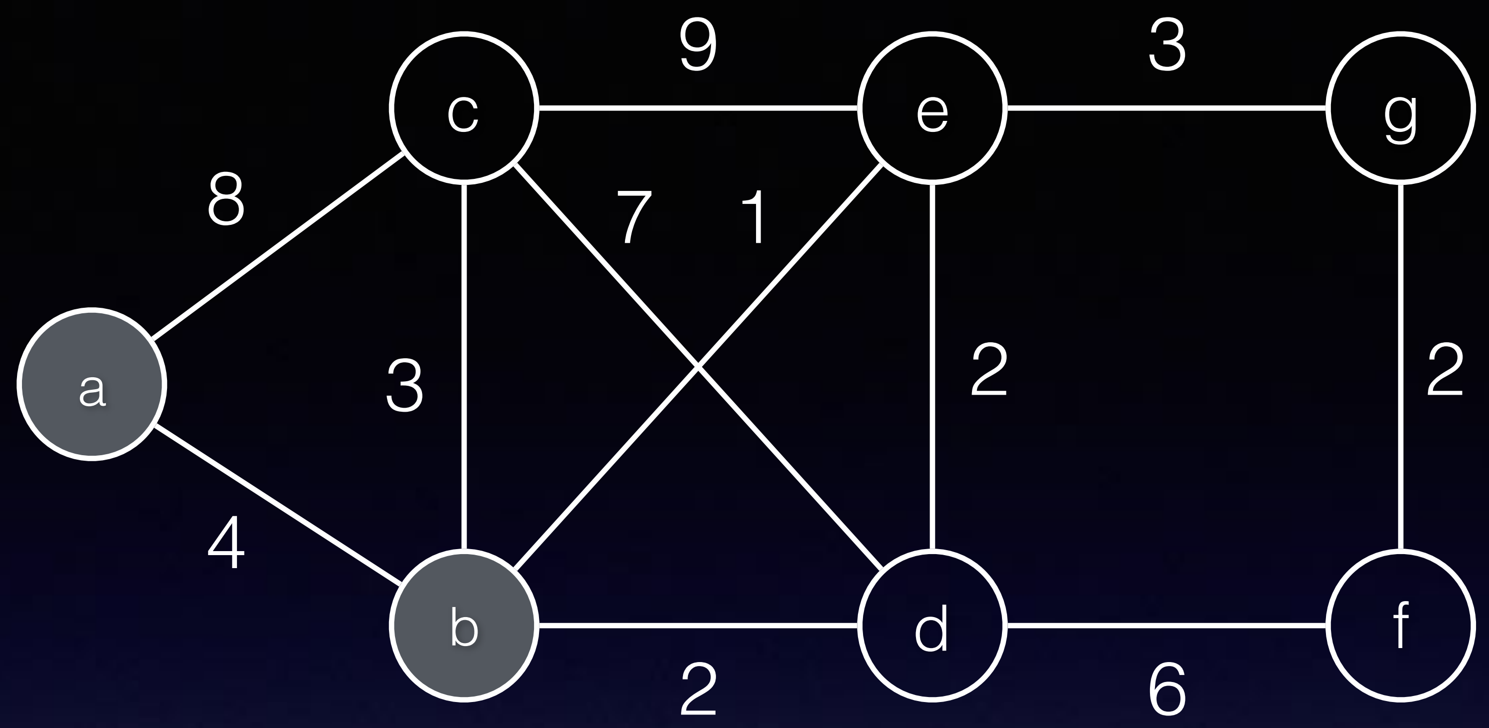
distance from **a** to **b** then to **d** is 6,
6 is less than ∞ so update **d(a,d)** and change **d**'s predecessor to current vertex **b**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	∞
predecessor:	?	a	b	b	b	?	?

current vertex is **b** adjacent unvisited vertices are **c e d**

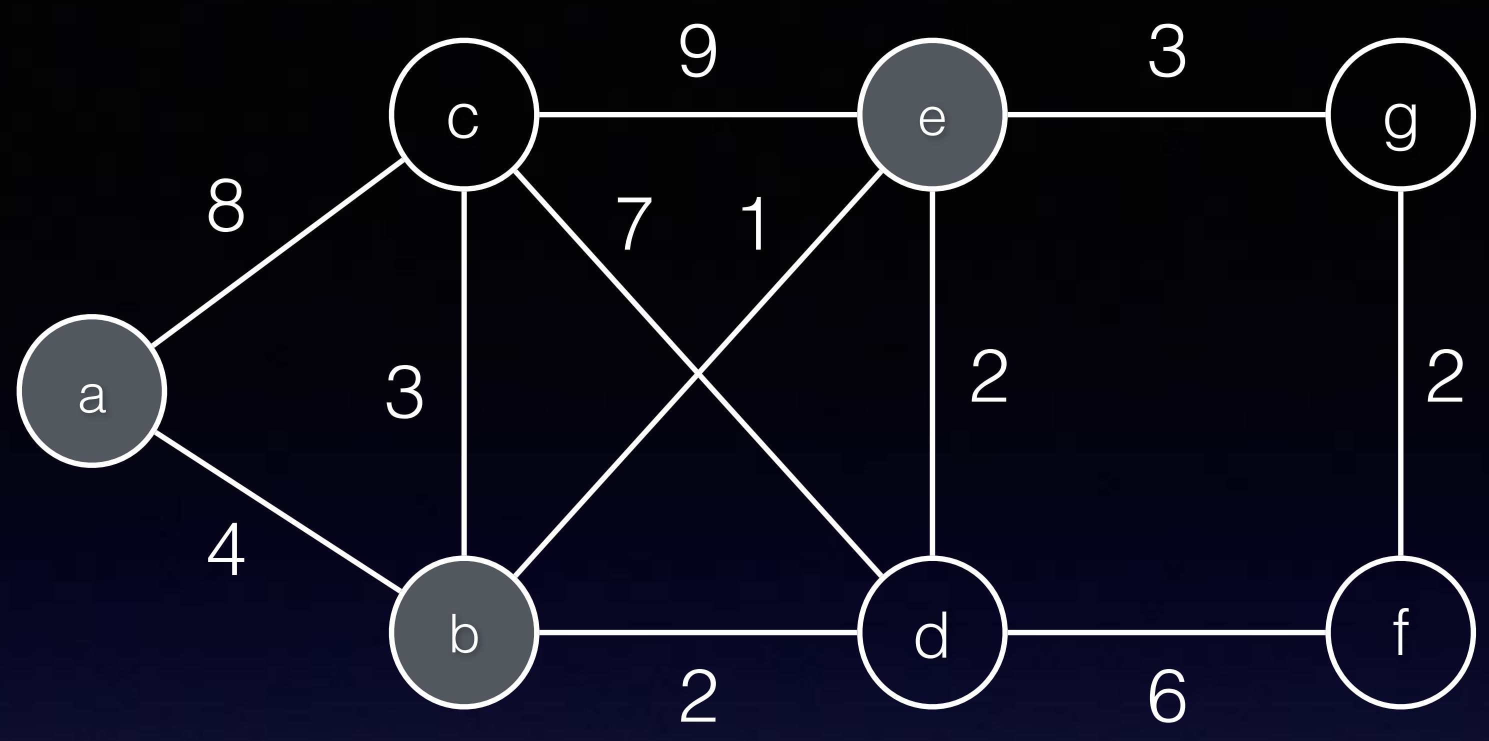
distance from **a** to **b** then to **d** is 6,
6 is less than ∞ so update **d(a,d)** and change **d**'s predecessor to current vertex **b**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	∞
predecessor:	?	a	b	b	b	?	?

dequeue unvisited vertex with shortest distance to **a**:

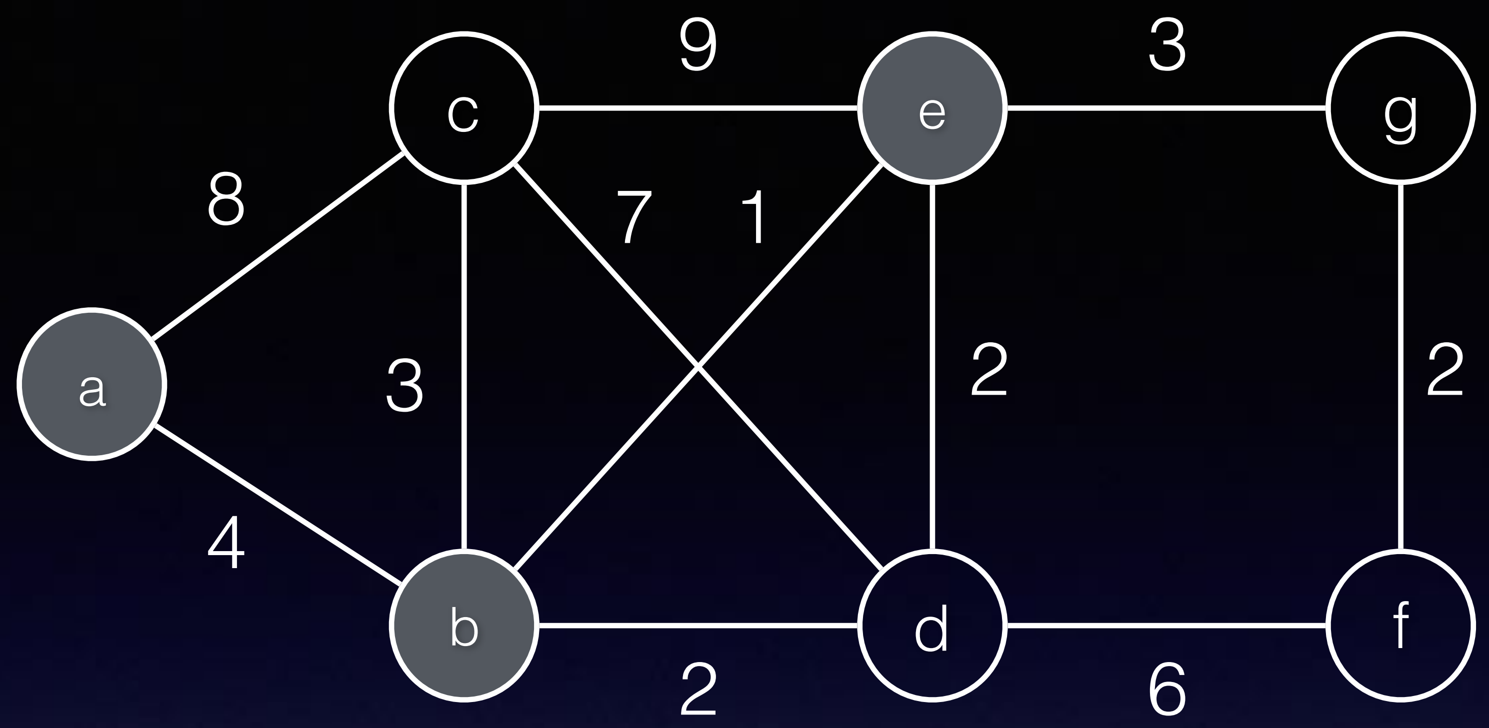
current vertex: ?



vertex:	a	b	c	d	**e**	f	g
d(a,v):	0	4	7	6	5	∞	∞
predecessor:	?	a	b	b	b	?	?

dequeue unvisited vertex with shortest distance to **a**:

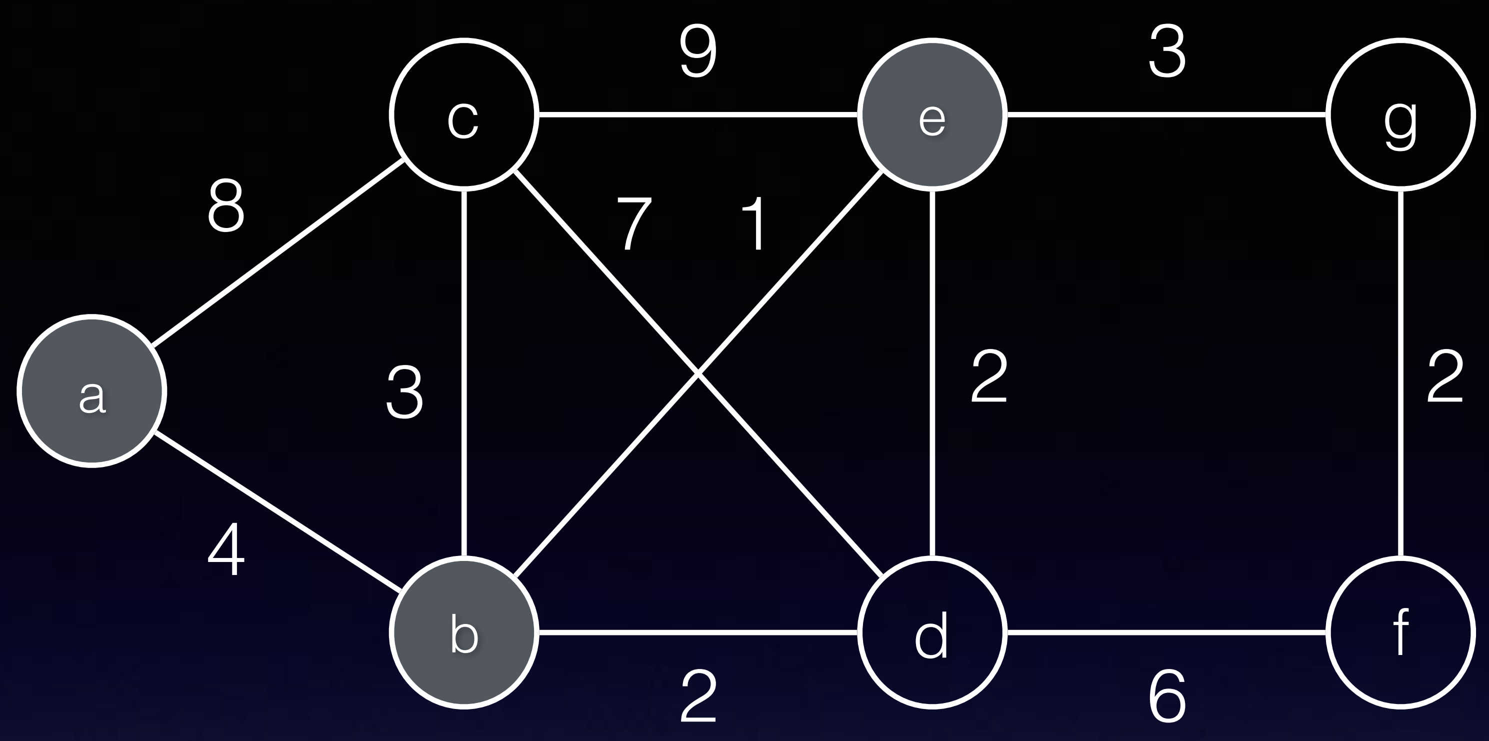
current vertex: **e**
adjacent unvisited vertices: ?



vertex:	a	b	c	d	**e**	f	g
d(a,v):	0	4	7	6	5	∞	∞
predecessor:	?	a	b	b	b	?	?

dequeue unvisited vertex with shortest distance to **a**:

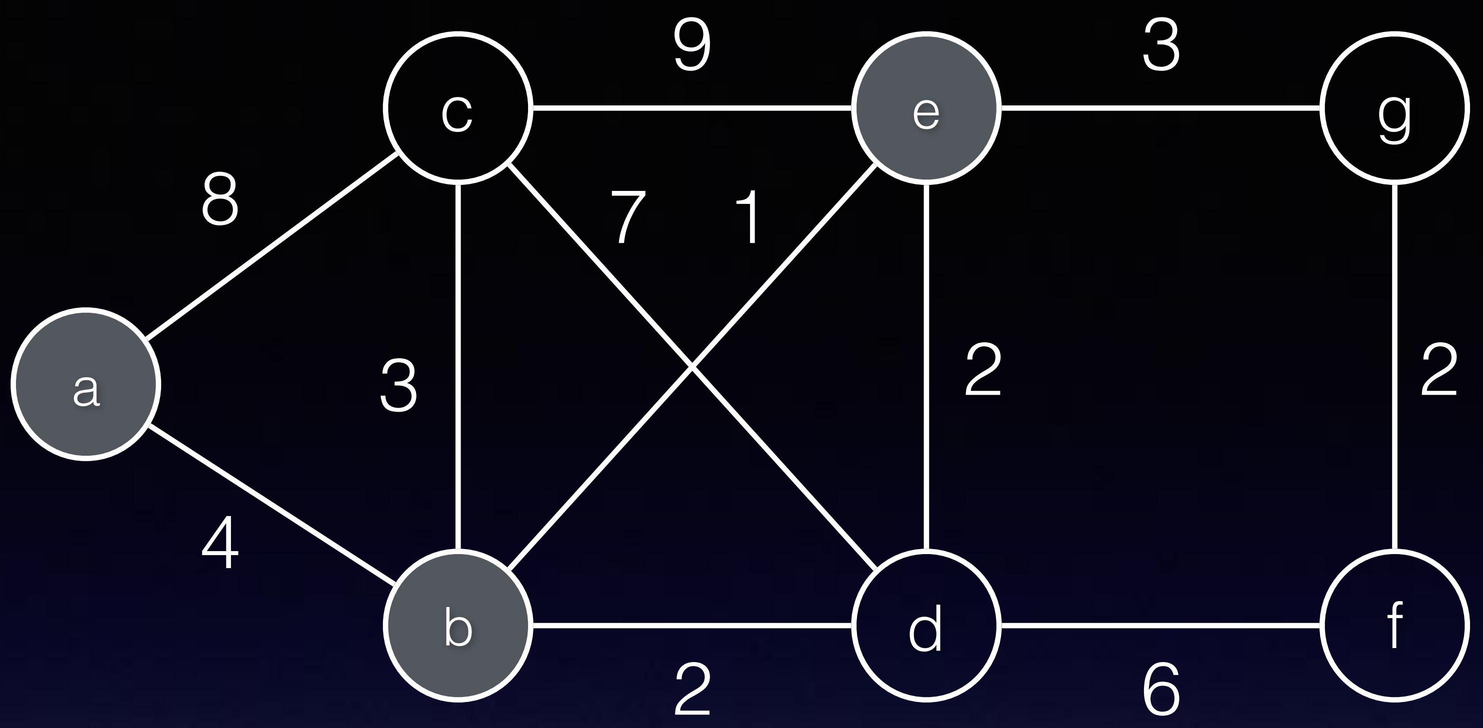
current vertex: **e**
adjacent unvisited vertices: **c d g**



vertex:	a	b	c	d	**e**	f	g
d(a,v):	0	4	7	6	5	∞	∞
predecessor:	?	a	b	b	b	?	?

for adjacent unvisited vertices **c d g**

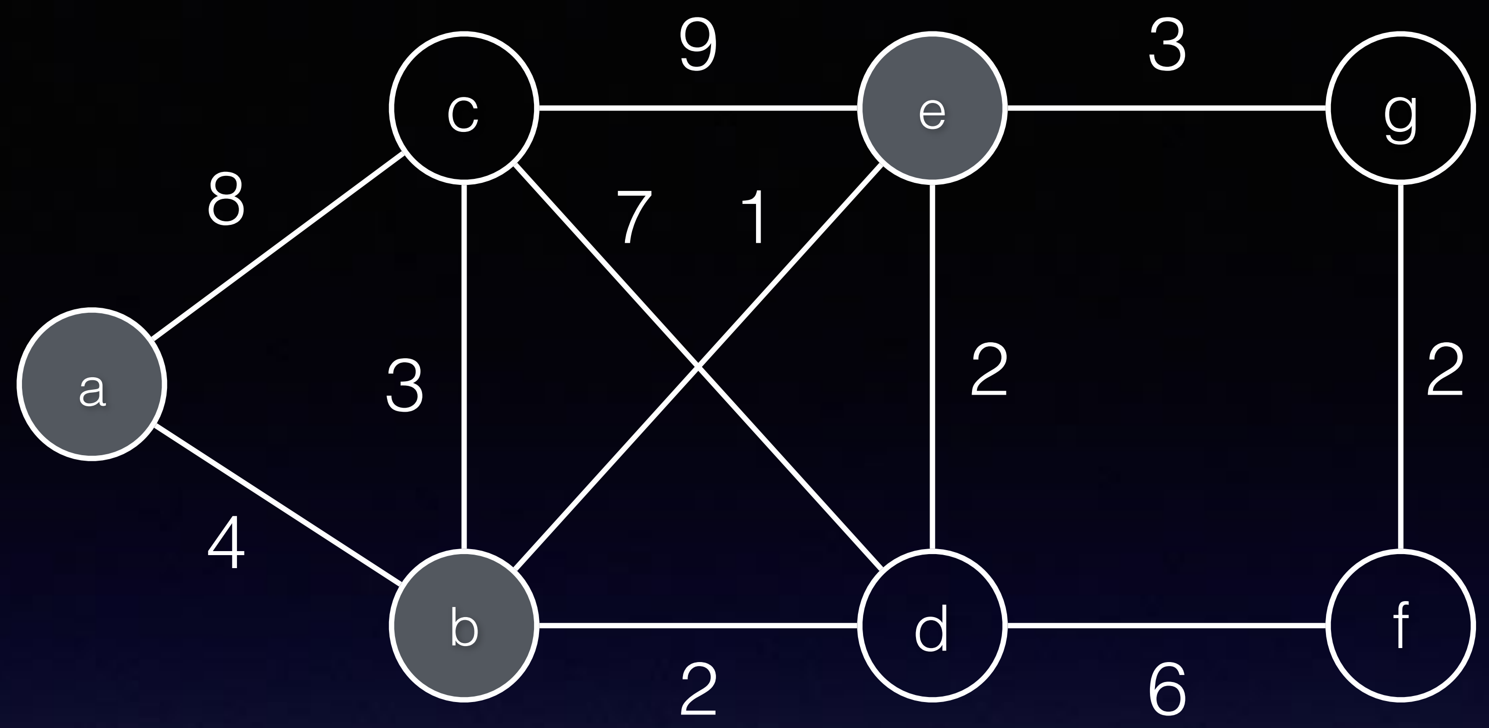
- compute distance from **a** to **e** and continuing on to the adjacent vertex
- if that path's distance is shorter than the adjacent vertex's current distance, update adjacent vertex's distance and predecessor



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	∞
predecessor:	?	a	b	b	b	?	?

current vertex is **e** adjacent vertices are **c d g**

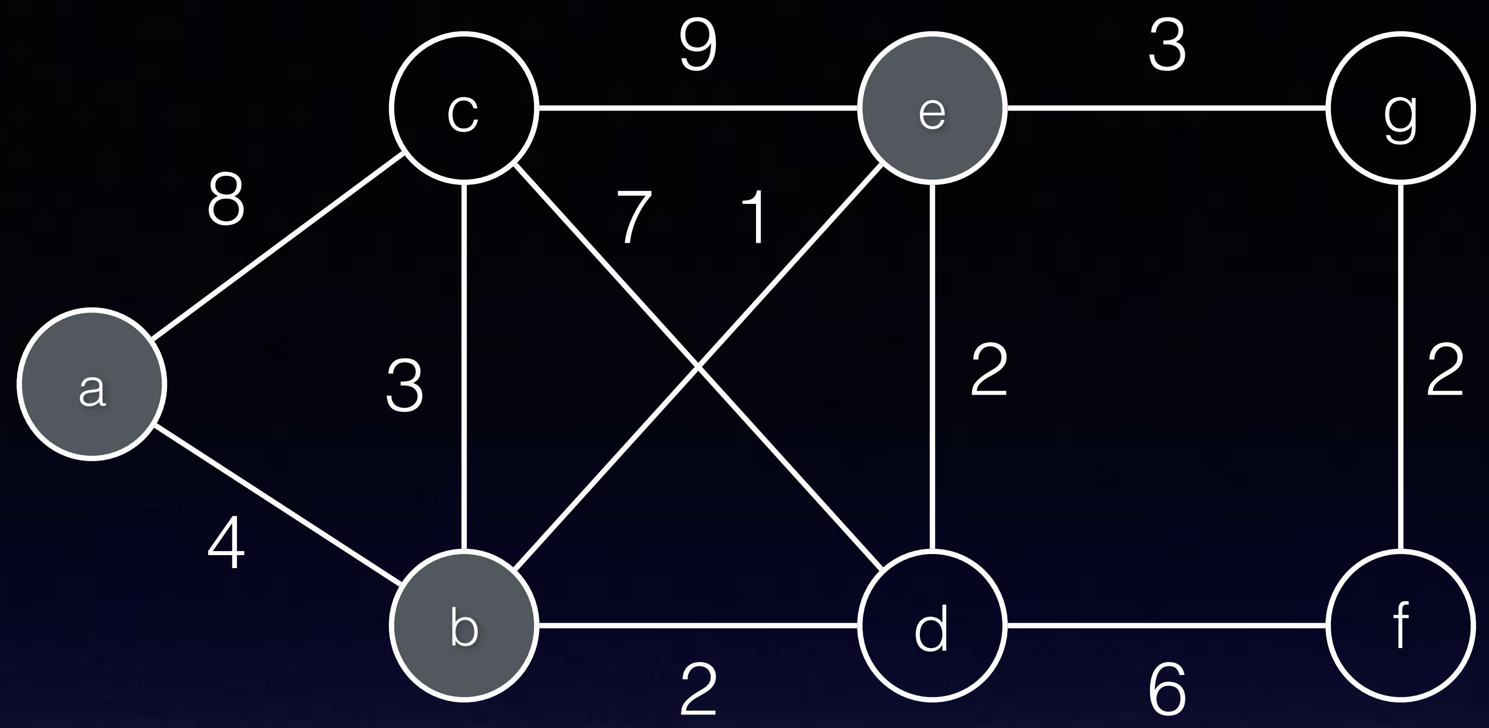
distance from **a** to **e** then to **c** is 5 + 9 = 14,
14 is not less than 7 so do nothing



vertex:	a	b	c	d	**e**	f	g
d(a,v):	0	4	7	6	5	∞	∞
predecessor:	?	a	b	b	b	?	?

current vertex is **e** adjacent vertices are **c d g**

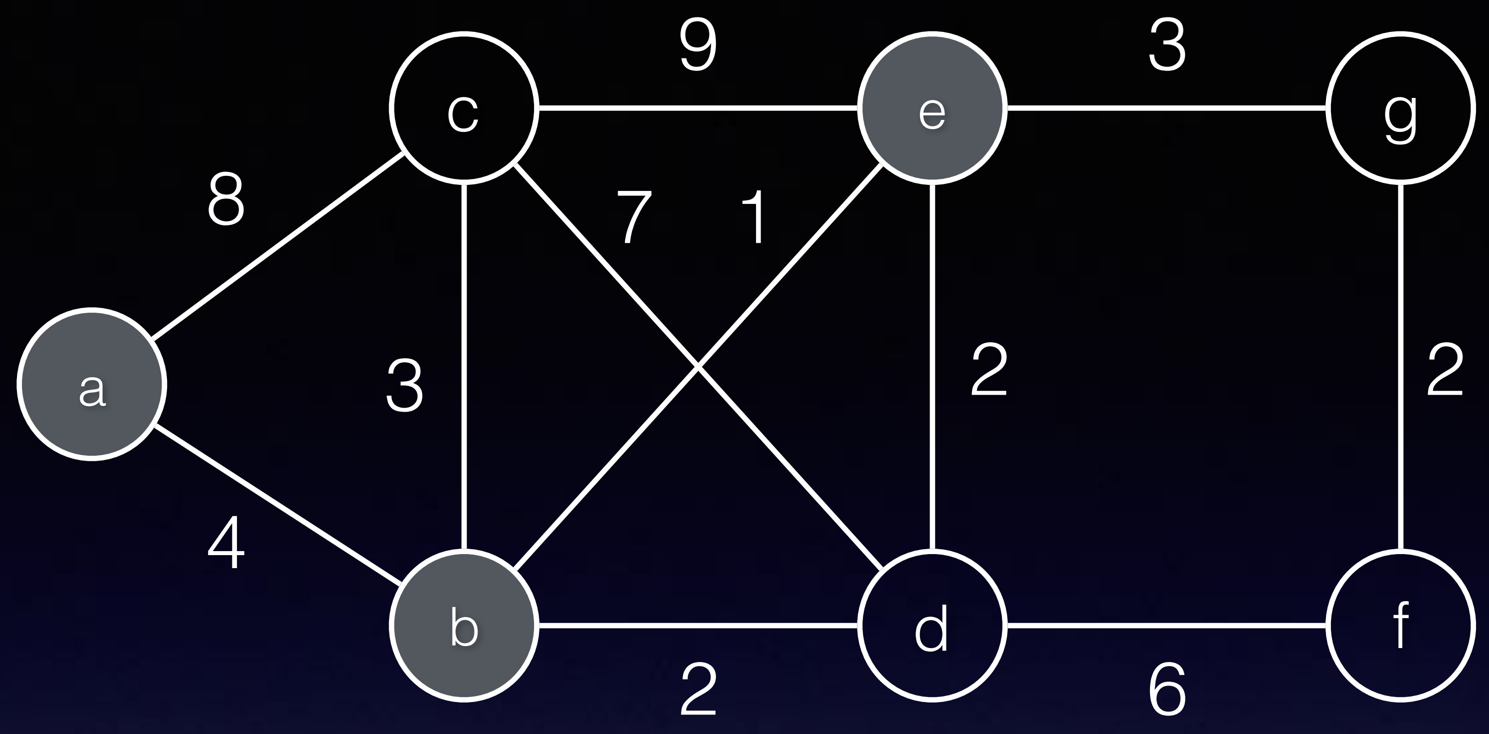
distance from **a** to **e** then to **d** is 5 + 2 = 7,
7 is not less than 6 so do nothing



vertex:	a	b	c	d	**e**	f	g
d(a,v):	0	4	7	6	5	∞	∞
predecessor:	?	a	b	b	b	?	?

current vertex is **e** adjacent vertices are **c d g**

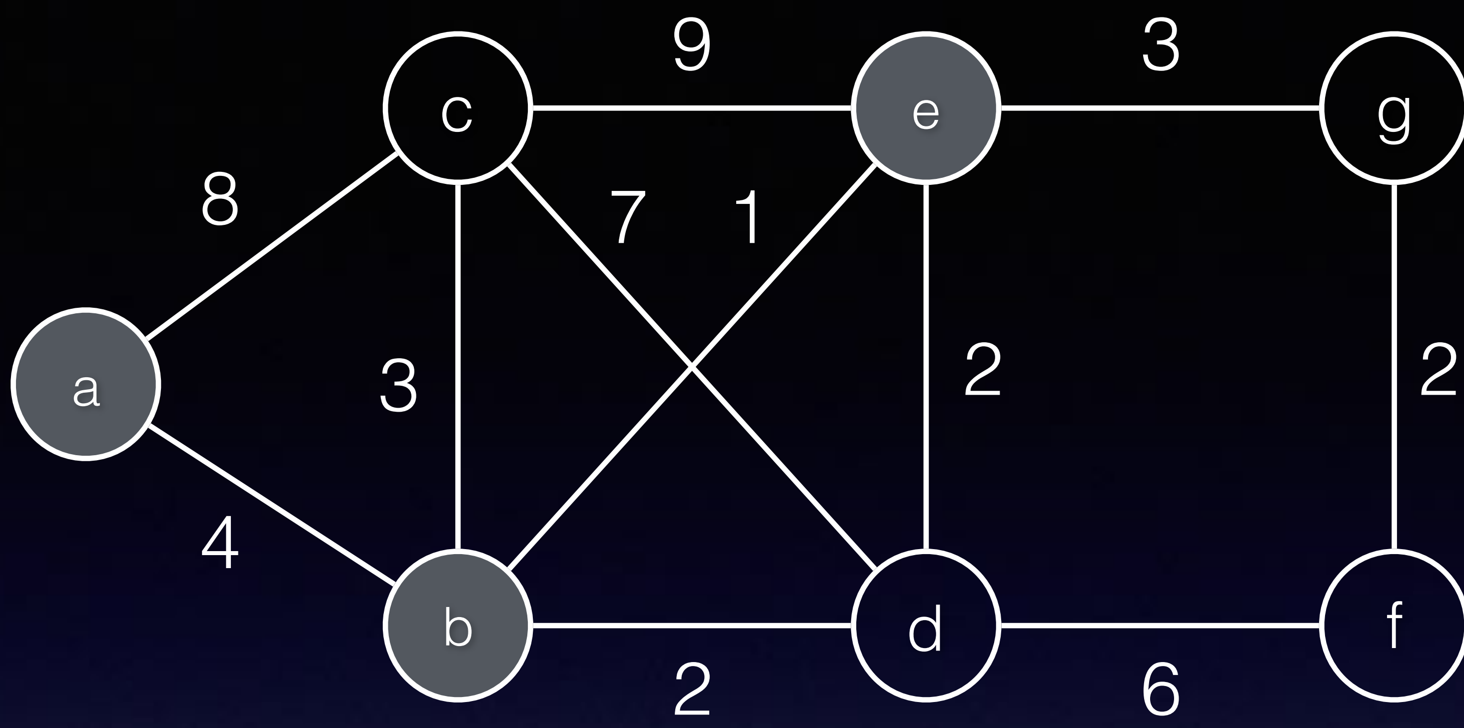
distance from **a** to **e** then to **g** is 5 + 3 = 8,
8 is less than ∞ so set **d(a,g)** = 8 and predecessor to **e**



vertex:	a	b	c	d	**e**	f	g
d(a,v):	0	4	7	6	5	∞	8
predecessor:	?	a	b	b	b	?	e

current vertex is **e** adjacent vertices are **c d g**

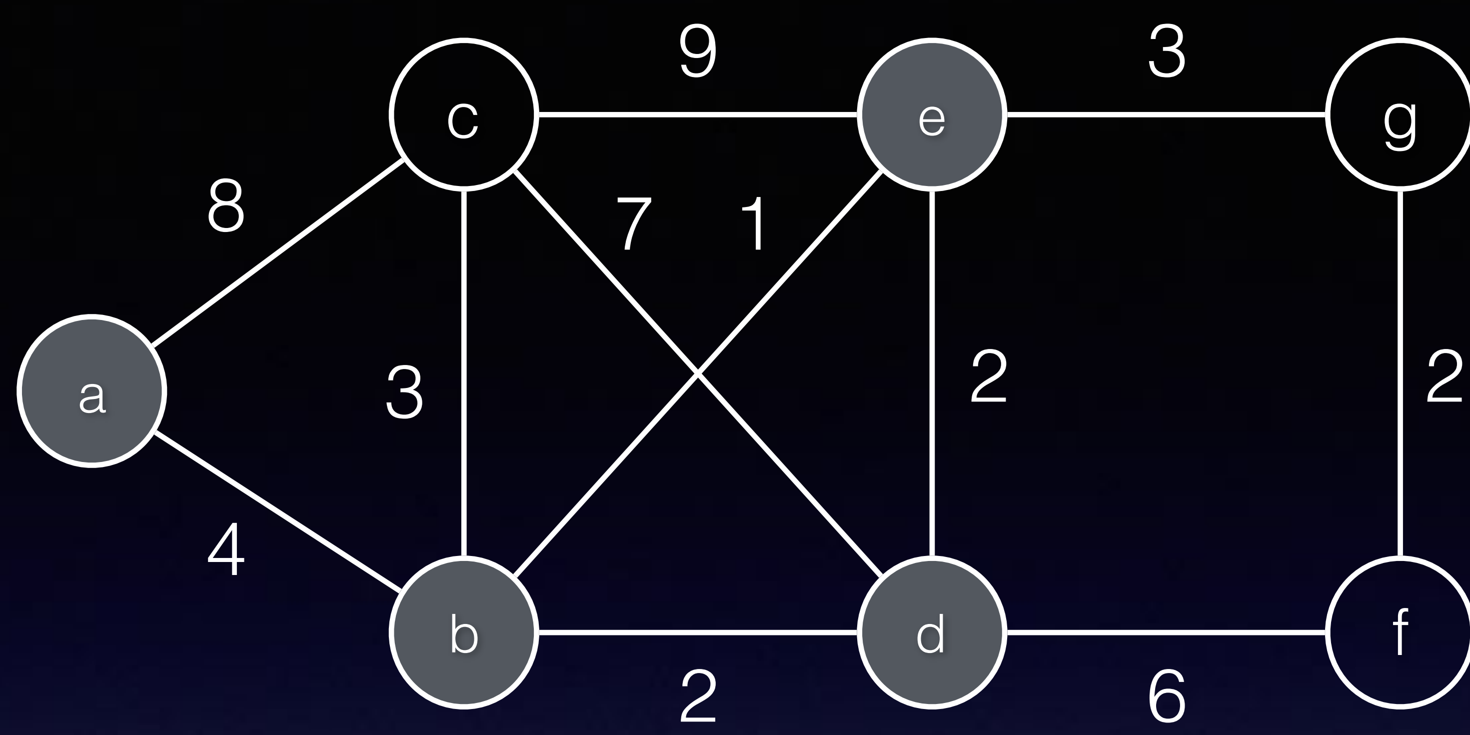
distance from **a** to **e** then to **g** is 5 + 3 = 8,
8 is less than ∞ so set **d(a,g)** = 8 and predecessor to **e**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	8
predecessor:	?	a	b	b	b	?	e

dequeue unvisited vertex with shortest distance to **a**

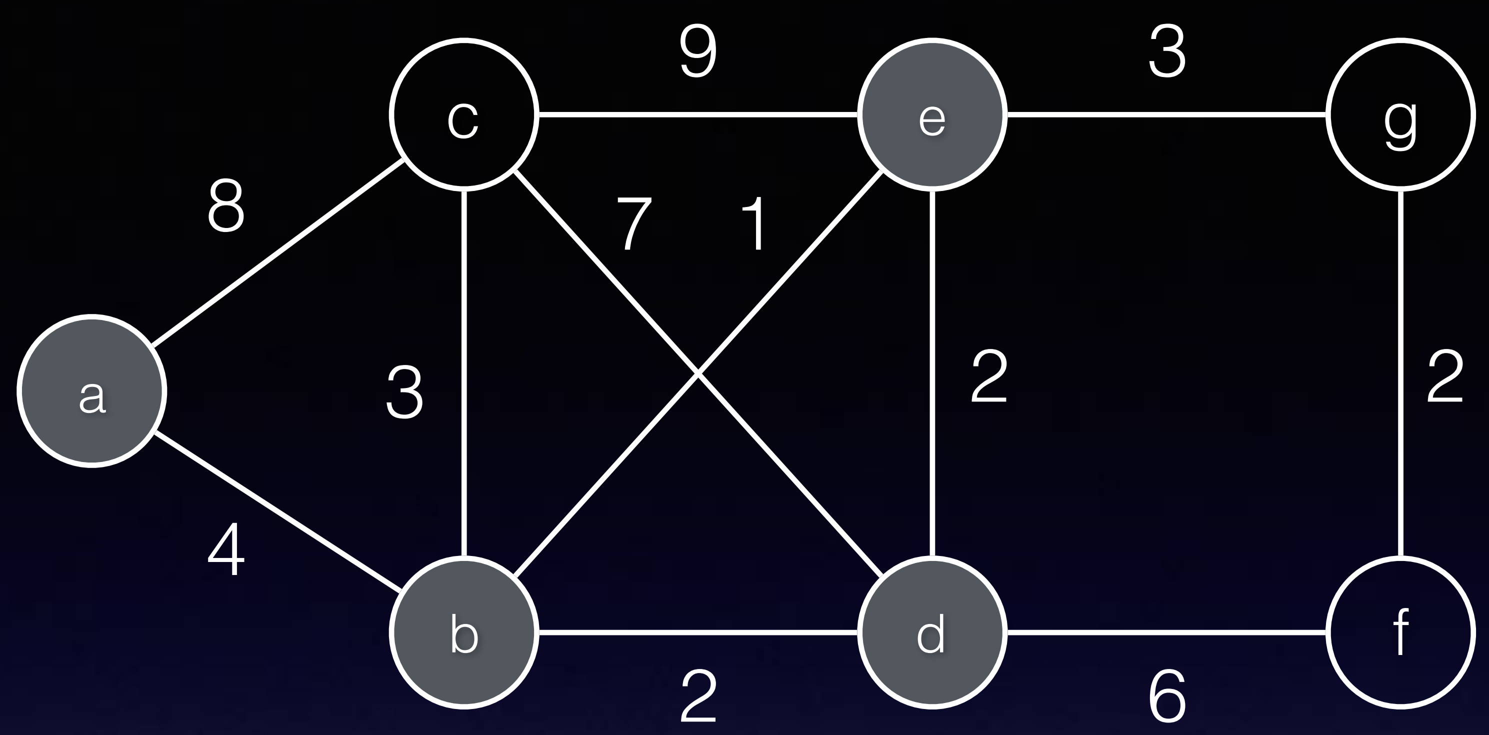
current vertex is ?
adjacent unvisited vertices are ?



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	8
predecessor:	?	a	b	b	b	?	e

dequeue unvisited vertex with shortest distance to **a**

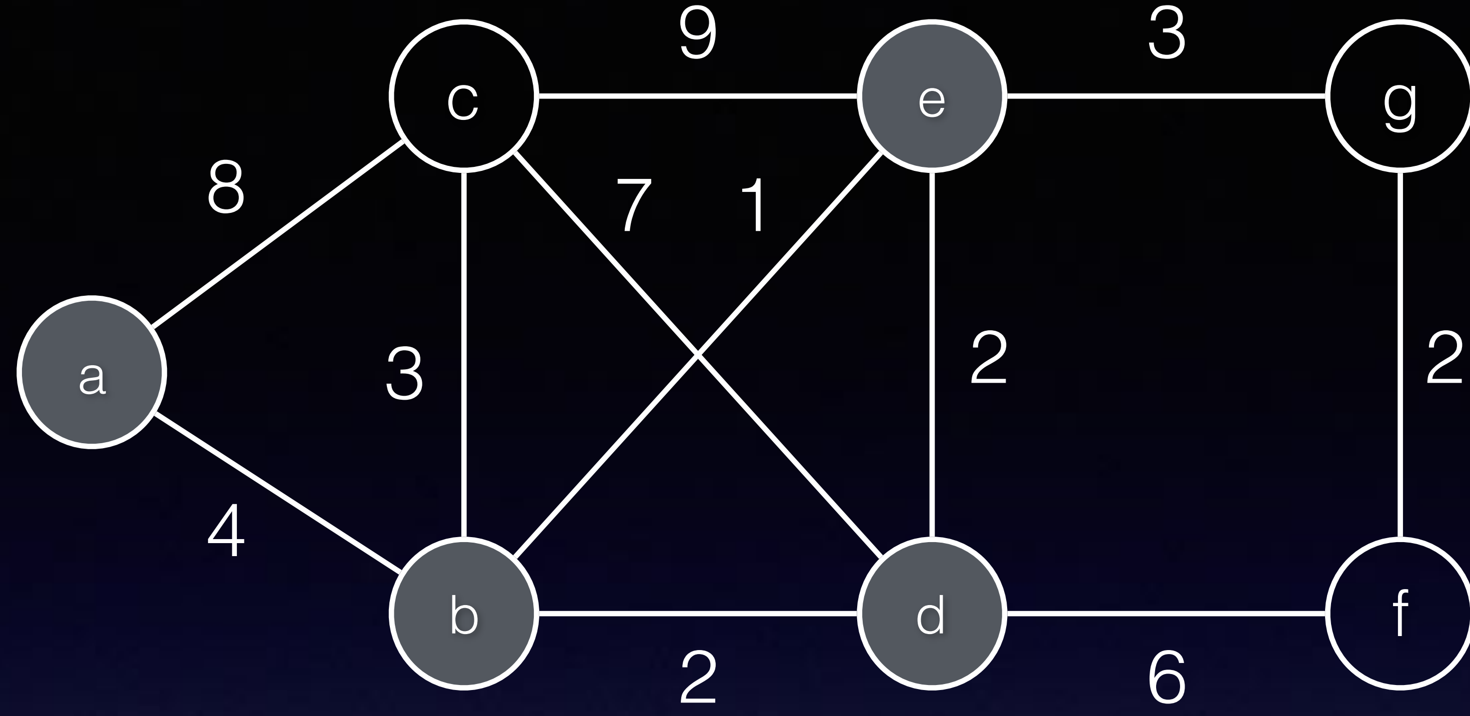
current vertex is **d**
adjacent unvisited vertices are ?



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	8
predecessor:	?	a	b	b	b	?	e

dequeue unvisited vertex with shortest distance to **a**

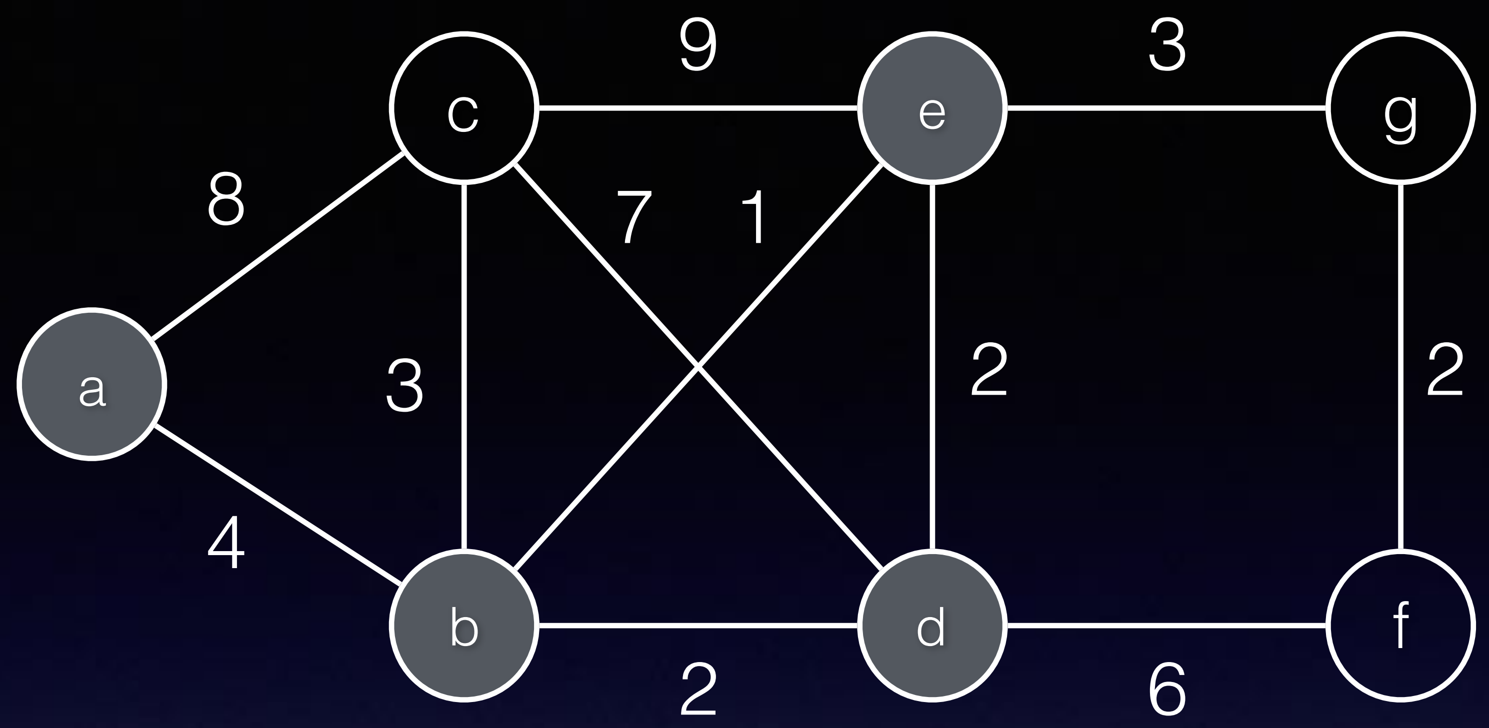
current vertex is **d**
adjacent unvisited vertices are **c f**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	8
predecessor:	?	a	b	b	b	?	e

for adjacent unvisited vertices **c f**

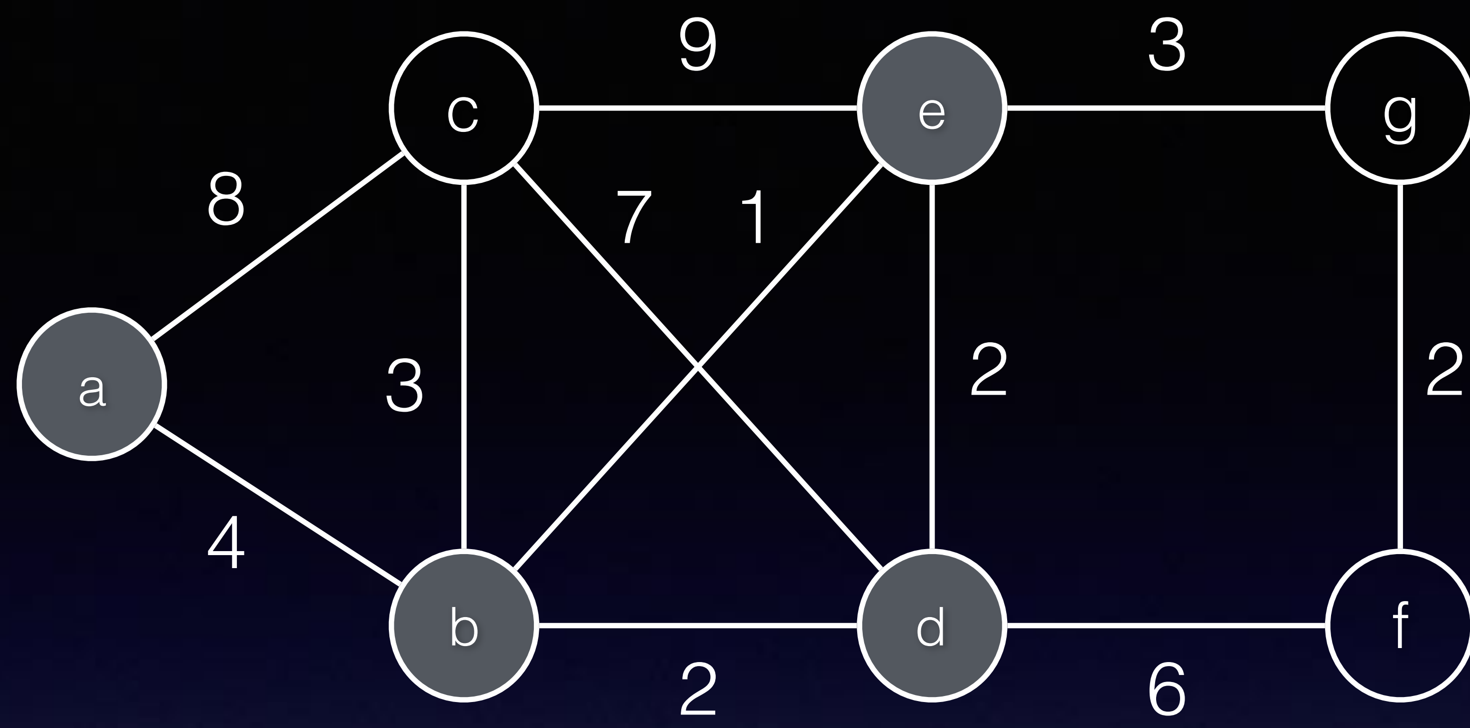
- compute distance from **a** to **d** and continuing on to adjacent vertex
- if that path's distance is shorter than the adjacent vertex's current distance, update adjacent vertex's distance and predecessor



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	8
predecessor:	?	a	b	b	b	?	e

current vertex is **d** adjacent unvisited vertices are **c f**

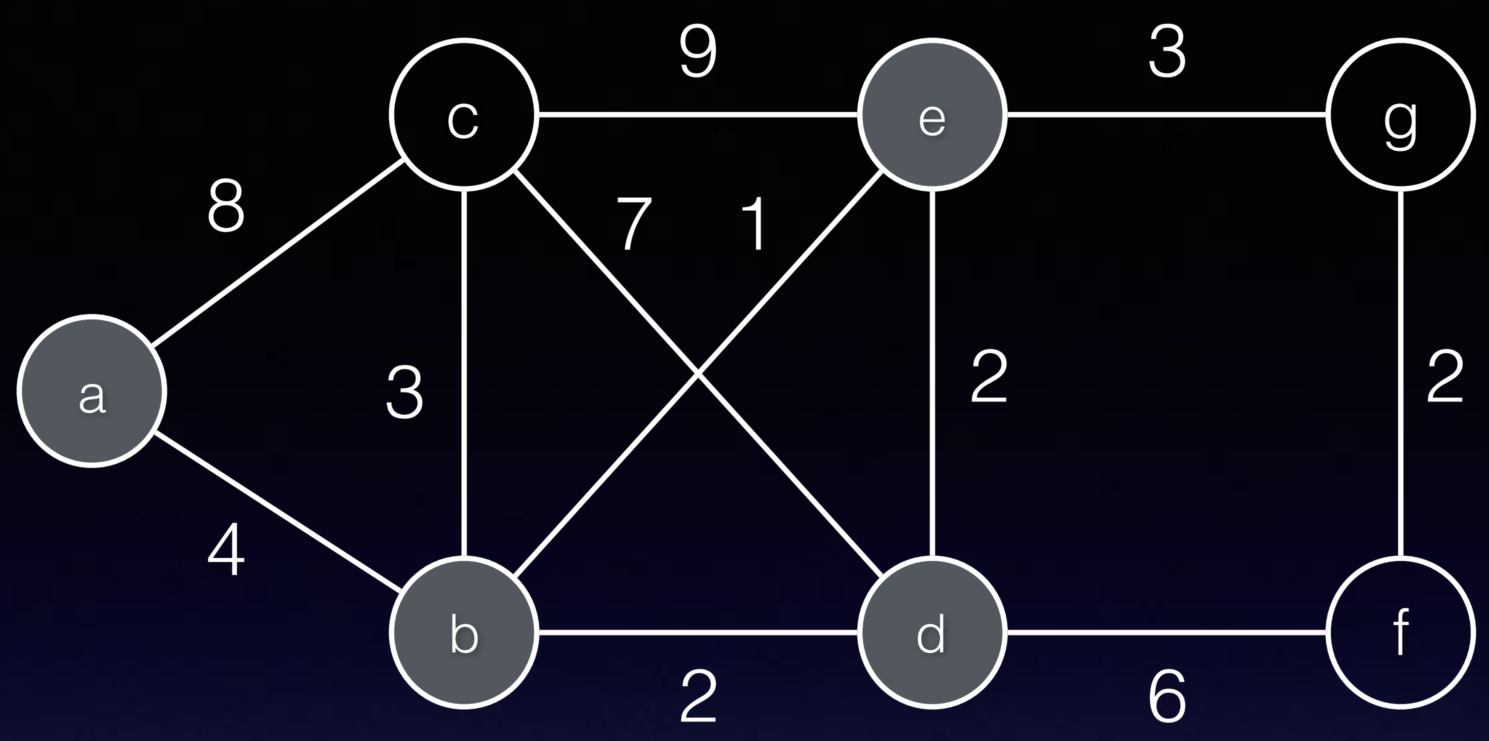
distance from **a** to **d** then to **c** is 6 + 7 = 13,
13 is not less than 7 so do nothing



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	∞	8
predecessor:	?	a	b	b	b	?	e

current vertex is **d** adjacent unvisited vertices are **c f**

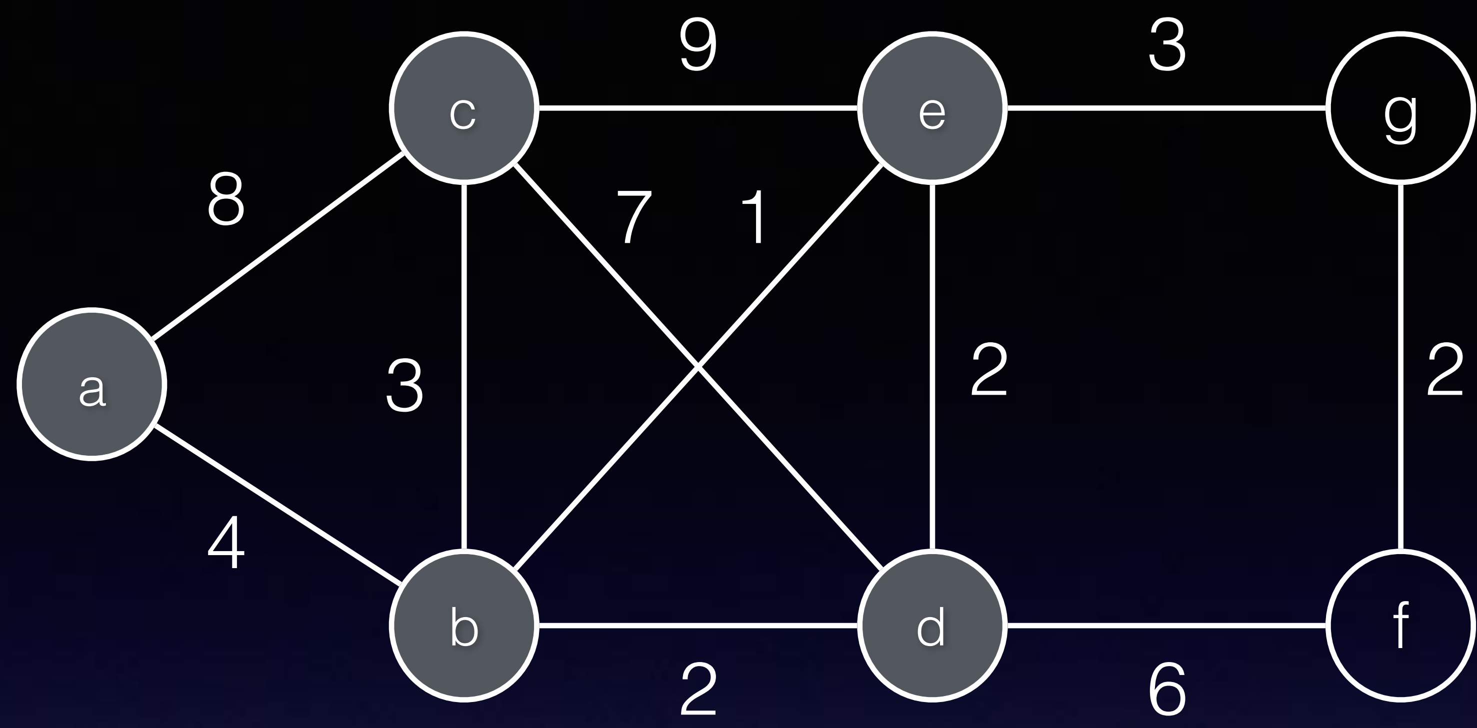
distance from **c** to **d** then to **f** is 6 + 6 = 12,
12 is less than ∞ so set **d(a,g)**=12 and predecessor to **d**



vertex:	a	b	c	**d**	e	f	g
d(a,v):	0	4	7	6	5	12	8
predecessor:	?	a	b	b	b	d	e

dequeue unvisited vertex with shortest distance to **a**:

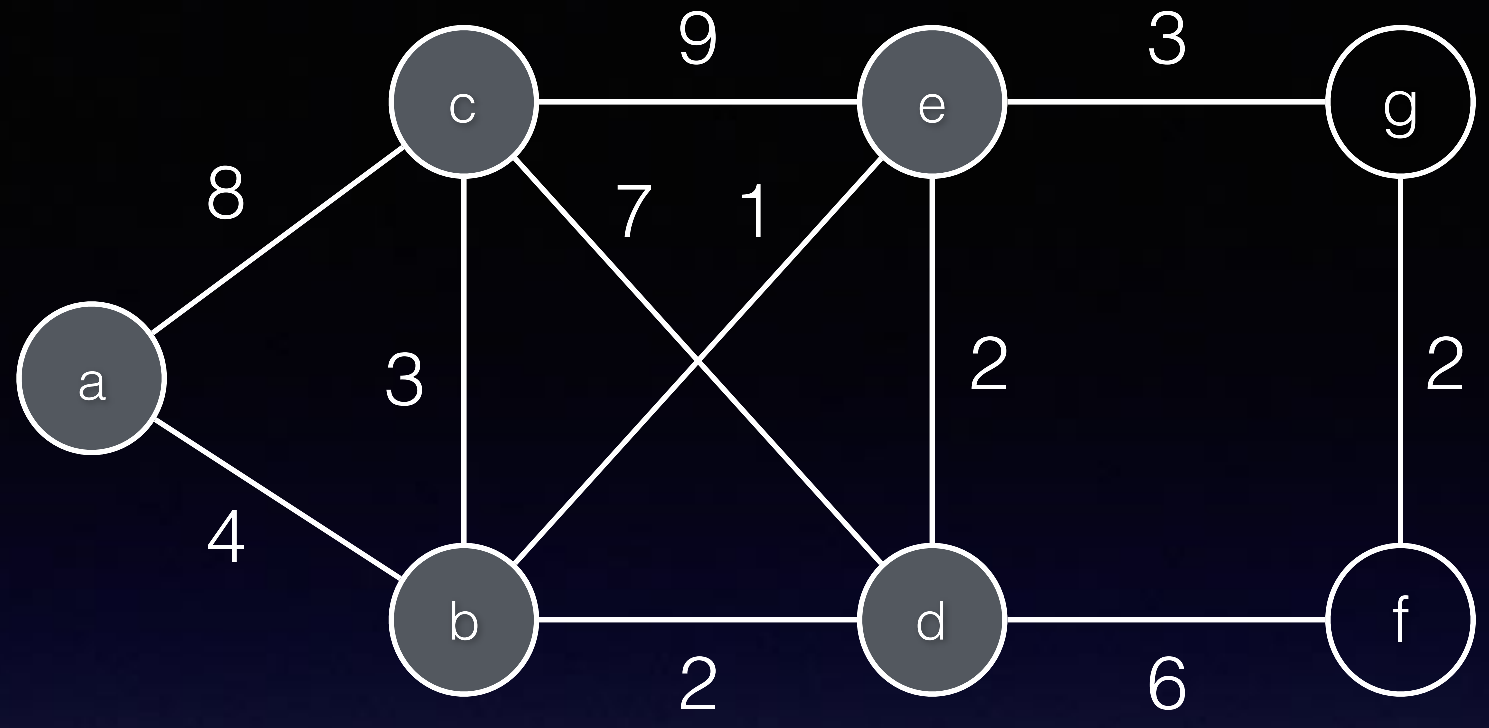
current vertex: ?
adjacent unvisited vertices: ?



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	12	8
predecessor:	?	a	b	b	b	d	e

dequeue unvisited vertex with shortest distance to **a**:

current vertex: **c**
adjacent unvisited vertices: ?

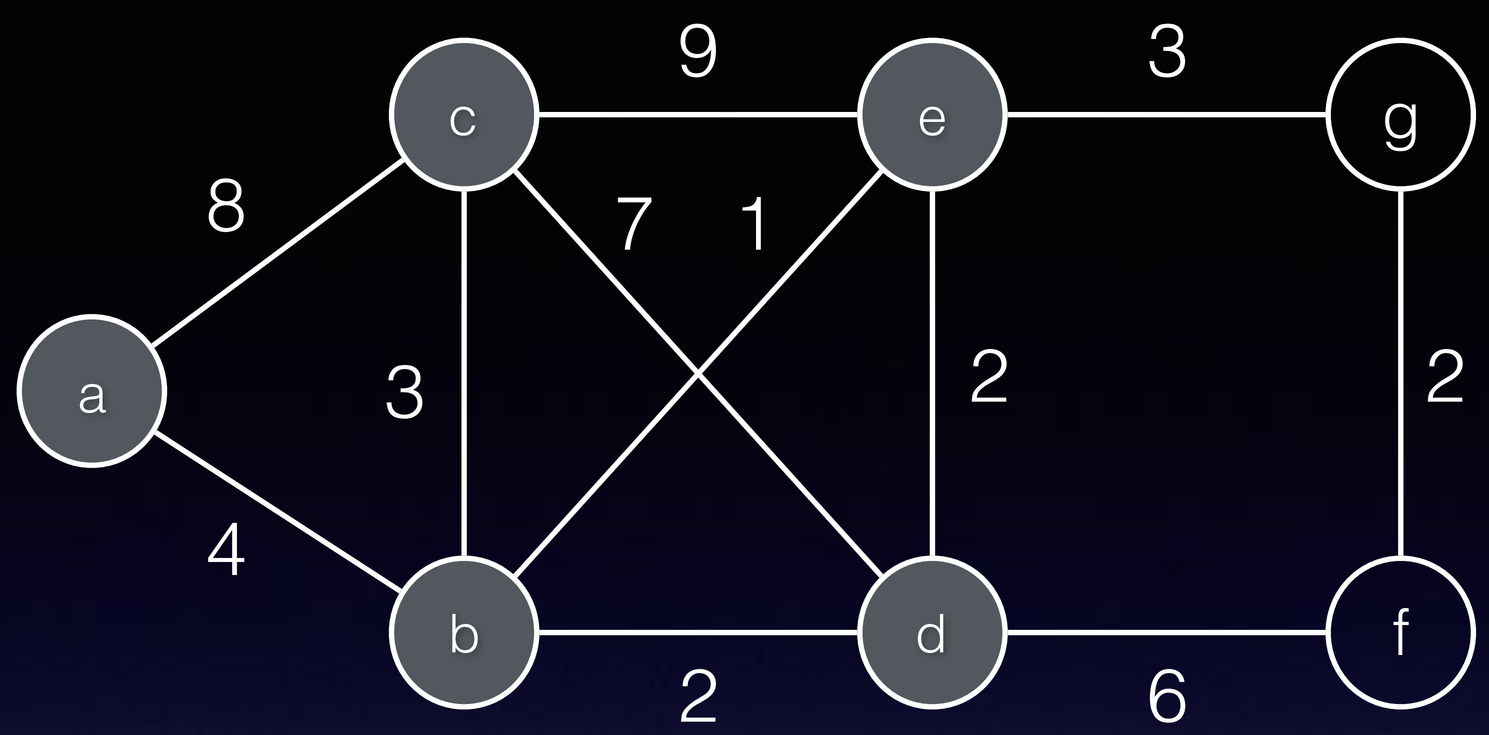


vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	12	8
predecessor:	?	a	b	b	b	d	e

dequeue unvisited vertex with shortest distance to **a**:

current vertex: **c**

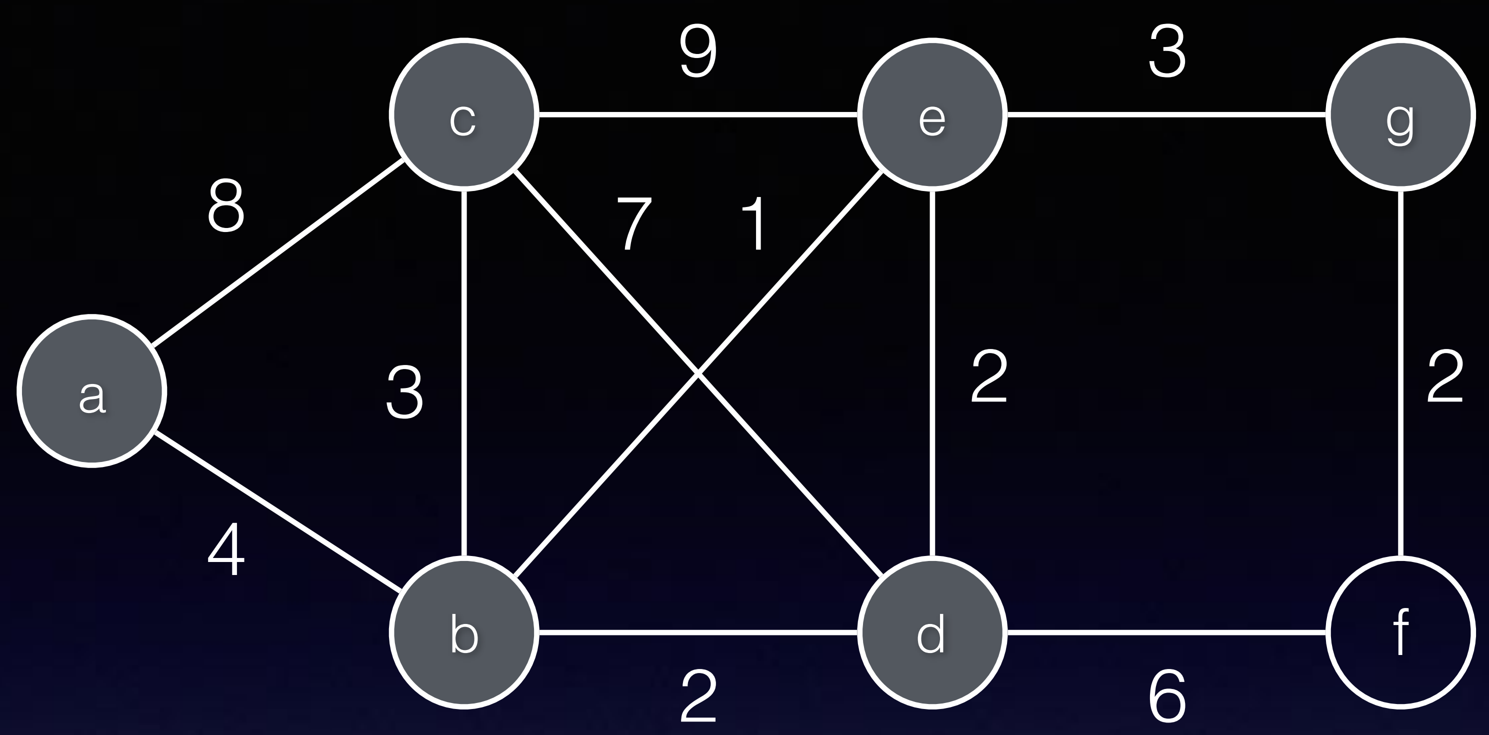
adjacent unvisited vertices: none, so nothing to do here



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	12	8
predecessor:	?	a	b	b	b	d	e

dequeue unvisited vertex with shortest distance to **a**:

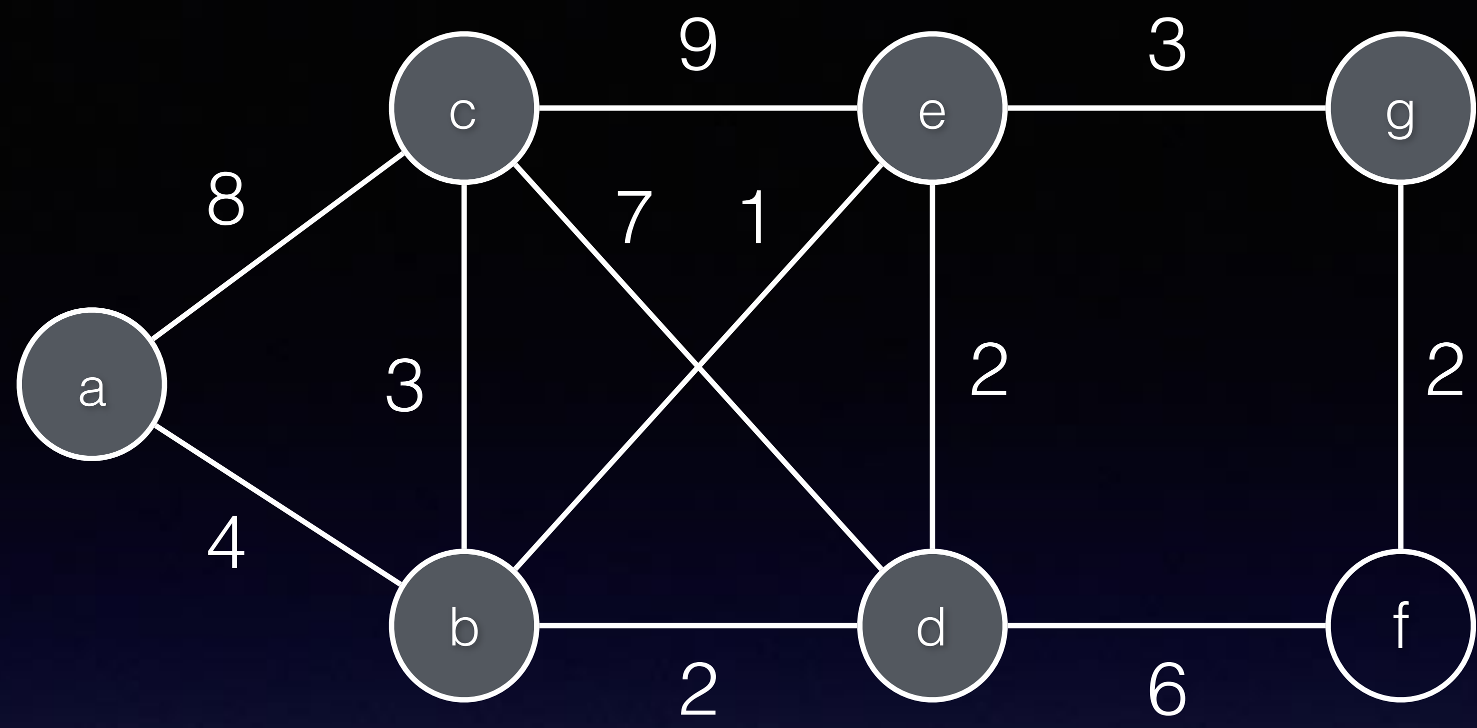
current vertex: ?
adjacent unvisited vertices: ?



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	12	8
predecessor:	?	a	b	b	b	d	e

dequeue unvisited vertex with shortest distance to **a**:

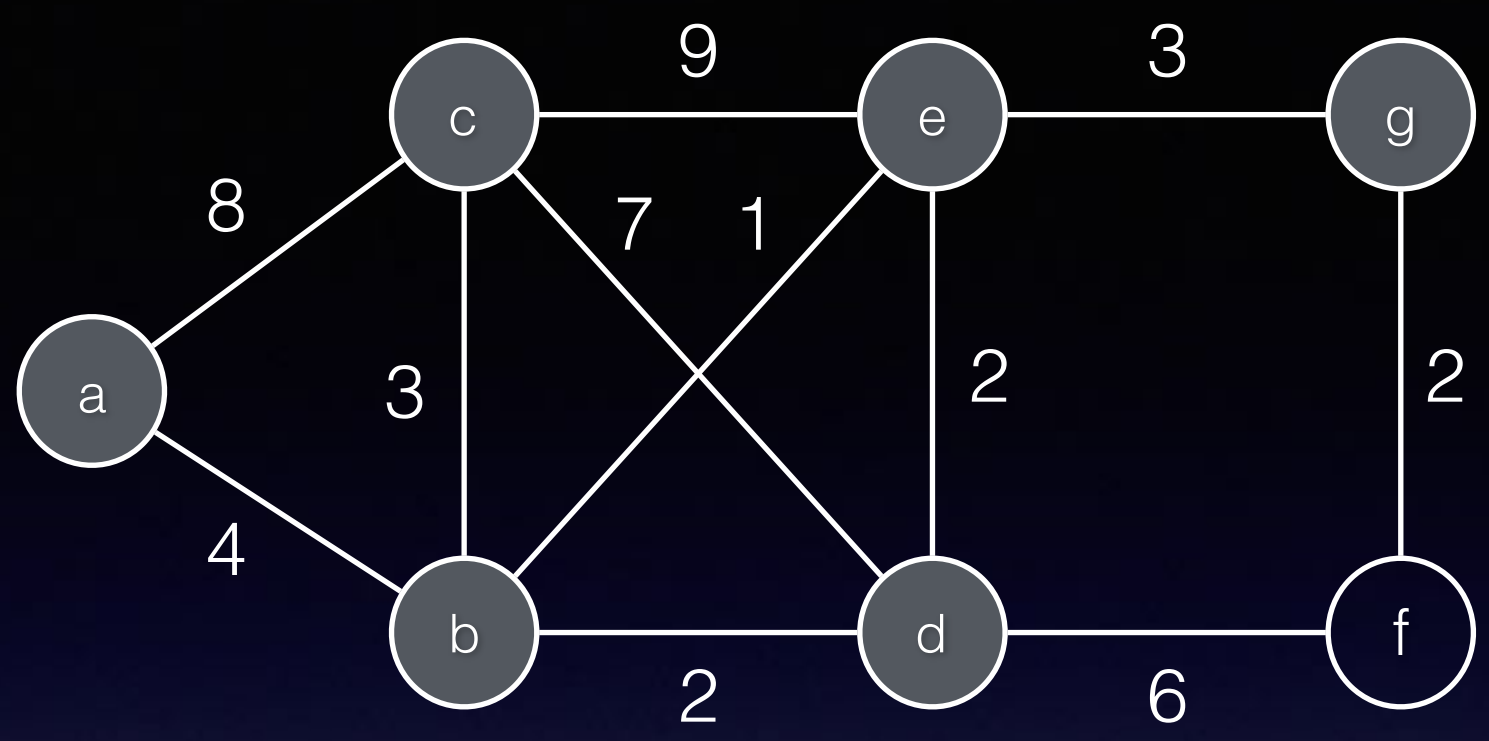
current vertex: **g**
adjacent unvisited vertices: ?



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	12	8
predecessor:	?	a	b	b	b	d	e

dequeue unvisited vertex with shortest distance to **a**:

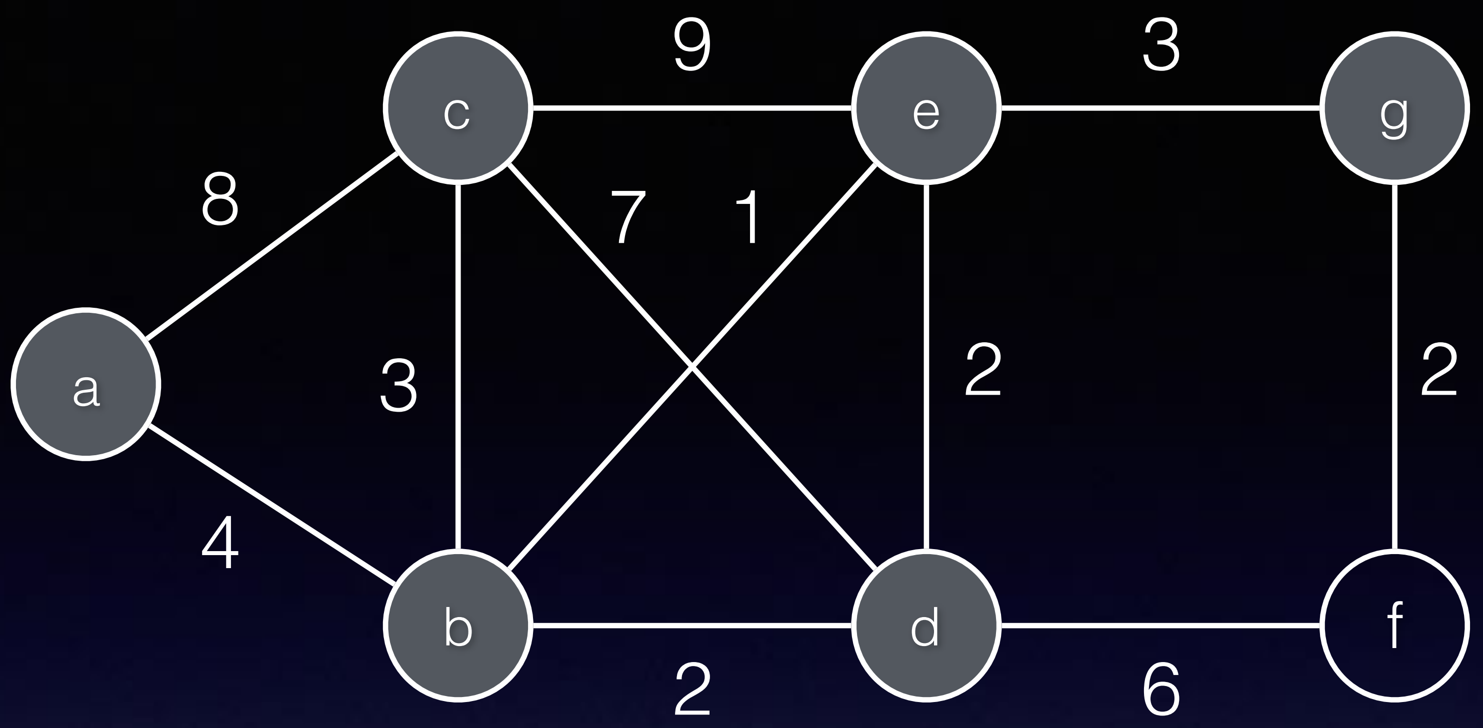
current vertex: **g**
adjacent unvisited vertices: **f**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	12	8
predecessor:	?	a	b	b	b	d	e

current vertex is **g** adjacent vertex is **f**

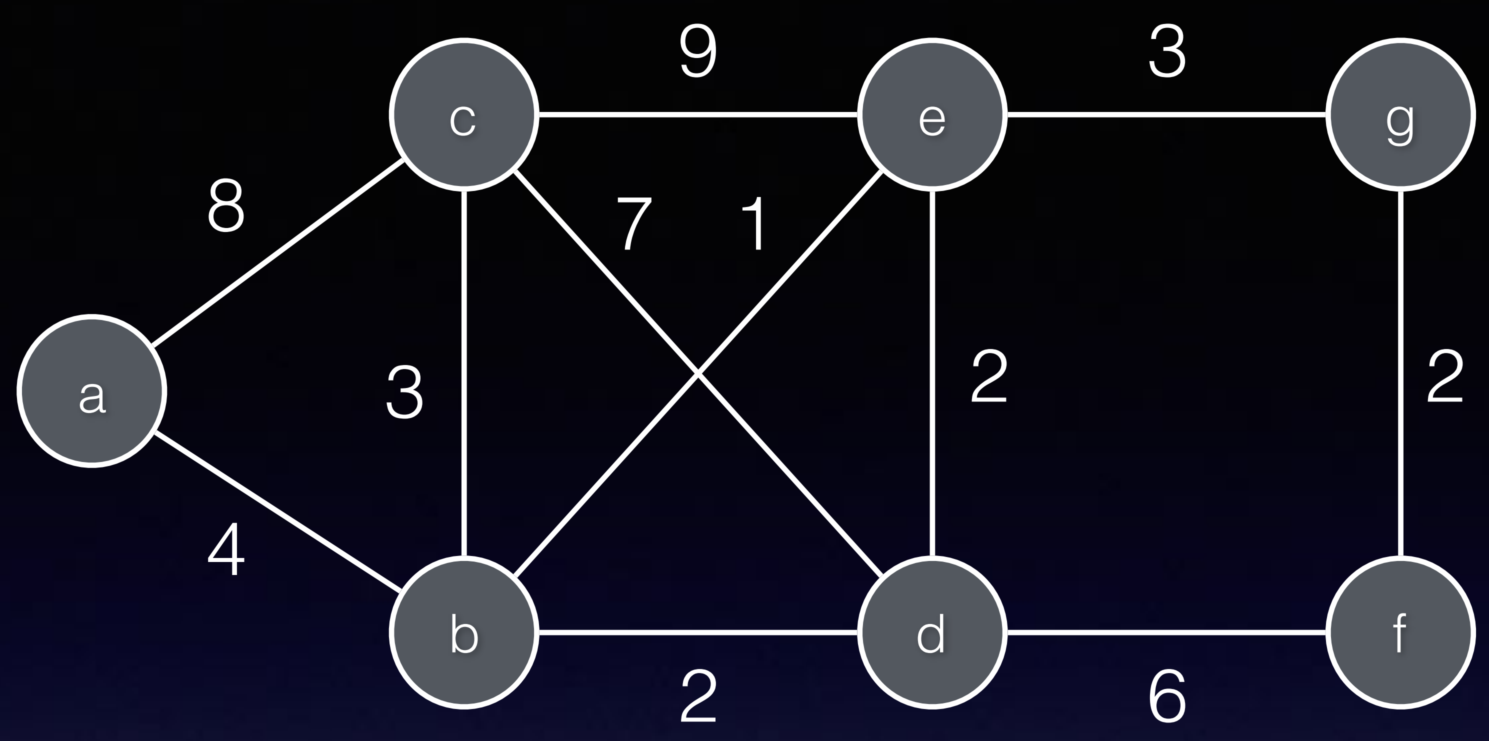
distance from **a** to **g** then to **f** is 8 + 2 = 10,
10 is less than 12 so set **d(a,f)** =10 and predecessor to **g**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	10	8
predecessor:	?	a	b	b	b	g	e

dequeue unvisited vertex with shortest distance to **a**:

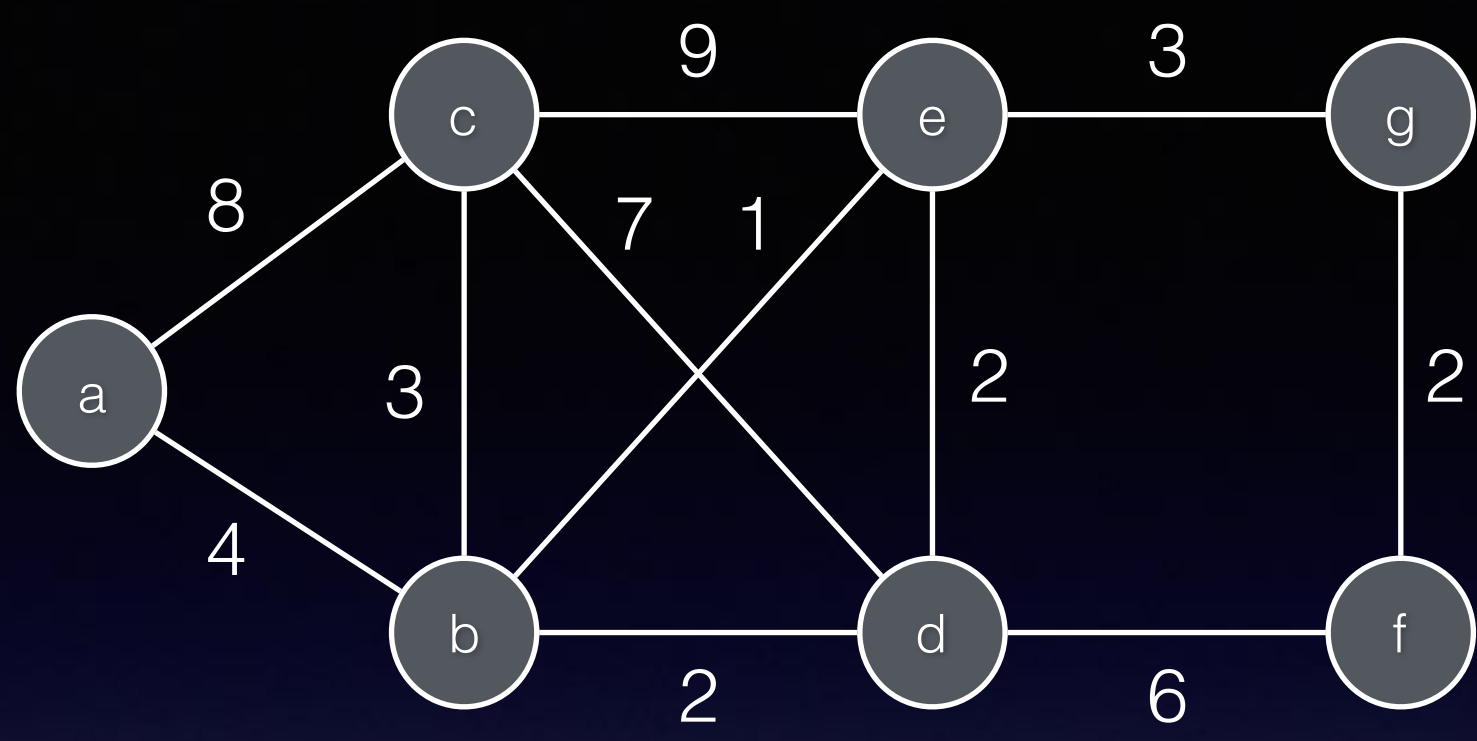
current vertex: ?
adjacent unvisited vertices: ?



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	10	8
predecessor:	?	a	b	b	b	g	e

dequeue unvisited vertex with shortest distance to **a**:

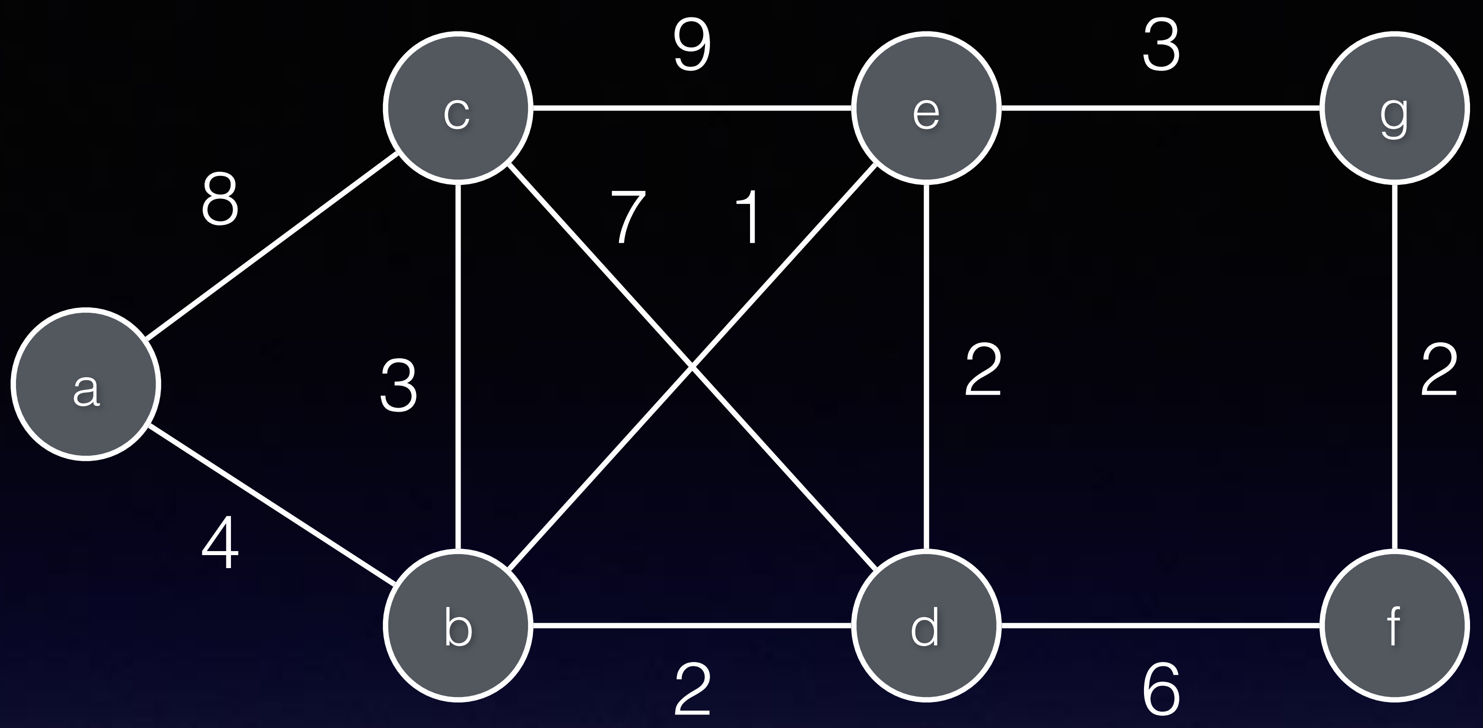
current vertex: **f**
adjacent unvisited vertices: none, nothing to do here



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	10	8
predecessor:	?	a	b	b	b	g	e

The priority queue is empty, so we're done!

But how do we know what the shortest path from **a** to anywhere is?

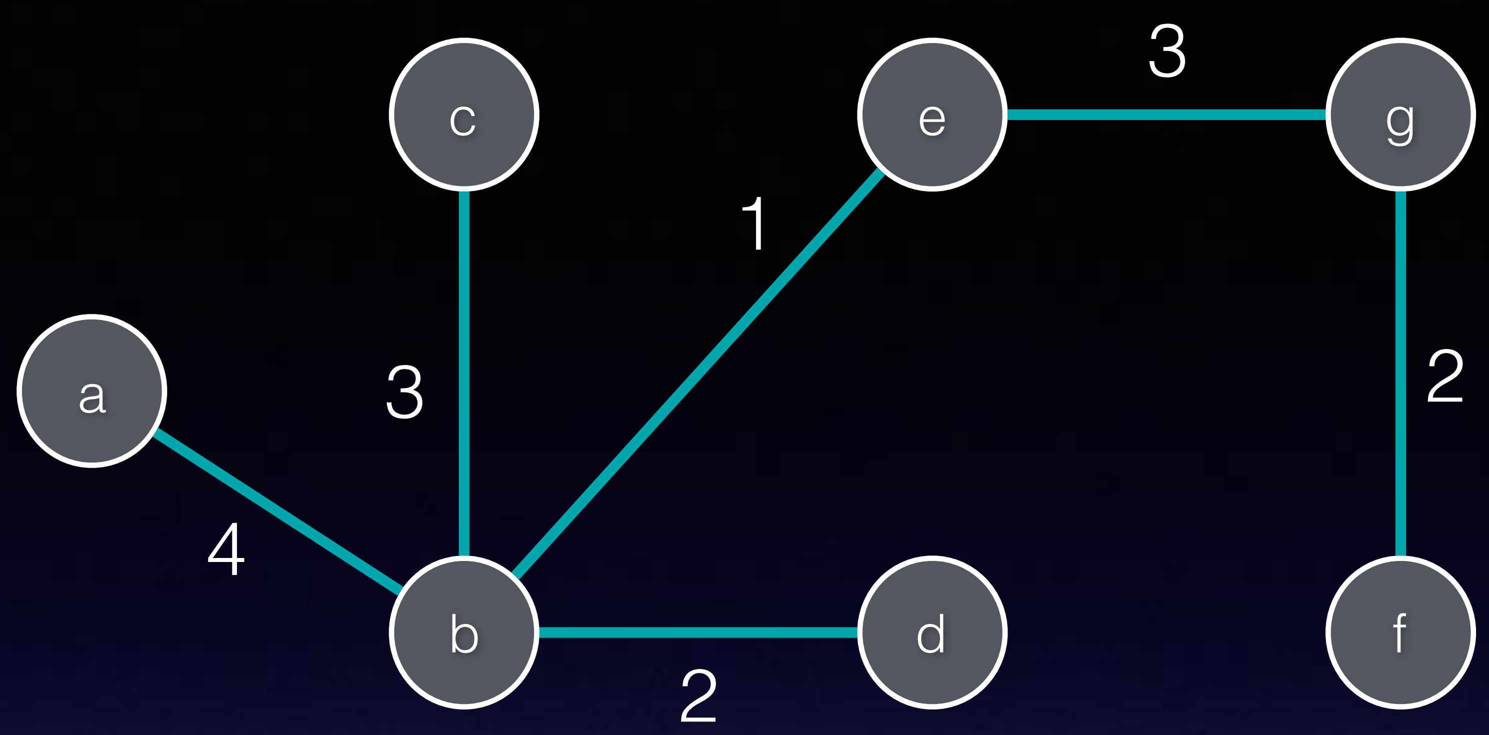


vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	10	8
predecessor:	?	a	b	b	b	g	e

Choose the vertex that you want to go to, then follow the predecessor pointers back to **a**.

Let's choose **g**, for example.

Shortest path from **g** to **a**: **g, e, b, a**



vertex:	a	b	c	d	e	f	g
d(a,v):	0	4	7	6	5	10	8
predecessor:	?	a	b	b	b	g	e

Predecessor always leads you closer to the start.

The links define a tree with the start as the root.

Similar to the BFS tree but with weights!