Tutorial 6

BFS, DFS, Dijkstra and A* in practice

Jan Bures

18YZALG – Basics of Algorithmization, Summer Semester 2026

How today works

- This is a **practical** session: representation → container → implementation → measured example.
- We keep the theory light and focus on what you will actually code in Python.
- Our recurring questions are simple: **can I reach it, how far is it, what is the cheapest route, how do I get to one goal fast?**

Ground rule

When a graph algorithm feels mysterious, look at **what goes into the queue, stack or heap** and **what key decides the next pop**.

How today works

- This is a **practical** session: representation → container → implementation → measured example.
- We keep the theory light and focus on what you will actually code in Python.
- Our recurring questions are simple: **can I reach it, how far is it, what is the cheapest route, how do I get to one goal fast?**

Ground rule

When a graph algorithm feels mysterious, look at **what goes into the queue, stack or heap** and **what key decides the next pop**.

How today works

- This is a **practical** session: representation $\rightarrow$ container $\rightarrow$ implementation $\rightarrow$ measured example.
- We keep the theory light and focus on what you will actually code in Python.
- Our recurring questions are simple: **can I reach it, how far is it, what is the cheapest route, how do I get to one goal fast?**

Ground rule

When a graph algorithm feels mysterious, look at **what goes into the queue, stack or heap** and **what key decides the next pop**.

How today works

- This is a **practical** session: representation $\rightarrow$ container $\rightarrow$ implementation $\rightarrow$ measured example.
- We keep the theory light and focus on what you will actually code in Python.
- Our recurring questions are simple: **can I reach it, how far is it, what is the cheapest route, how do I get to one goal fast?**

Ground rule

When a graph algorithm feels mysterious, look at **what goes into the queue, stack or heap** and **what key decides the next pop**.

How today works

- This is a **practical** session: representation $\rightarrow$ container $\rightarrow$ implementation $\rightarrow$ measured example.
- We keep the theory light and focus on what you will actually code in Python.
- Our recurring questions are simple: **can I reach it, how far is it, what is the cheapest route, how do I get to one goal fast?**

Ground rule

When a graph algorithm feels mysterious, look at **what goes into the queue, stack or heap** and **what key decides the next pop**.

Today

- Quick BFS/DFS refresher from Part 4.1
- Why graph representation matters immediately in practice
- Weighted shortest paths: why BFS fails, how Dijkstra fixes it
- A* as a goal-directed version for map-like search
- Benchmarks, complexity and a simple choice table

Today

- Quick BFS/DFS refresher from Part 4.1
- Why graph representation matters immediately in practice
- Weighted shortest paths: why BFS fails, how Dijkstra fixes it
- A* as a goal-directed version for map-like search
- Benchmarks, complexity and a simple choice table

Today

- Quick BFS/DFS refresher from Part 4.1
- Why graph representation matters immediately in practice
- Weighted shortest paths: why BFS fails, how Dijkstra fixes it
- A* as a goal-directed version for map-like search
- Benchmarks, complexity and a simple choice table

Today

- Quick BFS/DFS refresher from Part 4.1
- Why graph representation matters immediately in practice
- Weighted shortest paths: why BFS fails, how Dijkstra fixes it
- A^* as a goal-directed version for map-like search
- Benchmarks, complexity and a simple choice table

Today

- Quick BFS/DFS refresher from Part 4.1
- Why graph representation matters immediately in practice
- Weighted shortest paths: why BFS fails, how Dijkstra fixes it
- A^* as a goal-directed version for map-like search
- Benchmarks, complexity and a simple choice table

Same graph, four very different questions

Question	First tool	Why
Can I reach vertex t from s at all?	BFS or DFS	just visit systematically
Fewest edges from s to every vertex?	BFS	layers are distances
Cheapest total cost from s when edges have weights ≥ 0?	Dijkstra	pop smallest tentative cost
One known goal on a map, and I have a good hint toward it?	A^*	bias the search toward the goal

Practical mindset

Do not ask first: *"Which graph algorithm chapter is this from?"* Ask: *"What exactly is being minimized? edges, total cost, or just work toward one goal?"*

Representation first: same graph, different cost

Representation	Space	Neighbors of u	Traversal on sparse graph	Good when
Adjacency list	$\mathcal{O}(V+E)$	iterate only real neighbors	$\mathcal{O}(V+E)$	default choice
Adjacency matrix	$\mathcal{O}(V^2)$	scan an entire row	effectively $\mathcal{O}(V^2)$	dense graphs / fast edge test

Practical note

Most graphs in programming contests, grids, road nets and meshes are **sparse**. Adjacency lists let BFS, DFS and Dijkstra touch only edges that actually exist.

BFS: what it computes

Breadth-First Search

Start from a source s, keep a **queue**, and process vertices **level by level**.

Useful outputs

- `dist[v]`: number of edges from s to v (or "unreachable").
- `parent[v]`: who discovered v first.
- Following parents backward reconstructs a **shortest path** in an **unweighted** graph.

BFS: what it computes

Breadth-First Search

Start from a source s, keep a **queue**, and process vertices **level by level**.

Useful outputs

- `dist[v]`: number of edges from s to v (or "unreachable").
- `parent[v]`: who discovered v first.
- Following parents backward reconstructs a **shortest path** in an **unweighted** graph.

BFS: what it computes

Breadth-First Search

Start from a source s, keep a **queue**, and process vertices **level by level**.

Useful outputs

- `dist[v]`: number of edges from s to v (or "unreachable").
- `parent[v]`: who discovered v first.
- Following parents backward reconstructs a **shortest path** in an **unweighted** graph.

BFS: what it computes

Breadth-First Search

Start from a source s, keep a **queue**, and process vertices **level by level**.

Useful outputs

- `dist[v]`: number of edges from s to v (or "unreachable").
- `parent[v]`: who discovered v first.
- Following parents backward reconstructs a **shortest path** in an **unweighted** graph.

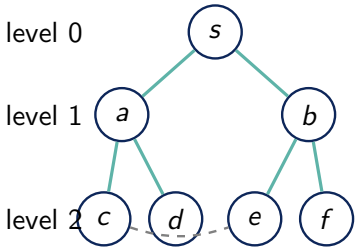
BFS (Python)

Implementation

```
from collections import deque

def bfs(adj, s):
    dist = [-1] * len(adj)
    parent = [-1] * len(adj)
    dist[s] = 0
    q = deque([s])
    while q:
        u = q.popleft()
        for v in adj[u]:
            if dist[v] == -1:
                dist[v] = dist[u] + 1
                parent[v] = u
                q.append(v)
    return dist, parent
```

BFS walk-through: 7-vertex example

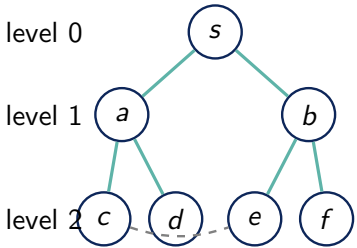


- Start with queue `[s]`. After processing s, the queue becomes `[a, b]`.
- Because the queue is FIFO, **every level-1 vertex leaves the queue before level-2 vertices**.
- That is exactly why `dist` counts the fewest number of edges.
- The dashed edge $c \leftrightarrow e$ exists, but it does not change the level structure once both vertices are already discovered.

Mental picture

BFS grows like a **wavefront**. The queue stores the current frontier of that wave.

BFS walk-through: 7-vertex example

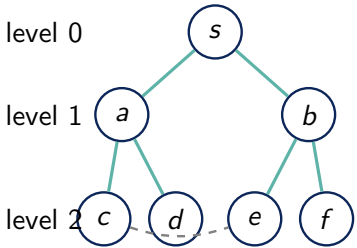


- Start with queue `[s]`. After processing s, the queue becomes `[a, b]`.
- Because the queue is FIFO, **every level-1 vertex leaves the queue before level-2 vertices**.
- That is exactly why `dist` counts the fewest number of edges.
- The dashed edge $c \leftrightarrow e$ exists, but it does not change the level structure once both vertices are already discovered.

Mental picture

BFS grows like a **wavefront**. The queue stores the current frontier of that wave.

BFS walk-through: 7-vertex example

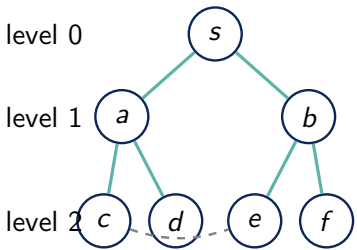


- Start with queue `[s]`. After processing s, the queue becomes `[a, b]`.
- Because the queue is FIFO, **every level-1 vertex leaves the queue before level-2 vertices**.
- That is exactly why `dist` counts the fewest number of edges.
- The dashed edge $c \leftrightarrow e$ exists, but it does not change the level structure once both vertices are already discovered.

Mental picture

BFS grows like a **wavefront**. The queue stores the current frontier of that wave.

BFS walk-through: 7-vertex example

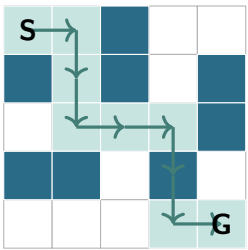


- Start with queue `[s]`. After processing s, the queue becomes `[a, b]`.
- Because the queue is FIFO, **every level-1 vertex leaves the queue before level-2 vertices**.
- That is exactly why `dist` counts the fewest number of edges.
- The dashed edge $c \leftrightarrow e$ exists, but it does not change the level structure once both vertices are already discovered.

Mental picture

BFS grows like a **wavefront**. The queue stores the current frontier of that wave.

BFS shortest path in a grid maze

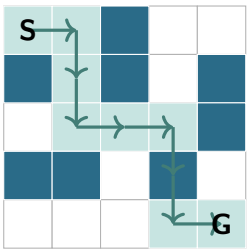


- In a **unit-cost grid**, BFS visits cells in increasing path length.
- Once G is reached, follow `parent` backward to reconstruct a shortest route.
- In this example, the shortest path has **8 moves**.
- If edges have different weights, plain BFS is no longer enough; use **Dijkstra**.

Why not DFS?

DFS can find *a* path, but the first path it finds need not be shortest.

BFS shortest path in a grid maze

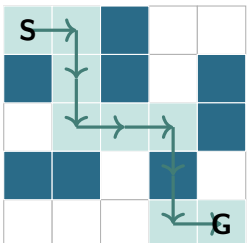


- In a **unit-cost grid**, BFS visits cells in increasing path length.
- Once G is reached, follow `parent` backward to reconstruct a shortest route.
- In this example, the shortest path has **8 moves**.
- If edges have different weights, plain BFS is no longer enough; use **Dijkstra**.

Why not DFS?

DFS can find *a* path, but the first path it finds need not be shortest.

BFS shortest path in a grid maze

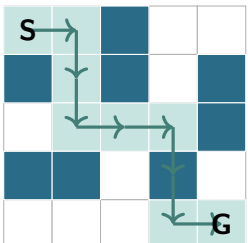


- In a **unit-cost grid**, BFS visits cells in increasing path length.
- Once G is reached, follow `parent` backward to reconstruct a shortest route.
- In this example, the shortest path has **8 moves**.
- If edges have different weights, plain BFS is no longer enough; use **Dijkstra**.

Why not DFS?

DFS can find *a* path, but the first path it finds need not be shortest.

BFS shortest path in a grid maze



- In a **unit-cost grid**, BFS visits cells in increasing path length.
- Once G is reached, follow `parent` backward to reconstruct a shortest route.
- In this example, the shortest path has **8 moves**.
- If edges have different weights, plain BFS is no longer enough; use **Dijkstra**.

Why not DFS?

DFS can find *a* path, but the first path it finds need not be shortest.

Measured example: BFS and DFS on sparse grid graphs

grid side	V	E	BFS [ms]	DFS [ms]
50×50	2500	4900	1.050	1.077
100×100	10000	19800	3.832	4.530
150×150	22500	44700	10.888	10.314
200×200	40000	79600	22.255	19.005

Best-of-7 on one machine; pure-Python reference implementations. In a square grid, $E \approx 2V$, so both traversals grow roughly linearly.

Interpretation

On sparse graphs, BFS and DFS have the **same asymptotic cost**. Choose between them by the **question you need answered**, not by hoping one is magically faster.

Representation benchmark: adjacency list vs adjacency matrix

grid side	V	E	list BFS [ms]	matrix BFS [ms]	speedup
20×20	400	760	0.115	6.543	$56.7\times$
30×30	900	1740	0.290	36.008	$124.0\times$
40×40	1600	3120	0.572	130.078	$227.6\times$
50×50	2500	4900	0.957	321.845	$336.3\times$

What happened?

The matrix version scans **every possible neighbor slot** in a row, even though the grid is sparse. The algorithm name is the same, but the representation changes the cost completely.

DFS: what it computes

Depth-First Search

Follow one branch deeply, then backtrack. In code this is a **stack** or **recursion**.

Useful outputs

- `disc[v]`: when the visit of v starts.
- `fin[v]`: when all descendants of v are finished.
- A **back edge** to a vertex still "in progress" reveals a cycle.

DFS: what it computes

Depth-First Search

Follow one branch deeply, then backtrack. In code this is a **stack** or **recursion**.

Useful outputs

- `disc[v]`: when the visit of v starts.
- `fin[v]`: when all descendants of v are finished.
- A **back edge** to a vertex still "in progress" reveals a cycle.

DFS: what it computes

Depth-First Search

Follow one branch deeply, then backtrack. In code this is a **stack** or **recursion**.

Useful outputs

- `disc[v]`: when the visit of v starts.
- `fin[v]`: when all descendants of v are finished.
- A **back edge** to a vertex still "in progress" reveals a cycle.

DFS: what it computes

Depth-First Search

Follow one branch deeply, then backtrack. In code this is a **stack** or **recursion**.

Useful outputs

- `disc[v]`: when the visit of v starts.
- `fin[v]`: when all descendants of v are finished.
- A **back edge** to a vertex still "in progress" reveals a cycle.

DFS with timestamps (Python)

Implementation

```
def dfs_timestamps(adj, s):
    seen = [False] * len(adj); parent = [-1] * len(adj)
    disc = [0] * len(adj); fin = [0] * len(adj); time = 0
    def visit(u):
        nonlocal time
        seen[u] = True; time += 1; disc[u] = time
        for v in adj[u]:
            if not seen[v]:
                parent[v] = u
                visit(v)
        time += 1; fin[u] = time
    visit(s)
    return disc, fin, parent
```

Python note: very deep graphs may need an explicit stack instead of recursion.

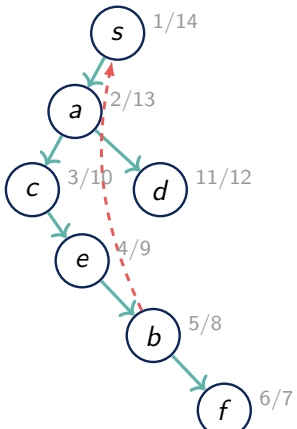
DFS with timestamps (Python)

Implementation

```
def dfs_timestamps(adj, s):
    seen = [False] * len(adj); parent = [-1] * len(adj)
    disc = [0] * len(adj); fin = [0] * len(adj); time = 0
    def visit(u):
        nonlocal time
        seen[u] = True; time += 1; disc[u] = time
        for v in adj[u]:
            if not seen[v]:
                parent[v] = u
                visit(v)
        time += 1; fin[u] = time
    visit(s)
    return disc, fin, parent
```

Python note: very deep graphs may need an explicit stack instead of recursion.

DFS on the same graph: timestamps and a back edge



v	s	a	c	e	b	f	d
$\texttt{disc}[v]$	1	2	3	4	5	6	11
$\texttt{fin}[v]$	14	13	10	9	8	7	12

Cycle detection in practice

When DFS from b sees the edge back to s, vertex s is already visited but not finished yet. That is a **back edge**, so a cycle exists.

In undirected graphs, DFS edge classification collapses to **tree edges** and **back edges**.

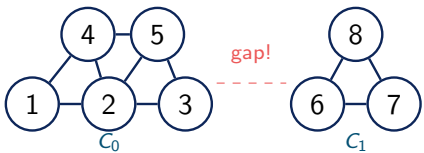
Connected components = repeated traversal

Implementation idea

```
from collections import deque

def connected_components(adj):
    comp = [-1] * len(adj); c = 0
    for s in range(len(adj)):
        if comp[s] != -1: continue
        q = deque([s]); comp[s] = c
        while q:
            u = q.popleft()
            for v in adj[u]:
                if comp[v] == -1:
                    comp[v] = c
                    q.append(v)
        c += 1
    return comp, c
```

Application 1: mesh connectivity check

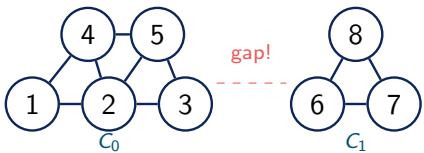


- Run one BFS/DFS from node 1.
- If `visited` $\neq |V|$, the mesh is disconnected.
- Or label every component using the repeated-traversal pattern.

Why this matters

A disconnected finite-element mesh can break the solver or silently invalidate the model. Traversal is a cheap pre-flight check.

Application 1: mesh connectivity check

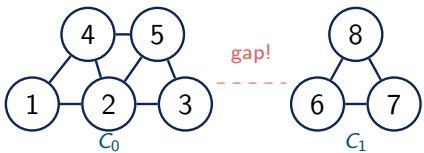


- Run one BFS/DFS from node 1.
- If `visited` $\neq |V|$, the mesh is disconnected.
- Or label every component using the repeated-traversal pattern.

Why this matters

A disconnected finite-element mesh can break the solver or silently invalidate the model. Traversal is a cheap pre-flight check.

Application 1: mesh connectivity check

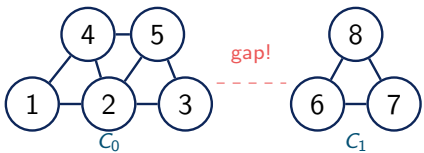


- Run one BFS/DFS from node 1.
- If `visited` $\neq |V|$, the mesh is disconnected.
- Or label every component using the repeated-traversal pattern.

Why this matters

A disconnected finite-element mesh can break the solver or silently invalidate the model. Traversal is a cheap pre-flight check.

Application 1: mesh connectivity check

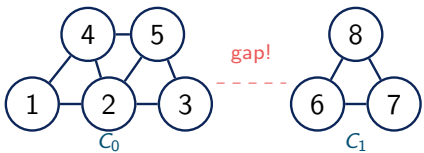


- Run one BFS/DFS from node 1.
- If `visited` $\neq |V|$, the mesh is disconnected.
- Or label every component using the repeated-traversal pattern.

Why this matters

A disconnected finite-element mesh can break the solver or silently invalidate the model. Traversal is a cheap pre-flight check.

Application 1: mesh connectivity check

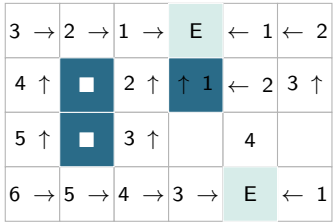


- Run one BFS/DFS from node 1.
- If `visited` $\neq |V|$, the mesh is disconnected.
- Or label every component using the repeated-traversal pattern.

Why this matters

A disconnected finite-element mesh can break the solver or silently invalidate the model. Traversal is a cheap pre-flight check.

Application 2: multi-source BFS for nearest exit



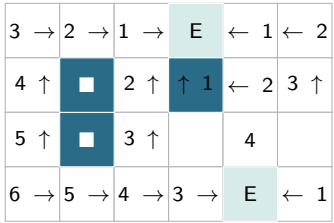
- Put **all exits** in the queue at distance 0.
- One BFS gives every cell its distance to the **nearest** exit.
- Reverse the parent pointers to get a movement direction field.

Why it scales

If a station grid has 16,000 cells and 500 passengers:

- one BFS per passenger: about $500 \times 16{,}000 = 8\,000\,000$ cell visits
- one multi-source BFS: about 16,000 cell visits

Application 2: multi-source BFS for nearest exit



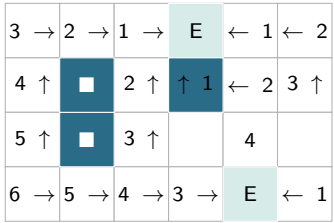
- Put **all exits** in the queue at distance 0.
- One BFS gives every cell its distance to the **nearest** exit.
- Reverse the parent pointers to get a movement direction field.

Why it scales

If a station grid has 16,000 cells and 500 passengers:

- one BFS per passenger: about $500 \times 16{,}000 = 8\,000\,000$ cell visits
- one multi-source BFS: about 16,000 cell visits

Application 2: multi-source BFS for nearest exit



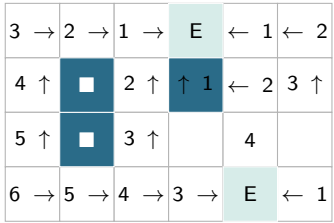
- Put **all exits** in the queue at distance 0.
- One BFS gives every cell its distance to the **nearest** exit.
- Reverse the parent pointers to get a movement direction field.

Why it scales

If a station grid has 16,000 cells and 500 passengers:

- one BFS per passenger: about $500 \times 16{,}000 = 8\,000\,000$ cell visits
- one multi-source BFS: about 16,000 cell visits

Application 2: multi-source BFS for nearest exit



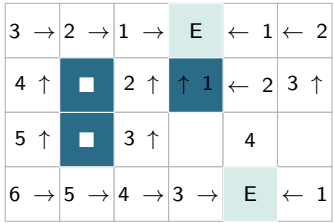
- Put **all exits** in the queue at distance 0.
- One BFS gives every cell its distance to the **nearest** exit.
- Reverse the parent pointers to get a movement direction field.

Why it scales

If a station grid has 16,000 cells and 500 passengers:

- one BFS per passenger: about $500 \times 16{,}000 = 8\,000\,000$ cell visits
- one multi-source BFS: about 16,000 cell visits

Application 2: multi-source BFS for nearest exit



- Put **all exits** in the queue at distance 0.
- One BFS gives every cell its distance to the **nearest** exit.
- Reverse the parent pointers to get a movement direction field.

Why it scales

If a station grid has 16,000 cells and 500 passengers:

- one BFS per passenger: about $500 \times 16{,}000 = 8\,000\,000$ cell visits
- one multi-source BFS: about 16,000 cell visits

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **"same loop, different container and different priority"**.

One data structure per search style

Queue (FIFO)

BFS

- oldest item leaves first
- natural for layers
- answers fewest-edges questions

Stack / recursion

DFS

- newest item leaves first
- natural for deep exploration
- reveals structure and cycles

Min-heap / priority queue

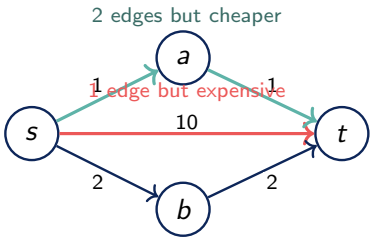
Dijkstra, A*

- smallest key leaves first
- natural for cheapest or most promising state
- key is `dist` or `g+h`

Unifying idea

Many search algorithms are just **“same loop, different container and different priority”**.

Why BFS fails on weighted graphs

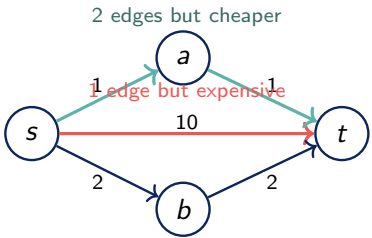


- BFS minimizes **number of edges**, not the **sum of weights**.
- From s, vertex t is only one edge away, so BFS can stop with cost 10.
- But the true cheapest path is $s \to a \to t$ with total cost 2.
- As soon as edge costs differ, we need a search ordered by **current best total cost**.

Takeaway

Unweighted shortest path and **weighted shortest path** are different problems.

Why BFS fails on weighted graphs

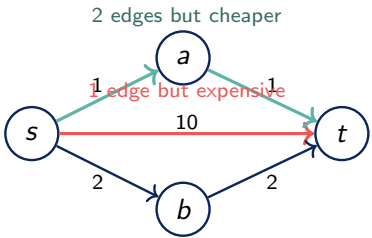


- BFS minimizes **number of edges**, not the **sum of weights**.
- From s, vertex t is only one edge away, so BFS can stop with cost 10.
- But the true cheapest path is $s \to a \to t$ with total cost 2.
- As soon as edge costs differ, we need a search ordered by **current best total cost**.

Takeaway

Unweighted shortest path and **weighted shortest path** are different problems.

Why BFS fails on weighted graphs

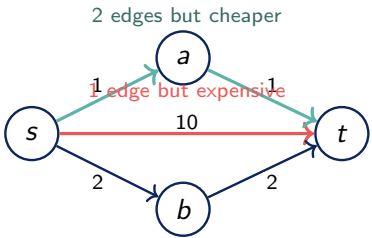


- BFS minimizes **number of edges**, not the **sum of weights**.
- From s, vertex t is only one edge away, so BFS can stop with cost 10.
- But the true cheapest path is $s \to a \to t$ with total cost 2.
- As soon as edge costs differ, we need a search ordered by **current best total cost**.

Takeaway

Unweighted shortest path and **weighted shortest path** are different problems.

Why BFS fails on weighted graphs



- BFS minimizes **number of edges**, not the **sum of weights**.
- From s, vertex t is only one edge away, so BFS can stop with cost 10.
- But the true cheapest path is $s \to a \to t$ with total cost 2.
- As soon as edge costs differ, we need a search ordered by **current best total cost**.

Takeaway

Unweighted shortest path and **weighted shortest path** are different problems.

Priority queue in Python = `heapq`

Minimal pattern

```
import heapq
pq = []
heapq.heappush(pq, (0, "s"))
heapq.heappush(pq, (5, "a"))
heapq.heappush(pq, (2, "b"))
d, u = heapq.heappop(pq)   # pops (0, "s")
```

- Push `(priority, vertex)`; the smallest priority comes out first.
- Dijkstra uses priority = current best distance. A* uses priority = $g + h$.
- Practical advice: use Python's heap abstraction; do not implement heap internals from scratch today.

Priority queue in Python = heapq

Minimal pattern

```
import heapq
pq = []
heapq.heappush(pq, (0, "s"))
heapq.heappush(pq, (5, "a"))
heapq.heappush(pq, (2, "b"))
d, u = heapq.heappop(pq)   # pops (0, "s")
```

- Push `(priority, vertex)`; the smallest priority comes out first.
- Dijkstra uses priority = current best distance. A* uses priority = `g + h`.
- Practical advice: use Python's heap abstraction; do not implement heap internals from scratch today.

Priority queue in Python = heapq

Minimal pattern

```
import heapq
pq = []
heapq.heappush(pq, (0, "s"))
heapq.heappush(pq, (5, "a"))
heapq.heappush(pq, (2, "b"))
d, u = heapq.heappop(pq)   # pops (0, "s")
```

- Push `(priority, vertex)`; the smallest priority comes out first.
- Dijkstra uses priority = current best distance. A* uses priority = `g + h`.
- Practical advice: use Python's heap abstraction; do not implement heap internals from scratch today.

Priority queue in Python = `heapq`

Minimal pattern

```
import heapq
pq = []
heapq.heappush(pq, (0, "s"))
heapq.heappush(pq, (5, "a"))
heapq.heappush(pq, (2, "b"))
d, u = heapq.heappop(pq)   # pops (0, "s")
```

- Push `(priority, vertex)`; the smallest priority comes out first.
- Dijkstra uses priority = current best distance. A* uses priority = `g + h`.
- Practical advice: use Python's heap abstraction; do not implement heap internals from scratch today.

Dijkstra: what it computes

Dijkstra's algorithm

Start with `dist[s] = 0`, keep a **min-priority queue**, and always process the unprocessed vertex with the **smallest tentative distance**.

Useful outputs

- `dist[v]`: cheapest total cost from s to v.
- `parent[v]`: previous vertex on that cheapest route.
- With **nonnegative** weights, once a vertex leaves the heap with the smallest valid key, its distance is final.

Dijkstra: what it computes

Dijkstra's algorithm

Start with `dist[s] = 0`, keep a **min-priority queue**, and always process the unprocessed vertex with the **smallest tentative distance**.

Useful outputs

- `dist[v]`: cheapest total cost from s to v.
- `parent[v]`: previous vertex on that cheapest route.
- With **nonnegative** weights, once a vertex leaves the heap with the smallest valid key, its distance is final.

Dijkstra: what it computes

Dijkstra's algorithm

Start with `dist[s] = 0`, keep a **min-priority queue**, and always process the unprocessed vertex with the **smallest tentative distance**.

Useful outputs

- `dist[v]`: cheapest total cost from s to v.
- `parent[v]`: previous vertex on that cheapest route.
- With **nonnegative** weights, once a vertex leaves the heap with the smallest valid key, its distance is final.

Dijkstra: what it computes

Dijkstra's algorithm

Start with `dist[s] = 0`, keep a **min-priority queue**, and always process the unprocessed vertex with the **smallest tentative distance**.

Useful outputs

- `dist[v]`: cheapest total cost from s to v.
- `parent[v]`: previous vertex on that cheapest route.
- With **nonnegative** weights, once a vertex leaves the heap with the smallest valid key, its distance is final.

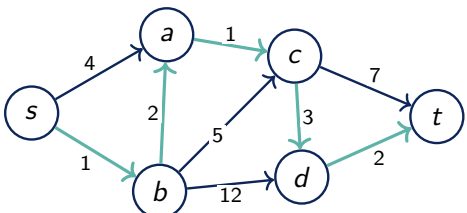
Dijkstra (Python)

Implementation

```
import heapq

def dijkstra(adj, s):
    dist = [float("inf")] * len(adj)
    parent = [-1] * len(adj); dist[s] = 0
    pq = [(0, s)]
    while pq:
        d, u = heapq.heappop(pq)
        if d != dist[u]: continue
        for v, w in adj[u]:
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd; parent[v] = u
                heapq.heappush(pq, (nd, v))
    return dist, parent
```

Dijkstra on a tiny road network



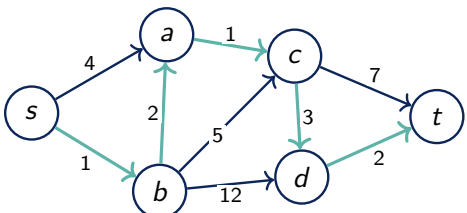
v	s	b	a	c	d	t
`dist[v]`	0	1	3	4	7	9
`parent[v]`	–	s	b	a	c	d

- Pop order is `s, b, a, c, d, t` because those are the cheapest tentative distances.
- The cheapest route to t is $s \to b \to a \to c \to d \to t$ with cost **9**.
- The algorithm keeps improving guesses until no cheaper improvement is possible.

Relaxation

The line `if nd < dist[v]` means: **"I found a cheaper route to v; remember it."**

Dijkstra on a tiny road network



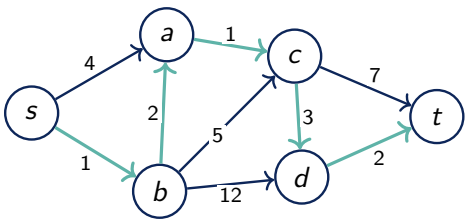
v	s	b	a	c	d	t
`dist[v]`	0	1	3	4	7	9
`parent[v]`	–	s	b	a	c	d

- Pop order is `s, b, a, c, d, t` because those are the cheapest tentative distances.
- The cheapest route to t is $s \to b \to a \to c \to d \to t$ with cost **9**.
- The algorithm keeps improving guesses until no cheaper improvement is possible.

Relaxation

The line `if nd < dist[v]` means: **"I found a cheaper route to v; remember it."**

Dijkstra on a tiny road network



v	s	b	a	c	d	t
`dist[`v`]`	0	1	3	4	7	9
`parent[`v`]`	–	s	b	a	c	d

- Pop order is `s, b, a, c, d, t` because those are the cheapest tentative distances.
- The cheapest route to t is $s \to b \to a \to c \to d \to t$ with cost **9**.
- The algorithm keeps improving guesses until no cheaper improvement is possible.

Relaxation

The line `if nd < dist[v]` means: **"I found a cheaper route to v; remember it."**

Dijkstra in practice: four things to remember

- **Relaxation** is the heart of the algorithm: improve `dist[v]` when a cheaper path through u appears.
- Python's `heapq` has no fast decrease-key, so we push a new pair and later skip stale ones with `if d != dist[u]: continue`.
- With adjacency lists and a binary heap, the running time is $\mathcal{O}((V + E) \log V)$.
- Dijkstra assumes **nonnegative edge weights**. Negative weights break the greedy logic.

Very practical rule

If all weights are exactly 1, **use BFS**. Dijkstra still works, but it pays an unnecessary `heapq` overhead.

Dijkstra in practice: four things to remember

- **Relaxation** is the heart of the algorithm: improve `dist[v]` when a cheaper path through u appears.
- Python's `heapq` has no fast decrease-key, so we push a new pair and later skip stale ones with `if d != dist[u]: continue`.
- With adjacency lists and a binary heap, the running time is $\mathcal{O}((V + E) \log V)$.
- Dijkstra assumes **nonnegative edge weights**. Negative weights break the greedy logic.

Very practical rule

If all weights are exactly 1, **use BFS**. Dijkstra still works, but it pays an unnecessary `heapq` overhead.

Dijkstra in practice: four things to remember

- **Relaxation** is the heart of the algorithm: improve `dist[v]` when a cheaper path through u appears.
- Python's `heapq` has no fast decrease-key, so we push a new pair and later skip stale ones with `if d != dist[u]: continue`.
- With adjacency lists and a binary heap, the running time is $\mathcal{O}((V + E) \log V)$.
- Dijkstra assumes **nonnegative edge weights**. Negative weights break the greedy logic.

Very practical rule

If all weights are exactly 1, **use BFS**. Dijkstra still works, but it pays an unnecessary `heapq` overhead.

Dijkstra in practice: four things to remember

- **Relaxation** is the heart of the algorithm: improve `dist[v]` when a cheaper path through u appears.
- Python's `heapq` has no fast decrease-key, so we push a new pair and later skip stale ones with `if d != dist[u]: continue`.
- With adjacency lists and a binary heap, the running time is $\mathcal{O}((V + E) \log V)$.
- Dijkstra assumes **nonnegative edge weights**. Negative weights break the greedy logic.

Very practical rule

If all weights are exactly 1, **use BFS**. Dijkstra still works, but it pays an unnecessary `heapq` overhead.

Dijkstra in practice: four things to remember

- **Relaxation** is the heart of the algorithm: improve `dist[v]` when a cheaper path through u appears.
- Python's `heapq` has no fast decrease-key, so we push a new pair and later skip stale ones with `if d != dist[u]: continue`.
- With adjacency lists and a binary heap, the running time is $\mathcal{O}((V + E) \log V)$.
- Dijkstra assumes **nonnegative edge weights**. Negative weights break the greedy logic.

Very practical rule

If all weights are exactly 1, **use BFS**. Dijkstra still works, but it pays an unnecessary `heapq` overhead.

Dijkstra in practice: four things to remember

- **Relaxation** is the heart of the algorithm: improve `dist[v]` when a cheaper path through u appears.
- Python's `heapq` has no fast decrease-key, so we push a new pair and later skip stale ones with `if d != dist[u]: continue`.
- With adjacency lists and a binary heap, the running time is $\mathcal{O}((V + E) \log V)$.
- Dijkstra assumes **nonnegative edge weights**. Negative weights break the greedy logic.

Very practical rule

If all weights are exactly 1, **use BFS**. Dijkstra still works, but it pays an unnecessary `heapq` overhead.

Same answer, different cost: BFS vs Dijkstra on unit grids

grid side	V	E	BFS [ms]	Dijkstra [ms]	overhead
50×50	2500	4900	0.934	1.962	2.1×
100×100	10000	19800	4.372	8.868	2.0×
150×150	22500	44700	10.827	20.979	1.9×
200×200	40000	79600	19.307	43.362	2.2×

Pure-Python reference implementations on the same unit-weight grid graphs. Both return the same distances.

Interpretation

Dijkstra is more general, but BFS is the better tool when every edge cost is 1. Generality often costs extra time.

Choosing among BFS, Dijkstra and A*

Problem shape	First choice	Why
Unweighted graph, need all distances from one source	BFS	no heap needed
Weighted graph with costs ≥ 0, need cheapest costs from one source	Dijkstra	exact and general
One target on a map, and a good heuristic is available	A*	same optimal path, often less search
Negative weights appear	not these tools	greedy order is unsafe

Safe summary sentence

BFS = fewest edges. Dijkstra = cheapest total cost. A* = Dijkstra plus a hint toward one goal.

A*: Dijkstra plus a hint toward the goal

Priority rule

Instead of ordering by only `g(v)` = cost from start so far, A* orders by

$$f(v) = g(v) + h(v),$$

where `h(v)` estimates the remaining cost from v to the goal.

- `g(v)` is exact so far; `h(v)` is a hint.
- On a 4-neighbor grid, **Manhattan distance** is a common heuristic.
- If `h` never overestimates, A* still returns an **optimal** path.
- In practice, A* explores less when the hint points roughly toward the goal.

A*: Dijkstra plus a hint toward the goal

Priority rule

Instead of ordering by only `g(v)` = cost from start so far, A* orders by

$$f(v) = g(v) + h(v),$$

where `h(v)` estimates the remaining cost from v to the goal.

- `g(v)` is exact so far; `h(v)` is a hint.
- On a 4-neighbor grid, **Manhattan distance** is a common heuristic.
- If `h` never overestimates, A* still returns an **optimal** path.
- In practice, A* explores less when the hint points roughly toward the goal.

A*: Dijkstra plus a hint toward the goal

Priority rule

Instead of ordering by only `g(v)` = cost from start so far, A* orders by

$$f(v) = g(v) + h(v),$$

where `h(v)` estimates the remaining cost from v to the goal.

- `g(v)` is exact so far; `h(v)` is a hint.
- On a 4-neighbor grid, **Manhattan distance** is a common heuristic.
- If `h` never overestimates, A* still returns an **optimal** path.
- In practice, A* explores less when the hint points roughly toward the goal.

A*: Dijkstra plus a hint toward the goal

Priority rule

Instead of ordering by only `g(v)` = cost from start so far, A* orders by

$$f(v) = g(v) + h(v),$$

where `h(v)` estimates the remaining cost from v to the goal.

- `g(v)` is exact so far; `h(v)` is a hint.
- On a 4-neighbor grid, **Manhattan distance** is a common heuristic.
- If `h` never overestimates, A* still returns an **optimal** path.
- In practice, A* explores less when the hint points roughly toward the goal.

A*: Dijkstra plus a hint toward the goal

Priority rule

Instead of ordering by only `g(v)` = cost from start so far, A* orders by

$$f(v) = g(v) + h(v),$$

where `h(v)` estimates the remaining cost from v to the goal.

- `g(v)` is exact so far; `h(v)` is a hint.
- On a 4-neighbor grid, **Manhattan distance** is a common heuristic.
- If `h` never overestimates, A* still returns an **optimal** path.
- In practice, A* explores less when the hint points roughly toward the goal.

A* (Python)

Implementation skeleton

```
import heapq
def astar(adj, start, goal, h):
    g = [float("inf")] * len(adj)
    parent = [-1] * len(adj); g[start] = 0
    pq = [(h(start), 0, start)]   # (f, g, vertex)
    while pq:
        f, gu, u = heapq.heappop(pq)
        if gu != g[u]: continue
        if u == goal: break
        for v, w in adj[u]:
            ng = gu + w
            if ng < g[v]:
                g[v] = ng; parent[v] = u
                heapq.heappush(pq, (ng + h(v), ng, v))
    return g[goal], parent
```

Targeted path search: Dijkstra vs A*

grid side	avg open cells	Dijkstra settled	A* settled	Dijkstra [ms]	A* [ms]	reduction
40×40	1113.8	1051.6	336.2	0.922	0.460	$3.1\times$
60×60	2504.6	2435.4	791.6	2.278	1.076	$3.1\times$
80×80	4472.2	4372.4	1643.2	4.231	2.417	$2.7\times$
100×100	7007.4	6833.0	1695.4	6.436	2.442	$4.0\times$

Average over 5 connected random obstacle maps with 30% blocked cells. Both algorithms returned the same optimal path cost.

Interpretation

Dijkstra spreads everywhere that could still be optimal. A* uses the heuristic to spend less work behind and sideways, so it often reaches the goal sooner.

When A* helps — and when it does not

Usually helps when ...

- there is **one goal**, not all-pairs or all-distances
- the graph has a **spatial meaning** (grid, map, navigation mesh)
- the heuristic is informative, e.g. Manhattan distance

Usually does not help much when ...

- you need distances to **every** vertex anyway
- no useful heuristic exists
- the heuristic overestimates — then optimality is no longer guaranteed

Plain-English summary

A* is a great **point-to-point search**. Dijkstra is the safer default when you need a full shortest-path map from one source.

When A* helps — and when it does not

Usually helps when ...

- there is **one goal**, not all-pairs or all-distances
- the graph has a **spatial meaning** (grid, map, navigation mesh)
- the heuristic is informative, e.g. Manhattan distance

Usually does not help much when ...

- you need distances to **every** vertex anyway
- no useful heuristic exists
- the heuristic overestimates — then optimality is no longer guaranteed

Plain-English summary

A* is a great **point-to-point search**. Dijkstra is the safer default when you need a full shortest-path map from one source.

When A* helps — and when it does not

Usually helps when ...

- there is **one goal**, not all-pairs or all-distances
- the graph has a **spatial meaning** (grid, map, navigation mesh)
- the heuristic is informative, e.g. Manhattan distance

Usually does not help much when ...

- you need distances to **every** vertex anyway
- no useful heuristic exists
- the heuristic overestimates — then optimality is no longer guaranteed

Plain-English summary

A* is a great **point-to-point search**. Dijkstra is the safer default when you need a full shortest-path map from one source.

When A* helps — and when it does not

Usually helps when ...

- there is **one goal**, not all-pairs or all-distances
- the graph has a **spatial meaning** (grid, map, navigation mesh)
- the heuristic is informative, e.g. Manhattan distance

Usually does not help much when ...

- you need distances to **every** vertex anyway
- no useful heuristic exists
- the heuristic overestimates — then optimality is no longer guaranteed

Plain-English summary

A* is a great **point-to-point search**. Dijkstra is the safer default when you need a full shortest-path map from one source.

When A* helps — and when it does not

Usually helps when ...

- there is **one goal**, not all-pairs or all-distances
- the graph has a **spatial meaning** (grid, map, navigation mesh)
- the heuristic is informative, e.g. Manhattan distance

Usually does not help much when ...

- you need distances to **every** vertex anyway
- no useful heuristic exists
- the heuristic overestimates — then optimality is no longer guaranteed

Plain-English summary

A* is a great **point-to-point search**. Dijkstra is the safer default when you need a full shortest-path map from one source.

When A* helps — and when it does not

Usually helps when ...

- there is **one goal**, not all-pairs or all-distances
- the graph has a **spatial meaning** (grid, map, navigation mesh)
- the heuristic is informative, e.g. Manhattan distance

Usually does not help much when ...

- you need distances to **every** vertex anyway
- no useful heuristic exists
- the heuristic overestimates — then optimality is no longer guaranteed

Plain-English summary

A* is a great **point-to-point search**. Dijkstra is the safer default when you need a full shortest-path map from one source.

When A* helps — and when it does not

Usually helps when ...

- there is **one goal**, not all-pairs or all-distances
- the graph has a **spatial meaning** (grid, map, navigation mesh)
- the heuristic is informative, e.g. Manhattan distance

Usually does not help much when ...

- you need distances to **every** vertex anyway
- no useful heuristic exists
- the heuristic overestimates — then optimality is no longer guaranteed

Plain-English summary

A* is a great **point-to-point search**. Dijkstra is the safer default when you need a full shortest-path map from one source.

When A* helps — and when it does not

Usually helps when ...

- there is **one goal**, not all-pairs or all-distances
- the graph has a **spatial meaning** (grid, map, navigation mesh)
- the heuristic is informative, e.g. Manhattan distance

Usually does not help much when ...

- you need distances to **every** vertex anyway
- no useful heuristic exists
- the heuristic overestimates — then optimality is no longer guaranteed

Plain-English summary

A* is a great **point-to-point search**. Dijkstra is the safer default when you need a full shortest-path map from one source.

Common mistakes across all four tools

- Saying "BFS always finds the shortest path." It does so only when all edges have equal cost.
- Forgetting that an adjacency matrix can make a sparse-graph search dramatically slower.
- Using recursive DFS on a very deep graph without thinking about recursion depth.
- Forgetting the stale-pop check in Dijkstra or A* when using Python's `heapq`.
- Using an overestimating heuristic in A* and still claiming the answer must be optimal.

Most expensive bug today

The right algorithm with the **wrong representation or wrong priority rule** can be slower — or simply wrong.

Common mistakes across all four tools

- Saying "BFS always finds the shortest path." It does so only when all edges have equal cost.
- Forgetting that an adjacency matrix can make a sparse-graph search dramatically slower.
- Using recursive DFS on a very deep graph without thinking about recursion depth.
- Forgetting the stale-pop check in Dijkstra or A* when using Python's `heapq`.
- Using an overestimating heuristic in A* and still claiming the answer must be optimal.

Most expensive bug today

The right algorithm with the **wrong representation or wrong priority rule** can be slower — or simply wrong.

Common mistakes across all four tools

- Saying "BFS always finds the shortest path." It does so only when all edges have equal cost.
- Forgetting that an adjacency matrix can make a sparse-graph search dramatically slower.
- Using recursive DFS on a very deep graph without thinking about recursion depth.
- Forgetting the stale-pop check in Dijkstra or A* when using Python's `heapq`.
- Using an overestimating heuristic in A* and still claiming the answer must be optimal.

Most expensive bug today

The right algorithm with the **wrong representation or wrong priority rule** can be slower — or simply wrong.

Common mistakes across all four tools

- Saying "BFS always finds the shortest path." It does so only when all edges have equal cost.
- Forgetting that an adjacency matrix can make a sparse-graph search dramatically slower.
- Using recursive DFS on a very deep graph without thinking about recursion depth.
- Forgetting the stale-pop check in Dijkstra or A* when using Python's `heapq`.
- Using an overestimating heuristic in A* and still claiming the answer must be optimal.

Most expensive bug today

The right algorithm with the **wrong representation or wrong priority rule** can be slower — or simply wrong.

Common mistakes across all four tools

- Saying "BFS always finds the shortest path." It does so only when all edges have equal cost.
- Forgetting that an adjacency matrix can make a sparse-graph search dramatically slower.
- Using recursive DFS on a very deep graph without thinking about recursion depth.
- Forgetting the stale-pop check in Dijkstra or A* when using Python's `heapq`.
- Using an overestimating heuristic in A* and still claiming the answer must be optimal.

Most expensive bug today

The right algorithm with the **wrong representation or wrong priority rule** can be slower — or simply wrong.

Common mistakes across all four tools

- Saying "BFS always finds the shortest path." It does so only when all edges have equal cost.
- Forgetting that an adjacency matrix can make a sparse-graph search dramatically slower.
- Using recursive DFS on a very deep graph without thinking about recursion depth.
- Forgetting the stale-pop check in Dijkstra or A* when using Python's `heapq`.
- Using an overestimating heuristic in A* and still claiming the answer must be optimal.

Most expensive bug today

The right algorithm with the **wrong representation or wrong priority rule** can be slower — or simply wrong.

Common mistakes across all four tools

- Saying "BFS always finds the shortest path." It does so only when all edges have equal cost.
- Forgetting that an adjacency matrix can make a sparse-graph search dramatically slower.
- Using recursive DFS on a very deep graph without thinking about recursion depth.
- Forgetting the stale-pop check in Dijkstra or A* when using Python's `heapq`.
- Using an overestimating heuristic in A* and still claiming the answer must be optimal.

Most expensive bug today

The right algorithm with the **wrong representation or wrong priority rule** can be slower — or simply wrong.

Quick check: which tool fits?

1. A 2D maze with equal move cost: which algorithm gives the fewest moves from S to G?
2. A directed graph where you want to know whether a cycle exists.
3. A disconnected mesh where you need to label all connected pieces.
4. A road network where each edge has a nonnegative travel time.
5. A game map with one target and a Manhattan-distance heuristic.

Likely answers

1) BFS, 2) DFS, 3) either BFS or DFS, 4) Dijkstra, 5) A*.

Quick check: which tool fits?

1. A 2D maze with equal move cost: which algorithm gives the fewest moves from S to G?
2. A directed graph where you want to know whether a cycle exists.
3. A disconnected mesh where you need to label all connected pieces.
4. A road network where each edge has a nonnegative travel time.
5. A game map with one target and a Manhattan-distance heuristic.

Likely answers

1) BFS, 2) DFS, 3) either BFS or DFS, 4) Dijkstra, 5) A*.

Quick check: which tool fits?

1. A 2D maze with equal move cost: which algorithm gives the fewest moves from S to G?
2. A directed graph where you want to know whether a cycle exists.
3. A disconnected mesh where you need to label all connected pieces.
4. A road network where each edge has a nonnegative travel time.
5. A game map with one target and a Manhattan-distance heuristic.

Likely answers

1) BFS, 2) DFS, 3) either BFS or DFS, 4) Dijkstra, 5) A*.

Quick check: which tool fits?

1. A 2D maze with equal move cost: which algorithm gives the fewest moves from S to G?
2. A directed graph where you want to know whether a cycle exists.
3. A disconnected mesh where you need to label all connected pieces.
4. A road network where each edge has a nonnegative travel time.
5. A game map with one target and a Manhattan-distance heuristic.

Likely answers

1) BFS, 2) DFS, 3) either BFS or DFS, 4) Dijkstra, 5) A*.

Quick check: which tool fits?

1. A 2D maze with equal move cost: which algorithm gives the fewest moves from S to G?
2. A directed graph where you want to know whether a cycle exists.
3. A disconnected mesh where you need to label all connected pieces.
4. A road network where each edge has a nonnegative travel time.
5. A game map with one target and a Manhattan-distance heuristic.

Likely answers

1) BFS, 2) DFS, 3) either BFS or DFS, 4) Dijkstra, 5) A*.

Quick check: which tool fits?

1. A 2D maze with equal move cost: which algorithm gives the fewest moves from S to G?
2. A directed graph where you want to know whether a cycle exists.
3. A disconnected mesh where you need to label all connected pieces.
4. A road network where each edge has a nonnegative travel time.
5. A game map with one target and a Manhattan-distance heuristic.

Likely answers

1) BFS, 2) DFS, 3) either BFS or DFS, 4) Dijkstra, 5) A*.

Quick check: which tool fits?

1. A 2D maze with equal move cost: which algorithm gives the fewest moves from S to G?
2. A directed graph where you want to know whether a cycle exists.
3. A disconnected mesh where you need to label all connected pieces.
4. A road network where each edge has a nonnegative travel time.
5. A game map with one target and a Manhattan-distance heuristic.

Likely answers

1) BFS, 2) DFS, 3) either BFS or DFS, 4) Dijkstra, 5) A*.

Summary

- **Representation first:** on sparse graphs, adjacency lists are the default choice.
- **BFS:** queue, layers, distances in number of edges, shortest paths in unweighted graphs.
- **DFS:** stack/recursion, timestamps, back edges, cycles, components.
- **Dijkstra:** min-heap, cheapest total cost with nonnegative weights.
- **A*:** Dijkstra plus a heuristic for faster point-to-point search when a good hint exists.

Summary

- **Representation first:** on sparse graphs, adjacency lists are the default choice.
- **BFS:** queue, layers, distances in number of edges, shortest paths in unweighted graphs.
- **DFS:** stack/recursion, timestamps, back edges, cycles, components.
- **Dijkstra:** min-heap, cheapest total cost with nonnegative weights.
- **A*:** Dijkstra plus a heuristic for faster point-to-point search when a good hint exists.

Summary

- **Representation first:** on sparse graphs, adjacency lists are the default choice.
- **BFS:** queue, layers, distances in number of edges, shortest paths in unweighted graphs.
- **DFS:** stack/recursion, timestamps, back edges, cycles, components.
- **Dijkstra:** min-heap, cheapest total cost with nonnegative weights.
- **A*:** Dijkstra plus a heuristic for faster point-to-point search when a good hint exists.

Summary

- **Representation first:** on sparse graphs, adjacency lists are the default choice.
- **BFS:** queue, layers, distances in number of edges, shortest paths in unweighted graphs.
- **DFS:** stack/recursion, timestamps, back edges, cycles, components.
- **Dijkstra:** min-heap, cheapest total cost with nonnegative weights.
- **A*:** Dijkstra plus a heuristic for faster point-to-point search when a good hint exists.

Summary

- **Representation first:** on sparse graphs, adjacency lists are the default choice.
- **BFS:** queue, layers, distances in number of edges, shortest paths in unweighted graphs.
- **DFS:** stack/recursion, timestamps, back edges, cycles, components.
- **Dijkstra:** min-heap, cheapest total cost with nonnegative weights.
- **A*:** Dijkstra plus a heuristic for faster point-to-point search when a good hint exists.