

Lecture 2: Search

September 2nd, 2026



Gabriel Poesia

Assistant Professor, CSE

University of Michigan, Ann Arbor

poesia@umich.edu

<https://gpoesia.com>

0. Administrivia
1. Search problems
2. Best-first search
3. Uninformed search
4. Heuristics and A*
5. Live coding
6. Search with learned heuristics

Slides (including these) are on the course website and on Canvas

Lecture recordings also on Canvas

Live coding will be pushed to Github

Small fixes to schedule:

- Assignment 1 released next Wednesday, Sep 9th (then every ~2 weeks, standard with other assignments)
- Review lecture on Wed, Oct 21, before midterm (schedule was incorrectly showing "Fall study break", which ends Tuesday. Calendar was correct!)

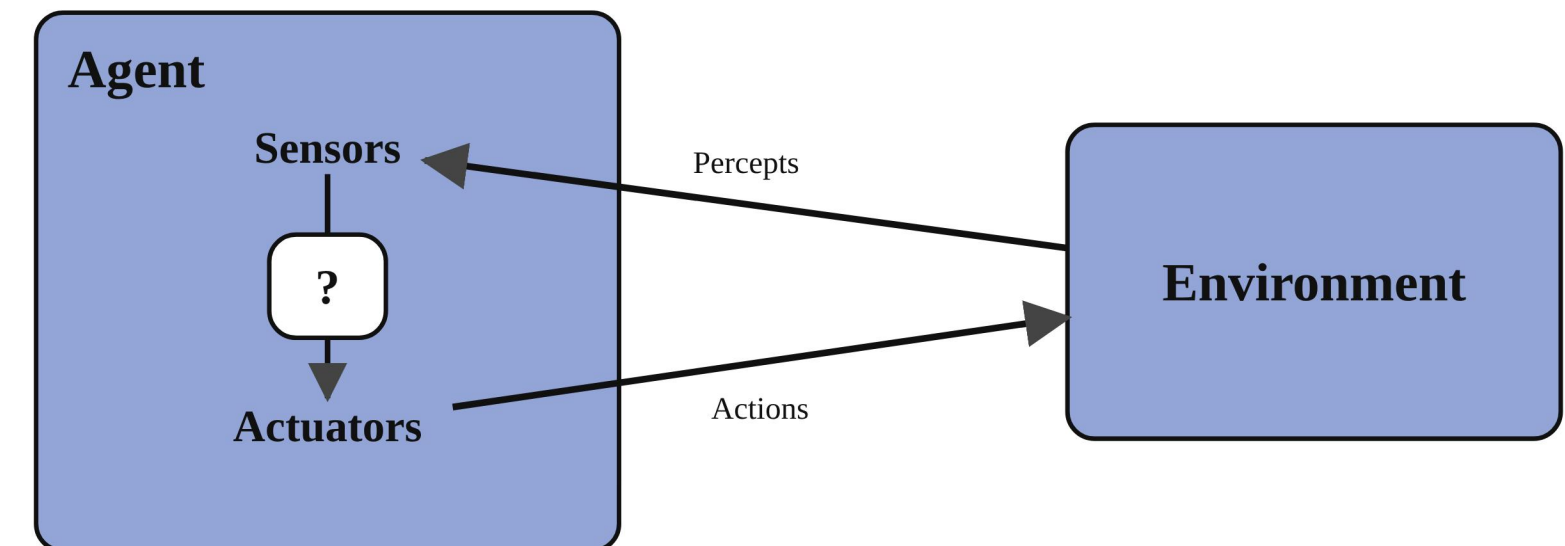
Where we left off

An **agent** perceives an **environment** and acts on it.

We want **rational** agents: ones that "do the right thing" given their knowledge and performance measure.

Today: the simplest environments where "the right thing" can be found by search.

Fully observable, deterministic, static, discrete, single-agent.



SEARCH PROBLEMS

Problem-solving agents

An agent that plans by search is called a **problem-solving agent**.

What does such an environment have to give us?

DEFINITION (SEARCH PROBLEM)

- A set of states S (atomic: we can only compare them)
- An initial state $s_0 \in S$
- Actions $A(s)$ available in each state
- A transition function $T(s, a)$
- An action cost $Cost(s, a) \geq \epsilon$ for some fixed $\epsilon > 0$
- A goal predicate $IsGoal(s)$

Binary performance measure: did we land in a goal state?

Problem-solving agents

These are mathematical *functions*: T and $Cost$ are deterministic, and $A(s)$ is a finite set.

Goal: find a sequence of actions from s_0 to a state where $IsGoal$ holds, with least total cost.

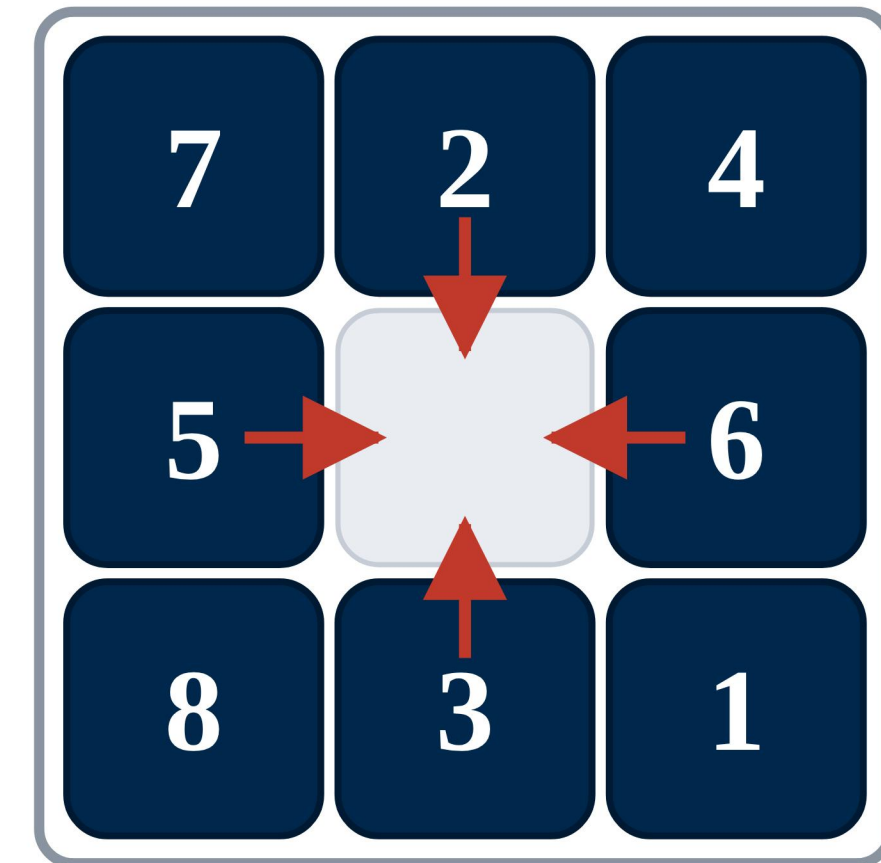
Note what we *don't* get: the set of states, or the graph. Only a starting point and the ability to ask "what's next?".

Methods from this lecture work for *any* environment with this interface.

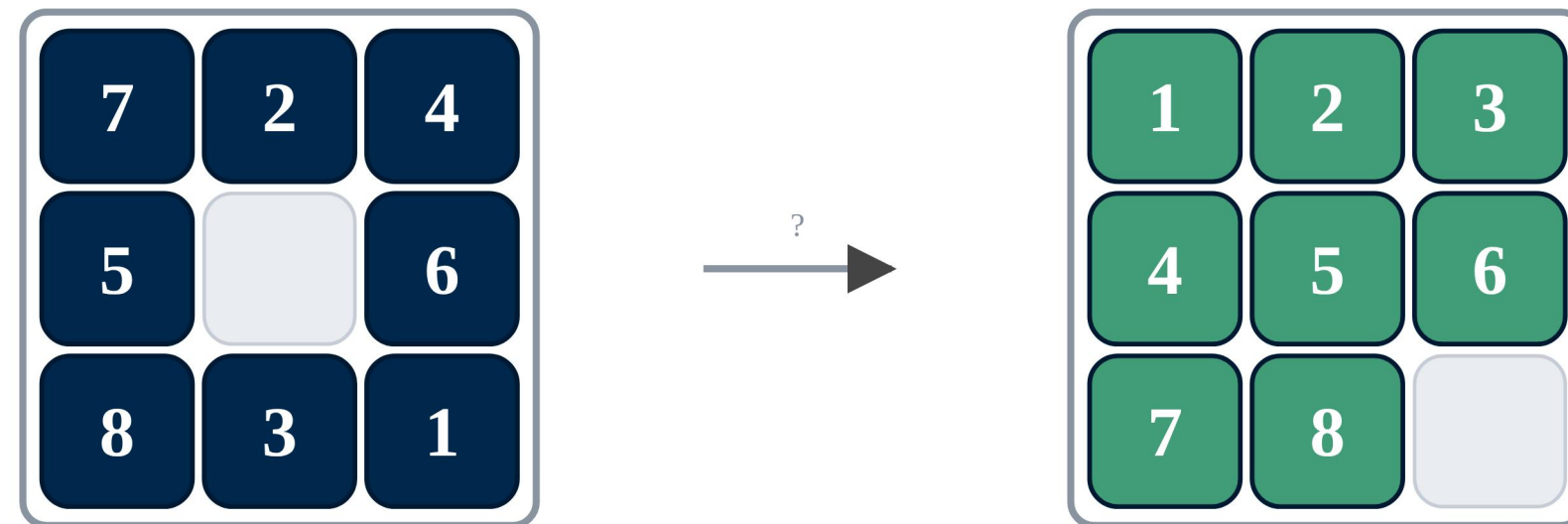
Example: the 8-puzzle

- **State:** grid with tile numbers
- s_0 : a scrambled board
- **Actions:** slide a neighbor tile into the blank
- T : the board after the slide
- **Cost:** 1 per move
- *IsGoal*: tiles in order

Here, 4 actions are available.



Example: the 8-puzzle



181,440 reachable states ($9!/2$), branching factor $b \leq 4$.
15-puzzle: 1.3×10^{13} states. 24-puzzle: 10^{25} .

State space might be larger than memory

Non-example: chess



Source: Wikipedia

Chess has states, actions, and a transition function. But:

- The next state (in our turn) depends on my *opponent*, so $T(s, a)$ is not a function
- *IsGoal* is unclear: checkmate? Sometimes a draw is the best outcome

Two-player games need adversarial search - we'll come back to this in the course.

Example: Sudoku

- **State:** a partially filled grid
- **Actions:** write a digit in an empty cell without violating constraints
- **Cost:** 1 per digit
- *IsGoal:* full grid

Valid search problem by our definition, but we ignore useful structure in the states and actions.

We also don't care about the particular *path* - actions can be freely reordered (so we ignore symmetry)

We'll have better methods for this and similar problems in the next lecture (Constraint Satisfaction).

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Search in the real world

EXAMPLE: VLSI LAYOUT

Place and route components on a chip

Goals: minimize area, wire length, etc.

EXAMPLE: ROBOT NAVIGATION

Navigate a robot through an environment to reach a destination while avoiding obstacles

Challenges: need to deal with uncertainty in (continuous) actuators and sensors

EXAMPLE: ASSEMBLY SEQUENCING

Find order for a robot to assemble the parts of some goal object

Challenges: generating valid actions is itself a complex geometrical/navigation problem

EXAMPLE: PROTEIN DESIGN

Find a sequence of amino acids that folds into a desired structure

Challenges: *IsGoal* (does it fold into the desired structure?) is itself approximate

We will see general ideas that can be adapted to each domain

BEST-FIRST SEARCH

One loop, many search algorithms

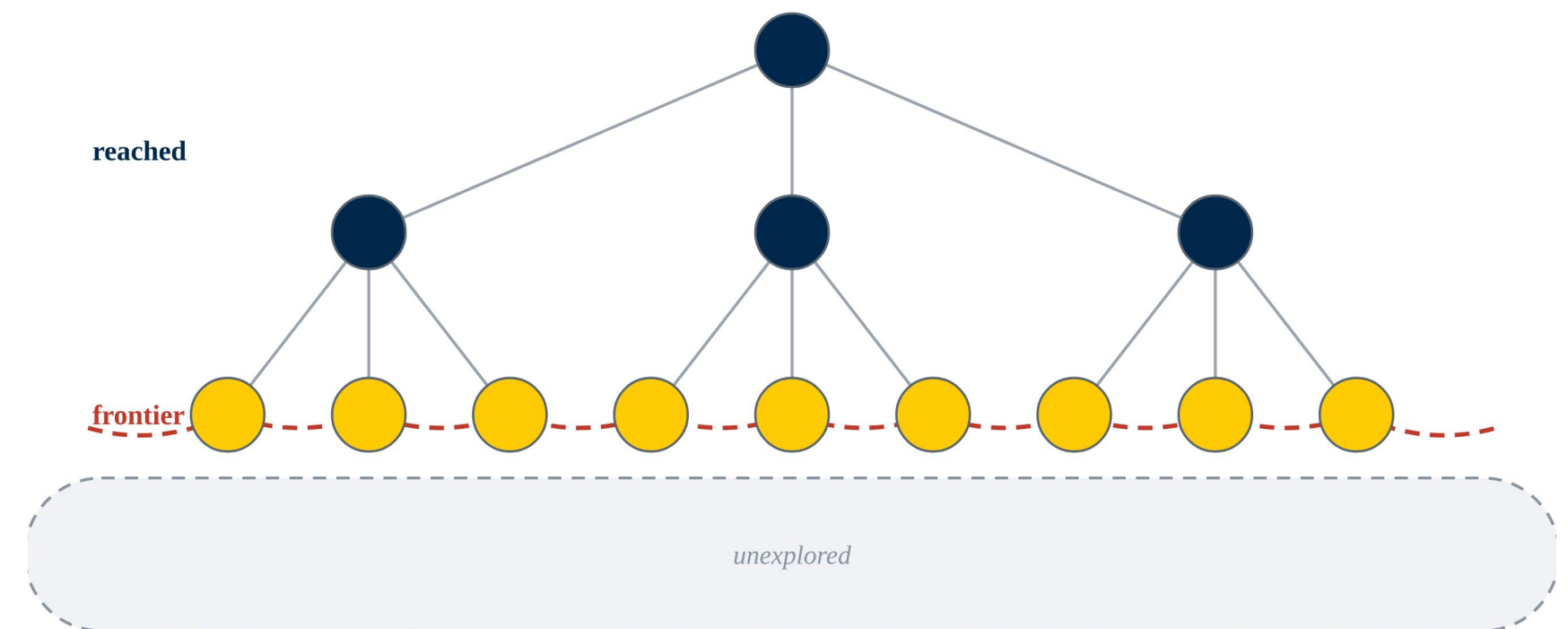
We will keep two data structures:

reached	discovered states so far + best known path to them
frontier	states to be expanded

The idea of Best-First Search is to repeat:

1. Remove the "best" node n from the frontier
2. If n is a goal, return path to n
3. If not, expand n and update both sets

Changing what is "best" will yield different algorithms



Best-first search

```
1  function BEST-FIRST-SEARCH(problem, f):
2      node = Node(problem.s0, path_cost=0)
3      frontier = PriorityQueue(key=f, items=[node])
4      reached = {problem.s0: node}
5      while frontier is not empty:
6          node = frontier.pop()          # node has smallest value of f(.)
7          if problem.IsGoal(node.state): return node
8          for child in EXPAND(problem, node):
9              if child.state not in reached
10                 or child.path_cost < reached[child.state].path_cost:
11                 reached[child.state] = child
12                 frontier.add(child)
13      return failure
```

Changing the priority f allows us to get a whole family of algorithms.

A cheaper path to a reached state is not discarded: in general, states might re-enter the frontier.

The frontier bound

Suppose P is any path from s_0 to a goal state.

If search has not returned yet, P must cross some state represented in the frontier.

In fact, we can show that best-first search reached that frontier state at no greater cost than P did.

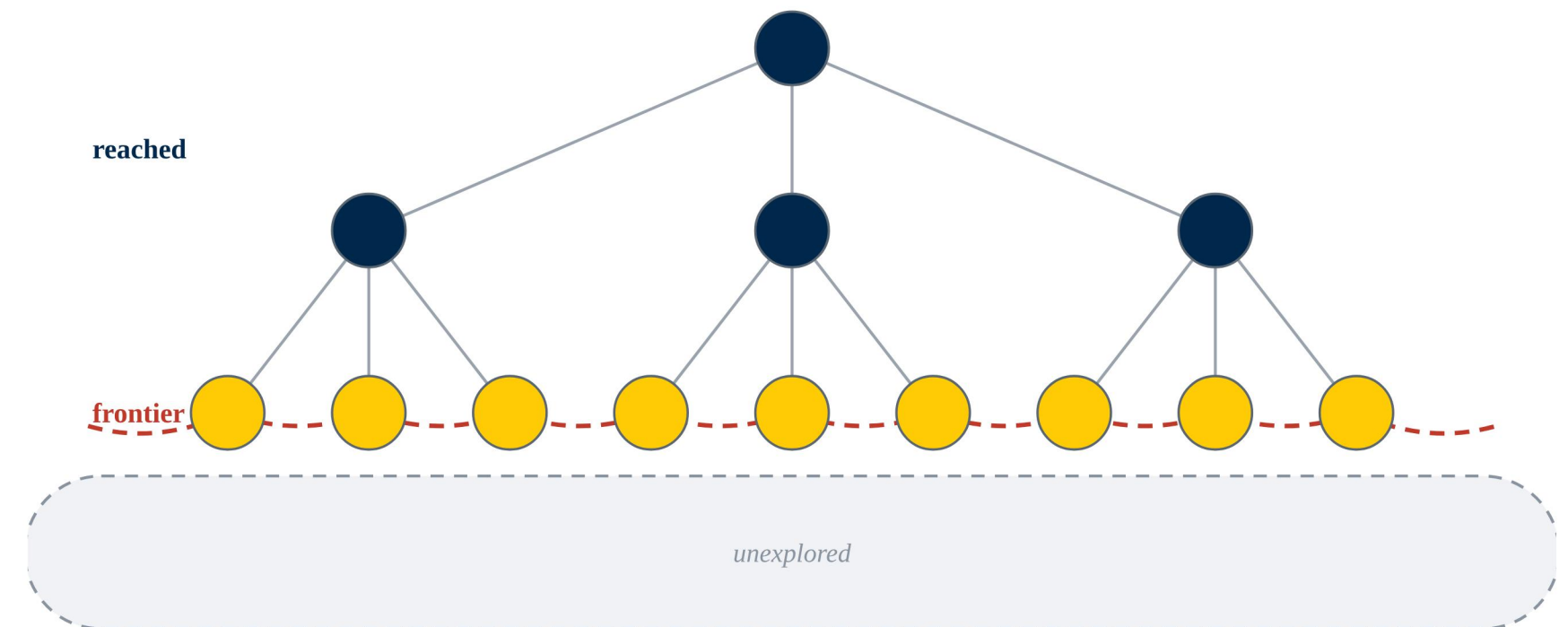
LEMMA (FRONTIER BOUND)

At the start of each iteration, every solution path P contains a state P_i represented by some frontier node n with $g(n) \leq \text{cost}(P_0 \rightarrow \dots \rightarrow P_i)$

So the frontier gives us a lower bound on every solution: even ones we have not reached yet.

We'll use this to prove optimality.

Note: we haven't specified f yet. This holds regardless of how we prioritize the frontier!



What is "best"?

We need a priority $f(n)$. What do we have to work with?

Uninformed search: f uses only the structure of the search (e.g., depth, cost so far)

Informed search: f also uses problem-specific heuristics of how promising a state is.

UNINFORMED SEARCH

Breadth-first search

Simplest choice: $f(n) = \mathbf{depth}$ of n (number of actions).

Shallowest nodes first (here, a FIFO queue would work as well)

Is this a good choice? We'll judge every algorithm on four criteria:

DEFINITION (COMPLETENESS)

A search algorithm is **complete** if it is guaranteed to find a solution whenever one exists.

DEFINITION (COST-OPTIMALITY)

A search algorithm is **cost-optimal** if whenever it returns a solution with cost C , any other solutions have cost at least C .

plus **time** and **space** complexities

In graph algorithms, we typically analyze time and space in terms of $|V|$ and $|E|$.

Here, we don't care about the whole state space. We can be more informative by using:

- b = branching factor (max number of children per node)
- d = depth of the shallowest solution

BFS: properties

Complete?	Yes
Cost-optimal?	Only if all action costs are equal
Time	$O(b^d)$
Space	$O(b^d)$

Frontier grows exponentially with depth: BFS struggles with deep solutions

Uniform-cost search

Fix the optimality problem: let $f(n) = g(n)$, the **cost so far**.

This gives *uniform-cost search* (UCS)

(why? think of BFS as "uniform-depth search")

This is Dijkstra's algorithm on a graph we discover as we go.

NOTE

If all action costs are equal, UCS degenerates to BFS.

Uniform-cost search is optimal

PROOF

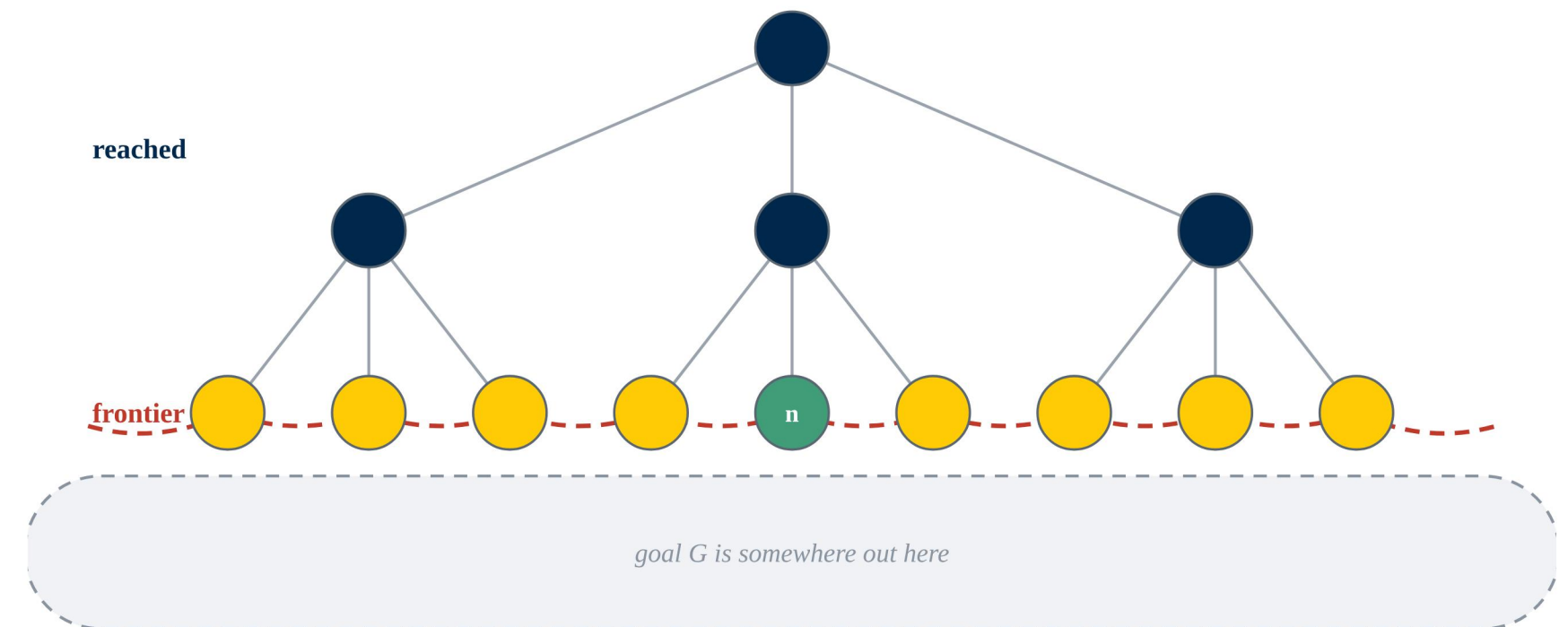
Suppose UCS is about to pop a goal G with cost c .

Consider any solution path $P = (P_0, \dots, P_k)$, with total cost C .

By the frontier bound, some frontier node n represents a state P_i on P , with $g(n) \leq \text{cost}(P_0 \rightarrow \dots \rightarrow P_i) \leq C$.

UCS pops the node with smallest g , so $c = g(G) \leq g(n) \leq C$.

This holds for every solution P . Thus, G is cost-optimal. $\square$



UCS: properties

Complete?

Yes

Cost-optimal?

Yes

Time

$O(b^{1+\lfloor C^*/\epsilon \rfloor})$

Space

$O(b^{1+\lfloor C^*/\epsilon \rfloor})$

C^* = optimal cost, ϵ = a global lower bound on action cost.

So, $\frac{C^*}{\epsilon}$ is how deep UCS has to go to find a solution.

When is UCS slow (compared to BFS)? When goal is shallow but expensive, and lots of cheap actions leading away from the goal.

Cost-optimal, but still slow and memory hungry

Depth-first search

Invert the cost in BFS: instead of shallowest, take deepest node in the frontier.

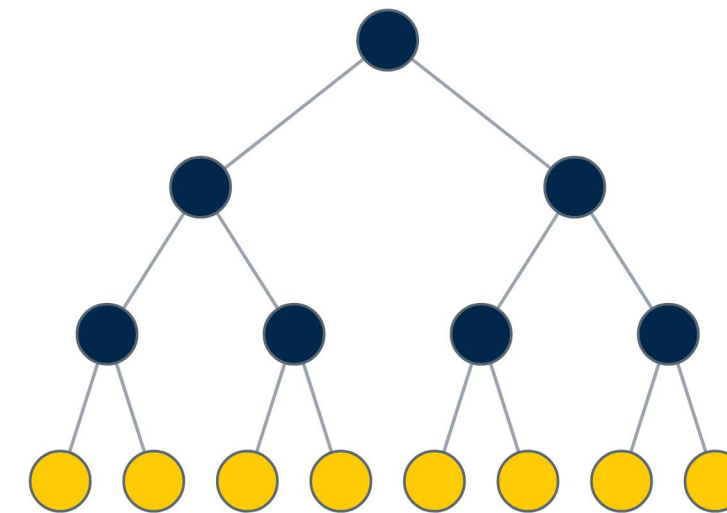
Can be drastically more efficient by using a stack

Complete? No! can get stuck in infinite branches.

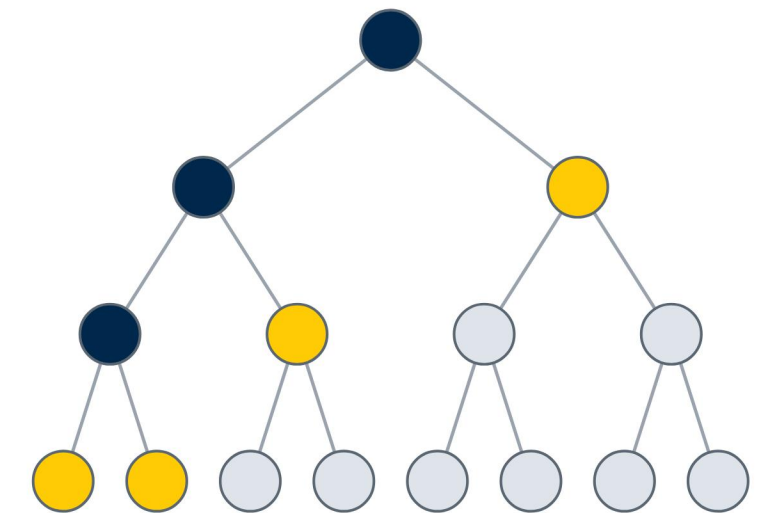
Optimal? No! returns the first solution it finds.

But: space $O(bm)$, linear in max depth (assuming we limit depth at m to avoid non-termination).

That might be useful: in hard problems, memory is often the constraint



BFS: frontier is a whole level



DFS: frontier is around one path

Iterative Deepening

Can we get cost-optimality guarantees with DFS's memory complexity?

ID-DFS: run DFS with a depth limit L . Try $L = 0, 1, 2, \dots$ until a solution appears.

Each iteration starts from scratch.

Asymptotically, how much more time does this take compared to BFS?

A. b times more

B. d times more

C. The same

D. Exponentially more

Iterative deepening: analysis

The tree grows geometrically, so the *bottom level dominates*:

$$N_{ID-DFS} = (d+1)b^0 + db^1 + \dots + 2b^{d-1} + b^d = O(b^d)$$

$$N_{BFS} = b^0 + b^1 + \dots + b^d = O(b^d)$$

Concretely, with $b = 10$, $d = 5$:

	Nodes generated	Nodes in memory
BFS	111,110	111,110
ID-DFS	123,450	~50

In practice, saving memory can often lead to saving time (due to caching, locality, etc.)

ID-DFS: properties

Complete?	Yes
Cost-optimal?	Only if all action costs are equal (like BFS)
Time	$O(b^d)$
Space	$O(bd)$

So we've recovered BFS in linear space, but we're back to ignoring costs.

How can we get cost-optimality in linear space?

Iterate on *cost* instead of depth?

Iterating on cost

Natural attempt: DFS that prunes any node with $g(n) > limit$, then iteratively increase limit.

By how much do we increase it?

Simple idea: pick a fixed increment δ , so $L = \delta, 2\delta, 3\delta, \dots$

Cost-optimal? Only if δ divides C^* , otherwise we might overshoot and miss the optimal solution.

Without knowing C^* , we can't guarantee our choice of δ is good.

Satisficing search

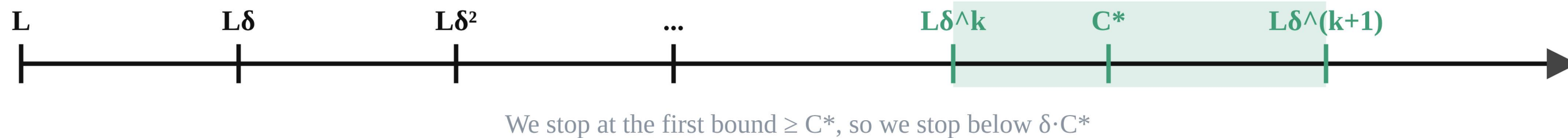
Suppose we give up on *exact* optimality, and accept to be within a factor δ of it.

DEFINITION (δ -OPTIMAL)

A search algorithm is δ -**optimal** ($\delta \geq 1$) if it returns a solution of cost at most $\delta \cdot C^*$.

$\delta = 1.1$: "I'll accept 10% worse than the best possible."

For $\delta > 1$, start with some limit $0 < L \leq C^*$ and **multiply** by δ each iteration.



DFS passes: at most $1 + \lceil \log_\delta(C^*/L) \rceil$

Recap: uninformed search

	Complete?	Optimal?	Time	Space
BFS	Yes	If costs all equal	$O(b^d)$	$O(b^d)$
UCS	Yes	Yes	$O(b^{C^*/\epsilon})$	$O(b^{C^*/\epsilon})$
DFS	No	No	$O(b^m)$	$O(bm)$
ID-DFS	Yes	If costs all equal	$O(b^d)$	$O(bd)$
Satisficing ID-DFS	Yes	δ -optimal	$O(b^{\delta C^*/\epsilon})$	$O(b\delta C^*/\epsilon)$

Very general: we assume nothing about what the states actually look like.

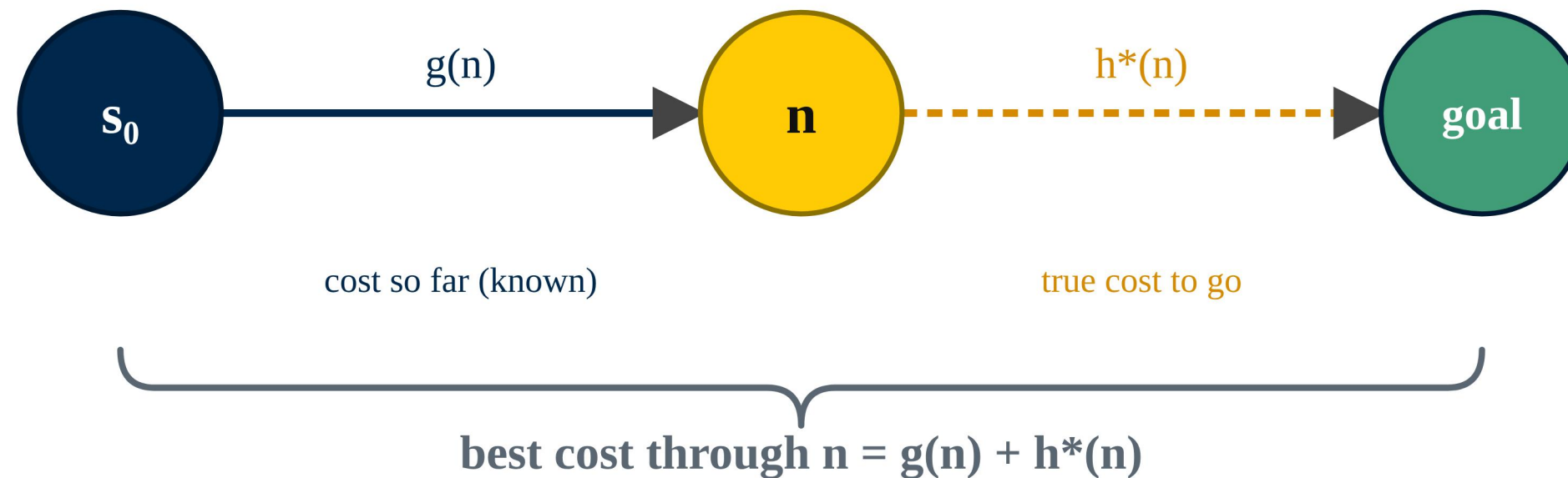
Downside of generality: we ignore useful domain-specific assumptions that can allow us to do much better.

Can we leverage domain-specific heuristics that know something about the states?

HEURISTICS AND A*

If we knew the cost to go

Suppose search has reached a node n .



Any solution through n includes the **cost so far**, $g(n)$.

It also includes the true **cost to go**, $h^*(n)$.

So the best solution through n has cost $g(n) + h^*(n)$.

If we knew h^* , we would not expand a node that cannot lead to an optimal solution.

Usually, we do not know h^* .

But we can often estimate it.

Replace the unknown h^* with an estimate.

DEFINITION (HEURISTIC)

A **heuristic** $h(n) \geq 0$ estimates the cost from n to the nearest goal, with $h(n) = 0$ at every goal.

$g(n)$ comes from the path. $h(n)$ may inspect the representation of the state.

NOTE

$h(n) = 0$ for every node ignores the state and gives uniform-cost search.

A* is best-first search with $f(n) = g(n) + h(n)$:

Known cost so far + estimated cost remaining.

In best-first search, only the priority-queue ordering changes.

Does A return an optimal solution?* Not for an arbitrary h .

But yes if we choose h carefully

DEFINITION (ADMISSIBLE HEURISTIC)

h is **admissible** if $h(n) \leq h^*(n)$ for every node n : it never overestimates the cost to go.

Underestimates are OK: A^* may expand a node optimistically and find out it is expensive.

Overestimates are not: otherwise, we might misjudge a node on an optimal path.

Admissibility only restricts the direction of the error, not its size.

Example: path finding

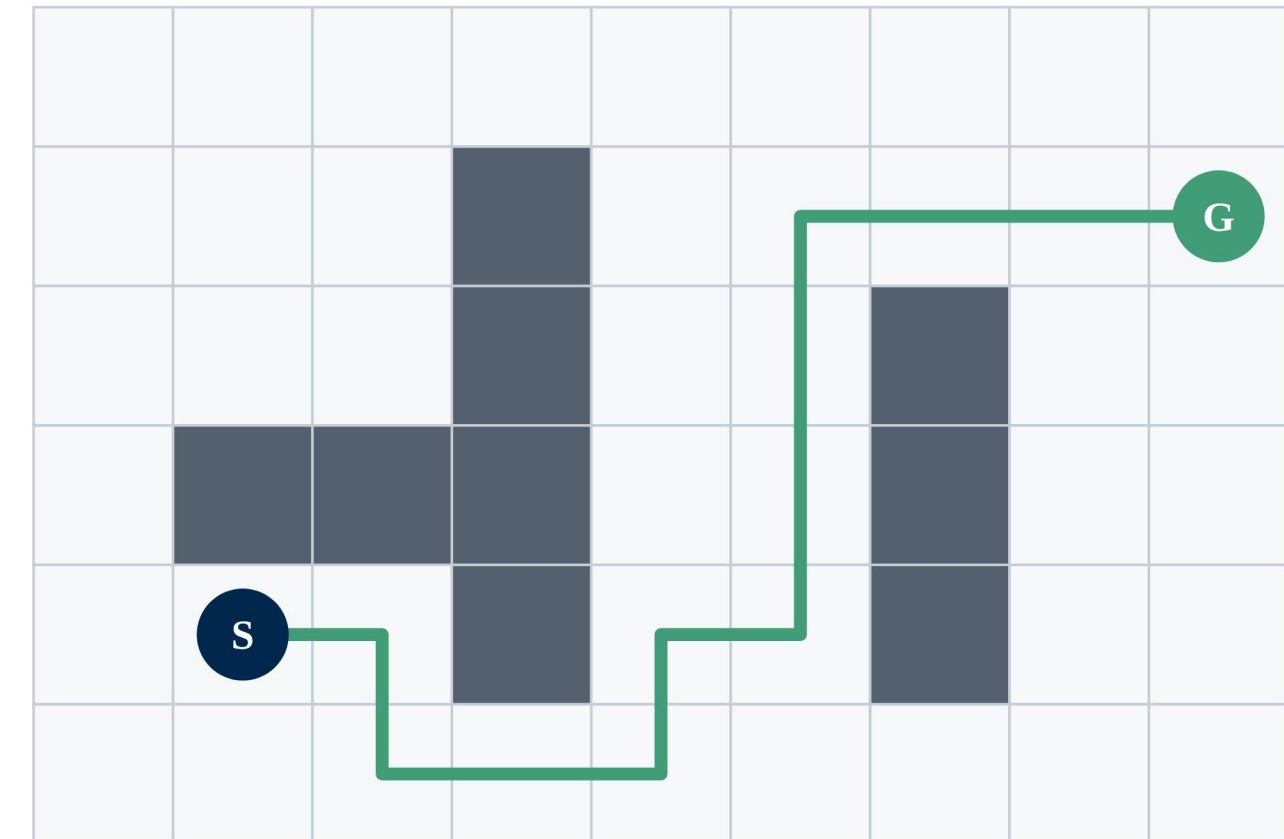
4 moves in a grid with walls.

One move changes x or y by 1. So at least $|x_n - x_G| + |y_n - y_G|$ moves are required.

$$h(n) = |x_n - x_G| + |y_n - y_G| \text{ (Manhattan distance)}$$

Walls can force detours, but not shorten the best path.

Thus $h(n) \leq h^*(n)$.



Deriving heuristics for the 8-puzzle

In a solution, every misplaced tile must move at least once.
One action moves only one tile.

$$h_1(n) = \text{number of misplaced tiles} = 6$$

More precisely: each tile must cross the horizontal and vertical gap to its goal square. One action moves one tile by one square.

$$h_2(n) = \sum_i \text{Manhattan distance of tile } i \text{ to its goal} \\ = 10$$

Both are admissible.

For every state, $h_2 \geq h_1$: h_2 **dominates** h_1 . With the same tie-breaking, A* using h_2 expands no more nodes.



goal

A systematic source of heuristics

General idea: make the original problem easier.

DEFINITION (RELAXATION)

P_{rel} is a **relaxation** of P if every solution to P is also a solution to P_{rel} .

The relaxed problem has at least as many solutions, so its optimal cost cannot be larger: $h_{rel}^*(n) \leq h^*(n)$.

Solve the relaxed problem exactly; use its cost as an admissible heuristic.

Two relaxations of the 8-puzzle

Original action: move a tile to an **adjacent** square **if that square is blank**.

Relaxed problem h_1 solves: move a tile to ~~an adjacent~~ **any** square ~~if that square is blank~~ .

Relaxed problem h_2 solves: move a tile to an **adjacent** square ~~if that square is blank~~

Should be careful to not add any restrictions: all solutions to the original problem must still be valid in the relaxed problem.

A* is cost-optimal

THEOREM

If h is admissible, every solution returned by A* is cost-optimal.

PROOF

Suppose A* is about to pop a goal G with cost c .

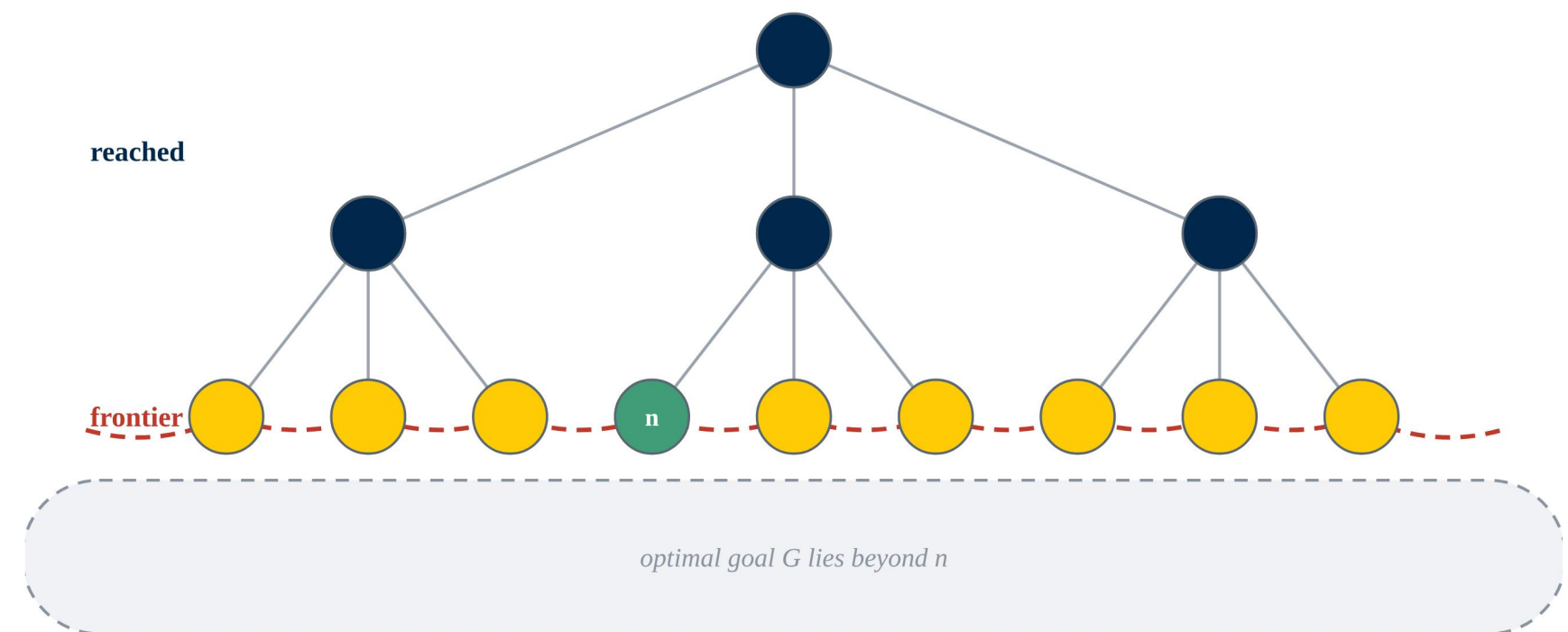
Consider any solution path $P = (P_0, \dots, P_k)$, with total cost C .

By the frontier bound, some frontier node n represents P_i , with $g(n) \leq \text{cost}(P_0 \rightarrow \dots \rightarrow P_i)$.

By admissibility, $h(n) \leq h^*(n) \leq \text{cost}(P_i \rightarrow \dots \rightarrow P_k)$. Therefore $f(n) = g(n) + h(n) \leq C$.

A* pops the smallest f . Since $h(G) = 0$, $c = f(G) \leq f(n) \leq C$.

This holds for every solution P . Thus, G is cost-optimal. $\square$



A*: properties

Complete?	Yes
Cost-optimal?	Yes, if h is admissible
Time	Exponential in worst case - better in practice with good h
Space	Same as time

As with BFS and UCS, memory is usually the limiting resource.

Possible variations:

- Weighted A*: $f(n) = g(n) + W \cdot h(n)$ - rely more heavily on heuristic, get W -optimality
- Iterative Deepening A* (IDA*): linear space by redoing work

LIVE CODING

Live coding

Let's implement A* and try out some heuristics!

SEARCH WITH LEARNED HEURISTICS

Search with learned heuristics

h is a function $S \rightarrow \mathbb{R}$; it need not be written by hand.

A neural network can learn to predict cost-to-go from solved instances.

Learned heuristics are generally not admissible, but can be very good in practice.

Also, learning allows the algorithm to improve over time.

Solving the Rubik's Cube with Deep Reinforcement Learning and Search

Agostinelli, McAleer, Shmakov & Baldi — Nature Machine Intelligence, 2019

Learn h from scrambled cubes, then use it in batched weighted A*. State space: 4.3×10^{19} .

In some cases the action space might be infinite (e.g., language, code, theorem proving), but we can learn to generate actions (e.g., from human data)

BFS-Prover: Scalable Best-First Tree Search for LLM-based Automatic Theorem Proving

Xin et al. — arXiv, 2025

- **Uninformed search:** BFS, UCS, DFS, ID-DFS: optimality + space/time trade-offs
- **Informed search:** A*, optimality with admissible heuristics
- **Relaxations:** heuristics can be derived from relaxations of the original problem
- **Modern search:** often uses learned heuristics and action generators

Next lecture

Today: **atomic** states. Search used equality, actions, costs, and a goal test.

Next lecture (Wed): constraint satisfaction, where we assume **factored** states and constraints (e.g., Sudoku).

NOTE

Reading: AIMA chapter 3. Monday is Labor Day!