Graph Algorithms: Depth-First Search (DFS)

Lecture 14A

Danqi Chen (Princeton University)

AddisCoder 2026

August 13, 2026

About me

About me

- Professor at **Princeton University** since 2019

About me

- Professor at **Princeton University** since 2019
- I work on **AI**, especially language models
  (tomorrow's lecture!)

About me

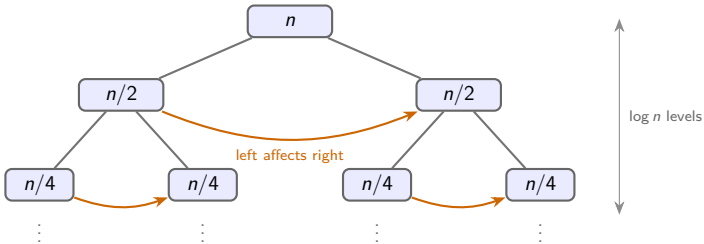
- Professor at **Princeton University** since 2019
- I work on **AI**, especially language models (tomorrow's lecture!)
- I started learning **programming and algorithms** in high school — and was a competitive programmer, many many years ago

About me

- Professor at **Princeton University** since 2019
- I work on **AI**, especially language models (tomorrow's lecture!)
- I started learning **programming and algorithms** in high school — and was a competitive programmer, many many years ago
- My **first time in Ethiopia** — I hope to come back often!

Fun fact: CDQ's divide and conquer

An **offline** divide-and-conquer algorithm for problems where **updates** and **queries** are interleaved over time.



Learn more:

robert1003.github.io/2020/01/31/cdq-divide-and-conquer.html

codeforces.com/blog/entry/150934

My lecture plan

Thursday	Morning	**14A**	**Depth-First Search (DFS)** ← we are here
	Afternoon	**14B**	Breadth-First Search (BFS)
Friday	Morning	**15A**	Quiz
	Afternoon	**15B**	Language Models

Yesterday: graphs in code

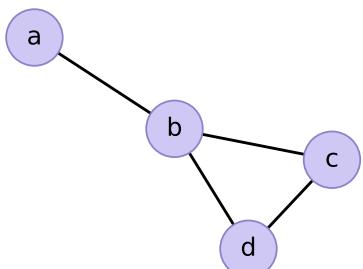
Recall that a graph G is made up of **vertices** V and **edges** E.

```
import networkx as nx

G = nx.Graph()
G.add_edges_from([
    ("a", "b"), ("b", "c"),
    ("d", "b"), ("d", "c")])
nx.draw(G, with_labels=True,
        node_size=1000,
        node_color='#cfc8f4')
```

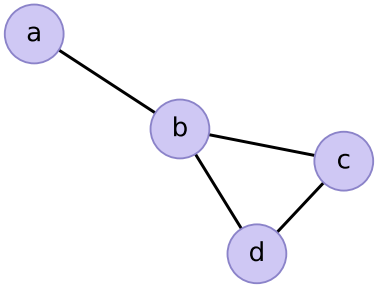
Yesterday: graphs in code

Recall that a graph G is made up of **vertices** V and **edges** E.

```
import networkx as nx

G = nx.Graph()
G.add_edges_from([
    ("a", "b"), ("b", "c"),
    ("d", "b"), ("d", "c")])
nx.draw(G, with_labels=True,
        node_size=1000,
        node_color='#cfc8f4')
```

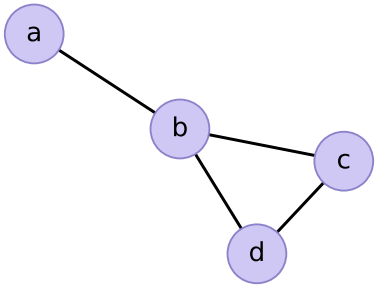


. . . but how do we store it?




```
graph = {"a": ["b"],
         "b": ["a", "c", "d"],
         "c": ["b", "d"],
         "d": ["b", "c"]}
```

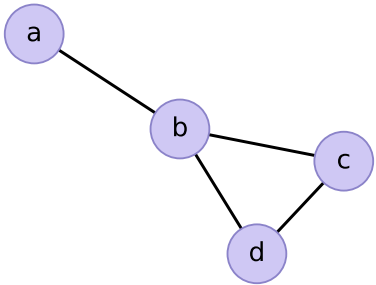
. . . but how do we store it?




```
graph = {"a": ["b"],
         "b": ["a", "c", "d"],
         "c": ["b", "d"],
         "d": ["b", "c"]}
```

- An **adjacency dictionary**: each vertex maps to its **neighbors**

. . . but how do we store it?




```
graph = {"a": ["b"],
         "b": ["a", "c", "d"],
         "c": ["b", "d"],
         "d": ["b", "c"]}
```

- An **adjacency dictionary**: each vertex maps to its **neighbors**
- **Undirected**, so it is symmetric: `"a"` lists `"b"`, and `"b"` lists `"a"`

Directed graphs

```
G = nx.DiGraph()
G.add_edges_from([
    ("a", "b"), ("b", "c"),
    ("d", "b"), ("d", "c")])
```

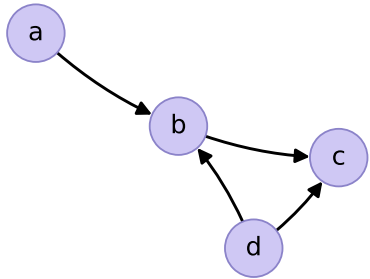
Directed graphs

```
G = nx.DiGraph()
G.add_edges_from([
    ("a", "b"), ("b", "c"),
    ("d", "b"), ("d", "c")])
```

Directed graphs

```
G = nx.DiGraph()
G.add_edges_from([
    ("a", "b"), ("b", "c"),
    ("d", "b"), ("d", "c")])
```

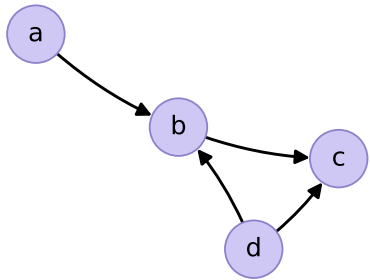
```
graph = {"a": ["b"],
         "b": ["c"],
         "c": [],
         "d": ["b", "c"]}
```



Directed graphs

```
G = nx.DiGraph()
G.add_edges_from([
    ("a", "b"), ("b", "c"),
    ("d", "b"), ("d", "c")])
```

```
graph = {"a": ["b"],
         "b": ["c"],
         "c": [],
         "d": ["b", "c"]}
```



- `nx.DiGraph()` instead of `nx.Graph()` — same edges, now one-way
- No longer symmetric: `"a"` lists `"b"`, but `"b"` does **not** list `"a"`

Reachability

Reachability is one of the most important properties of a graph:

Reachability

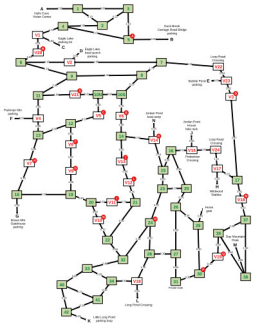
Reachability is one of the most important properties of a graph:

*Can I get from **vertex** i to **vertex** j?*

Reachability

Reachability is one of the most important properties of a graph:

*Can I get from **vertex** i to **vertex** j?*

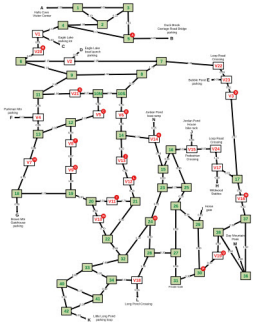


Is there a road route from Addis Ababa to Nairobi?

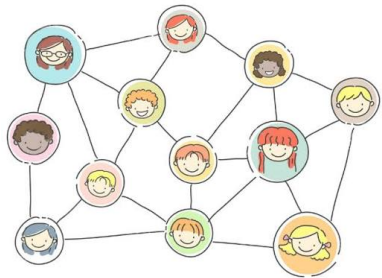
Reachability

Reachability is one of the most important properties of a graph:

*Can I get from **vertex** i to **vertex** j?*



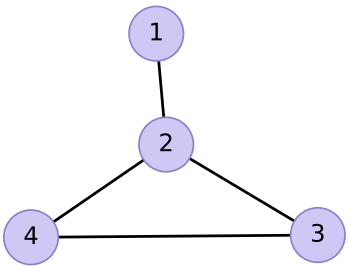
Is there a road route from Addis Ababa to Nairobi?



Are two people connected through friends of friends?

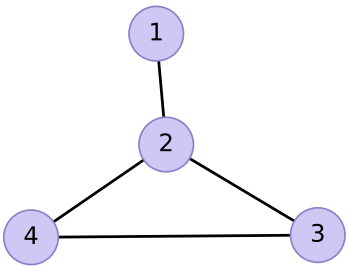
Reachability, precisely

- Vertex j is **reachable** from vertex i if there is a **path** from i to j



Reachability, precisely

- Vertex j is **reachable** from vertex i if there is a **path** from i to j

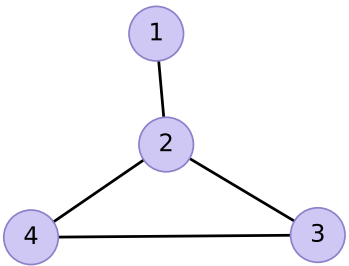


Discuss:

Is vertex 4 reachable from vertex 1?

Reachability, precisely

- Vertex j is **reachable** from vertex i if there is a **path** from i to j



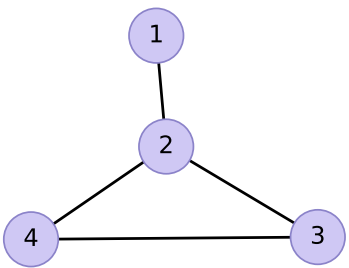
Discuss:

Is vertex 4 reachable from vertex 1?

Yes.

Reachability, precisely

- Vertex j is **reachable** from vertex i if there is a **path** from i to j



Discuss:

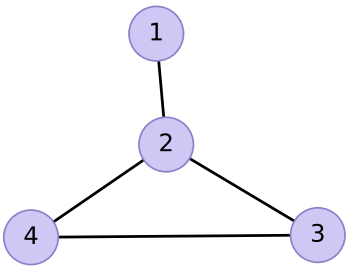
Is vertex 4 reachable from vertex 1?

Yes.

Is vertex 1 reachable from vertex 4?

Reachability, precisely

- Vertex j is **reachable** from vertex i if there is a **path** from i to j



Discuss:

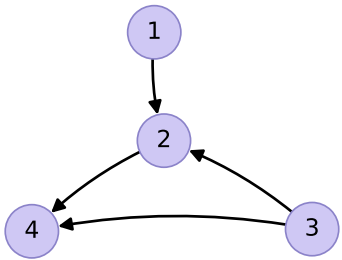
Is vertex 4 reachable from vertex 1?

Yes.

Is vertex 1 reachable from vertex 4?

Yes.

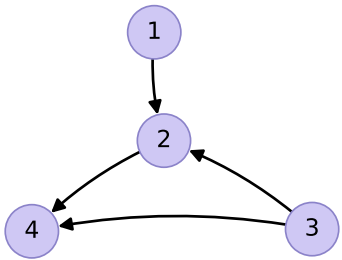
Applies to directed graphs too



Discuss:

Is vertex 4 reachable from vertex 1?

Applies to directed graphs too

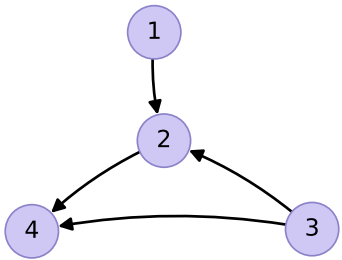


Discuss:

Is vertex 4 reachable from vertex 1?

Yes.

Applies to directed graphs too



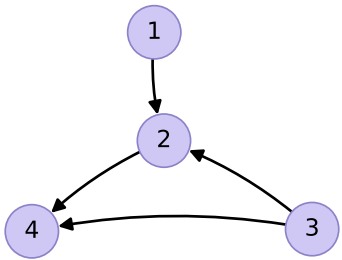
Discuss:

Is vertex 4 reachable from vertex 1?

Yes.

Is vertex 1 reachable from vertex 4?

Applies to directed graphs too



Discuss:

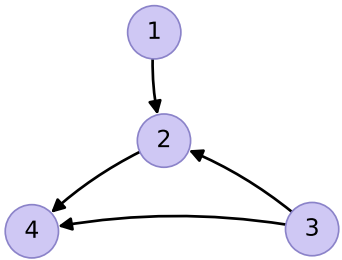
Is vertex 4 reachable from vertex 1?

Yes.

Is vertex 1 reachable from vertex 4?

No.

Applies to directed graphs too



Discuss:

Is vertex 4 reachable from vertex 1?

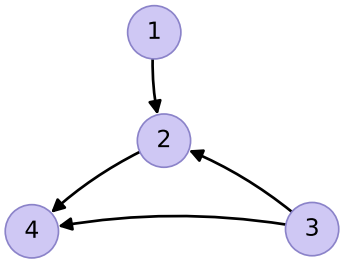
Yes.

Is vertex 1 reachable from vertex 4?

No.

*On a directed graph, reachability is **not symmetric**.*

Applies to directed graphs too



Discuss:

Is vertex 4 reachable from vertex 1?

Yes.

Is vertex 1 reachable from vertex 4?

No.

*On a directed graph, reachability is **not symmetric**.*

But how do you know if vertex j is reachable from vertex i?

How would a computer check this?

- You just traced paths **with your eyes** — a computer cannot do that

How would a computer check this?

- You just traced paths **with your eyes** — a computer cannot do that

> *One natural algorithm we can think about here is that two vertices are **connected** if i **is** j, or if there is some **neighbor** k of i (`k in G[i]`) such that k is **connected** to j.*

How would a computer check this?

- You just traced paths **with your eyes** — a computer cannot do that

> *One natural algorithm we can think about here is that two vertices are **connected** if i **is** j, or if there is some **neighbor** k of i (`k in G[i]`) such that k is **connected** to j.*

- A definition in terms of itself... that's **recursion**!

Let's turn that idea into code

```
def connected(i, j, G):
```

Let's turn that idea into code

```
def connected(i, j, G):
    if i == j:
```

Let's turn that idea into code

```
def connected(i, j, G):
   if i == j:
      return True
```

Let's turn that idea into code

```
def connected(i, j, G):
    if i == j:
        return True
    for k in G[i]:
```

Let's turn that idea into code

```
def connected(i, j, G):
    if i == j:
        return True
    for k in G[i]:
        if connected(k, j, G):
```

Let's turn that idea into code

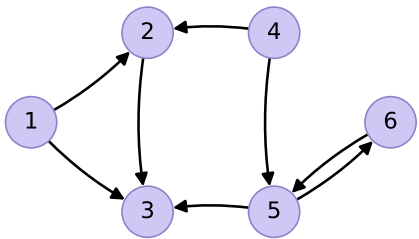
```
def connected(i, j, G):
    if i == j:
        return True
    for k in G[i]:
        if connected(k, j, G):
            return True
```

Let's turn that idea into code

```
def connected(i, j, G):
  if i == j:
    return True
  for k in G[i]:
    if connected(k, j, G):
      return True
  return False
```

Our test graph

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```



What will this return?

```
def connected(i, j, G):
    if i == j:
        return True
    for k in G[i]:
        if connected(k, j, G):
            return True
    return False
```

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```

What will this return?

```
def connected(i, j, G):
    if i == j:
        return True
    for k in G[i]:
        if connected(k, j, G):
            return True
    return False
```

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```

Discuss: What will `connected(5, 2, g_connected)` return — `True`, `False`, or something else?

Let's follow the calls

`connected(5, 2, G)` $5 \neq 2$, and `G[5] = [3, 6]`

Let's follow the calls

```
connected(5, 2, G)
   connected(3, 2, G)
```

$5 \neq 2$, and `G[5] = [3, 6]`

`G[3] = []` $\Rightarrow$ `False`

Let's follow the calls

```
connected(5, 2, G)
    connected(3, 2, G)
    connected(6, 2, G)
```

$5 \neq 2$, and `G[5] = [3, 6]`

`G[3] = []` $\Rightarrow$ `False`

next one in `G[5]`...

Let's follow the calls

`connected(5, 2, G)`	$5 \neq 2$, and `G[5] = [3, 6]`
`connected(3, 2, G)`	`G[3] = []` $\Rightarrow$ `False`
`connected(6, 2, G)`	next one in `G[5]`...
`connected(5, 2, G)`	`G[6] = [5]` — **we've been here before!**

Let's follow the calls

`connected(5, 2, G)`	$5 \neq 2$, and `G[5] = [3, 6]`
`connected(3, 2, G)`	`G[3] = []` $\Rightarrow$ `False`
`connected(6, 2, G)`	next one in `G[5]`...
`connected(5, 2, G)`	`G[6] = [5]` — **we've been here before!**
`connected(3, 2, G)`	`False` again

Let's follow the calls

`connected(5, 2, G)`	$5 \neq 2$, and `G[5] = [3, 6]`
`connected(3, 2, G)`	`G[3] = []` $\Rightarrow$ `False`
`connected(6, 2, G)`	next one in `G[5]`...
`connected(5, 2, G)`	`G[6] = [5]` — **we've been here before!**
`connected(3, 2, G)`	`False` again
`connected(6, 2, G)`	

Let's follow the calls

`connected(5, 2, G)`	$5 \neq 2$, and `G[5] = [3, 6]`
`connected(3, 2, G)`	`G[3] = []` $\Rightarrow$ `False`
`connected(6, 2, G)`	next one in `G[5]`...
`connected(5, 2, G)`	`G[6] = [5]` — **we've been here before!**
`connected(3, 2, G)`	`False` again
`connected(6, 2, G)`	
`connected(5, 2, G)`	...and again

Let's follow the calls

`connected(5, 2, G)`	$5 \neq 2$, and `G[5] = [3, 6]`
`connected(3, 2, G)`	`G[3] = []` $\Rightarrow$ `False`
`connected(6, 2, G)`	next one in `G[5]`...
`connected(5, 2, G)`	`G[6] = [5]` — **we've been here before!**
`connected(3, 2, G)`	`False` again
`connected(6, 2, G)`	
`connected(5, 2, G)`	...and again
⋮	

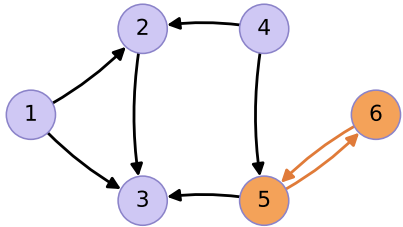
Let's follow the calls

`connected(5, 2, G)`	$5 \neq 2$, and `G[5] = [3, 6]`
`connected(3, 2, G)`	`G[3] = []` $\Rightarrow$ `False`
`connected(6, 2, G)`	next one in `G[5]`...
`connected(5, 2, G)`	`G[6] = [5]` — **we've been here before!**
`connected(3, 2, G)`	`False` again
`connected(6, 2, G)`	
`connected(5, 2, G)`	...and again
⋮	

*The calls **never stop**. Python gives up:* `RecursionError`.

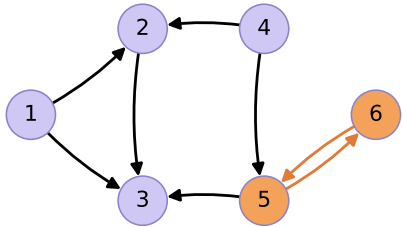
What went wrong?

- 5 and 6 point at **each other**



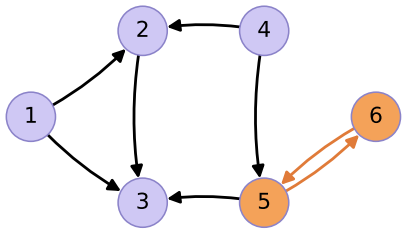
What went wrong?

- 5 and 6 point at **each other**
- The function **never remembers** where it has already been



What went wrong?

- 5 and 6 point at **each other**
- The function **never remembers** where it has already been



Discuss: How would **you** avoid walking in circles in a maze?

The fix: remember where you've been

```
def connected(i, j, G, visited):
    if i == j:
        return True
    visited[i] = True    # leave a breadcrumb
    for k in G[i]:
        if k not in visited:    # skip where we've been
            if connected(k, j, G, visited):
                return True
    return False
```

The fix: remember where you've been

```
def connected(i, j, G, visited):
    if i == j:
        return True
    visited[i] = True    # leave a breadcrumb
    for k in G[i]:
        if k not in visited:    # skip where we've been
            if connected(k, j, G, visited):
                return True
    return False
```

- Two new lines — that's the whole fix

Does it work?

```
def connected(i, j, G, visited):
    if i == j:
        return True
    visited[i] = True
    for k in G[i]:
        if k not in visited:
            if connected(k, j, G, visited):
                return True
    return False
```

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```

Does it work?

```
def connected(i, j, G, visited):
    if i == j:
        return True
    visited[i] = True
    for k in G[i]:
        if k not in visited:
            if connected(k, j, G, visited):
                return True
    return False
```

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```

```
connected(5, 2, g_connected, {})
```

Does it work?

```
def connected(i, j, G, visited):
    if i == j:
        return True
    visited[i] = True
    for k in G[i]:
        if k not in visited:
            if connected(k, j, G, visited):
                return True
    return False
```

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```

`connected(5, 2, g_connected, {})` $\Rightarrow$ **`False`**

Does it work?

```
def connected(i, j, G, visited):
    if i == j:
        return True
    visited[i] = True
    for k in G[i]:
        if k not in visited:
            if connected(k, j, G, visited):
                return True
    return False
```

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```

`connected(5, 2, g_connected, {})` $\Rightarrow$ **False**

`connected(5, 6, g_connected, {})`

Does it work?

```
def connected(i, j, G, visited):
    if i == j:
        return True
    visited[i] = True
    for k in G[i]:
        if k not in visited:
            if connected(k, j, G, visited):
                return True
    return False
```

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```

`connected(5, 2, g_connected, {})` $\Rightarrow$ **False**

`connected(5, 6, g_connected, {})` $\Rightarrow$ **True**

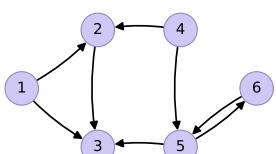
Does it work?

```
def connected(i, j, G, visited):
    if i == j:
        return True
    visited[i] = True
    for k in G[i]:
        if k not in visited:
            if connected(k, j, G, visited):
                return True
    return False
```

```
g_connected = {1: [2, 3],
               2: [3],
               4: [2, 5],
               3: [],
               5: [3, 6],
               6: [5]}
```

`connected(5, 2, g_connected, {})` $\Rightarrow$ **False**

`connected(5, 6, g_connected, {})` $\Rightarrow$ **True**



Depth-first search

The `connected` function we just implemented is better known as

depth-first search (DFS)

Depth-first search

The `connected` function we just implemented is better known as

depth-first search (DFS)

- DFS is a **graph traversal** algorithm: it lets us move through every node of a graph in a structured way

Depth-first search

The `connected` function we just implemented is better known as

depth-first search (DFS)

- DFS is a **graph traversal** algorithm: it lets us move through every node of a graph in a structured way
- To do that, we have to **keep track of the nodes we have already seen** — so we know which ones are still left to visit

DFS, step by step

1. Start at a node

DFS, step by step

1. Start at a node
2. Mark it as **visited**

DFS, step by step

1. Start at a node
2. Mark it as **visited**
3. Pick one unvisited neighbor and **go as deep as possible**

DFS, step by step

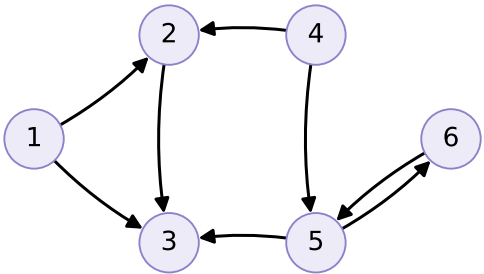
1. Start at a node
2. Mark it as **visited**
3. Pick one unvisited neighbor and **go as deep as possible**
4. When stuck, **backtrack** and try the next neighbor

DFS, step by step

1. Start at a node
2. Mark it as **visited**
3. Pick one unvisited neighbor and **go as deep as possible**
4. When stuck, **backtrack** and try the next neighbor

Go deep first. Back up only when you must.

Let's trace it by hand: DFS from 4

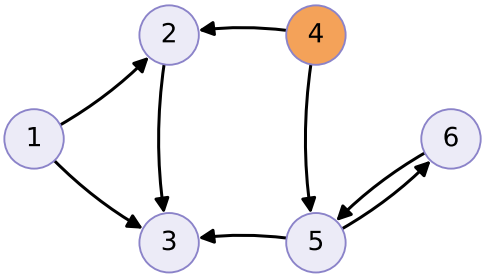


Now: start at 4

Call stack:

`visited: {}`

Let's trace it by hand: DFS from 4

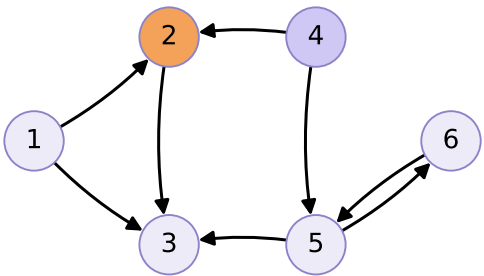


Now: visit 4 — `G[4] = [2, 5]`

Call stack: `4`

`visited`: `{4}`

Let's trace it by hand: DFS from 4

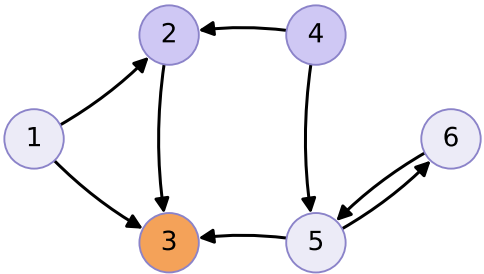


Now: visit 2 — `G[2] = [3]`

Call stack: `4, 2`

`visited`: `{4, 2}`

Let's trace it by hand: DFS from 4

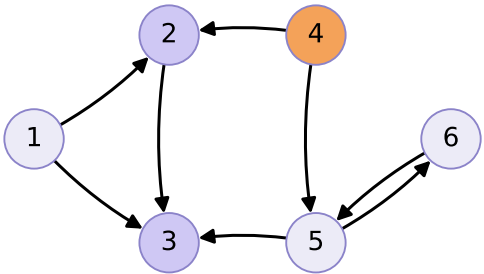


Now: visit 3 — `G[3] = []`

Call stack: `4, 2, 3`

`visited`: `{4, 2, 3}`

Let's trace it by hand: DFS from 4

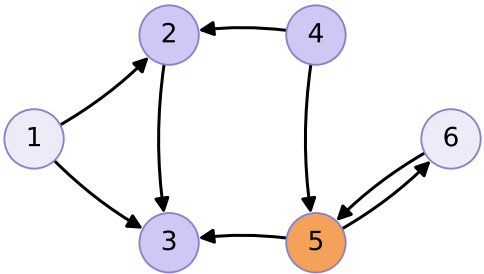


Now: `G[3] = []` → backtrack

Call stack: `4`

`visited`: `{4, 2, 3}`

Let's trace it by hand: DFS from 4

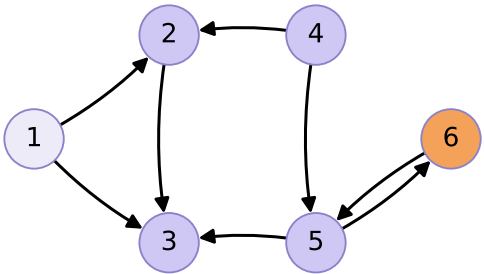


Now: visit 5 — `G[5] = [3, 6]`

Call stack: `4, 5`

`visited`: `{4, 2, 3, 5}`

Let's trace it by hand: DFS from 4

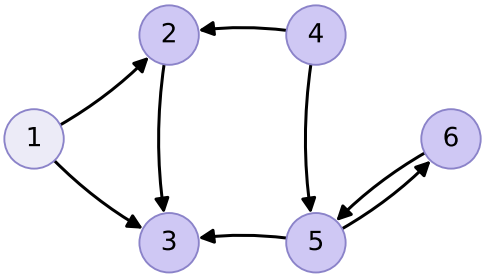


Now: 3 visited → visit 6

Call stack: `4, 5, 6`

`visited`: `{4, 2, 3, 5, 6}`

Let's trace it by hand: DFS from 4

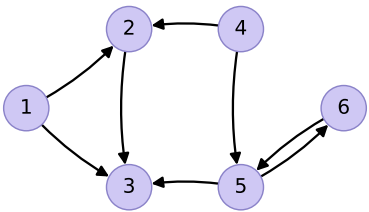


Now: 5 visited → done!

Call stack:

`visited:` {4, 2, 3, 5, 6}

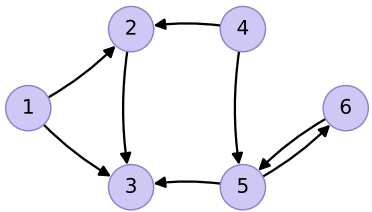
What if we start at 1?



Discuss:

In what order does DFS visit the nodes?

What if we start at 1?

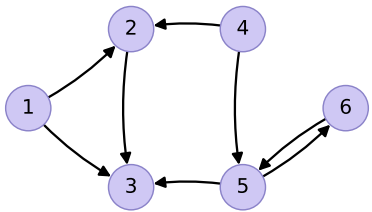


Discuss:

In what order does DFS visit the nodes?

1, 2, 3

What if we start at 1?



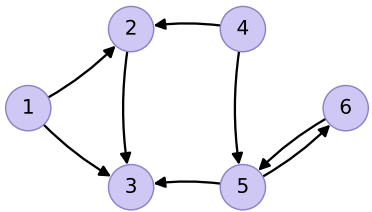
Discuss:

In what order does DFS visit the nodes?

1, 2, 3

Which nodes are **never** visited?

What if we start at 1?



Discuss:

In what order does DFS visit the nodes?

1, 2, 3

Which nodes are **never** visited?

4, 5, 6

Let's implement DFS

Converting graph edges to an adjacency dictionary

```
# convert graph edges to adjacency dictionary
def convert_adjacency(edges):
    res = {}
    for i in edges:
        res[i[0]] = []
        res[i[1]] = []
    for i in edges:
        res[i[0]] += [i[1]]
    return res
```

Converting graph edges to an adjacency dictionary

```
# convert graph edges to adjacency dictionary
def convert_adjacency(edges):
    res = {}
    for i in edges:
        res[i[0]] = []
        res[i[1]] = []
    for i in edges:
        res[i[0]] += [i[1]]
    return res
```

```
g = [[1, 2], [2, 3], [4, 2], [1, 3], [5, 3], [4, 5], [5, 6], [6, 5]]
g_adj = convert_adjacency(g)
print(g_adj)
# {1: [2, 3], 2: [3], 3: [], 4: [2, 5], 5: [3, 6], 6: [5]}
```

Depth-first search in code

```
def dfs(visited, graph, node):
```

Depth-first search in code

```
def dfs(visited, graph, node):
   visited[node] = True    # leave a breadcrumb
```

Depth-first search in code

```
def dfs(visited, graph, node):
   visited[node] = True    # leave a breadcrumb
   print(node)
```

Depth-first search in code

```
def dfs(visited, graph, node):
   visited[node] = True    # leave a breadcrumb
   print(node)
   for neighbor in graph[node]:
```

Depth-first search in code

```
def dfs(visited, graph, node):
   visited[node] = True    # leave a breadcrumb
   print(node)
   for neighbor in graph[node]:
      if neighbor not in visited:   # skip where we've been
```

Depth-first search in code

```
def dfs(visited, graph, node):
  visited[node] = True    # leave a breadcrumb
  print(node)
  for neighbor in graph[node]:
    if neighbor not in visited:   # skip where we've been
      dfs(visited, graph, neighbor)    # go deeper
```

Depth-first search in code

```
def dfs(visited, graph, node):
  visited[node] = True    # leave a breadcrumb
  print(node)
  for neighbor in graph[node]:
    if neighbor not in visited:   # skip where we've been
      dfs(visited, graph, neighbor)    # go deeper
```

Six lines — the same shape as `connected`, but with no target `j` it ***visits everything*** *it can reach.*

Running it from every node

```
# g_adj = {1: [2, 3], 2: [3], 3: [],
#          4: [2, 5], 5: [3, 6], 6: [5]}

for i in g_adj.keys():
    visited = {}
    print("DFS from", i)
    dfs(visited, g_adj, i)
```

Running it from every node

```
# g_adj = {1: [2, 3], 2: [3], 3: [],
#          4: [2, 5], 5: [3, 6], 6: [5]}

for i in g_adj.keys():
    visited = {}
    print("DFS from", i)
    dfs(visited, g_adj, i)
```

```
DFS from 1
1
2
3
DFS from 2
2
3
DFS from 3
3
DFS from 4
4
2
3
5
6
DFS from 5
5
3
6
DFS from 6
6
5
3
```

DFS time complexity

Discuss: How many times does DFS **visit** each vertex? How many times does it **look at** each edge?

DFS time complexity

Discuss: How many times does DFS **visit** each vertex? How many times does it **look at** each edge?

- Each **vertex**: visited once — `visited` makes sure of that

DFS time complexity

Discuss: How many times does DFS **visit** each vertex? How many times does it **look at** each edge?

- Each **vertex**: visited once — `visited` makes sure of that
- Each **edge**: looked at once — when we scan its start's neighbor list

DFS time complexity

Discuss: How many times does DFS **visit** each vertex? How many times does it **look at** each edge?

- Each **vertex**: visited once — `visited` makes sure of that
- Each **edge**: looked at once — when we scan its start's neighbor list
- Total work: $O(V + E)$ (V vertices, E edges)

Applications of DFS

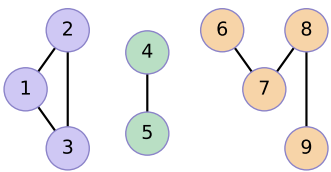
DFS has many applications. Some examples are:

1. **Finding a path** between two points u and v — that was `connected`

Applications of DFS

DFS has many applications. Some examples are:

1. **Finding a path** between two points u and v — that was `connected`
2. **Connected components** of a graph



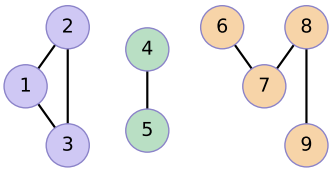
Run DFS from a node: everything it reaches is one group. Three here.

Applications of DFS

DFS has many applications. Some examples are:

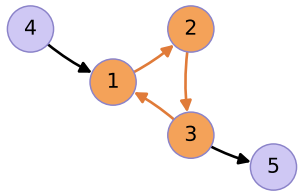
1. **Finding a path** between two points u and v — that was `connected`

2. **Connected components** of a graph



Run DFS from a node: everything it reaches is one group. Three here.

3. **Detecting cycles** in a directed graph



If DFS reaches a node still on the call stack — there's a cycle.

Today and this afternoon

Today:

- DFS explores **deep first**, backtracks when stuck
- `visited` prevents infinite loops
- Runs in $O(V + E)$

Today and this afternoon

Today:

- DFS explores **deep first**, backtracks when stuck
- `visited` prevents infinite loops
- Runs in $O(V + E)$

This afternoon:

- A search that explores **level by level** instead
- Almost the same code — **one** data structure changes