Graph Algorithms: Breadth-First Search (BFS)

Lecture 14B

Danqi Chen (Princeton University)

AddisCoder 2026

August 13, 2026

Lecture plan

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

This morning: depth-first search

- Go **as deep as possible**,
  backtrack when stuck

This morning: depth-first search

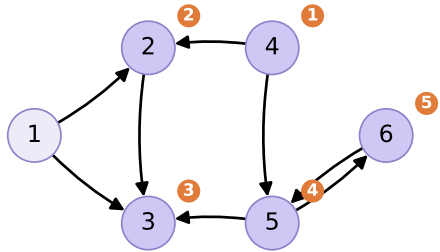
- Go **as deep as possible**, backtrack when stuck
- `visited` stops us walking in circles

This morning: depth-first search

- Go **as deep as possible**, backtrack when stuck
- `visited` stops us walking in circles
- Runs in $O(V + E)$

This morning: depth-first search

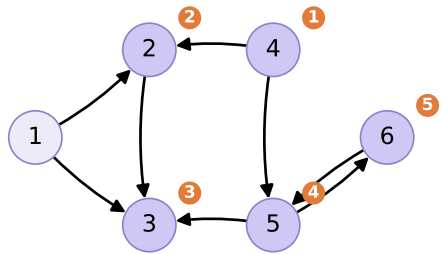
- Go **as deep as possible**, backtrack when stuck
- `visited` stops us walking in circles
- Runs in $O(V + E)$



DFS from 4: 4 2 3 5 6

This morning: depth-first search

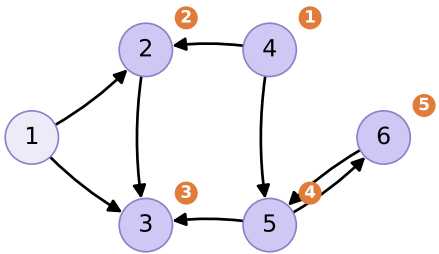
- Go **as deep as possible**, backtrack when stuck
- `visited` stops us walking in circles
- Runs in $O(V + E)$



DFS from 4: 4 2 3 5 6

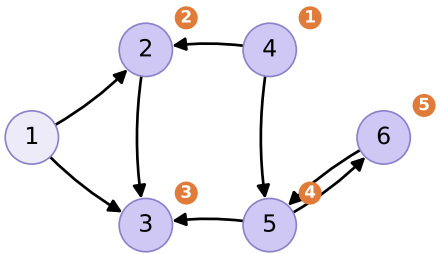
*This afternoon: a search that explores **level by level** instead — almost the same code, **one** data structure changes.*

Is there another sensible order?



DFS from 4: 4 2 3 5 6

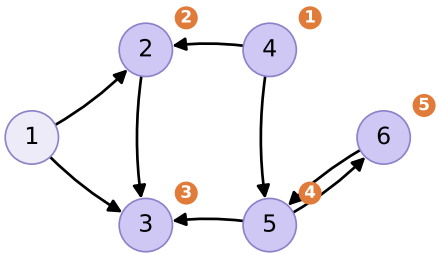
Is there another sensible order?



DFS from 4: 4 2 3 5 6

- DFS went **4** $\rightarrow$ **2** $\rightarrow$ **3**, all the way down, before it ever looked at **5**

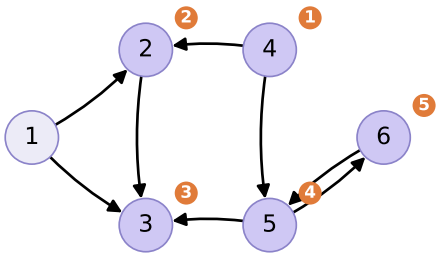
Is there another sensible order?



DFS from 4: 4 2 3 5 6

- DFS went **4** $\rightarrow$ **2** $\rightarrow$ **3**, all the way down, before it ever looked at **5**
- But 2 and 5 are both **one step** from 4

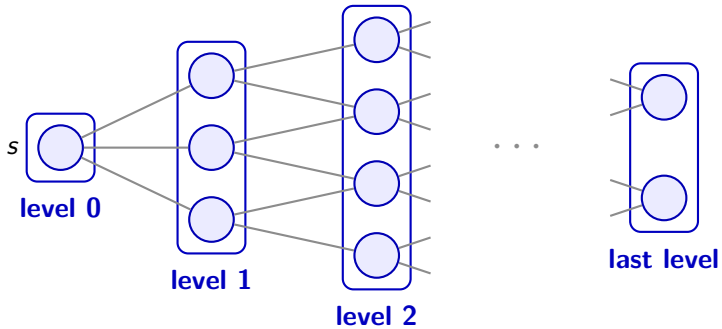
Is there another sensible order?



DFS from 4: 4 2 3 5 6

- DFS went **4** → **2** → **3**, all the way down, before it ever looked at **5**
- But 2 and 5 are both **one step** from 4
- **Why not visit both of them first?**

Breadth-first search: level by level



*Visit everything **one step** away,*
*then everything **two steps** away, and so on.*

The BFS algorithm

1. Start at the **starting node** of the graph

The BFS algorithm

1. Start at the **starting node** of the graph
2. Visit all nodes at a particular **level**

The BFS algorithm

1. Start at the **starting node** of the graph
2. Visit all nodes at a particular **level**
3. After visiting all nodes at that level, move to the **next level**

The BFS algorithm

1. Start at the **starting node** of the graph
2. Visit all nodes at a particular **level**
3. After visiting all nodes at that level, move to the **next level**
4. Repeat until all nodes are visited

The BFS algorithm

1. Start at the **starting node** of the graph
2. Visit all nodes at a particular **level**
3. After visiting all nodes at that level, move to the **next level**
4. Repeat until all nodes are visited

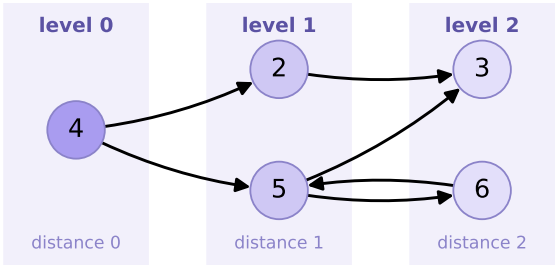
- Just like DFS, we keep a `visited` dictionary

The BFS algorithm

1. Start at the **starting node** of the graph
2. Visit all nodes at a particular **level**
3. After visiting all nodes at that level, move to the **next level**
4. Repeat until all nodes are visited

- Just like DFS, we keep a `visited` dictionary
- But to know **what to visit next**, we now need a **queue**

BFS on our graph: what order?

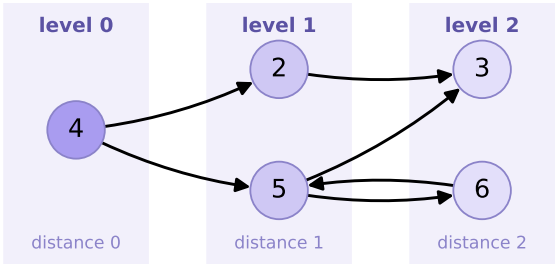


The same graph, redrawn by level. Vertex 1 is not reachable from 4, so it isn't here.

Discuss:

In what order does **BFS from 4** visit the nodes?

BFS on our graph: what order?



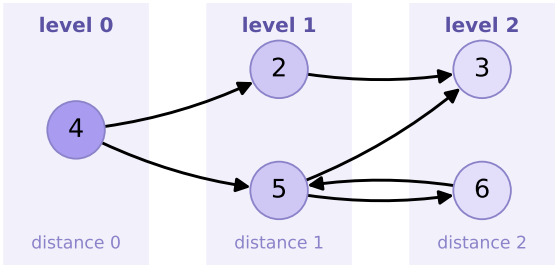
The same graph, redrawn by level. Vertex 1 is not reachable from 4, so it isn't here.

Discuss:

In what order does **BFS from 4** visit the nodes?

4 2 5 3 6

BFS on our graph: what order?



The same graph, redrawn by level. Vertex 1 is not reachable from 4, so it isn't here.

Discuss:

In what order does **BFS from 4** visit the nodes?

4 2 5 3 6

DFS said 4 2 3 5 6 — the same nodes, a different order.

What actually has to change?

- DFS never had a list of "places still to visit"... or did it?

What actually has to change?

- DFS never had a list of "places still to visit"... or did it?
- Every unfinished recursive call was waiting on the **call stack**

What actually has to change?

- DFS never had a list of "places still to visit"... or did it?
- Every unfinished recursive call was waiting on the **call stack**
- A stack gives back the **most recent** thing you put in — so DFS always carries on from the node it **just** arrived at, going **deeper and deeper**

What actually has to change?

- DFS never had a list of "places still to visit"... or did it?
- Every unfinished recursive call was waiting on the **call stack**
- A stack gives back the **most recent** thing you put in — so DFS always carries on from the node it **just** arrived at, going **deeper and deeper**

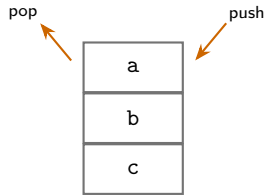
*To go level by level, we need the **opposite**: give back the **oldest** thing first. That data structure is a **queue**.*

Queues

First in, first out

Stack vs. queue

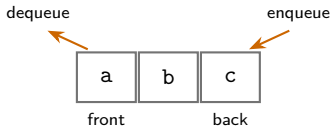
Stack (used by DFS)



Last **I**n, **F**irst **O**ut

In and out at the **same** end

Queue (used by BFS)

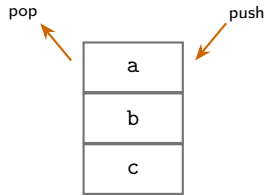


First **I**n, **F**irst **O**ut

In at one end, out at the **other**

Stack vs. queue

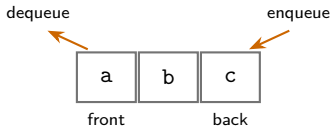
Stack (used by DFS)



Last **I**n, **F**irst **O**ut

In and out at the **same** end

Queue (used by BFS)



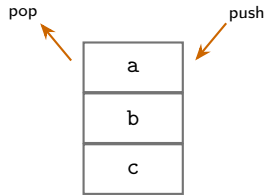
First **I**n, **F**irst **O**ut

In at one end, out at the **other**

Discuss: Which one is the queue you **stand in** at a shop?

Stack vs. queue

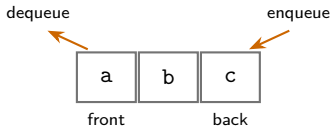
Stack (used by DFS)



Last **I**n, **F**irst **O**ut

In and out at the **same** end

Queue (used by BFS)



First **I**n, **F**irst **O**ut

In at one end, out at the **other**

Discuss: Which one is the queue you **stand in** at a shop?

And which one is a **pile of plates**?

Queues in Python: deque

```
# In Python, we can represent
# queues using deque:
from collections import deque

arr = [5, 7, 1, 8, 2, 10]
queue = deque(arr)
print(queue)
```

deque is short for **d**ouble-**e**nded **que**ue — Python's built-in queue

Queues in Python: deque

```
# In Python, we can represent
# queues using deque:
from collections import deque

arr = [5, 7, 1, 8, 2, 10]
queue = deque(arr)
print(queue)

front = queue.popleft()
print(front)
```

deque is short for **d**ouble-**e**nded **que**ue — Python's built-in queue

`popleft()`
take from the **front**

Queues in Python: deque

```
# In Python, we can represent
# queues using deque:
from collections import deque

arr = [5, 7, 1, 8, 2, 10]
queue = deque(arr)
print(queue)

front = queue.popleft()
print(front)

print(queue)
queue.append(3)
print(queue)
```

deque is short for **d**ouble-**e**nded **que**ue — Python's built-in queue

`popleft()`
take from the **front**

`append(x)`
add at the **back**

Queues in Python: deque

```
# In Python, we can represent
# queues using deque:
from collections import deque

arr = [5, 7, 1, 8, 2, 10]
queue = deque(arr)
print(queue)

front = queue.popleft()
print(front)

print(queue)
queue.append(3)
print(queue)
```

deque is short for **d**ouble-**e**nded **que**ue — Python's built-in queue

`popleft()`
take from the **front**

`append(x)`
add at the **back**

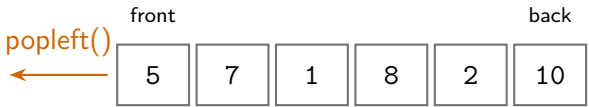
Discuss: What do the four `print()` lines put on the screen?

A queue, step by step


```
queue = deque(arr)
```

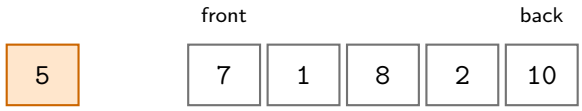
prints `deque([5, 7, 1, 8, 2, 10])`

A queue, step by step




```
front = queue.popleft()
```

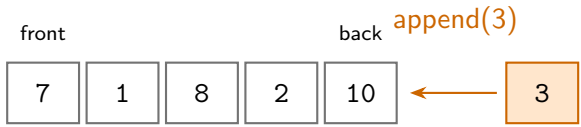
A queue, step by step



`print(front)` and `print(queue)`

prints `5` then `deque([7, 1, 8, 2, 10])`

A queue, step by step



`queue.append(3)`

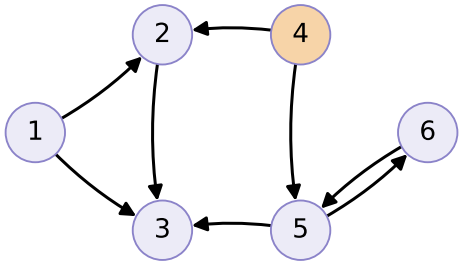
A queue, step by step

`print(queue)`

prints `deque([7, 1, 8, 2, 10, 3])`

*Out at the **front**, in at the **back** —*
and nothing ever jumps the queue.

Let's trace it by hand: BFS from 4



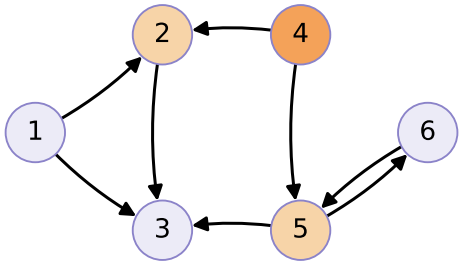
current in queue done not seen

Now: start at 4 — put it in the queue

`queue:` `[4]`

`visited:` `{4}`

Let's trace it by hand: BFS from 4



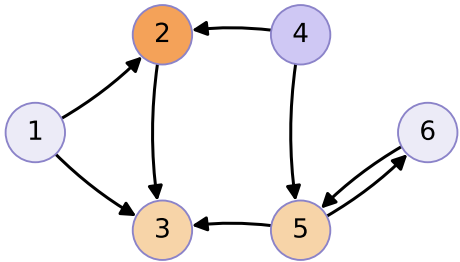
current in queue done not seen

Now: take **4** — `G[4] = [2, 5]`, add both

`queue: [2, 5]`

`visited: {4, 2, 5}`

Let's trace it by hand: BFS from 4



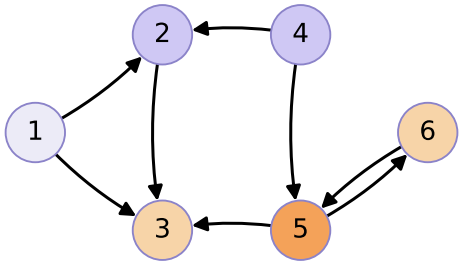
current in queue done not seen

Now: take **2** — `G[2] = [3]`, add 3

queue: `[5, 3]`

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

Let's trace it by hand: BFS from 4



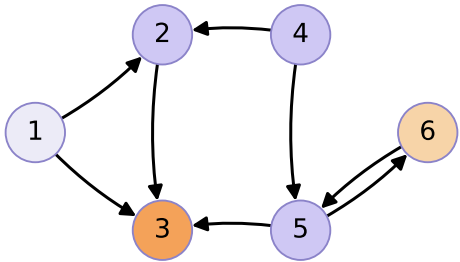
current in queue done not seen

Now: take **5** — 3 already seen, add 6

`queue:` `[3, 6]`

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

Let's trace it by hand: BFS from 4



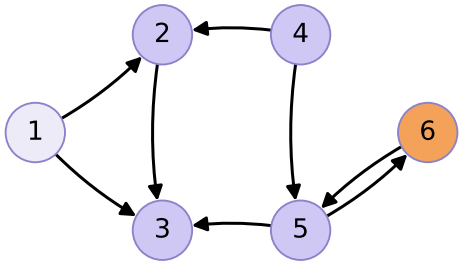
current in queue done not seen

Now: take **3** — `G[3] = []`, nothing to add

queue: `[6]`

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

Let's trace it by hand: BFS from 4



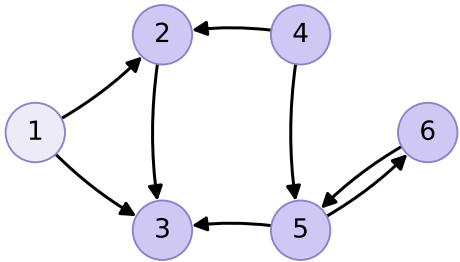
current in queue done not seen

Now: take **6** — 5 already seen

`queue:` `[]`

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

Let's trace it by hand: BFS from 4



current in queue done not seen

Now: queue empty → done!

queue: `[]`

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

Let's implement BFS

Reachability again — but breadth-first

```
def connected(source, target, G):
```

Reachability again — but breadth-first

```
def connected(source, target, G):
    visited = {}
    queue = deque([source])    # start with just the source
    visited[source] = True
```

Reachability again — but breadth-first

```
def connected(source, target, G):
    visited = {}
    queue = deque([source])    # start with just the source
    visited[source] = True
    while len(queue) > 0:    # still somewhere to go
```

Reachability again — but breadth-first

```
def connected(source, target, G):
   visited = {}
   queue = deque([source])    # start with just the source
   visited[source] = True
   while len(queue) > 0:    # still somewhere to go
      x = queue.popleft()      # take from the FRONT
```

Reachability again — but breadth-first

```
def connected(source, target, G):
    visited = {}
    queue = deque([source])    # start with just the source
    visited[source] = True
    while len(queue) > 0:    # still somewhere to go
        x = queue.popleft()      # take from the FRONT
        for y in G[x]:
```

Reachability again — but breadth-first

```
def connected(source, target, G):
   visited = {}
   queue = deque([source])    # start with just the source
   visited[source] = True
   while len(queue) > 0:    # still somewhere to go
      x = queue.popleft()      # take from the FRONT
      for y in G[x]:
         if y not in visited:
            visited[y] = True
            queue.append(y)    # join the BACK
```

Reachability again — but breadth-first

```
def connected(source, target, G):
   visited = {}
   queue = deque([source])    # start with just the source
   visited[source] = True
   while len(queue) > 0:    # still somewhere to go
      x = queue.popleft()      # take from the FRONT
      for y in G[x]:
         if y not in visited:
            visited[y] = True
            queue.append(y)    # join the BACK
         if y == target:
            return True
```

Reachability again — but breadth-first

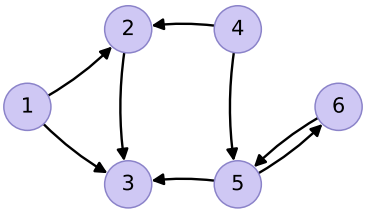
```
def connected(source, target, G):
    visited = {}
    queue = deque([source])    # start with just the source
    visited[source] = True
    while len(queue) > 0:    # still somewhere to go
        x = queue.popleft()      # take from the FRONT
        for y in G[x]:
            if y not in visited:
                visited[y] = True
                queue.append(y)    # join the BACK
            if y == target:
                return True
    return False
```

Reachability again — but breadth-first

```
def connected(source, target, G):
    visited = {}
    queue = deque([source])    # start with just the source
    visited[source] = True
    while len(queue) > 0:    # still somewhere to go
        x = queue.popleft()      # take from the FRONT
        for y in G[x]:
            if y not in visited:
                visited[y] = True
                queue.append(y)    # join the BACK
            if y == target:
                return True
    return False
```

No recursion — the nodes still to visit sit in a `queue` we can see.

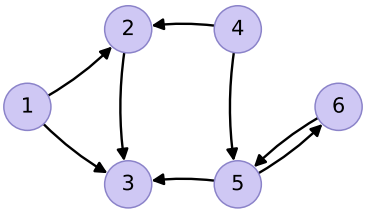
Does it work?



Discuss:

```
print(connected(1, 5, g_adj))
```

Does it work?

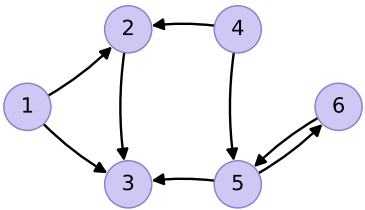


Discuss:

```
print(connected(1, 5, g_adj))
```

False

Does it work?



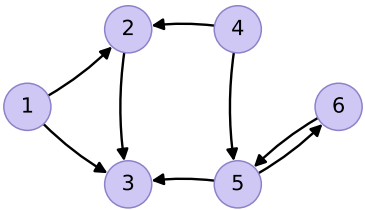
Discuss:

```
print(connected(1, 5, g_adj))
```

False

```
print(connected(4, 6, g_adj))
```

Does it work?



Discuss:

```
print(connected(1, 5, g_adj))
```

False

```
print(connected(4, 6, g_adj))
```

True

From searching to traversing

Drop the `target`, print on the way out — BFS now **visits everything** it can reach:

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

From searching to traversing

Drop the `target`, print on the way out — BFS now **visits everything** it can reach:

```
def bfs(visited, graph, node):
  queue = deque([node])     # the only node we know
  visited[node] = True
```

From searching to traversing

Drop the `target`, print on the way out — BFS now **visits everything** it can reach:

```
def bfs(visited, graph, node):
  queue = deque([node])     # the only node we know
  visited[node] = True
  while len(queue) > 0:
```

From searching to traversing

Drop the `target`, print on the way out — BFS now **visits everything** it can reach:

```
def bfs(visited, graph, node):
  queue = deque([node])     # the only node we know
  visited[node] = True
  while len(queue) > 0:
    x = queue.popleft()
    print(x)       # print on the way OUT
```

From searching to traversing

Drop the `target`, print on the way out — BFS now **visits everything** it can reach:

```
def bfs(visited, graph, node):
  queue = deque([node])     # the only node we know
  visited[node] = True
  while len(queue) > 0:
    x = queue.popleft()
    print(x)       # print on the way OUT
    for y in graph[x]:
```

From searching to traversing

Drop the `target`, print on the way out — BFS now **visits everything** it can reach:

```
def bfs(visited, graph, node):
  queue = deque([node])     # the only node we know
  visited[node] = True
  while len(queue) > 0:
    x = queue.popleft()
    print(x)        # print on the way OUT
    for y in graph[x]:
      if y not in visited:
        visited[y] = True
        queue.append(y)
```

From searching to traversing

Drop the `target`, print on the way out — BFS now **visits everything** it can reach:

```
def bfs(visited, graph, node):
  queue = deque([node])     # the only node we know
  visited[node] = True
  while len(queue) > 0:
    x = queue.popleft()
    print(x)        # print on the way OUT
    for y in graph[x]:
      if y not in visited:
        visited[y] = True
        queue.append(y)
```

Same arguments as `dfs(visited, graph, node)` — swap one for the other in any program.

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("BFS from", i)
    bfs(visited, g_adj, i)
```

Running it from every node

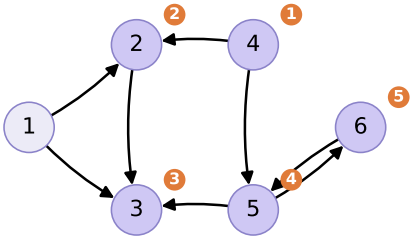
```
# g_adj = {1: [2, 3], 2: [3], 3: [],
#          4: [2, 5], 5: [3, 6], 6: [5]}

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

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

DFS vs. BFS: the same graph, from 4

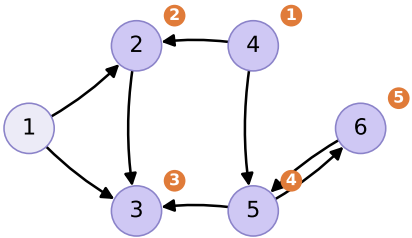
DFS dive, then backtrack



4 2 3 5 6

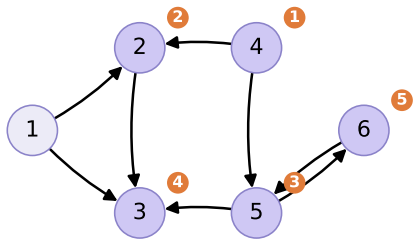
DFS vs. BFS: the same graph, from 4

DFS dive, then backtrack



4 2 3 5 6

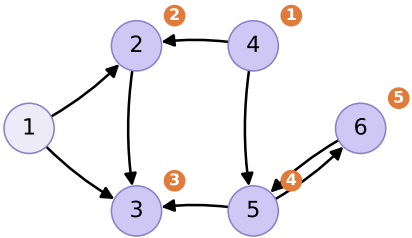
BFS level by level



4 2 5 3 6

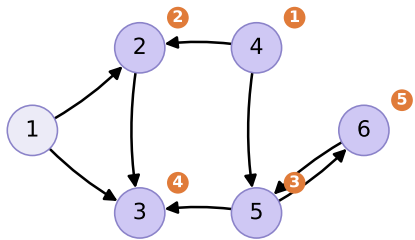
DFS vs. BFS: the same graph, from 4

DFS dive, then backtrack



4 2 3 5 6

BFS level by level



4 2 5 3 6

*Same graph, same five nodes — the **order** is the whole difference.*

One data structure apart

DFS

```
def dfs(visited, graph, node):
    visited[node] = True
    print(node)
    for y in graph[node]:
        if y not in visited:
            dfs(visited, graph, y)
```

The **call stack** holds what's left to do — newest first

BFS

```
def bfs(visited, graph, node):
    queue = deque([node])
    visited[node] = True
    while len(queue) > 0:
        x = queue.popleft()
        print(x)
        for y in graph[x]:
            if y not in visited:
                visited[y] = True
                queue.append(y)
```

The **queue** holds what's left to do — oldest first

One data structure apart

DFS

```
def dfs(visited, graph, node):
    visited[node] = True
    print(node)
    for y in graph[node]:
        if y not in visited:
            dfs(visited, graph, y)
```

The **call stack** holds what's left to do — newest first

BFS

```
def bfs(visited, graph, node):
    queue = deque([node])
    visited[node] = True
    while len(queue) > 0:
        x = queue.popleft()
        print(x)
        for y in graph[x]:
            if y not in visited:
                visited[y] = True
                queue.append(y)
```

The **queue** holds what's left to do — oldest first

*DFS lets **recursion** remember what's left — BFS has **no recursion**, just a queue.*

BFS time complexity

Discuss: How many times does BFS put each vertex **into** the queue? How many times does it **look at** each edge?

BFS time complexity

Discuss: How many times does BFS put each vertex **into** the queue? How many times does it **look at** each edge?

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

BFS time complexity

Discuss: How many times does BFS put each vertex **into** the queue? How many times does it **look at** each edge?

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

BFS time complexity

Discuss: How many times does BFS put each vertex **into** the queue? How many times does it **look at** each edge?

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

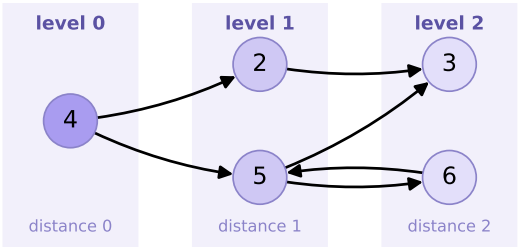
BFS time complexity

Discuss: How many times does BFS put each vertex **into** the queue? How many times does it **look at** each edge?

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

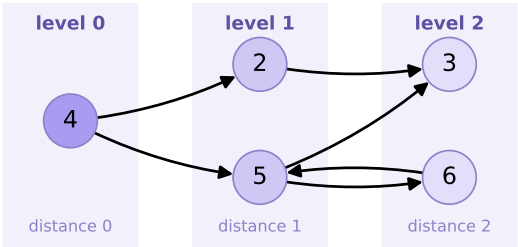
*Exactly the same as DFS. Choosing between them is never about speed — it is about the **order** you want.*

Why BFS: shortest paths



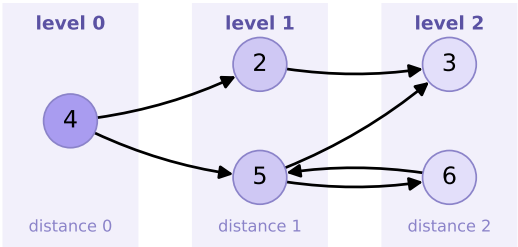
- BFS reaches level k only after finishing level $k - 1$

Why BFS: shortest paths



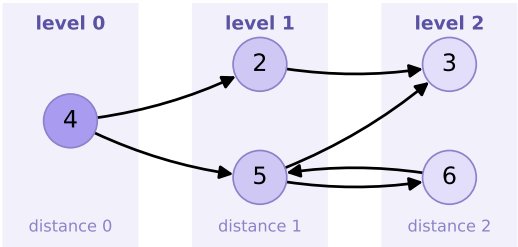
- BFS reaches level k only after finishing level $k - 1$
- So the level a node lands in **is** its distance from the start

Why BFS: shortest paths



- BFS reaches level k only after finishing level $k - 1$
- So the level a node lands in **is** its distance from the start
- The **fewest-edges** path, for free

Why BFS: shortest paths

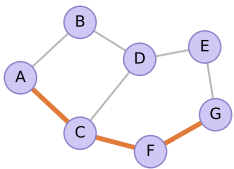


- BFS reaches level k only after finishing level $k-1$
- So the level a node lands in **is** its distance from the start
- The **fewest-edges** path, for free

*BFS finds the **shortest path** (in number of edges) from the start to every node it reaches. DFS does **not**.*

Applications of BFS

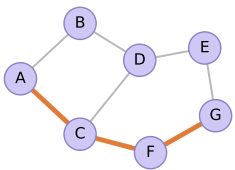
Fewest stops



`A` to `G` in 3 stops, not 4

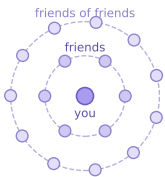
Applications of BFS

Fewest stops



`A` to `G` in 3 stops, not 4

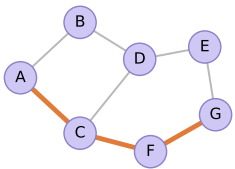
Degrees of separation



Each level: one more “friend of”

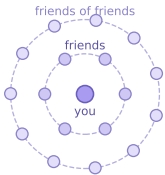
Applications of BFS

Fewest stops



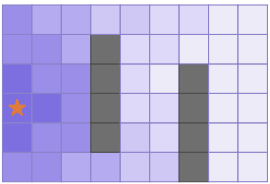
`A` to `G` in 3 stops, not 4

Degrees of separation



Each level: one more "friend of"

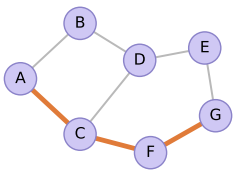
Flood fill



The paint bucket, pixel by pixel

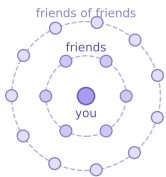
Applications of BFS

Fewest stops



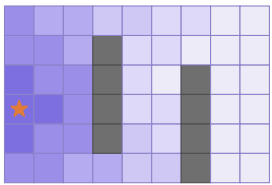
`A` to `G` in 3 stops, not 4

Degrees of separation



Each level: one more "friend of"

Flood fill

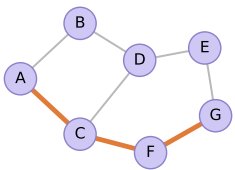


The paint bucket, pixel by pixel

Also **web crawling**, and **broadcasting**: how many hops to reach everyone?

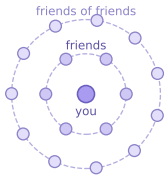
Applications of BFS

Fewest stops



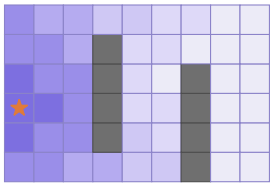
`A` to `G` in 3 stops, not 4

Degrees of separation



Each level: one more "friend of"

Flood fill



The paint bucket, pixel by pixel

Also **web crawling**, and **broadcasting**: how many hops to reach everyone?

*Whenever "**how far**" or "**how few steps**" is part of the question, reach for BFS.*

So which one should I use?

	DFS	**BFS**
Explores	deep along one branch	level by level
To-do list	the call stack — recursion keeps it **for you**	a `deque` you keep **yourself**
Shortest path?	no	yes (fewest edges)
Memory	one *path* at a time — usually far less	one whole *level* at a time — can be huge
Finds a target	sooner if it is **far** away	sooner if it is **near**
Good for	backtracking; cycles in a *directed* graph	distances, "how few steps", nearest match

Beyond DFS and BFS

For DFS and BFS, every edge counts the same: **one step**. Change how you pick the *next* node, and you get a whole family:

- **Dijkstra's algorithm** — edges have *lengths*; always take the **closest** node next (a **priority queue** instead of a queue)

Beyond DFS and BFS

For DFS and BFS, every edge counts the same: **one step**. Change how you pick the *next* node, and you get a whole family:

- **Dijkstra's algorithm** — edges have *lengths*; always take the **closest** node next (a **priority queue** instead of a queue)
- **A* search** — Dijkstra plus a guess of how far the goal still is; how a map app really routes you

Beyond DFS and BFS

For DFS and BFS, every edge counts the same: **one step**. Change how you pick the *next* node, and you get a whole family:

- **Dijkstra's algorithm** — edges have *lengths*; always take the **closest** node next (a **priority queue** instead of a queue)
- **A* search** — Dijkstra plus a guess of how far the goal still is; how a map app really routes you
- **Topological sort** — an order for tasks that depend on each other (built on DFS)

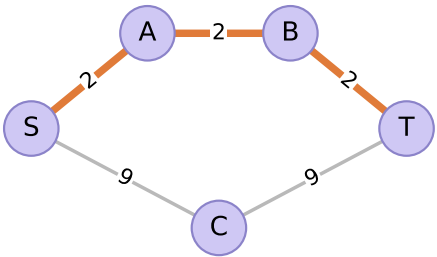
Beyond DFS and BFS

For DFS and BFS, every edge counts the same: **one step**. Change how you pick the *next* node, and you get a whole family:

- **Dijkstra's algorithm** — edges have *lengths*; always take the **closest** node next (a **priority queue** instead of a queue)
- **A* search** — Dijkstra plus a guess of how far the goal still is; how a map app really routes you
- **Topological sort** — an order for tasks that depend on each other (built on DFS)

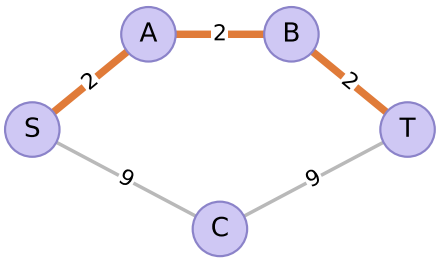
Stack → deep. Queue → level by level. Priority queue → nearest by distance. Same skeleton every time.

Dijkstra's algorithm: when steps have a *length*



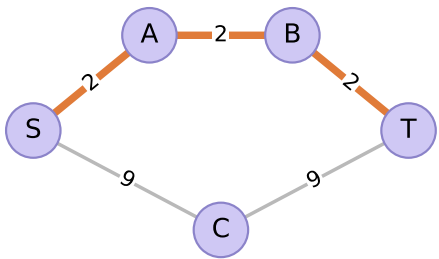
- **BFS** would pick `S – C – T`: only **2 steps**... but a length of **18**

Dijkstra's algorithm: when steps have a *length*



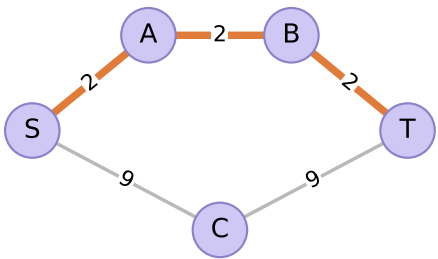
- **BFS** would pick `S – C – T`: only **2 steps**... but a length of **18**
- `S – A – B – T` is 3 steps and a length of only **6**

Dijkstra's algorithm: when steps have a *length*



- **BFS** would pick `S – C – T`: only **2 steps**... but a length of **18**
- `S – A – B – T` is 3 steps and a length of only **6**
- Take the **nearest** node next, not the oldest — a **priority queue**

Dijkstra's algorithm: when steps have a *length*

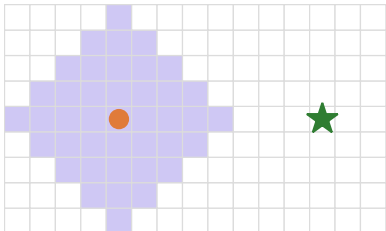


- **BFS** would pick `S – C – T`: only **2 steps**... but a length of **18**
- `S – A – B – T` is 3 steps and a length of only **6**
- Take the **nearest** node next, not the oldest — a **priority queue**

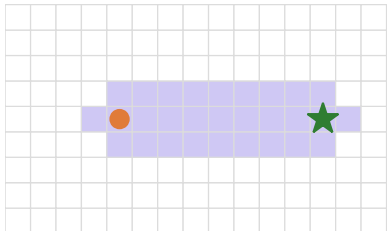
BFS is Dijkstra for the special case where every edge has length 1.

A* search: point the search at the goal

Dijkstra: spreads out in every direction

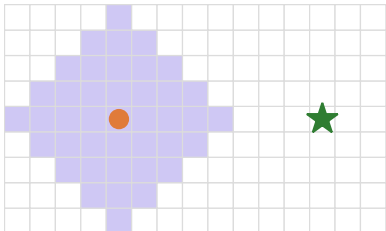


A*: only where the goal could be

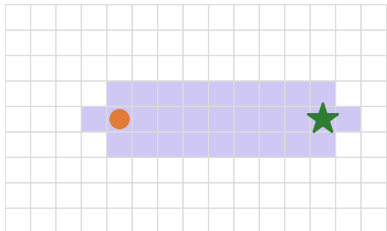


A* search: point the search at the goal

Dijkstra: spreads out in every direction



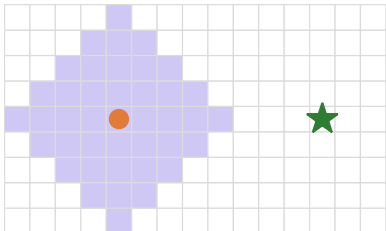
A*: only where the goal could be



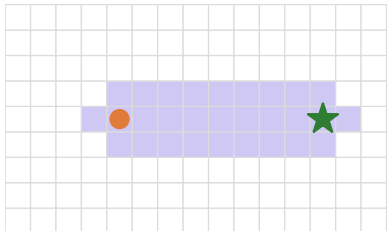
- Dijkstra has no idea where the goal is — it spreads out **everywhere**

A* search: point the search at the goal

Dijkstra: spreads out in every direction

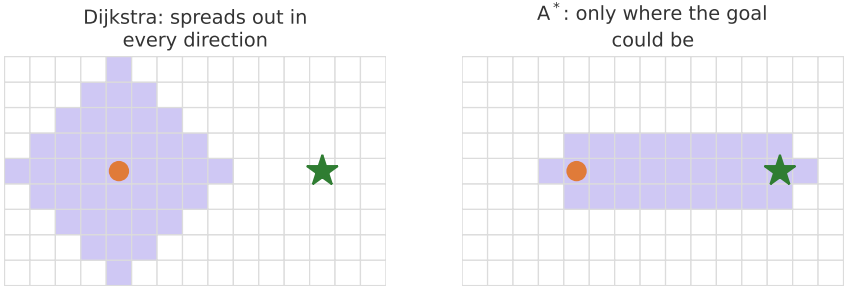


A*: only where the goal could be



- Dijkstra has no idea where the goal is — it spreads out **everywhere**
- A* adds a **guess** of how far the goal still is, and tries the most promising cells first

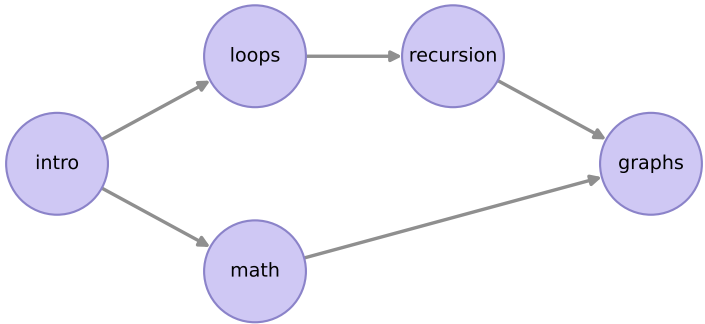
A* search: point the search at the goal



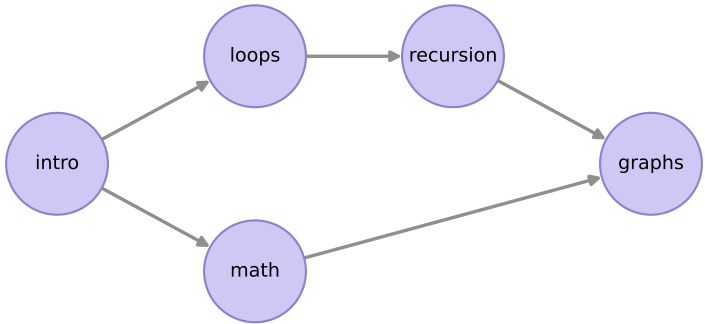
- Dijkstra has no idea where the goal is — it spreads out **everywhere**
- A* adds a **guess** of how far the goal still is, and tries the most promising cells first

Same answer, far less searching — this is what a map app is really doing.

Topological sort: what order can I do things in?

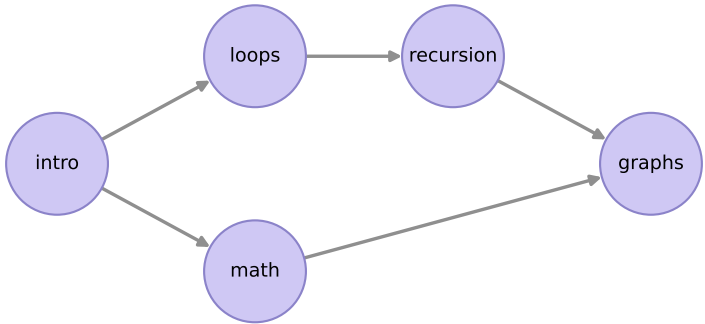


Topological sort: what order can I do things in?



- An arrow means "**must come before**" — you cannot do `graphs` before `math` and `recursion`

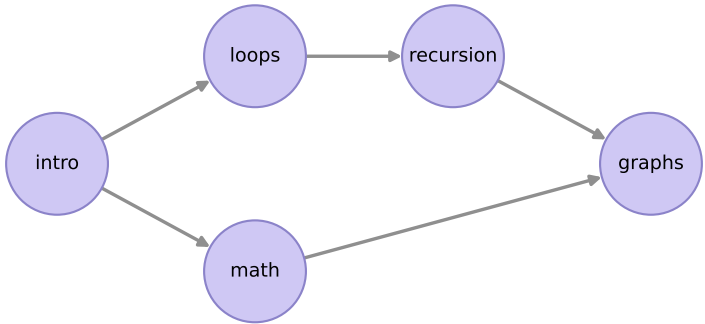
Topological sort: what order can I do things in?



- An arrow means "**must come before**" — you cannot do `graphs` before `math` and `recursion`
- A **topological sort** is any order that respects every arrow:

  `intro` → `loops` → `math` → `recursion` → `graphs`

Topological sort: what order can I do things in?



- An arrow means "**must come before**" — you cannot do `graphs` before `math` and `recursion`
- A **topological sort** is any order that respects every arrow:

  `intro` → `loops` → `math` → `recursion` → `graphs`
- Built on **DFS** — and it only exists if the graph has **no cycle**

Wrapping up

This afternoon:

- BFS explores **level by level** — nearest nodes first
- A **queue** replaces the call stack: `popleft()` instead of recursion
- Same $O(V + E)$ as DFS — but BFS gives you **shortest paths**

Wrapping up

This afternoon:

- BFS explores **level by level** — nearest nodes first
- A **queue** replaces the call stack: `popleft()` instead of recursion
- Same $O(V + E)$ as DFS — but BFS gives you **shortest paths**

Tomorrow:

- **Morning:** quiz — graphs, DFS and BFS
- **Afternoon:** language models

Wrapping up

This afternoon:

- BFS explores **level by level** — nearest nodes first
- A **queue** replaces the call stack: `popleft()` instead of recursion
- Same $O(V + E)$ as DFS — but BFS gives you **shortest paths**

Tomorrow:

- **Morning:** quiz — graphs, DFS and BFS
- **Afternoon:** language models

Thank you — see you tomorrow!