If we keep track of what vertices we've visited (requires linear space), then we can perform two simple searches.

\[
\text{DFS}(G, s, t)
\]

1. If \( s = t \)
2. Return true
3. Mark \( s \) as visited
4. For each neighbor \( u \) of \( s \) we haven't visited
   
   If \( u \) is not visited
   
   \[
   \text{DFS}(G, u, t)
   \]
   
   Return True

Return False

First note that we can turn this into an iterative algorithm via a stack data structure.

\[
\text{DFS}(G, s, t)
\]

1. Let \( S \) be a stack.
2. Push \( s \) onto \( S \)
3. While \( S \) is not empty
   
   a. Pop \( v \) off of \( S \)
   b. Mark \( v \) as visited
   c. If \( v = t \)
      
      Return True
   
   For each neighbor \( u \) of \( v \) not visited
   
   Push \( u \) onto \( S \)

Return False

In either case, the space requirement is \( O(n) \) and since we explore each edge at most once and each vertex at most once, the runtime is \( O(V+E) \). It is important to use \( V+E \) here even though \( E \in O(V^2) \) b/c it demonstrates that DFS is better on sparse graphs.
With DFS, we explore all the way to dead ends before exploring neighbors.

**Ex)**

![Graph](graph.png)

Start DFS from A. Explore neighbors alphabetically:

A → B → E → F  C → G → D

If we instead explore neighbors first, we get a breadth-first search.

**BFS** \((G, s, t)\)

Let \(Q\) be a queue

Enqueue \(s\) into \(Q\)

while \(Q\) is not empty

Dequeue \(u\) from \(Q\)

Mark \(u\) visited

If \(u = t\)

Return true

For each neighbor \(v\) of \(u\),

Enqueue \(v\) into \(Q\)

Return False

Like DFS, we need linear space \((O(n))\) and visit each vertex/edge at most once, and thus run in \(O(V+E)\) time.
Ex)

Start BFS at A
Explore neighbors alphabetically
A → B → C → D → E → F → G → H

DFS/BFS is not just useful to answer reachability. We can perform operations as we go. Consider if we wanted to evaluate the expression \((4 + 8) \times 2 - 4 / 2\). We can build this expression as a tree of expressions consisting of constants and operators. For example, \(4 + 8\) is

```
+  
  
  4  8
```

All together, we get

```
-  
  
  +  
    
    4  2
    
    +  
      
      4  8
```

If we evaluate constants as is and operators as the operator of their children, then a DFS will do this for us.

```python
Evaluate(T, root)
  If r is constant
    Return r's data.
  Return map(Evaluate(r, left), Evaluate(r, right))
```

\(\text{(\# goes to leaves before \# evaluated; hence a DFS)}\)