CSE 592 - Discussion 1

Search

Seung Hyun Lee / Fall 26

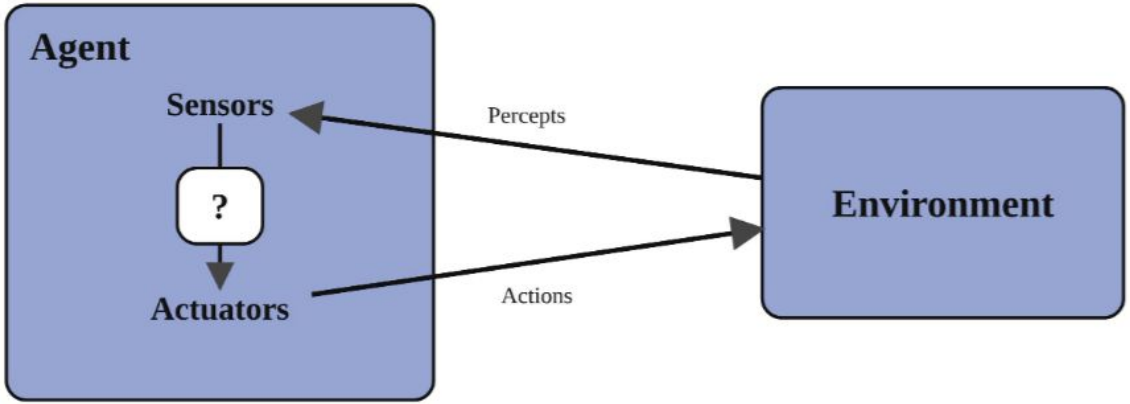
Logistics

- Slides: After discussion session
- Discussion: Friday, 9:30-10:30 am, 1010 DOW
- GSI Office hours: Thursday, 1:00-2:00pm, 1637 BBB (Table #1)
  - https://umich.zoom.us/j/99426471592
- Email: seungle@umich.edu
- **Assignment 1** will be released Wednesday, Sep. 9.

No class & OH Monday, Sep. 7 — Labor Day

Task environments and rational agents

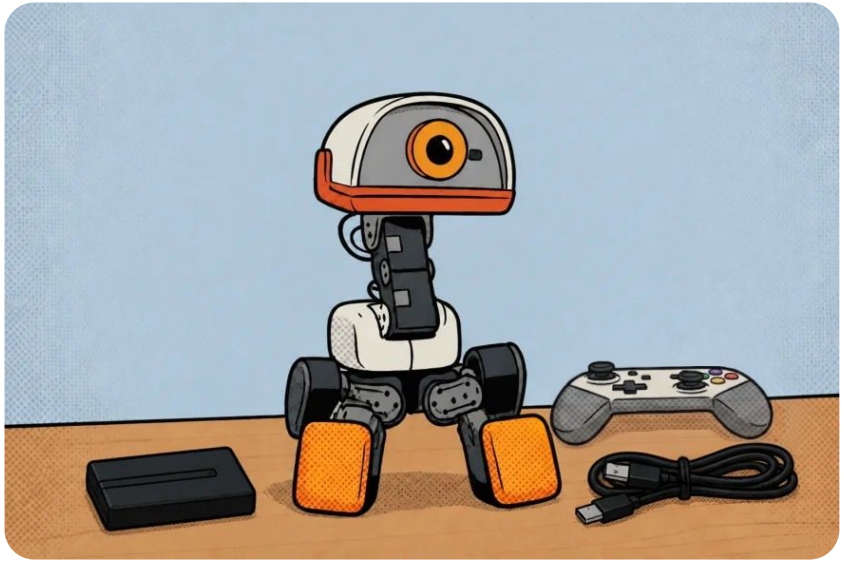
- PEAS
- **P**erformance measure
- **E**nvironment
- **A**ctuators
- **S**ensors



Meet Microduck: a robot that can sense, walk, and grasp

TASK

Find the requested object and pick it up safely.



https://microduck.net/#story

Design Microduck's PEAS description

Performance measure	Environment	Actuators	Sensors

A complete PEAS description for Microduck

PERFORMANCE

Correct object grasped
No drops, falls, or damage
Low time and energy use

ENVIRONMENT

Target + distractors
Floor, furniture, obstacles
People and lighting

ACTUATORS

Leg, head, neck motors
Articulated beak
Communication output

SENSORS

Front camera
Depth sensor
Two IMUs

When should Microduck refuse?

- The object may be too heavy or fragile.
- A person’s hand is nearby.
- Microduck may be uncertain whether it found the correct object.
- Its current pose is unstable or its battery is too low.

Classify Microduck's task environment

- Fully vs partially observable?
- Deterministic vs nondeterministic?
- Episodic vs sequential?
- Static vs dynamic?
- Discrete vs continuous?
- Single-agent vs multi-agent?

Microduck is partially observable, dynamic, and sequential

- Fully vs partially observable? **Partially observable**
- Deterministic vs nondeterministic? **Non-deterministic**
- Episodic vs sequential? **Sequential**
- Static vs dynamic? **Dynamic**
- Discrete vs continuous? **Continuous**
- Single-agent vs multi-agent? **Single-agent for this task**

The four frontier rules

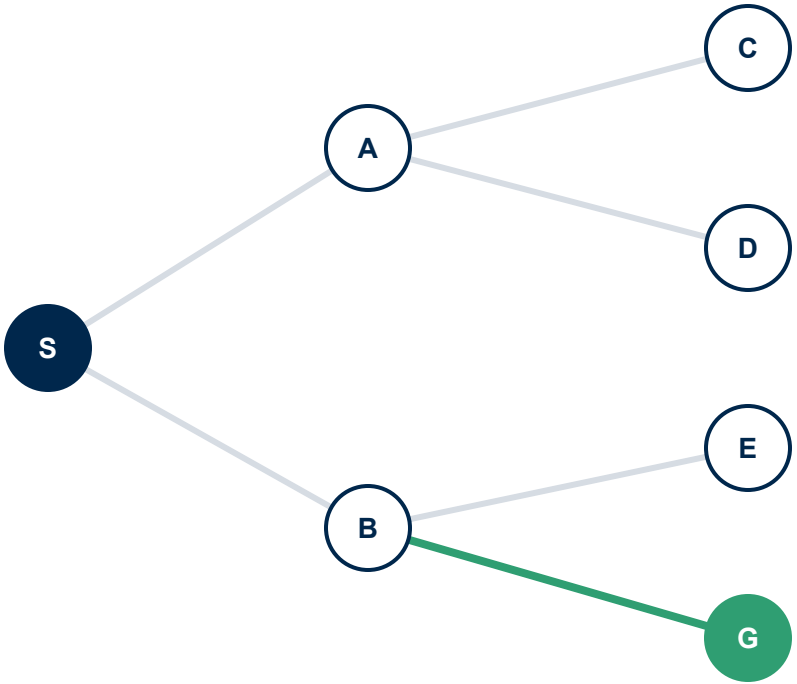
- The search loop is almost unchanged
- Each algorithm chooses a different node to pop next.

BFS	**FIFO / shallowest**	**UCS**	**smallest g**
Explore one depth layer at a time		Prefer the least real cost paid so far	
DFS	**LIFO / deepest**	**A***	**smallest g+h**
Follow one branch before its siblings		Combine paid cost with a remaining-cost estimate	

We will run each rule on small graphs and watch the frontier change.

Example

State = current graph node; action = follow an edge; step cost = 1; goal test = “is this G?”



TWO DATA STRUCTURES

frontier — **discovered nodes still waiting to pop**

reached — best node found for each state

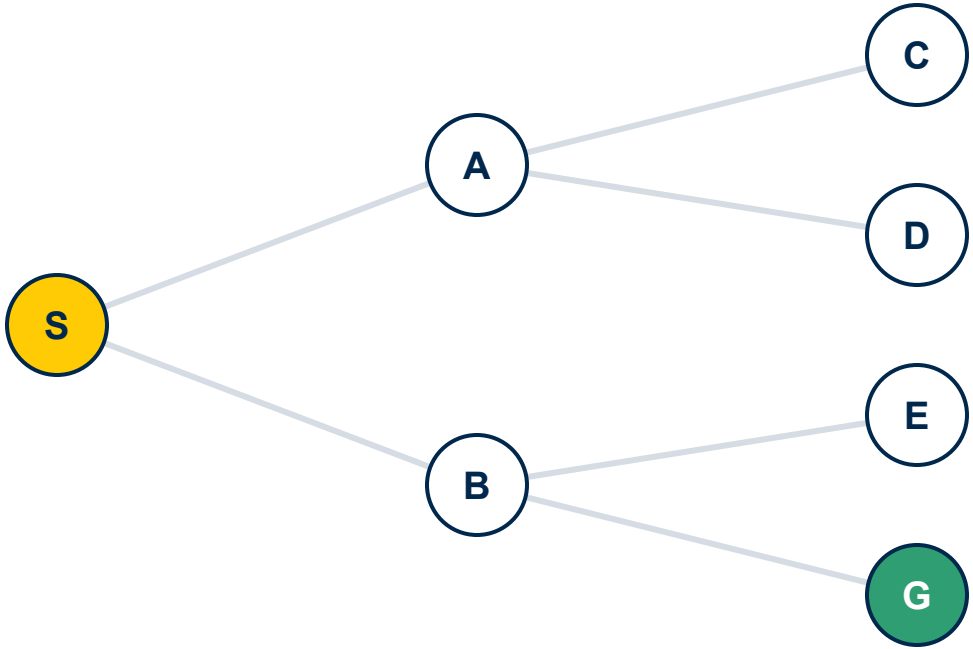
THE COMMON LOOP

1. **pop the first frontier node**
2. **if it is a goal, return its path**
3. **otherwise expand successors**
4. **update reached and frontier**

Only the frontier order changes.

BFS step 0: enqueue S

A FIFO queue pops from the left and appends new states on the right.



FIFO FRONTIER

S

front back

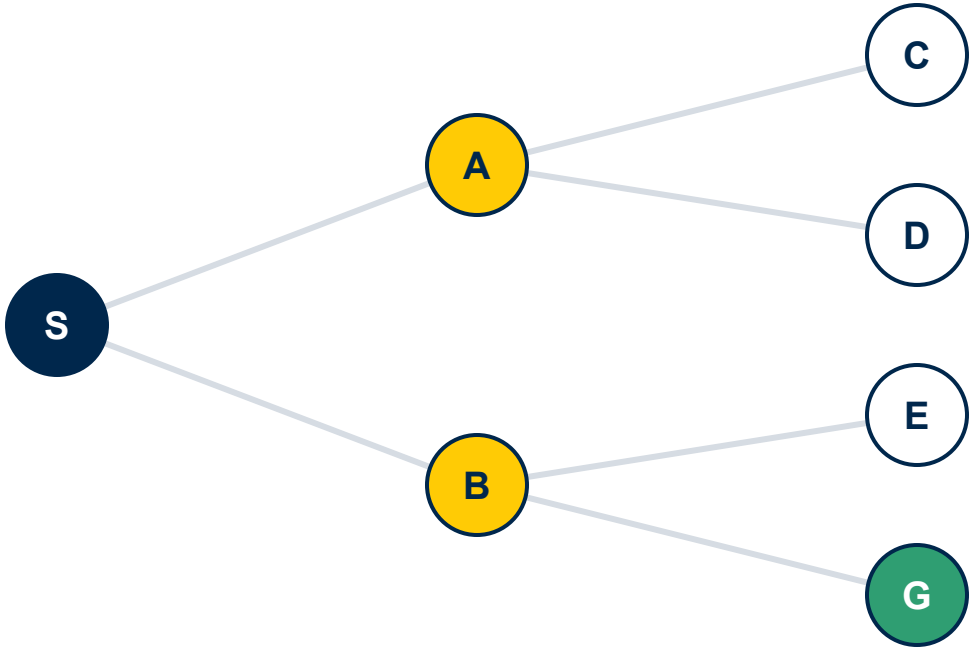
reached = { S }

expanded	—
successors	—

next pop: S

BFS step 1: expand S → enqueue A, B

Successor order breaks ties between nodes at the same depth.



FIFO FRONTIER

A	B

front back

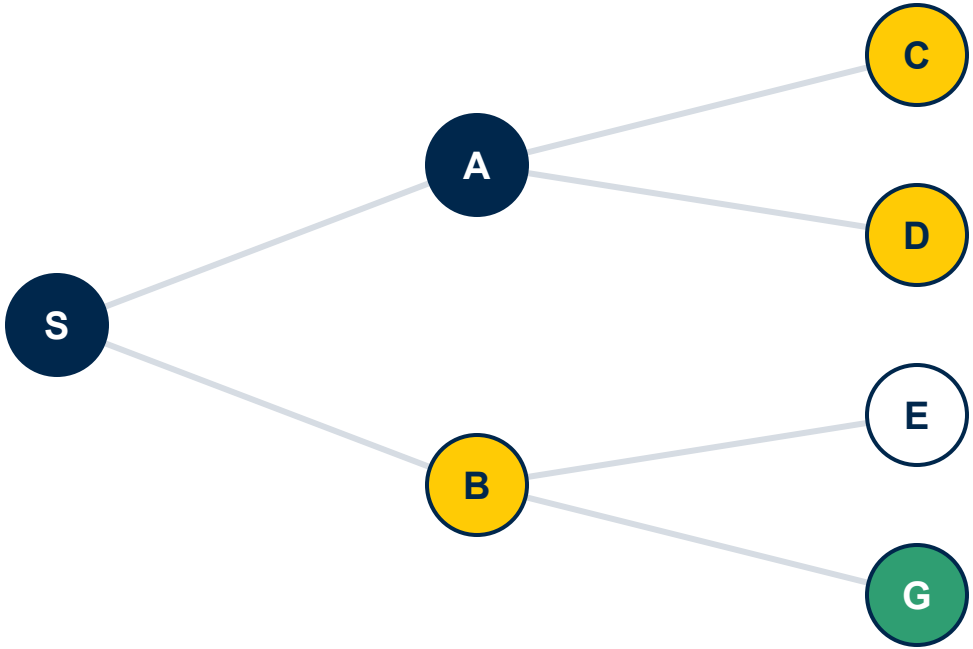
reached = { S, A, B }

expanded	S
successors	A, B

next pop: A

BFS step 2: expand A → enqueue C, D

B remains at the front because it entered the queue before C and D.



FIFO FRONTIER

B	C	D

front back

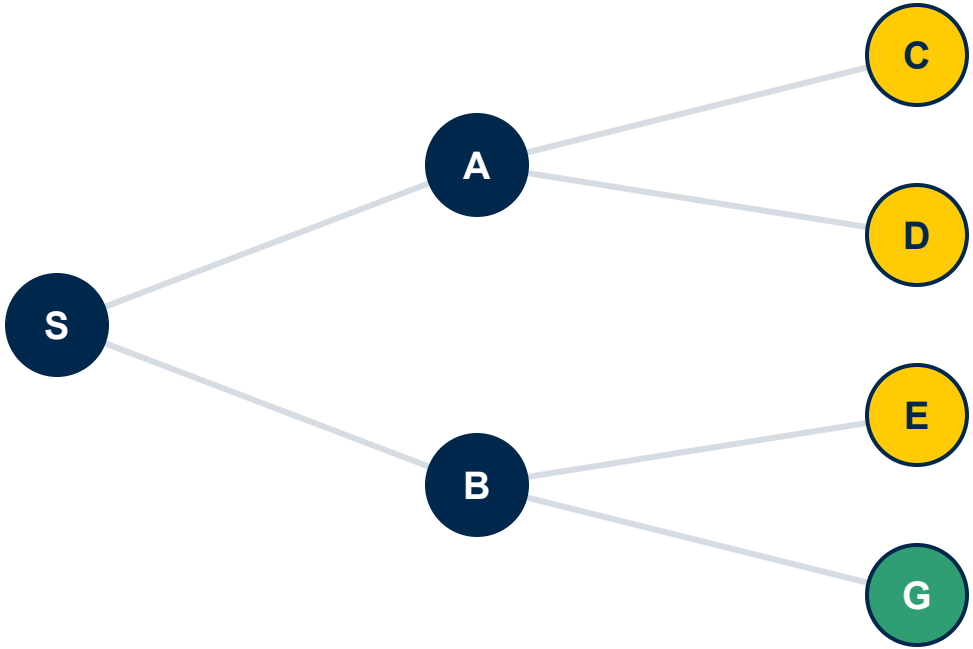
reached = { S, A, B, C, D }

expanded	A
successors	C, D

next pop: B

BFS step 3: expand B → enqueue E, G

G is generated now, but the course algorithm tests IsGoal only when a node is popped.



FIFO FRONTIER

C	D	E	G

front back

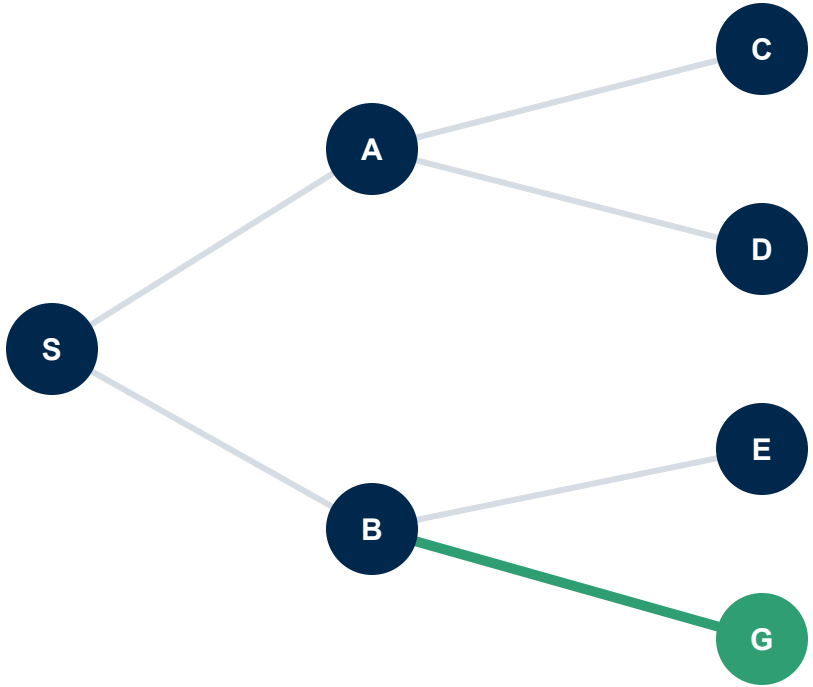
reached = { S, A, B, C, D, E, G }

expanded	B
successors	E, G

G waits behind C, D, E

BFS stops when G is popped

Goal testing happens immediately after pop, so G is returned without being expanded.



RETURNED PATH

S → B → G

FINAL FRONTIER

front [G] back

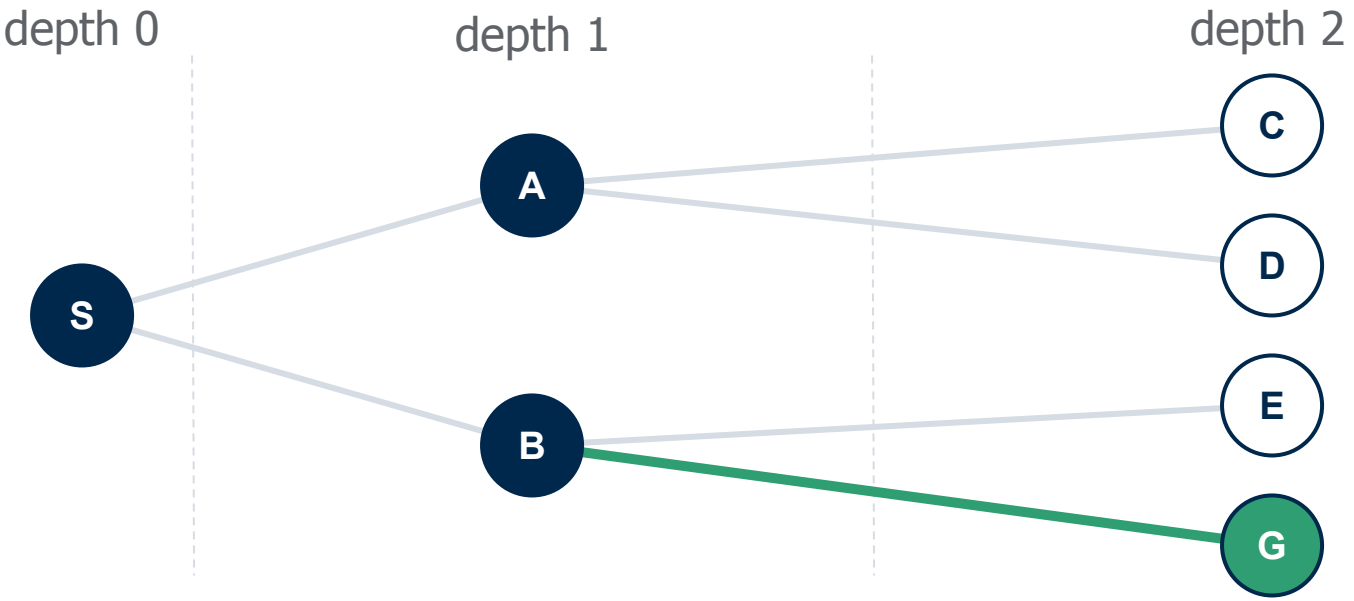
1 pop G

2 IsGoal(G) = true

return before expand

FIFO order makes the first goal shallowest

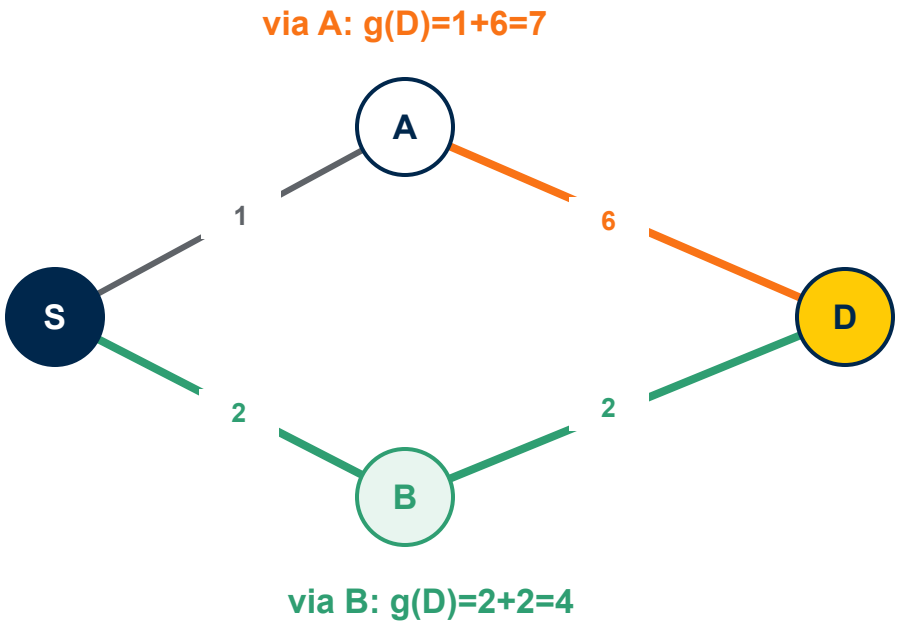
All depth-k nodes leave the queue before any depth-(k+1) node.



Depth never decreases along the pop order

Reached keeps the smallest g found for each state

If a new path reaches the same state with a lower cost, replace the old entry.



reached[D]

first: D with g=7

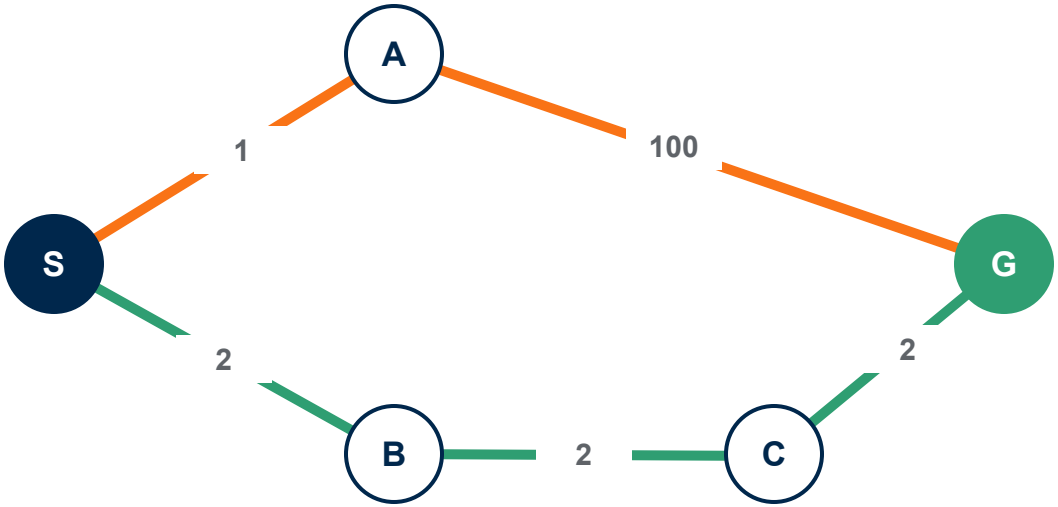
then find g=4 < 7

replace → D with g=4

```
if g_new < reached[state].cost:
    replace reached[state]
```

BFS minimizes actions, not path cost

A shallower goal can be much more expensive when edge costs differ.



BFS: depth 2

S → A → G

cost = 1 + 100 = 101

Cheapest: depth 3

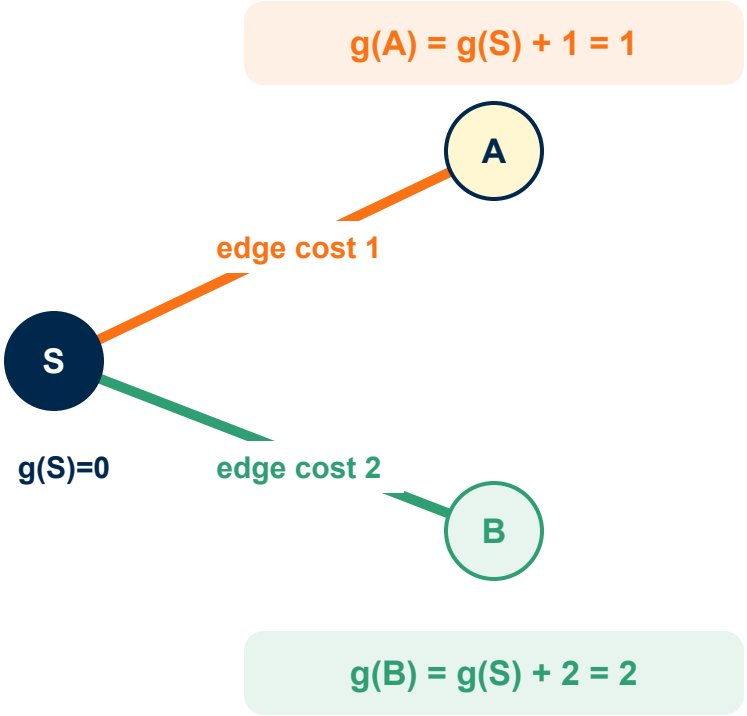
S → B → C → G

cost = 2 + 2 + 2 = 6

Equal edge costs ⇒ minimizing depth also minimizes cost.

UCS changes the priority from depth to g

g(n) is the real path cost accumulated from S to n. UCS pops the smallest g.



UCS PRIORITY QUEUE

[(A, g=1), (B, g=2)]

smallest g is at the front

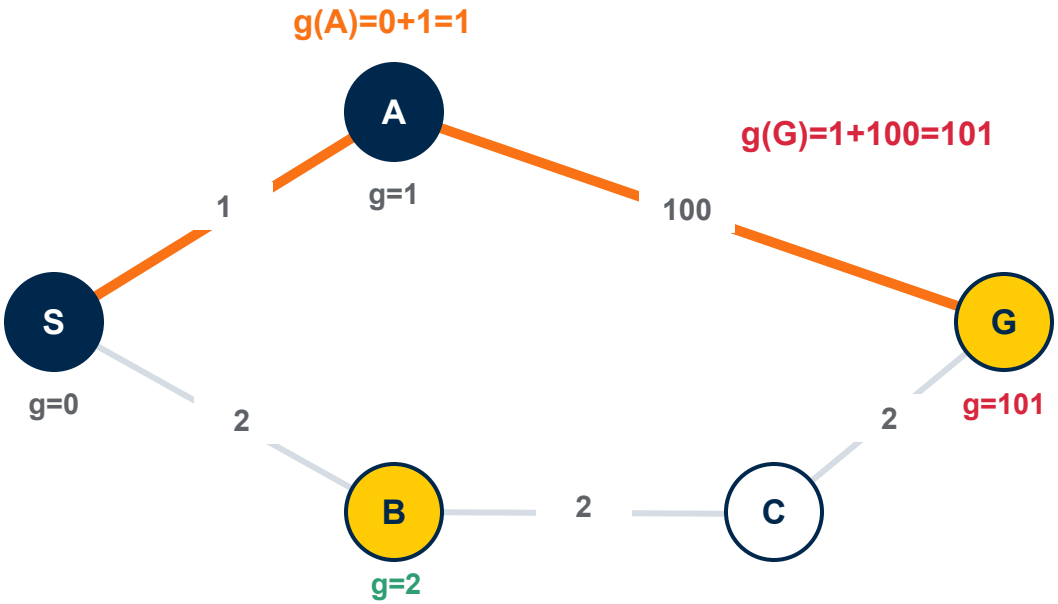
ONE UPDATE RULE

g_child = g_parent + edge_cost

No h yet: UCS uses only cost already paid.

UCS trace I: generating a goal is not enough

The queue is sorted by cumulative g. A goal is tested only when it is popped.



PRIORITY QUEUE · min g first

STEP 0 — **start**

[(S, 0)]

pop S

STEP 1 — **expand S**

[(A, 1), (B, 2)]

pop A because 1 < 2

STEP 2 — **expand A**

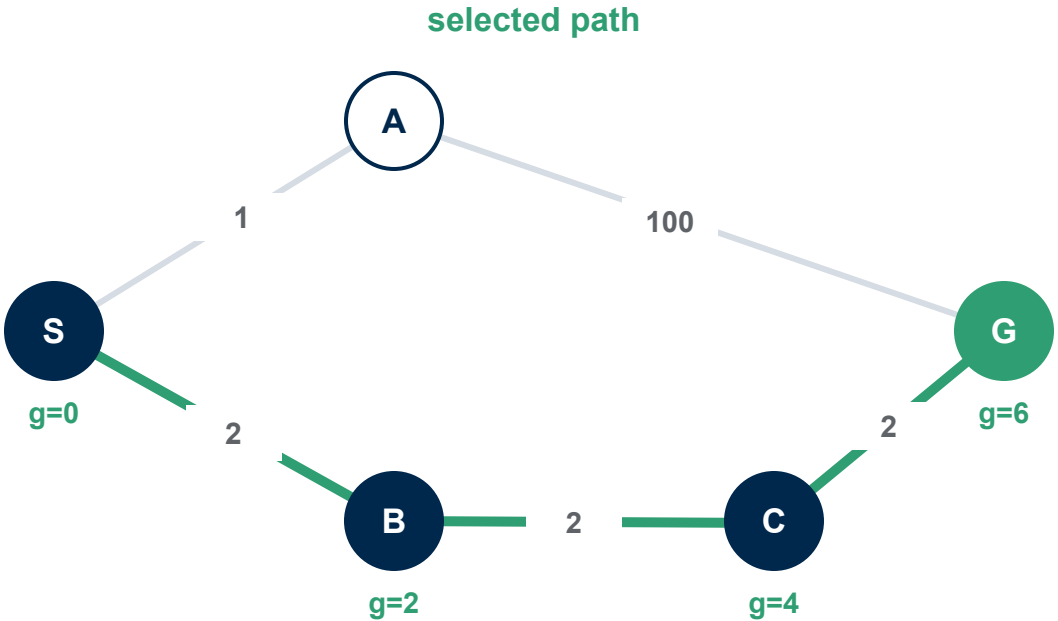
[(B, 2), (G, 101)]

next pop is B — not G

Generating G is not enough.
UCS returns only when G reaches the front and is popped.

UCS trace II: a cheaper path replaces reached[G]

The new G enters the heap. The old, more expensive heap entry may remain until later.



return S → B → C → G · total cost 6

PRIORITY QUEUE · min g first

STEP 3 **expand B**

[(C, 4), (G, 101)]

g(C)=2+2=4

STEP 4 **expand C**

[(G, 6), (G, 101)]

reached[G] now points to g=6

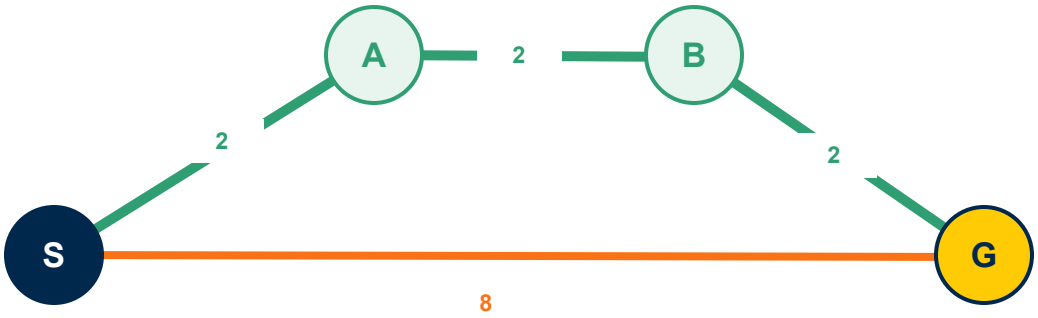
STEP 5 **pop G**

return path

parents recover S → B → C → G

Why UCS waits until the cheapest goal is popped

If a cheaper route is unfinished, one of its nodes must still appear ahead of the expensive goal.



direct goal: cost 8

cheaper path: cost 6

PRIORITY QUEUE · smallest g first

STEP 1 **expand S**

[(A, 2), (G, 8)]

A must be popped first

STEP 2 **expand A**

[(B, 4), (G, 8)]

the cheap route is still represented

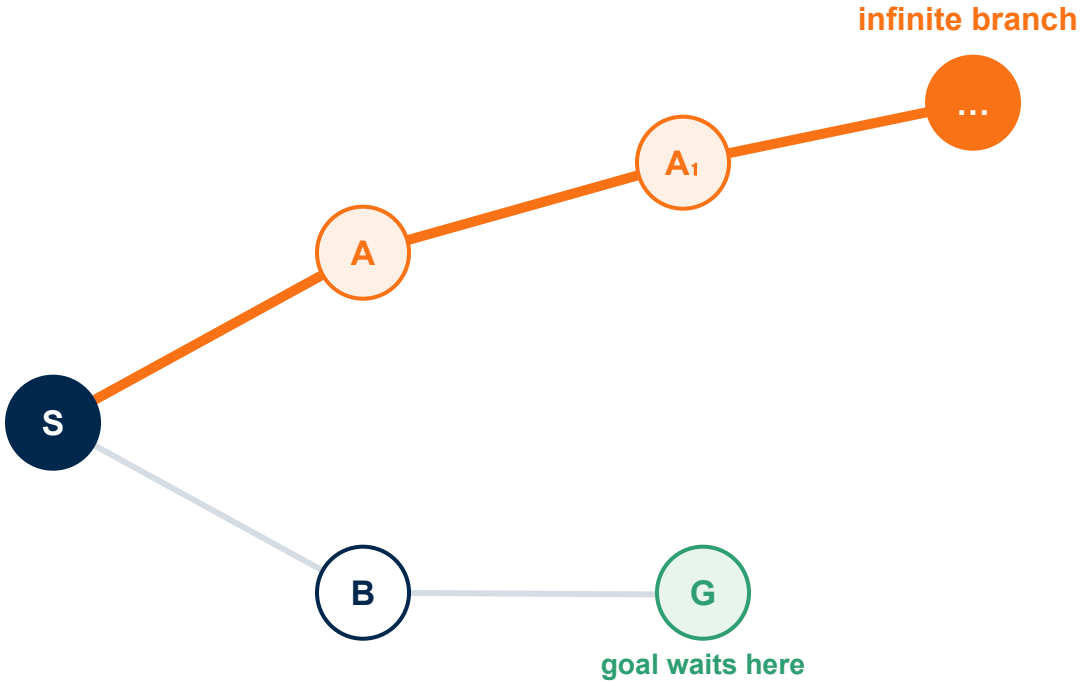
STEP 3 **expand B**

[(G, 6), (G, 8)]

pop G at 6; the cost-8 entry waits

DFS keeps following the newest branch

Successor order A before B makes a LIFO stack keep diving down the A branch.



LIFO STACK · top at left

start	[S]
expand S	[A, B]
expand A	[A_1, B]
expand A_1	[A_2, B]

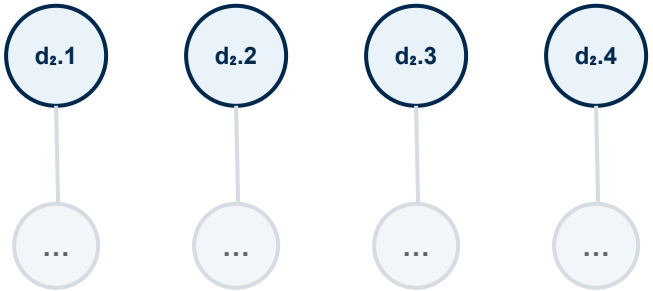
B and G may wait forever

DFS behavior depends on successor order.

Why DFS usually stores less than BFS

BFS keeps a whole layer. DFS keeps one active path plus a few waiting siblings.

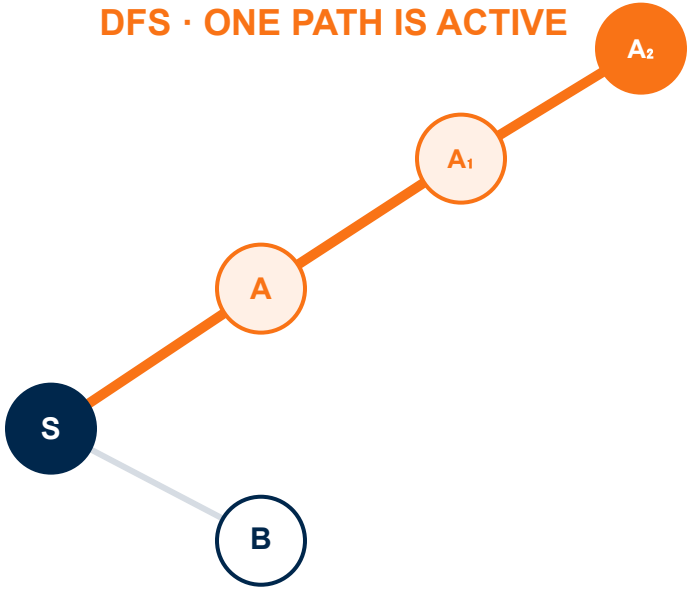
BFS · WHOLE LAYER WAITS



frontier = many nodes at depth 2

stores the width of the search tree

DFS · ONE PATH IS ACTIVE



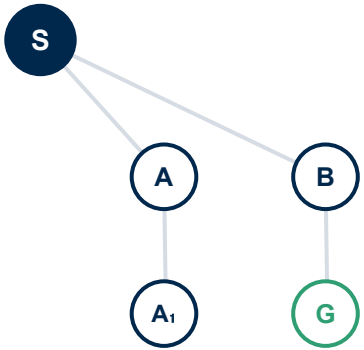
stack top: [A₂, waiting siblings, B]

Memory advantage does not make DFS complete or optimal.

IDDFS restarts depth-limited DFS

A cutoff means a deeper solution may exist. Each larger limit restarts from S.

DEPTH LIMIT = 0

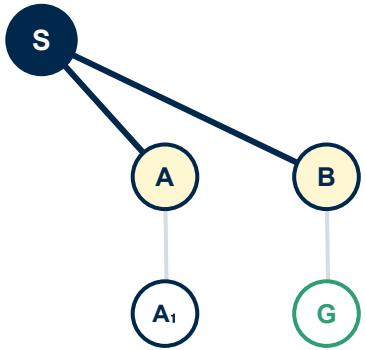


visited: S

cutoff

restart →

DEPTH LIMIT = 1

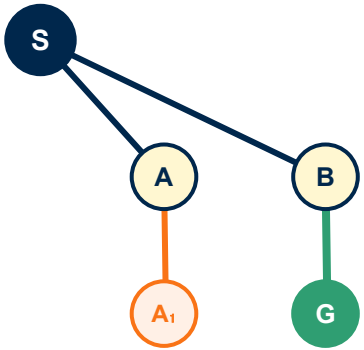


visited: S, A, B

cutoff

restart →

DEPTH LIMIT = 2



visited: S, A, A_1, B, G

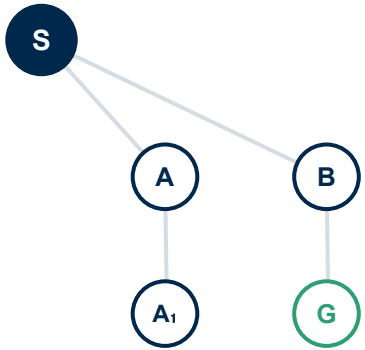
return S → B → G

Nodes at the limit are visited, but their children are not expanded.

IDDFS repeats shallow work to keep memory small

Each pass restarts at S, but only the current DFS path and waiting siblings stay in memory.

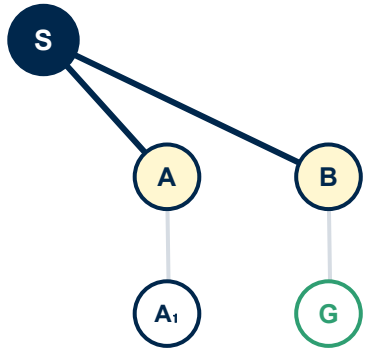
LIMIT 0



visit 1 node

max stack: 1

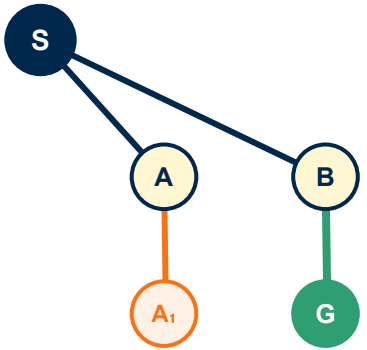
LIMIT 1



visit 3 nodes

max stack: 2

LIMIT 2



visit 5 nodes

max stack: 3

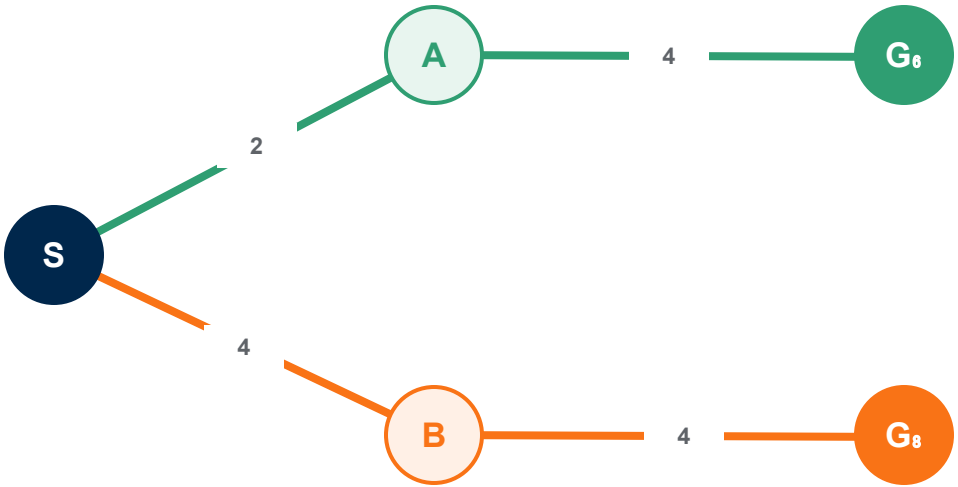
S is visited 3 times

A and B are visited twice

but the deepest layer is visited once

Why fixed cost-limit increments are not enough

Suppose we copy IDDFS but limit path cost instead of depth, adding 5 after every cutoff.



NAIVE COST-LIMITED DFS

STEP 1 — **limit = 5**

no goal allowed

both paths exceed 5

STEP 2 — **jump to 10**

G₈ is allowed

DFS sees B first → returns cost 8

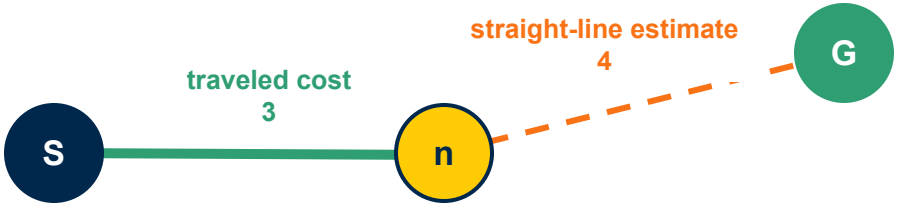
The jump skips C*=6; without C*, no fixed increment is guaranteed safe.

A* adds an estimate of the remaining cost

- Unlike uninformed search, the heuristic may inspect what the state looks like.

g(n)	real edge costs already taken	h(n)	quick estimate from the current state	f(n)	g(n)+h(n): A*'s heap priority

NAVIGATION INTUITION



Actual cheapest remaining road: unknown

g=3	h=4	f=7

COURSE CODE

BEFORE SEARCH

```
algorithm = AStar(heuristic)
```

WHEN A SUCCESSOR IS GENERATED

```
g_n = node.cost + a_cost
```

```
f_n = g_n + self.heuristic(env, next_s)
```

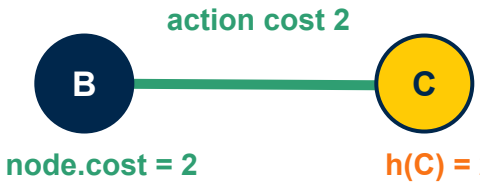
```
heapq.heappush(frontier, (f_n, it, next_node))
```

No useful estimate? h=0 ⇒ A* has UCS ordering.

A* in code: one pop-to-push iteration

Follow one successor B → C through pop, cost update, reached, and heap push.

1 POP	`_, _, node = heapq.heappop(frontier)`
2 EXPAND	`for a, a_cost in env.actions(node.state):`
3 ADD COST	`g_n = node.cost + a_cost`
4 COMPARE	`if next_s not in reached or g_n < reached[next_s].cost:`
5 PRIORITY	`f_n = g_n + self.heuristic(env, next_s)`
6 PUSH	`heapq.heappush(frontier, (f_n, it, next_node))`



BEFORE

g(C)=4, f(C)=6

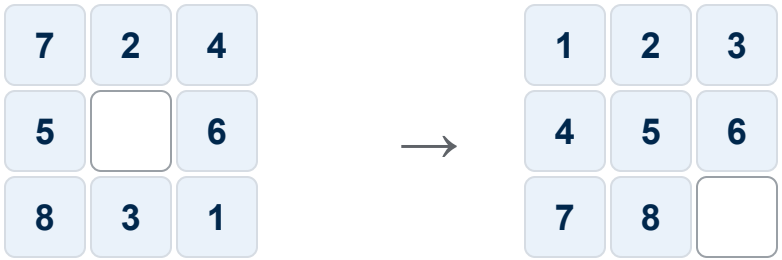
frontier [(B,f=5), (A,f=7)]

AFTER

frontier [(C,f=6), (A,f=7)]

8-puzzle heuristics: do not count the blank

The repository's puzzle state is a tuple; tile 0 represents the blank.



optimal path length = 20

misplaced tiles

count nonblank tiles outside their goal cell

h_misplaced(start) = 6

Manhattan distance

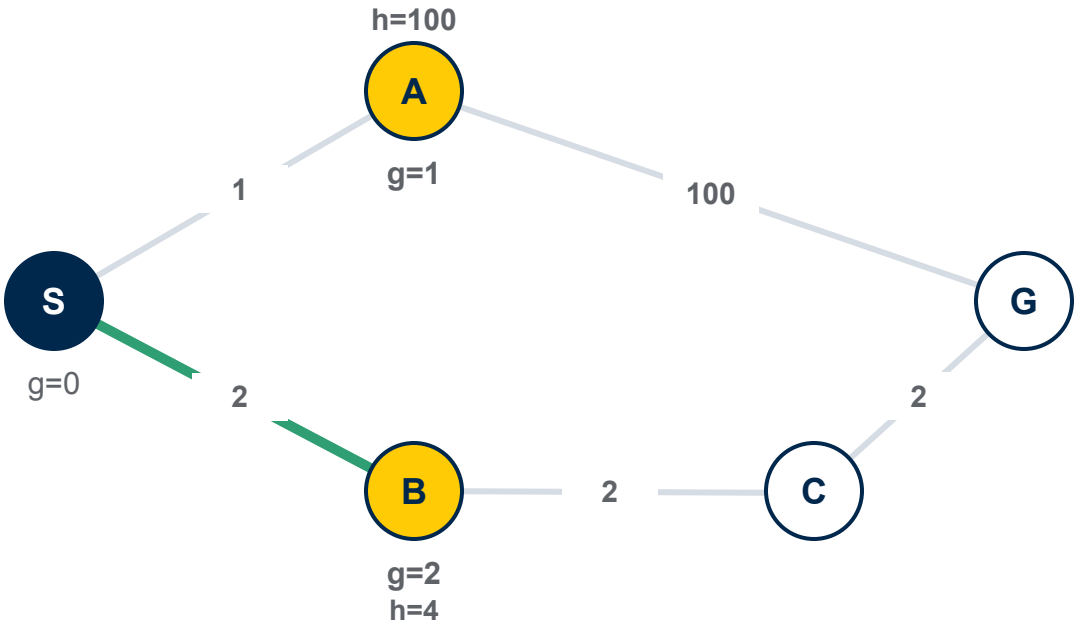
sum |Δrow| + |Δcolumn| for every nonblank tile

h_Manhattan(start) = 14

correct condition: tile != 0 and tile != goal[index]

A* ranks frontier nodes by f = g + h

g(n) is the cost already paid; h(n) estimates the remaining cost.



AFTER EXPANDING S

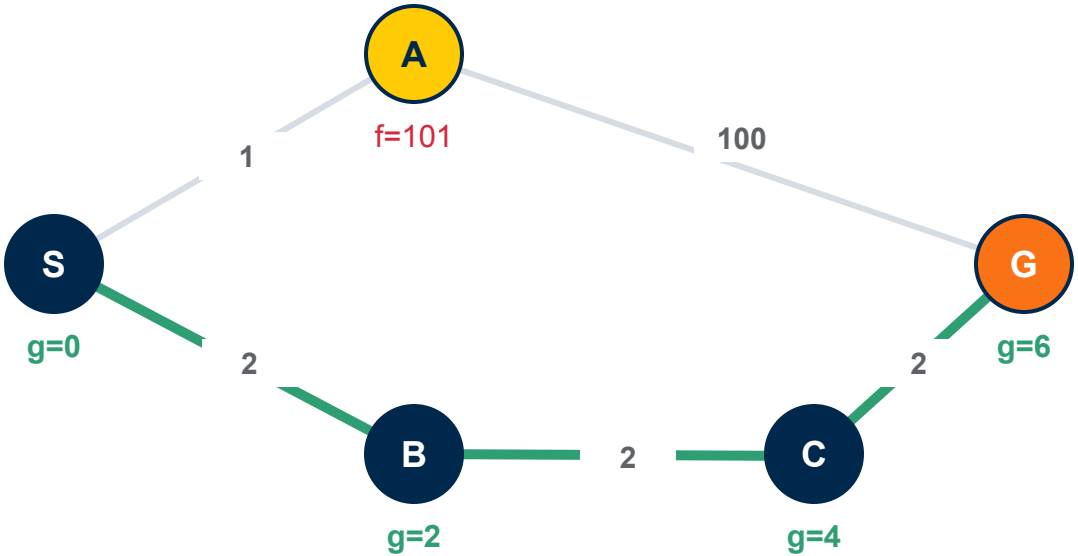
1	B	2 + 4 = 6 g + h = f
2	A	1 + 100 = 101 g + h = f

next pop: B

A has smaller g, but B has smaller estimated total cost f.

A* follows the smallest f while g tracks real cost

Along the selected path, g records 0 → 2 → 4 → 6; parent pointers recover the route.



SELECTED BY f = g + h

B: g=2, h=4 → f=6

C: g=4, h=2 → f=6

G: g=6, h=0 → f=6

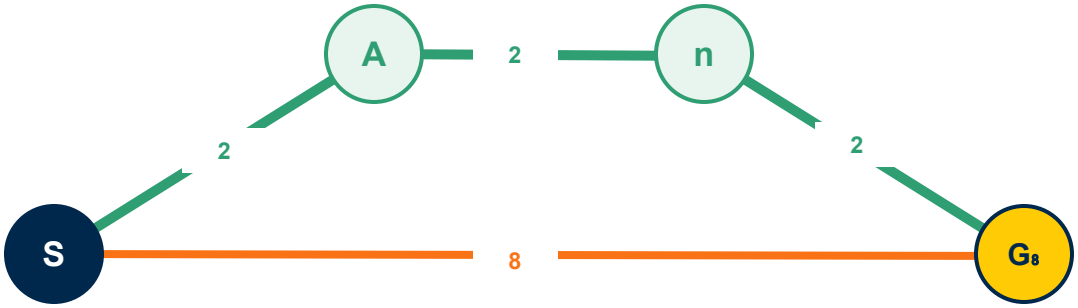
pop order: S → B → C → G

A remains in the frontier with f=101.

return path: S → B → C → G · total cost 6

Why an admissible h makes A* cost-optimal

An unfinished cheaper solution must still have a frontier node ahead of a more expensive goal.



Suppose G_8 is about to be popped

goal: $f(G_8)=g(G_8)=8$

But node n is still on the cost-6 path

g(n)=4

admissible h(n) ≤ 2

therefore $f(n)=g+h \le 6 < 8$

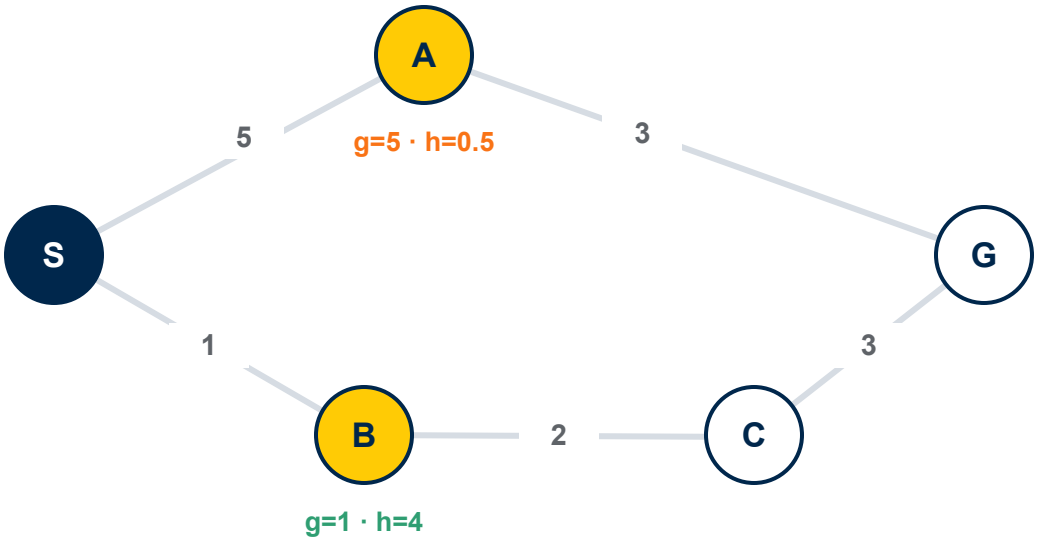
cheaper solution cost = 6

A* must pop n before G_8

So a cost-8 goal cannot be returned while a cost-6 solution remains.

Weighting h can reverse the next pop

After expanding S, A* tries B; Weighted A* with w=2 tries A.



AFTER EXPANDING S

A* · priority g+h

B: 1+4=5

A: 5+0.5=5.5

next pop: B

Weighted A* · priority g+2h

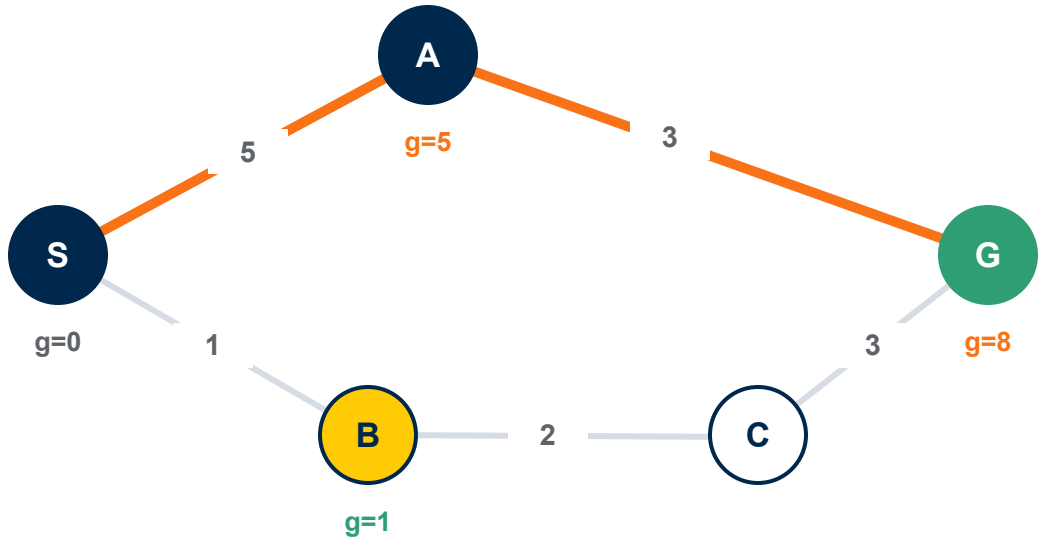
A: 5+2(0.5)=6

B: 1+2(4)=9

next pop: A

Weighted A* may return sooner but with a costlier path

With w=2, the goal through A reaches the front before the cheaper B route is explored.



returned cost 8 ≤ w·C* = 2·6 = 12

WEIGHTED A* QUEUE · smallest g+2h

STEP 1 — **expand S**

[(A, 6), (B, 9)]

pop A

STEP 2 — **expand A**

[(G, 8), (B, 9)]

A → G adds real cost 3

STEP 3 — **pop G**

return cost 8

the cost-6 route via B remains unexplored

One search loop, four frontier orders

The priority rule changes which node is expanded next

BFS

priority: FIFO / depth
good for equal-cost steps
complete for finite branching; optimal if costs match

UCS

priority: smallest g
handles different edge costs
complete and optimal if each step cost ≥ ε > 0

DFS

priority: LIFO / deepest
usually much smaller frontier
not generally complete or optimal

A*

priority: smallest g+h
uses a problem-specific estimate
optimal with admissible h; h=0 gives UCS ordering

COMMON EXTENSIONS

IDDFS: repeat depth limits

Weighted A*: use g+w·h

IDA*: repeat f limits