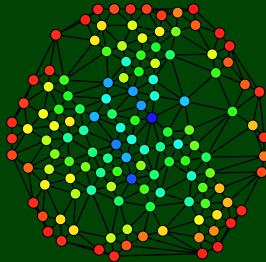


# CSE 332

## Data Abstractions

# Graphs 1: What is a Graph? DFS and BFS



## Where We've Been

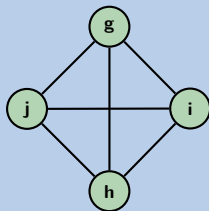
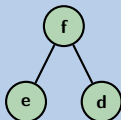
- Essential ADTs: Lists, Stacks, Queues, Priority Queues, Heaps, Vanilla Trees, BSTs, Balanced Trees, B-Trees, Hash Tables
- Important Algorithms: Traversals, Sorting, buildHeap, Prefix Sum, "Divide and Conquer Algorithms"
- Concurrency: Parallelism, Synchronization

So, what's next?

## Graphs and Graph Algorithms

A nearly universal data structure that will change the way you think about the world. (Seriously.)

Graphs are more common than all the other data structures combined (this is in part true, because they're a **generalization** of most of the other data structures).



$$V = \{a\}, E = \emptyset$$

$$V = \{b, c\}, \\ E = \{\{b, c\}\}$$

We call the circles **vertices** and the lines **edges**.

## Definition (Graph)

A **Graph** is a pair,  $G = (V, E)$ , where:

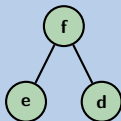
- $V$  is a set of **vertices**, and
- $E$  is a set of **edges** (pairs of vertices).



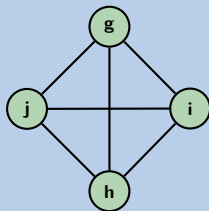
$$V = \{a\}, E = \emptyset$$



$$V = \{b, c\}, \\ E = \{\{b, c\}\}$$



$$V = \{d, e, f\}, \\ E = \{\{e, f\}, \{f, d\}\}$$



$$V = \{g, h, i, j\}, \\ E = \{\{x, y\} \mid x, y \in V \wedge x \neq y\}$$

We call the circles **vertices** and the lines **edges**.

## Definition (Graph)

A **Graph** is a pair,  $G = (V, E)$ , where:

- $V$  is a set of **vertices**, and
- $E$  is a set of **edges** (pairs of vertices).

We can think of graphs as an ADT with operations like `x.isNeighbor(y)`, but it's not clear what should be included:

- `x.reachableFrom(y)?`
- `x.shortestPathTo(y)?`
- `x.centraliity()?`
- ...

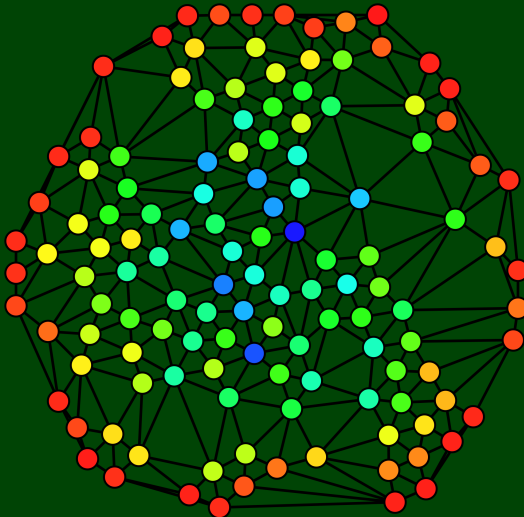
We will approach graphs differently:

- Graphs are an abstract concept that we can apply in different ways to the problem at hand.
- A “graph problem” is one that we can **mathematically model** as a graph. . .

Consider the following questions:

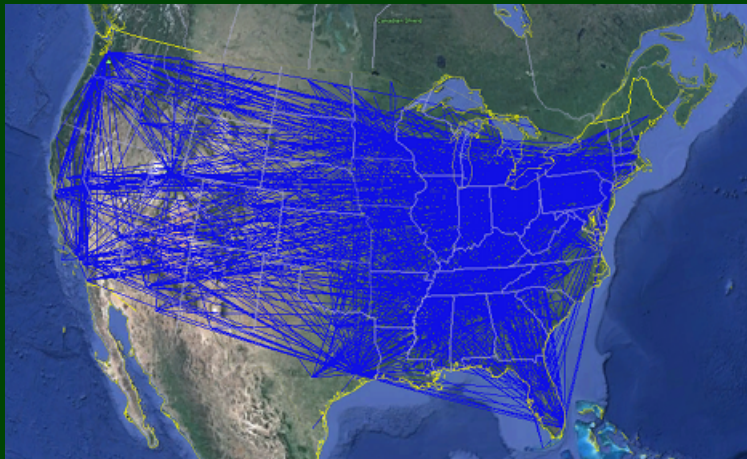
- How can I allocate registers to variables in a program?
- How popular am I?
- What's the minimum amount of wire I have to use to connect all these homes?
- Just **how does** Google work?
- Can I automatically tag the words of a sentence with their part of speech?
- How do I make look-ups in databases quick at Facebook's scale?

If this graph is a social network, who is most important?  
Who has the most influence?

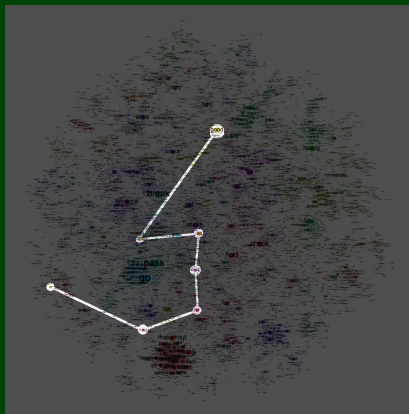




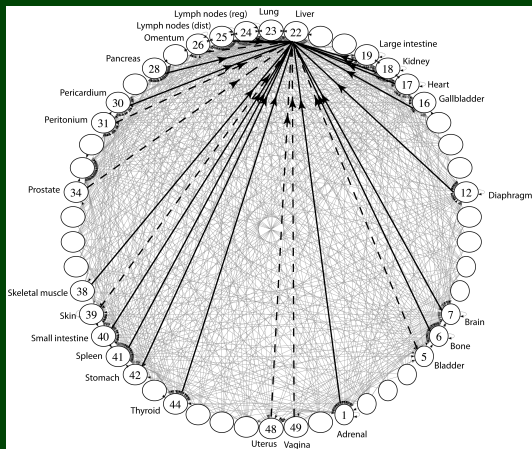
What is the cheapest/shortest/etc. flight from location A to B?



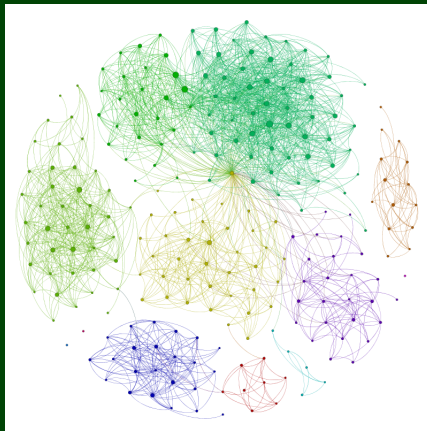
How do words associate with each other? How easy are words to confuse? How similar are spellings?



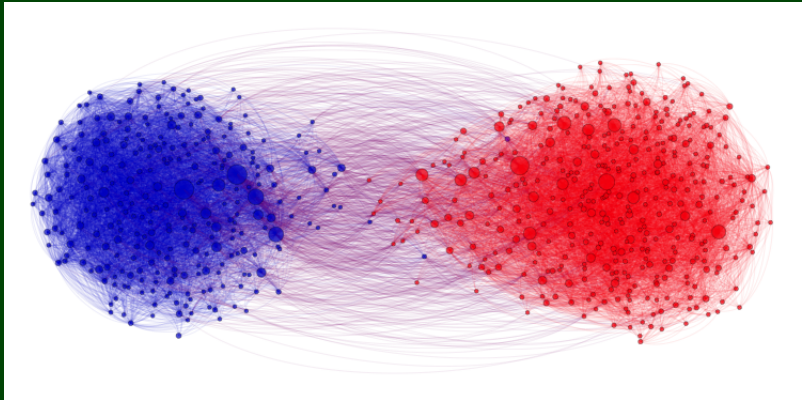
Where should we target cancer in a particular patient's body?



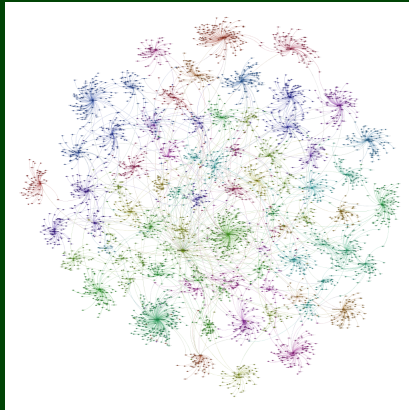
How far am I from my friends? Why does my social network look clustered? Why are all my friends more popular than I am?



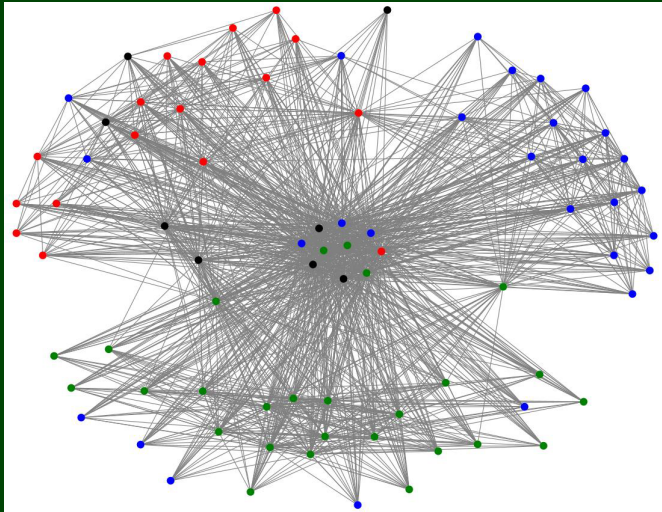
What can we find out from political blog data? Who listens to whom?



What effect did Robin Williams have on the people around him?



What happens when veteran teachers leave school networks?



To model a problem with a graph, you need to make two choices

- 1 What are the vertices?
- 2 What are the edges?

- Maps
- The Internet
- Social Networks
- A Running Program
- A Chess Game
- Telephone Lines
- CSE Courses

With these in mind, let's talk about more crucial definitions.



To model a problem with a graph, you need to make two choices

1 What are the vertices?

2 What are the edges?

- Maps

Vertices: regions; Edges: “is next to”

- The Internet

- Social Networks

- A Running Program

- A Chess Game

- Telephone Lines

- CSE Courses

With these in mind, let's talk about more crucial definitions.

To model a problem with a graph, you need to make two choices

1 What are the vertices?

2 What are the edges?

- Maps

Vertices: regions; Edges: “is next to”

- The Internet

Vertices: websites; Edges: “has a link to”

- Social Networks

- A Running Program

- A Chess Game

- Telephone Lines

- CSE Courses

With these in mind, let's talk about more crucial definitions.

To model a problem with a graph, you need to make two choices

1 What are the vertices?

2 What are the edges?

- Maps

Vertices: regions; Edges: “is next to”

- The Internet

Vertices: websites; Edges: “has a link to”

- Social Networks

Vertices: people; Edges: “is friends with”

- A Running Program

- A Chess Game

- Telephone Lines

- CSE Courses

With these in mind, let's talk about more crucial definitions.

To model a problem with a graph, you need to make two choices

1 What are the vertices?

2 What are the edges?

- Maps

Vertices: regions; Edges: “is next to”

- The Internet

Vertices: websites; Edges: “has a link to”

- Social Networks

Vertices: people; Edges: “is friends with”

- A Running Program

Vertices: methods; Edges: “calls”

- A Chess Game

- Telephone Lines

- CSE Courses

With these in mind, let's talk about more crucial definitions.

To model a problem with a graph, you need to make two choices

1 What are the vertices?

2 What are the edges?

- Maps

Vertices: regions; Edges: “is next to”

- The Internet

Vertices: websites; Edges: “has a link to”

- Social Networks

Vertices: people; Edges: “is friends with”

- A Running Program

Vertices: methods; Edges: “calls”

- A Chess Game

Vertices: boards; Edges: “can move to”

- Telephone Lines

- CSE Courses

With these in mind, let's talk about more crucial definitions.

To model a problem with a graph, you need to make two choices

1 What are the vertices?

2 What are the edges?

- Maps

Vertices: regions; Edges: “is next to”

- The Internet

Vertices: websites; Edges: “has a link to”

- Social Networks

Vertices: people; Edges: “is friends with”

- A Running Program

Vertices: methods; Edges: “calls”

- A Chess Game

Vertices: boards; Edges: “can move to”

- Telephone Lines

Vertices: houses; Edges: “telephone line between”

- CSE Courses

With these in mind, let's talk about more crucial definitions.

To model a problem with a graph, you need to make two choices

1 What are the vertices?

2 What are the edges?

- Maps

Vertices: regions; Edges: “is next to”

- The Internet

Vertices: websites; Edges: “has a link to”

- Social Networks

Vertices: people; Edges: “is friends with”

- A Running Program

Vertices: methods; Edges: “calls”

- A Chess Game

Vertices: boards; Edges: “can move to”

- Telephone Lines

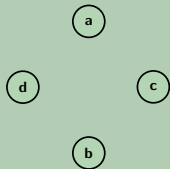
Vertices: houses; Edges: “telephone line between”

- CSE Courses

Vertices: courses; Edges: “is a pre-requisite of”

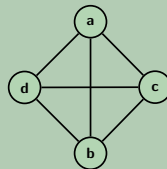
With these in mind, let’s talk about more crucial definitions.

## Empty Graph



...

## Complete Graph ( $K_n$ )

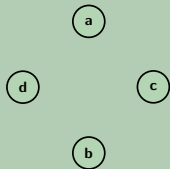


## Some Questions

- How many edges can a graph with  $|V| = n$  have?
- If we have  $|E| = n$ , what is the smallest number of vertices we can have? The largest?
  - Smallest:
  - Largest:

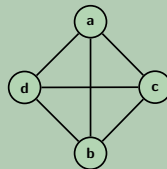


## Empty Graph



...

## Complete Graph ( $K_n$ )



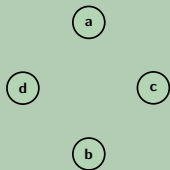
## Some Questions

- How many edges can a graph with  $|V| = n$  have?

$$|E| = \binom{n}{2} = \frac{n(n-1)}{2}$$

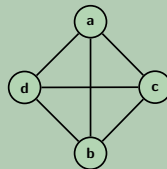
- If we have  $|E| = n$ , what is the smallest number of vertices we can have? The largest?
  - Smallest:
  - Largest:

## Empty Graph



...

## Complete Graph ( $K_n$ )



## Some Questions

- How many edges can a graph with  $|V| = n$  have?

$$|E| = \binom{n}{2} = \frac{n(n-1)}{2}$$

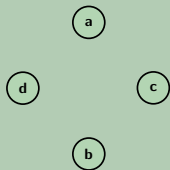
- If we have  $|E| = n$ , what is the smallest number of vertices we can have? The largest?

- Smallest:

$$\frac{v(v-1)}{2} = n \implies v^2 - v = v \in \mathcal{O}(\sqrt{n})$$

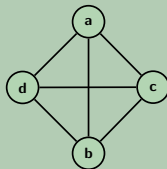
- Largest:

## Empty Graph



...

## Complete Graph ( $K_n$ )



## Some Questions

- How many edges can a graph with  $|V| = n$  have?

$$|E| = \binom{n}{2} = \frac{n(n-1)}{2}$$

- If we have  $|E| = n$ , what is the smallest number of vertices we can have? The largest?

- Smallest:

$$\frac{v(v-1)}{2} = n \implies v^2 - v = v \in \mathcal{O}(\sqrt{n})$$

- Largest: There is no largest!

## Definition (Walk)

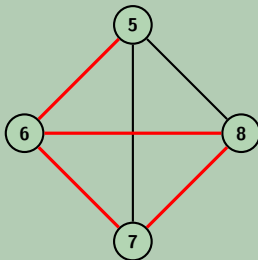
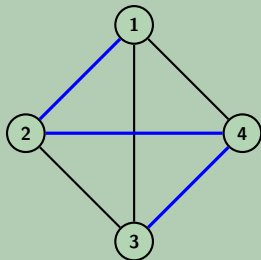
A **walk** in a graph  $G = (V, E)$  is a list of vertices:

$v_0, v_1, \dots, v_n$  such that  $\{v_i, v_{i+1}\} \in E$ .

Intuitively, a path from  $u$  to  $v$  is a continuous line drawn without picking up your pencil.

## Definition (Path)

A **path** in a graph  $G = (V, E)$  is a walk with no repeated vertices.



The blue edges are a path    The red edges are a walk but not a path

## Definition (Walk)

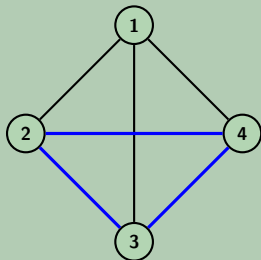
A **walk** in a graph  $G = (V, E)$  is a list of vertices:

$v_0, v_1, \dots, v_n$  such that  $\{v_i, v_{i+1}\} \in E$ .

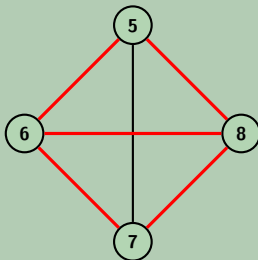
Intuitively, a path from  $u$  to  $v$  is a continuous line drawn without picking up your pencil.

## Definition (Cycle)

A **cycle** in a graph  $G = (V, E)$  is a walk  $(v_0, v_1, \dots, v_n)$  with no repeated vertices **except**  $v_0 = v_n$ .



The blue edges are a cycle

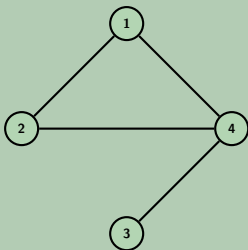


The red edges are not a cycle

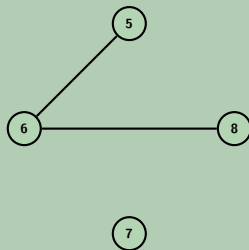
## Definition (Connected Graph)

We say a graph is connected if for every pair of vertices,  $u, v \in V$ , there is a path from  $u$  to  $v$ .

Intuitively, if we pick up the graph and shake it around, if anything isn't still in the air, then the graph isn't connected.



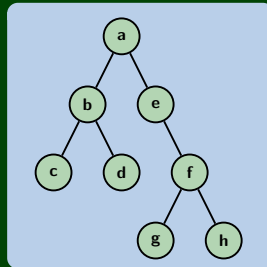
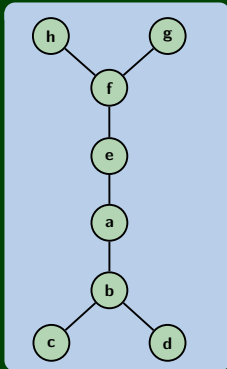
Connected!



Not Connected!

## Definition (Tree)

A **tree** is a **connected, acyclic** graph.



## Definition (Rooted Tree)

A **rooted tree** is a tree with one special node called out as the “root”.

It can be equally useful to work with trees as it is to work with rooted trees; we’re definitely more familiar with rooted trees so far. . .

A very common type of algorithm on graphs is a **worklist algorithm**.

First, we define a new (non-standard) ADT:

### WorkList ADT

<code>add(v)</code>	Notifies the worklist that it must handle <b>v</b>
<code>next()</code>	Returns the next vertex to work on
<code>hasWork()</code>	Returns true if there's any work left and false otherwise

Importantly, we **do not care how** the worklist manages the work. (Okay, we do, but not when coming up with the algorithm.)

Worklist algorithms will always look like the following:

```
1 worklist = /* add initial work to worklist */
2 while (worklist.hasWork()) {
3     v = worklist.next();
4     doWork(v);
5 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

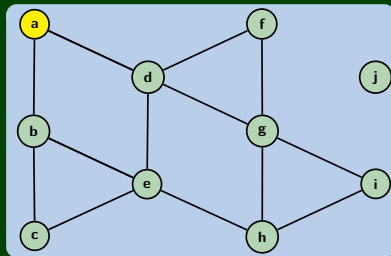
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

Relatedly, we might ask the question:

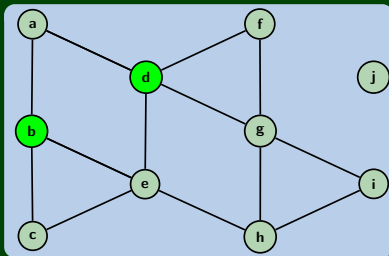
Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```

1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

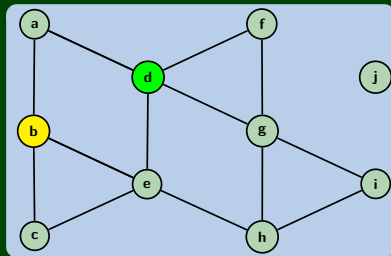
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

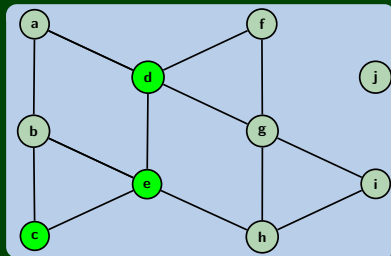
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

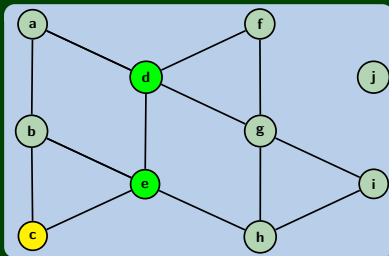
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

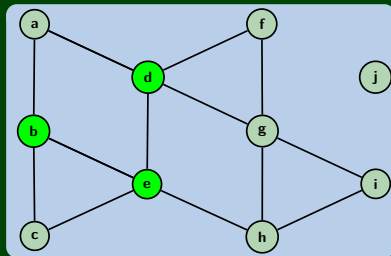
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

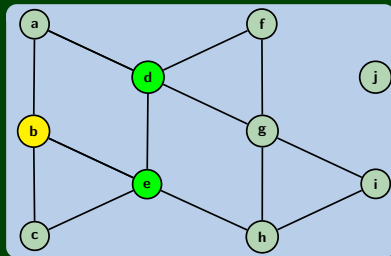
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

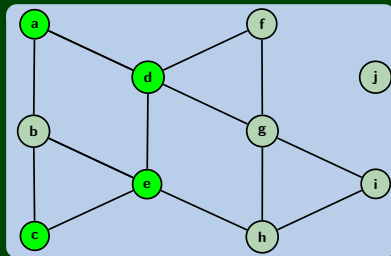
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```





We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

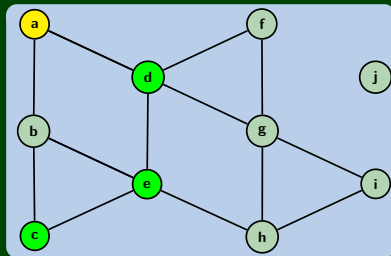
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

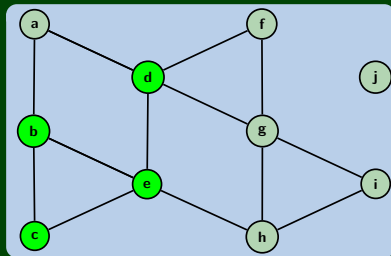
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

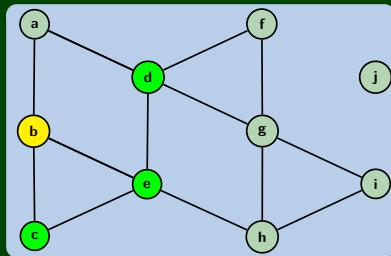
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



We said a graph is **connected** when there is a path between every pair of vertices. What if we don't **know** if a graph is connected?

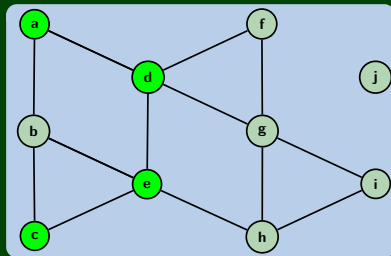
Relatedly, we might ask the question:

Is a vertex  $w$  **reachable** (does there exist a path) from a vertex  $v$ ?

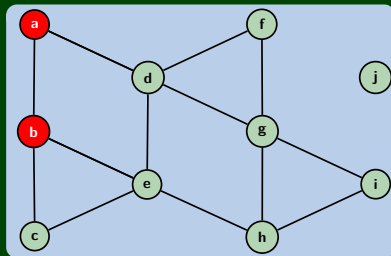
We could use this algorithm to **find a path**, **do something** with every node, **search** for a particular node, etc.

Unsurprisingly, this is a worklist algorithm:

```
1 search(v) {
2   worklist = [v];
3   while (worklist.hasWork()) {
4     v = worklist.next();
5     doSomething(v);
6     for (w : v.neighbors()) {
7       worklist.add(w);
8     }
9   }
10 }
```



```
1 search(v) {  
2   worklist = [v];  
3   while (worklist.hasWork()) {  
4     v = worklist.next();  
5     doSomething(v);  
6     for (w : v.neighbors()) {  
7       worklist.add(w);  
8     }  
9   }  
10 }
```



## What Happened?

We started searching paths in the graph and eventually went back and between **a** and **b**.

This happened, because there were **two distinct paths** between **b** and **c**:

$b \rightarrow e \rightarrow c$

and

$c \rightarrow b \rightarrow e$

We followed the cycle in our graph!

**A Corollary:** This wouldn't have happened on a tree!

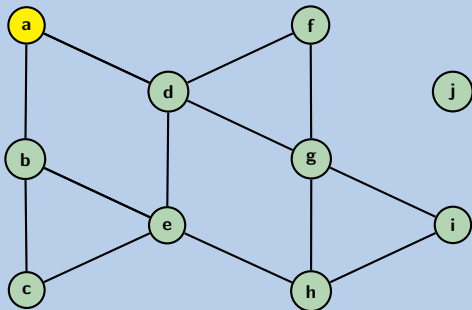
That is, we just **found an algorithm** to check for tree-ness!

```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist

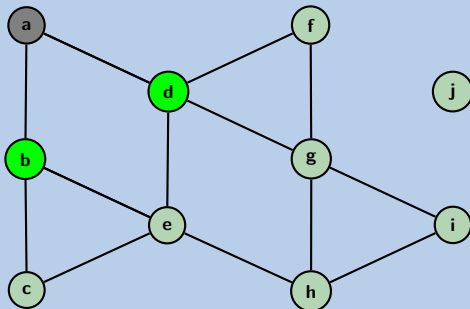


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist

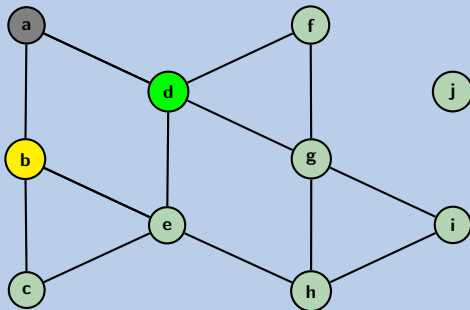


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



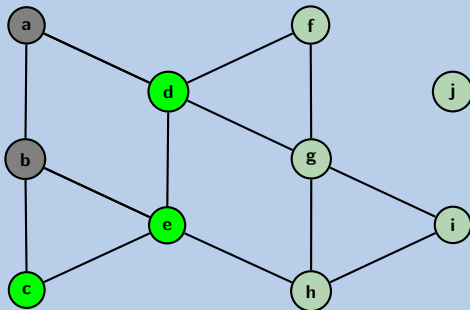


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

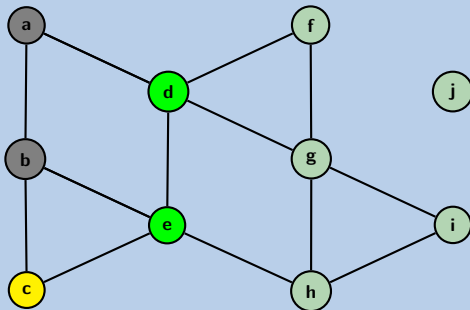
- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



```
1 search(v) {  
2   worklist = [v];  
3   seen = {v};  
4   while (worklist.hasWork()) {  
5     v = worklist.next();  
6     doSomething(v);  
7     for (w : v.neighbors()) {  
8       if (w not seen) {  
9         worklist.add(w);  
10        seen = seen ∪ {w};  
11      }  
12    }  
13  }  
14 }
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist

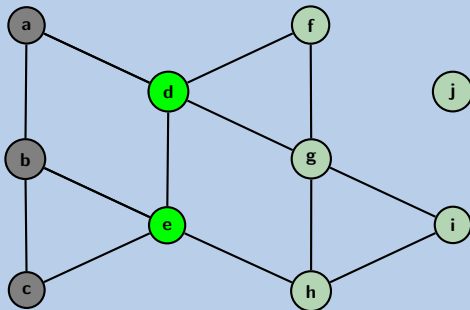


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist

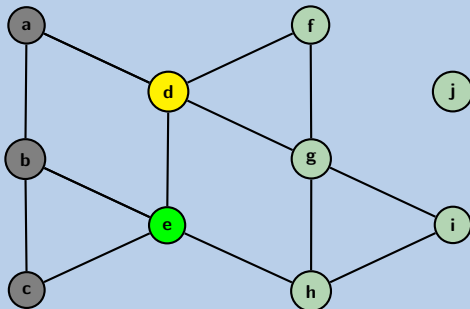


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

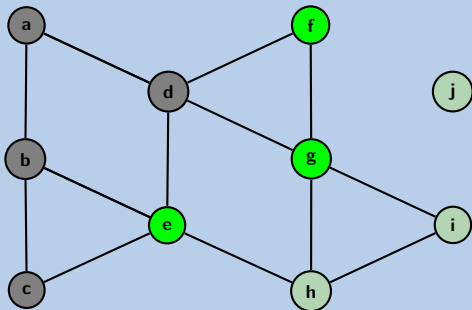
- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



```
1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
```

We have two ways of fixing this:

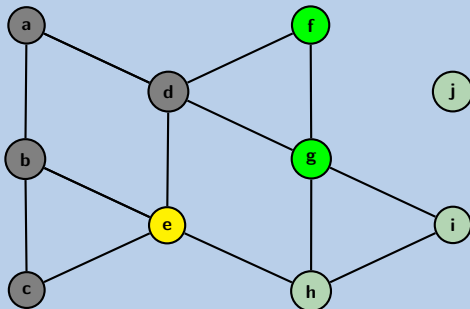
- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



```
1 search(v) {  
2   worklist = [v];  
3   seen = {v};  
4   while (worklist.hasWork()) {  
5     v = worklist.next();  
6     doSomething(v);  
7     for (w : v.neighbors()) {  
8       if (w not seen) {  
9         worklist.add(w);  
10        seen = seen ∪ {w};  
11      }  
12    }  
13  }  
14 }
```

We have two ways of fixing this:

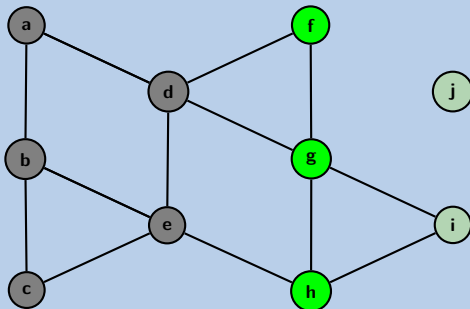
- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



```
1 search(v) {  
2   worklist = [v];  
3   seen = {v};  
4   while (worklist.hasWork()) {  
5     v = worklist.next();  
6     doSomething(v);  
7     for (w : v.neighbors()) {  
8       if (w not seen) {  
9         worklist.add(w);  
10        seen = seen ∪ {w};  
11      }  
12    }  
13  }  
14 }
```

We have two ways of fixing this:

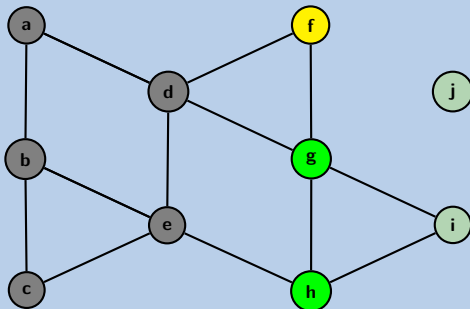
- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



```
1 search(v) {  
2   worklist = [v];  
3   seen = {v};  
4   while (worklist.hasWork()) {  
5     v = worklist.next();  
6     doSomething(v);  
7     for (w : v.neighbors()) {  
8       if (w not seen) {  
9         worklist.add(w);  
10        seen = seen ∪ {w};  
11      }  
12    }  
13  }  
14 }
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



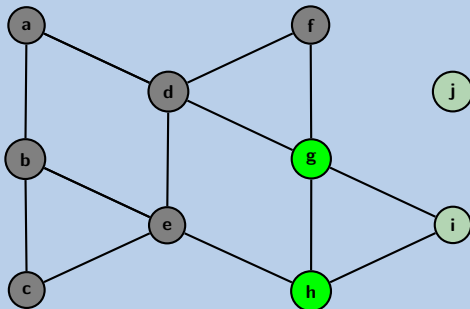


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist

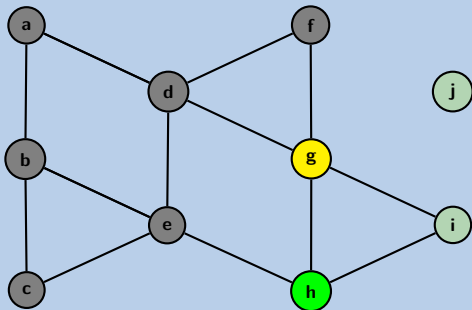


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

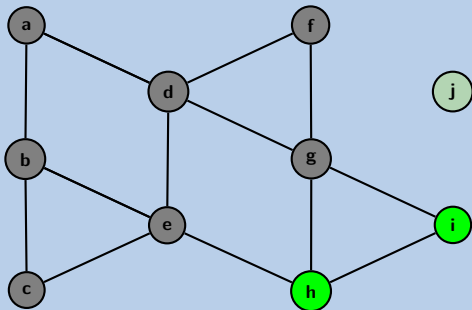
- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



```
1 search(v) {  
2   worklist = [v];  
3   seen = {v};  
4   while (worklist.hasWork()) {  
5     v = worklist.next();  
6     doSomething(v);  
7     for (w : v.neighbors()) {  
8       if (w not seen) {  
9         worklist.add(w);  
10        seen = seen ∪ {w};  
11      }  
12    }  
13  }  
14 }
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist

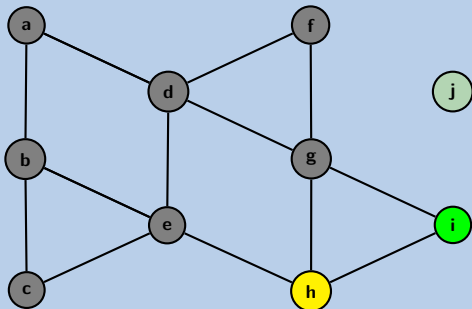


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist

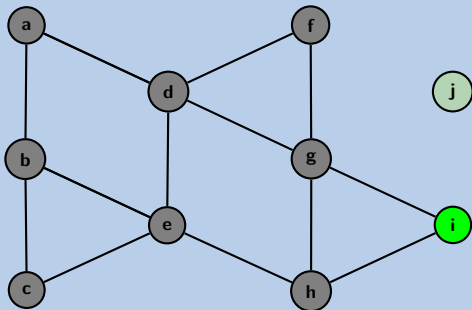


```

1 search(v) {
2   worklist = [v];
3   seen = {v};
4   while (worklist.hasWork()) {
5     v = worklist.next();
6     doSomething(v);
7     for (w : v.neighbors()) {
8       if (w not seen) {
9         worklist.add(w);
10        seen = seen ∪ {w};
11      }
12    }
13  }
14 }
    
```

We have two ways of fixing this:

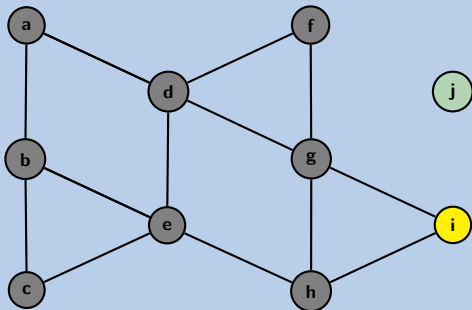
- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



```
1 search(v) {  
2   worklist = [v];  
3   seen = {v};  
4   while (worklist.hasWork()) {  
5     v = worklist.next();  
6     doSomething(v);  
7     for (w : v.neighbors()) {  
8       if (w not seen) {  
9         worklist.add(w);  
10        seen = seen ∪ {w};  
11      }  
12    }  
13  }  
14 }
```

We have two ways of fixing this:

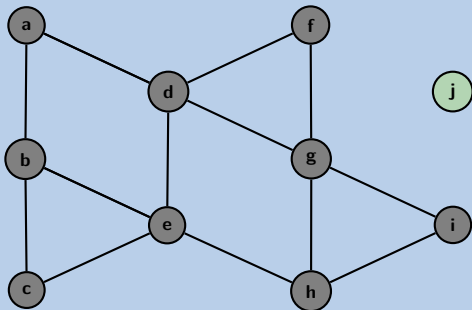
- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



```
1 search(v) {  
2   worklist = [v];  
3   seen = {v};  
4   while (worklist.hasWork()) {  
5     v = worklist.next();  
6     doSomething(v);  
7     for (w : v.neighbors()) {  
8       if (w not seen) {  
9         worklist.add(w);  
10        seen = seen ∪ {w};  
11      }  
12    }  
13  }  
14 }
```

We have two ways of fixing this:

- 1 insist that the worklist take care of duplicates, and
- 2 avoid feeding duplicates to the worklist



Okay, but we'd never actually use a SortedList. How about a Stack?

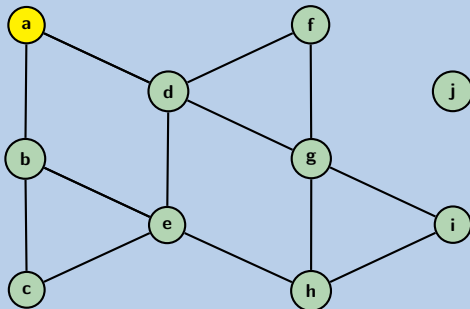
When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

↓↑

a



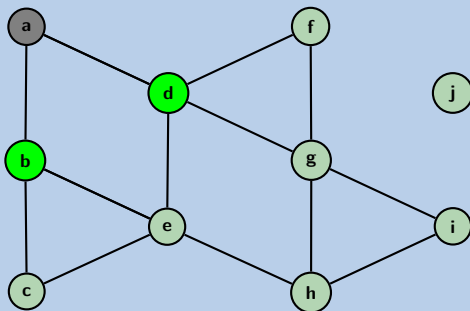


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

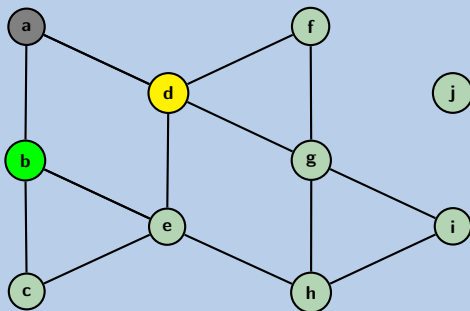


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist



Okay, but we'd never actually use a SortedList. How about a Stack?

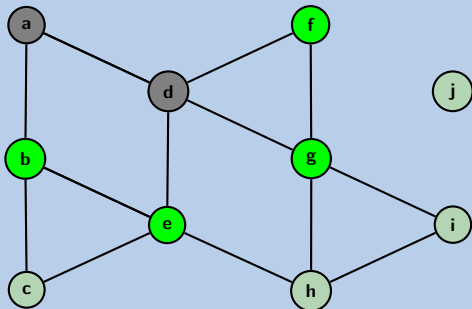
When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

↓↑

g
f
e
b



Okay, but we'd never actually use a SortedList. How about a Stack?

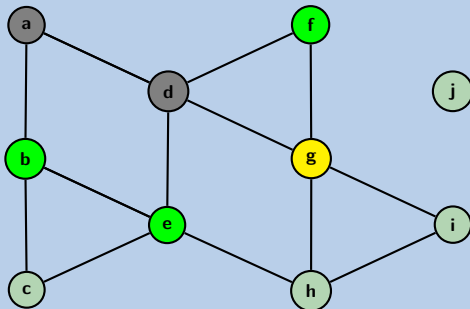
When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

↓↑

g
f
e
b



Okay, but we'd never actually use a SortedList. How about a Stack?

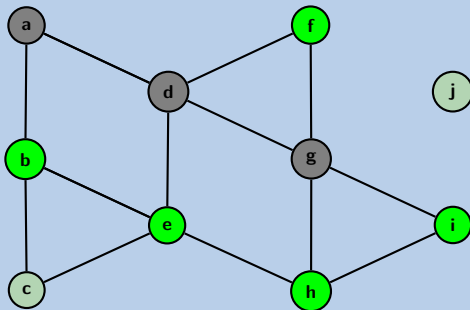
When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

↓↑

i
h
f
e
b

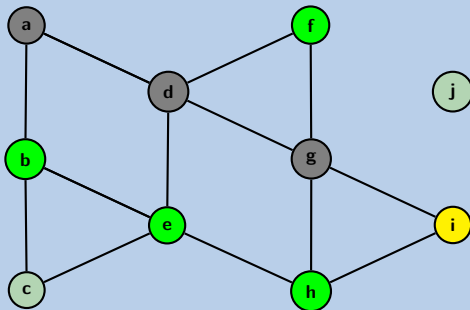
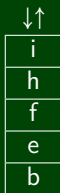


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

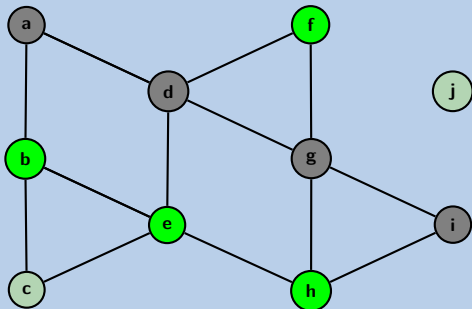


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

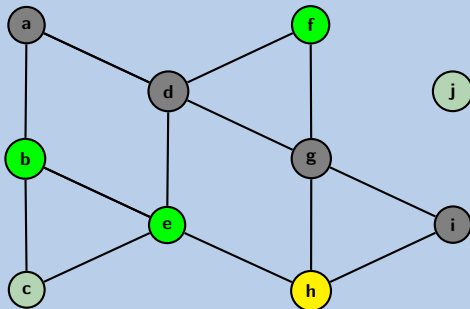


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist



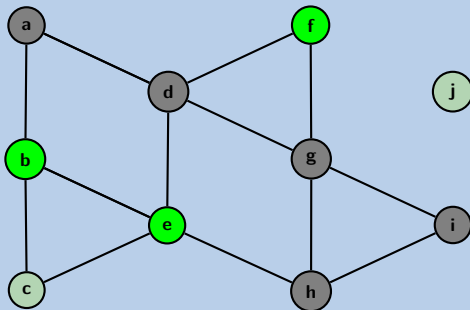


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

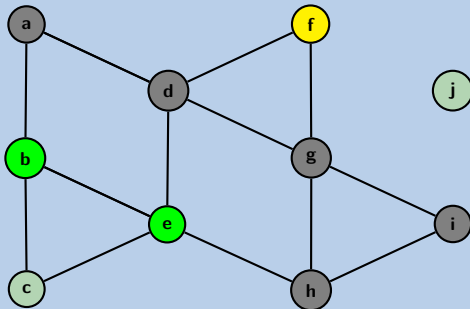


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

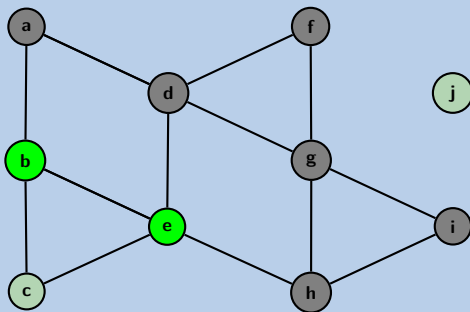


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

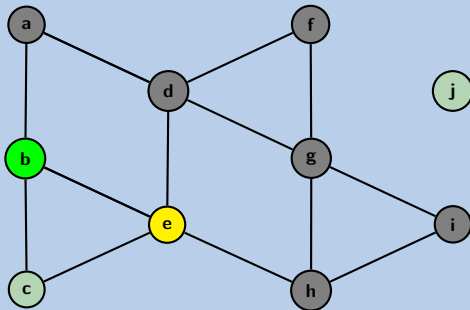


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

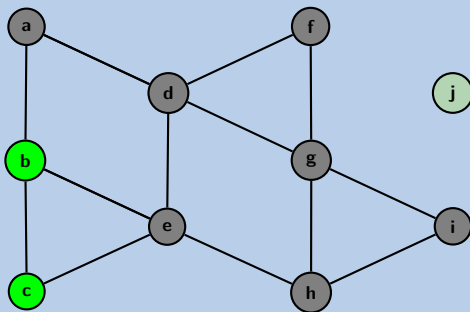


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

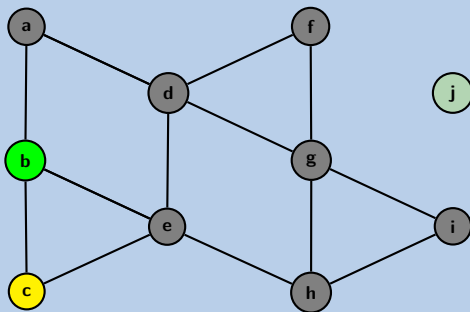


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist



Okay, but we'd never actually use a SortedList. How about a Stack?

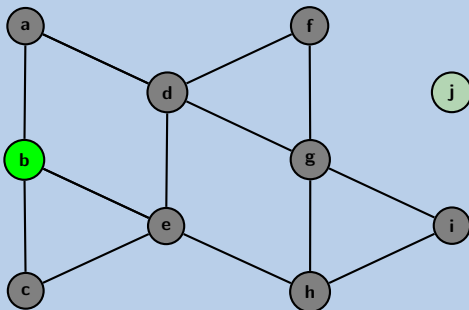
When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

↓↑

b



Okay, but we'd never actually use a SortedList. How about a Stack?

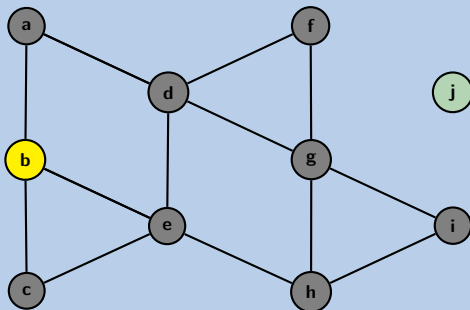
When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

↓↑

b



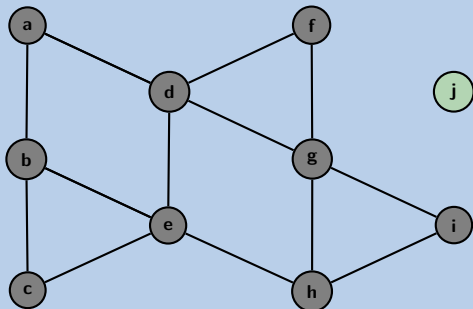


Okay, but we'd never actually use a SortedList. How about a Stack?

When we use a Stack:

- 1 This algorithm is called DFS (depth-first search)
- 2 We follow a path as far as possible, then back up
- 3 It can be written recursively

worklist

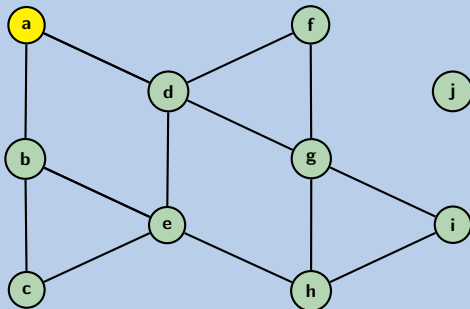


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← a ←

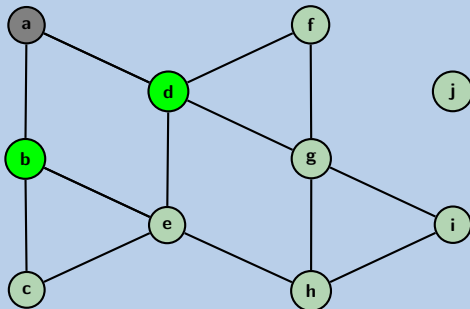


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ b | d ] ←

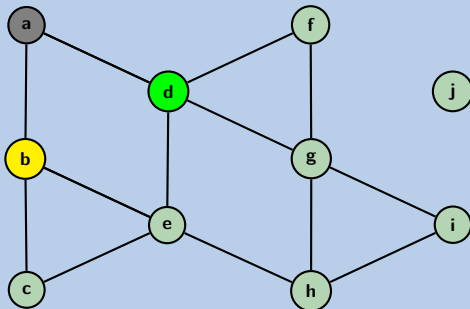


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ b | d ] ←

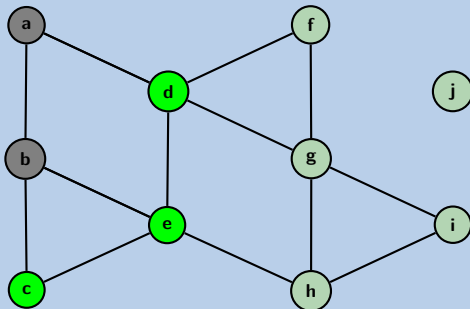


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ d | c | e ] ←

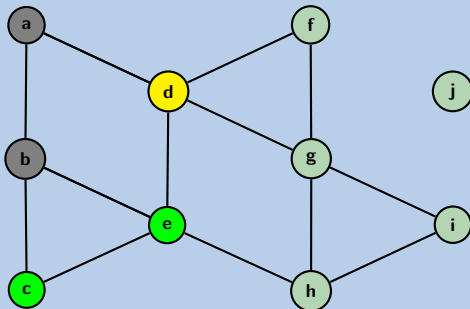


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ d c e ] ←

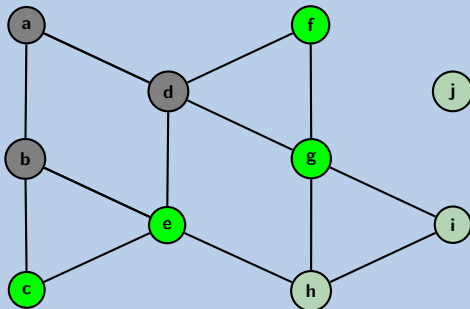


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ c e f g ] ←

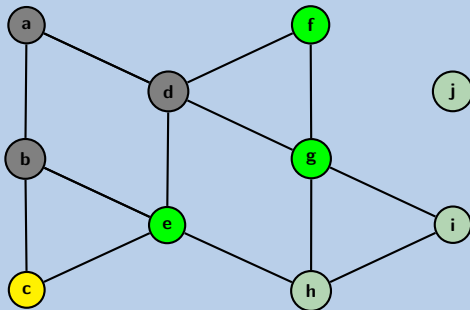


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ c e f g ] ←



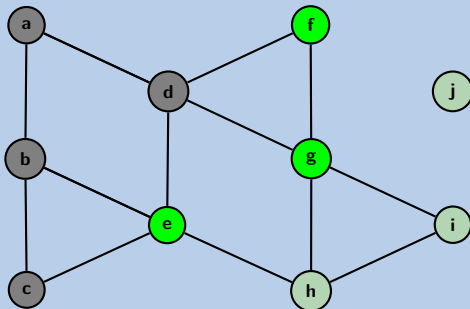


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ e | f | g ] ←

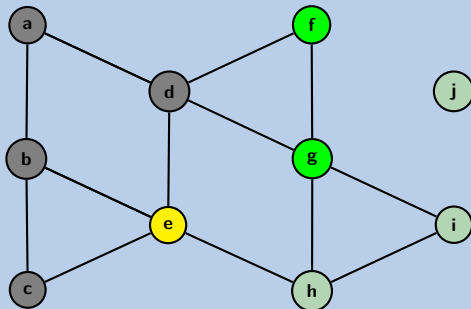


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ e | f | g ] ←

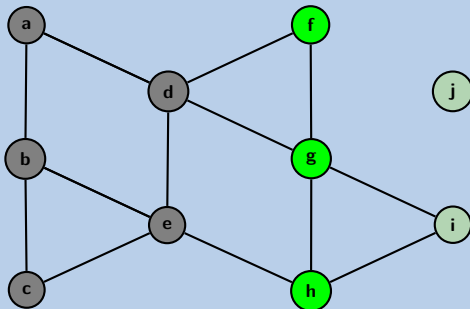


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ f | g | h ] ←

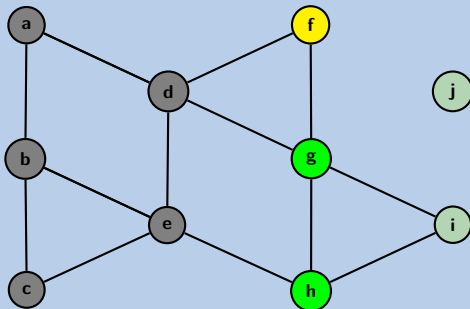


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ f | g | h ] ←

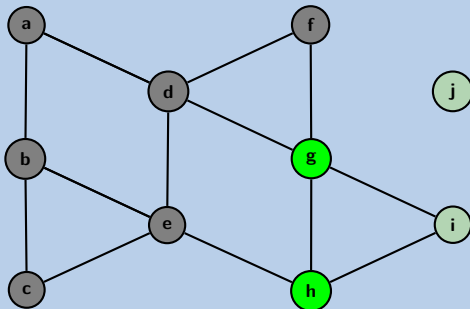


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ g h ] ←

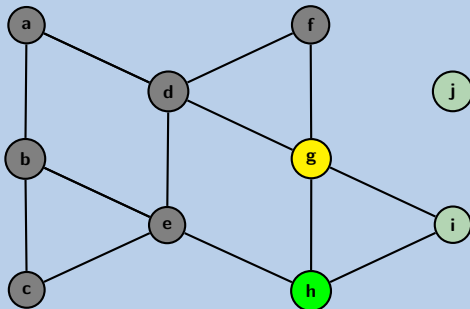


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ g h ] ←

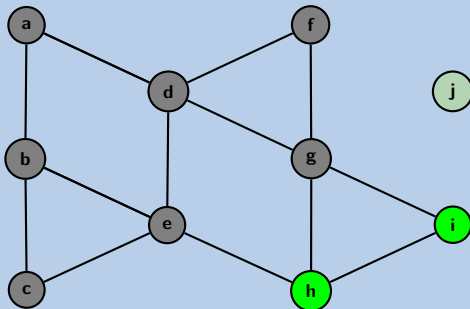


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ h | i ] ←

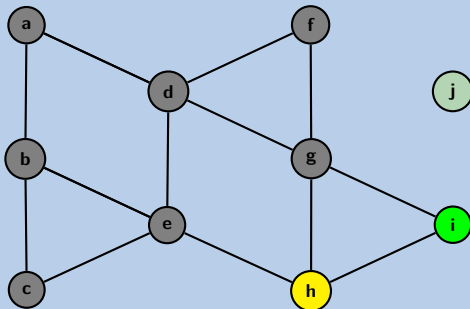


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← [ h | i ] ←



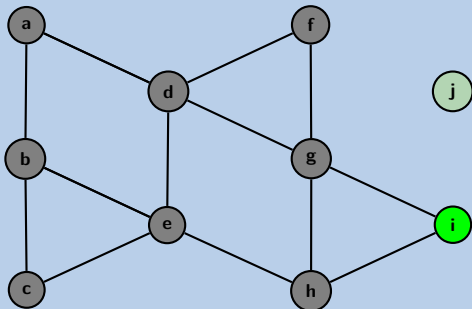


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← i ←

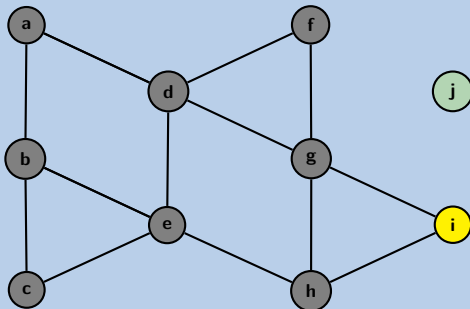


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ← i ←

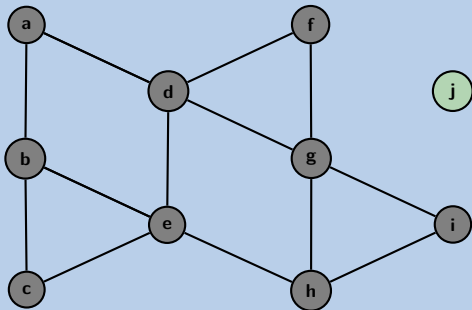


Any reason we shouldn't use a Queue?

When we use a Queue:

- 1 This algorithm is called BFS (breadth-first search)
- 2 We increase our horizon away from the starting node

worklist ←



## Use A Dictionary!

```
search(v) {
    worklist = [v];
    from = new Dictionary();
    from.put(v, null);
    while (worklist.hasWork()) {
        v = worklist.next();
        doSomething(v);
        for (w : v.neighbors()) {
            if (w not in from) {
                worklist.add(w);
                from.put(w, v);
            }
        }
    }
    return from;
}

findPath(v, w) {
    from = search(v);
    path = [];
    curr = w;
    while (curr != null) {
        path.add(0, curr);
        curr = from[curr];
    }
    return path;
}
```

## Runtime

- Both algorithms visit all nodes in the connected component:  $|V|$
- Both algorithms can visit a node once for each edge in the graph:  $|E|$

So, BFS and DFS are  $\mathcal{O}(|V|+|E|)$  (this is called “graph linear”).

## Space

- DFS: If the longest path has length  $p$  and the largest number of neighbors is  $n$ , then DFS stores at most  $pn$  vertices
- BFS: Consider a tree. BFS will hold the entire bottom level which is  $\mathcal{O}(|V|)$ .

## Trade-Offs

- DFS has better space usage, but it might find a circuitous path
- BFS will always find the shortest path to a node, but it will use more memory

## Iterative Deepening

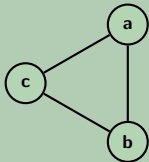
Iterative Deepening is a DFS that **bounds** the depth:

```
1  int depth = 1;
2  while (there are nodes to explore) {
3      dfs(v, depth);
4      depth++;
5  }
```

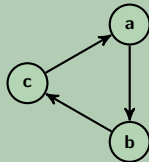
Since **most of the vertices are “leaves”**, this actually doesn't waste much time!

- Undirected vs. Directed (do the edges have arrows?)

## Undirected

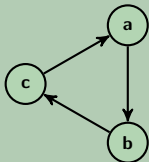


## Directed

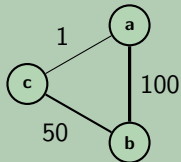


- Weighted vs. Unweighted (do the edges have weights?)

## Unweighted & Directed

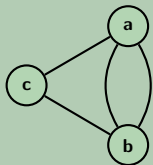


## Weighted & Undirected

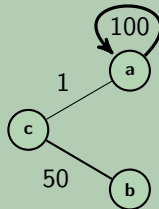


- Simple vs. Multi (loops on vertices? multiple edges?)

Multi-graph



Graph with Loops



These generalizations are all useful in different domains. We're going to talk a lot more about them over the next few lectures.

Next lecture, we'll be working mostly with **directed graphs**.



Back to counting edges. In a graph without multiple edges, if there are  $n$  vertices, there can be anywhere from 0 to  $n^2$  edges.

This is a very wide range. A graph with fewer edges is called **sparse** and one with closer to  $n^2$  is called **dense**.

We already saw that graph traversal was  $\mathcal{O}(|E| + |V|)$ :

- On a sparse graph, that's  $\mathcal{O}(|V|)$
- On a dense graph, that's  $\mathcal{O}(|V|^2)$ .

Sparsity makes a **huge** difference!