The depth of \( v \) is the length of the path from the root to \( v \).

The height of the tree is the maximum depth.
An ordered tree puts the children of each node into a specific order (pictorially represented as left-to-right).

The diagrams shown above are the same as unordered trees, but are different as ordered trees.

We’ll be looking at graphs and trees drawn in other ways; there will be other orderings besides left-to-right.
Depth first search of a tree
We will number the vertices by a depth first search (DFS), also called a depth first traversal.
Start at the root $r$ (the top vertex in this diagram), and assign it the number 1.
Visit 1’s first child (in left-to-right order), and assign it the number 2.
Visit 2’s first child (in left-to-right order), and assign it the number 3.
3 is a leaf, with no children!

We explored as far as possible, but now have to backtrack to explore more.

Back up how we came until a vertex (2) with at least one unnumbered child.
The next child of 2 is numbered 4. Continue going down this branch, choosing the leftmost option available.
Visit 4’s first child and assign it the number 5.
5 has no children.

Back up to the first vertex (4) that has a child not yet numbered.

Assign the first remaining child the number 6.
6 has no children.

Back up to the first vertex (4) that has a child not yet numbered.

Assign the first remaining child the number 7.
Depth first search of a tree

- 7 has no children.
- Back up to the first vertex (2) that has a child not yet numbered.
- Assign the first remaining child the number 8.
Depth first search of a tree

Continue in this fashion until all vertices are numbered.
Breadth first search of a tree
While depth first search explores as deep as possible, **Breadth first search** (BFS) works one layer (depth) at a time.

**Depth 0**: Number the root 1.

**Depth 1**: Consecutively number all neighbors of 1.

**Depth 2**: The depth 2 elements are all the neighbors of the depth 1 elements not yet accounted for. Number them consecutively.

Continue in this way for depth 3, 4, etc.
Breadth first search of a tree

Depth first search

Breadth first search
Depth first search (DFS) in a connected graph
Depth first search (DFS) in a connected graph

For trees, we showed the root at the top, and children below in left-to-right order. That doesn’t apply for arbitrary graph drawings.
Set up a counter: \( \text{time} = 0 \)
- We’ll keep adding 1 to it as we number the vertices.

Pick starting vertex: \( a \)
- The starting vertex can be any vertex; \( a \) is just an example.
  - You may get a different tree, depending on where you start.
  - This takes the place of the root in a rooted tree.
  - In some applications, it’s called the source.
Current vertex: \( u = a \)

Color \( u \) red: \( a \) is *discovered*

Time stamp \( u \): \( T(a) = \text{time} = 1 \)

Neighbors of \( u \): \( b, d, e \) (we’ll use alphabetical order)

1\textsuperscript{st} undiscovered neighbor: \( v = b \)

Draw red edge \( \{u, v\} \): \( \{a, b\} \)

Continue exploring: \( \text{DFS}(b) \)
Depth first search (DFS) in a connected graph

- **Current vertex:** \( u = b \)
- **Color \( u \) red:** \( b \) is *discovered*
- **Time stamp \( u \):** \( T(b) = \text{time} = 2 \)
- **Neighbors of \( u \):** \( a, c \)
- **1\text{st} undiscovered neighbor:** \( v = c \)
- **Draw red edge \( \{u, v\} \):** \( \{b, c\} \)
- **Continue exploring:** DFS(c)
Depth first search (DFS) in a connected graph

- Current vertex: $u = c$
- Color $u$ red: $c$ is discovered
- Time stamp $u$: $T(c) = \text{time} = 3$
- Neighbors of $u$: $b, f, h$
- 1st undiscovered neighbor: $v = f$
- Draw red edge $\{u, v\}$: $\{c, f\}$
- Continue exploring: DFS(f)
Depth first search (DFS) in a connected graph

Skipping ahead a few steps...

- Current vertex: $u = g$
- Color $u$ red: $g$ is discovered
- Time stamp $u$: $T(g) = \text{time} = 6$
- Neighbors of $u$: $d, e, j$
- 1st undiscovered neighbor: $v = d$
- Draw red edge $\{u, v\}$: $\{g, d\}$
- Continue exploring: DFS(d)
Current vertex: \( u = d \)
Color \( u \) red: \( d \) is \textit{discovered}
Time stamp \( u \): \( T(d) = \text{time} = 7 \)
Neighbors of \( u \): \( a, f, g \)
But all neighbors of \( u \) have already been discovered!
\textit{Backtrack} to find a vertex \((g)\) with an undiscovered neighbor.
Depth first search (DFS) in a connected graph

Current vertex: \( u = g \)

Neighbors of \( u \): \( d, e, j \)

1\textsuperscript{st} undiscovered neighbor: \( v = e \)

Draw red edge \( \{u, v\} \): \( \{g, e\} \)

Continue exploring: \( \text{DFS}(e) \)
Depth first search (DFS) in a connected graph

- Current vertex: \( u = e \)
- Color \( u \) red: \( e \) is discovered
- Time stamp \( u \): \( T(e) = \text{time} = 8 \)
- Neighbors of \( u \): \( a, g, h \)
- 1\textsuperscript{st} undiscovered neighbor: \( v = h \)
- Draw red edge \( \{u, v\} \): \( \{e, h\} \)
- Continue exploring: DFS(h)
Depth first search (DFS) in a connected graph

- Current vertex: \( u = h \)
- Color \( u \) red: \( h \) is \textit{discovered}
- Time stamp \( u \): \( T(h) = \text{time} = 9 \)
- Neighbors of \( u \): \( c, e, i \)
- 1\textsuperscript{st} undiscovered neighbor: \( v = i \)
- Draw red edge \( \{u, v\} \): \( \{h, i\} \)
- Continue exploring: \( \text{DFS}(i) \)
Depth first search (DFS) in a connected graph

- **Current vertex:** \( u = i \)
- **Color** \( u \) **red:** \( i \) is *discovered*
- **Time stamp** \( u \): \( T(i) = \text{time} = 10 \)
- **Neighbors of** \( u \): \( h, j \)

All neighbors are already discovered.

Backtracking doesn’t give any new branch, so we’re done.
Depth first search pseudocode, using recursion

Initialize: discovered[⋯] ← false
parent[⋯] ← null
$T[⋯] ← \infty$  ▶ Time stamp
time ← 0

Start with: DFS(root)

procedure DFS(u)
1:  discovered[u] ← true  ▶ We did this by coloring $u$ red
2:  time ← time + 1  ▶ Visit $u$ by doing something with it,
3:  $T[u] ←$ time  ▶ such as recording a time stamp.
4:  for all $v \in N(u)$ do
5:      if (not discovered[v]) then
6:        parent[v] ← u  ▶ Add tree edge
7:        DFS(v)  ▶ Recursively explore further
DFS only finds vertices reachable from the source.

To find a spanning forest for a disconnected graph, loop over all vertices. If a vertex isn’t marked as discovered, do DFS starting there to get a spanning tree for that component. This also lets you count the connected components.

This also applies to breadth first search (BFS), coming up next.

DFS and BFS also work for a directed graph (explore $v \in N^+(u)$). Similar issues arise if it’s not strongly connected.
Breadth first search (BFS) in a connected graph
Queues

- Consider customers waiting in a line, called a queue.  
  **Queue:** $B, C, E, A, D, H$

- The first customer is $B$. We dequeue $B$ and process their order.  
  **Queue:** $C, E, A, D, H$

- While processing $B$, customer $J$ comes along and is added to the end of the queue (called *enqueueing*); no cuts allowed!  
  **Queue:** $C, E, A, D, H, J$

- In Computer Science, a queue is a data structure that works in the same way.
  - New items are added to the end of the queue (*enqueued*).
  - Items at the front of the line are *dequeued* and processed.
  - We don't have the complete sequence when we start. The queue grows and shrinks over time as items are enqueued and dequeued.
Breadth first search (BFS) in a connected graph

Pick a starting vertex. We’ll use $a$.

Add $a$ to the end of the queue, $Q$, and mark $a$ as **discovered**.

- The undiscovered nodes are white.
- The discovered nodes are pink. By hand, just make a small mark like a dot.
Breadth first search (BFS) in a connected graph

Dequeue a vertex: $u = a.$

Time stamp $u$: $T(a) = \text{time} = 1$
Breadth first search (BFS) in a connected graph

Add any undiscovered neighbors of $u$ to the end of the queue: $b, d, e$. This is called enqueueing.

Mark those neighbors as discovered (color them pink).

Add an edge from $u$ to each of those neighbors to the tree (color the edges red).
Breadth first search (BFS) in a connected graph

Mark \( a \) as \textit{finished} (we’ll color it red).

By hand: Don’t literally color vertices. Just make a mark (like a dot) to show discovery, and write the number to show it’s finished.

In software:
  - There’s a variable \textit{discovered} or \textit{visited} for each vertex.
  - Various algorithms using DFS or BFS use 0, 2, or 3 of these states.
Breadth first search (BFS) in a connected graph

Dequeue:

Time stamp $u$:
$s = b$

Undiscovered neighbors:
$c$
Dequeue:

Time stamp $u$:

Undiscovered neighbors:

Append them to queue, mark them discovered (pink), and add tree edges to them (red)
Breadth first search (BFS) in a connected graph

Dequeue:
- $u = b$

Time stamp $u$:
- $T(b) = \text{time} = 2$

Undiscovered neighbors:
- $c$

Append them to queue, mark them discovered (pink), and add tree edges to them (red)

Mark $u$ finished (red)
Breadth first search (BFS) in a connected graph

- **Dequeque:**
- **Time stamp** \( u \): \( u = d \)
- **Undiscovered neighbors:** \( f, g \)

**Q:** \( x, xxx, e, c \)

**u:** \( d \)
Breadth first search (BFS) in a connected graph

Dequeue: \( u = d \)

Time stamp \( u \): \( T(d) = \text{time} = 3 \)

Undiscovered neighbors: \( f, g \)

Append them to queue, mark them discovered (pink), and add tree edges to them (red)
Breadth first search (BFS) in a connected graph

Q: xxxxx, e, c, f, g
u: d

- Dequeue: \( u = d \)
- Time stamp \( u \): \( T(d) = \text{time} = 3 \)
- Undiscovered neighbors: \( f, g \)
- Append them to queue, mark them discovered (pink), and add tree edges to them (red)
- Mark \( u \) finished (red)
Continue until forced to stop (no element to dequeue).

- All vertices reachable from \( a \) are included in the tree. If it’s a connected graph, then it’s a spanning tree reaching all vertices.
The BFS queue is in weakly increasing order of distance from \( a \).

Instead of marking time stamps, we could have marked the distance from \( a \) (= parent's distance + 1 = depth in tree).

Processing vertices in order by layer assures we get shortest paths from \( a \), although there may be ties for the shortest path to each vertex, e.g., \( a, d, g \) and \( a, e, g \).
Distance (in edges) from a vertex to all other vertices

Let $N_i(v)$ be the set of all vertices of $G$ a distance $i$ from $v$:

- $N_0(a) = \{a\}$
- $N_1(a) = \{b, d, e\}$
- $N_2(a) = \{c, f, g, h\}$
- $N_3(a) = \{i, j\}$

We used breadth first search to compute that for $v = a$. 
DFS vs. BFS

**DFS from** $a$

- Depth $= 8$

**BFS from** $a$

- Depth $= 3$

- DFS tends to give longer paths, and to branch out less.
- BFS tends to give shorter paths, and to branch out more.
A maze can be represented by a graph.

- Vertex for each cell and for **Start** and **End**.
- Edge between adjacent cells w/o wall in-between.

Pick an ordering of neighbors of the cells:
- Could go up, left, down, or right one cell.
- Or, **best first**: Use a heuristic to guess best neighbor (may not be right). Order neighbors by distance to **End**, using Manhattan distance

\[
d(((x, y), (x', y')) = |x - x'| + |y - y'|
\]

Break ties with a rule like down, left, right, up.

- Use DFS or BFS. But instead of exploring all cells, you can stop when you reach the goal.
- DFS may be better since you need to go deep rather than to find the shortest solution.
Diameter and radius of an undirected graph

The eccentricity of a vertex is the largest distance from the vertex.
- Eccentricity of $A$ is 2.
- Eccentricity of $F$ is 3.

The BFS algorithm we just used gives all the distances from a vertex to the other vertices, so it can be used to compute this.
The **diameter** of a graph is maximum eccentricity: it’s the largest distance between two vertices.

The **radius** of a graph is the minimum eccentricity.

- **Left graph:** diameter 3 (from C to F); radius 2 (using G)
- **Right graph:** diameter 4 (from (0,0) to (2,2)); radius 2 (using (1,1)).
Is a connected graph bipartite?
First solution

- We’ll label all vertices as being in part $A$ or part $B$.
- Pick a starting vertex and label it $A$.
- Use DFS, BFS, or any other traversal that adds on one edge at a time to form a spanning tree.
- As you add new vertices, label them $A/B$ opposite of their parent.
- As you explore neighbors of $u$, if any neighbor $v$ was already discovered and has the same label $A/B$ as $u$, then it’s not bipartite.
- If it is bipartite, then $A$ and $B$ are its two parts.
Is a connected graph bipartite?

Second solution

- Pick any starting vertex, \( u \).
- Do BFS starting at \( u \), and set
  \[
  A = N_0(u) \cup N_2(u) \cup N_4(u) \cup \cdots \quad \text{(even distance)}
  \]
  \[
  B = N_1(u) \cup N_3(u) \cup N_5(u) \cup \cdots \quad \text{(odd distance)}
  \]
- In BFS, all edges either connect vertices at two consecutive layers, or two vertices in the same layer.
  E.g., if \( d(u, v) = 8 \) and \( \{v, w\} \) is an edge, then \( d(u, w) \in \{7, 8, 9\} \).
- If there is any edge between two vertices of \( A \) (or two vertices of \( B \)), then there is an odd-length cycle, so it’s not bipartite.
- Otherwise, the graph is bipartite and \( A \) and \( B \) are its two parts.
Is a connected graph bipartite?

Second solution

Vertices are marked with their distance from $a$, found from BFS.

There are edges between vertices at the same level (such as $\{c, f\}$ in $N_2(a)$, as well as others), so it’s not bipartite.
Is a disconnected graph bipartite?

- Apply either procedure starting from any vertex in each component.
- If any component isn’t bipartite, then the graph isn’t bipartite.
- Otherwise, $A$ and $B$ gives a bipartition of the graph.
- Inverting $A/B$ in any component gives other solutions.
Does a graph have a cycle?

- **Undirected simple graph**: In BFS or DFS, when checking the neighbors of a vertex, if any neighbor besides the parent was already discovered, then there’s a cycle.

- **Multigraphs/pseudographs**: Loops are cycles of length 1. Multiple edges give cycles of length 2. If there are no loops or multiple edges, use the test for a simple graph.

- **Directed graph**: DFS can be used to determine if there’s a directed cycle, but it’s more involved.
  - Use DFS with three vertex colors.
  - Initialize all vertices to white (not discovered).
  - When entering a vertex, color it pink (discovered).
  - It stays pink while recursing to its out-neighbors.
  - After returning from the recursion, color it red (finished).
  - When checking out-neighbors of a vertex, if any is pink (discovered), it must be an ancestor in the tree, so there’s a cycle.