Artificial Intelligence

Search
(Chapter 3)

Instructor: Qiang Yu

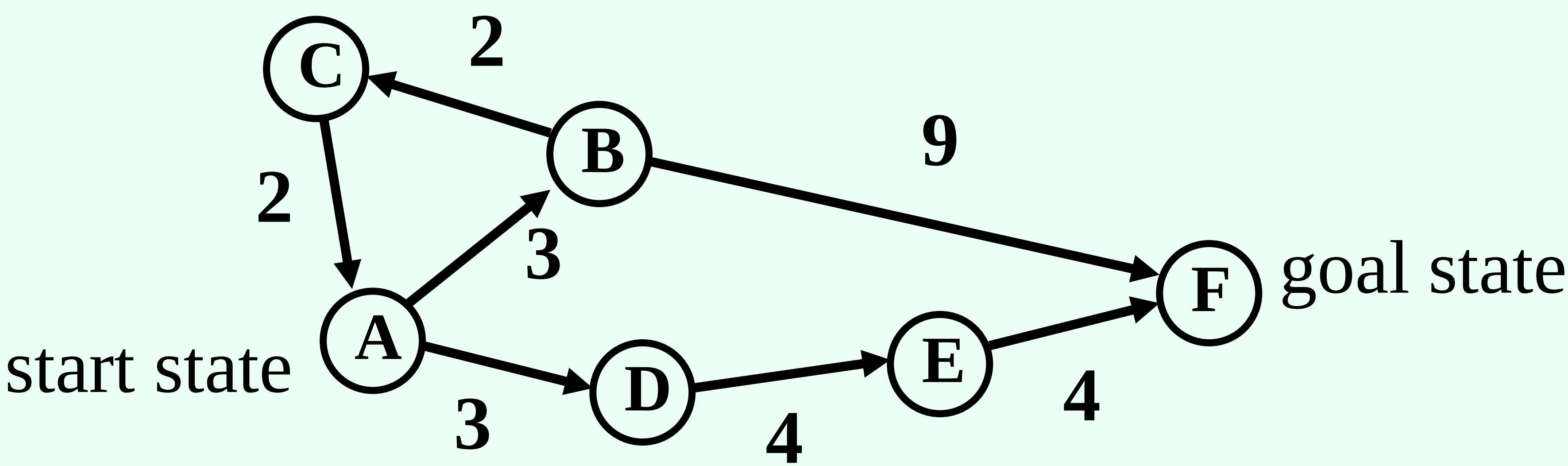
Rubik’s Cube robot

- https://www.youtube.com/watch?v=iBE46R-fD6M

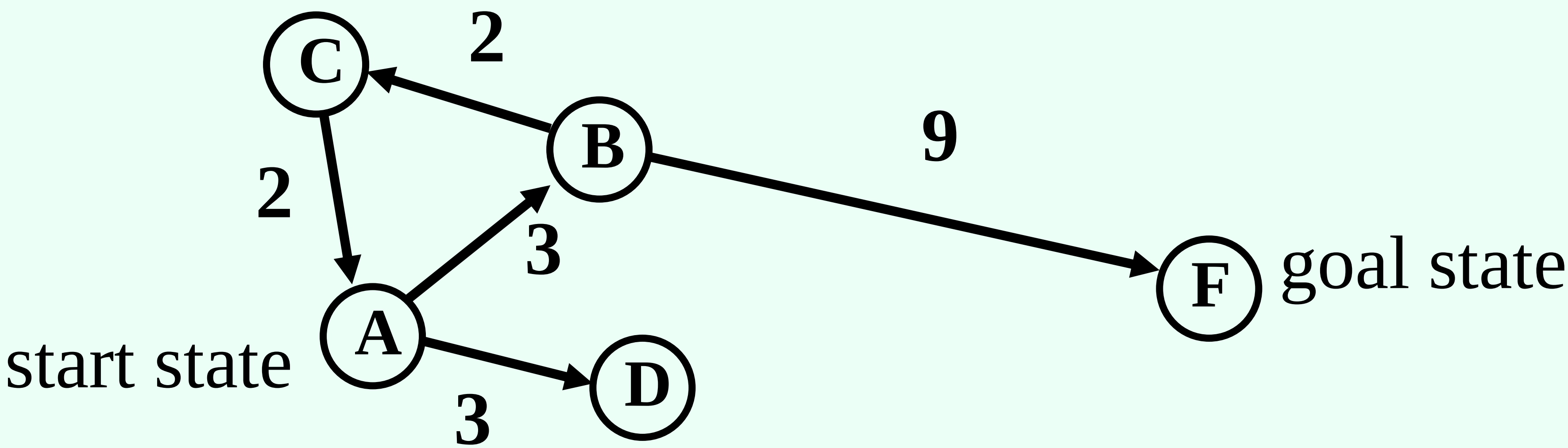
Search

- We have some actions that can change the state of the world
  - Change induced by an action is perfectly predictable
- Try to come up with a sequence of actions that will lead us to a goal state
  - May want to minimize number of actions
  - More generally, may want to minimize total cost of actions
- Do not need to execute actions in real life while searching for solution!
  - Everything perfectly predictable anyway

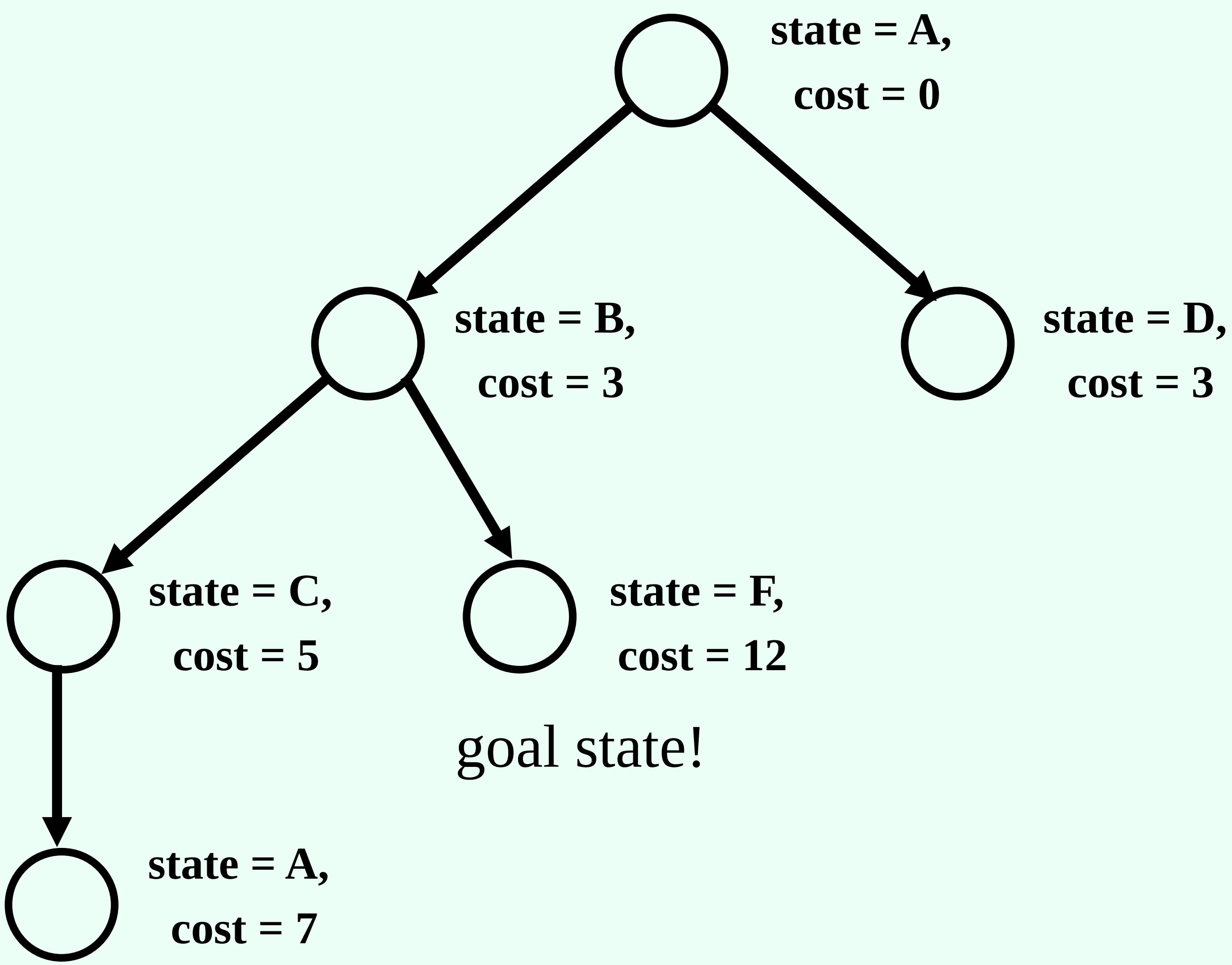
A simple example: traveling on a graph



Searching for a solution

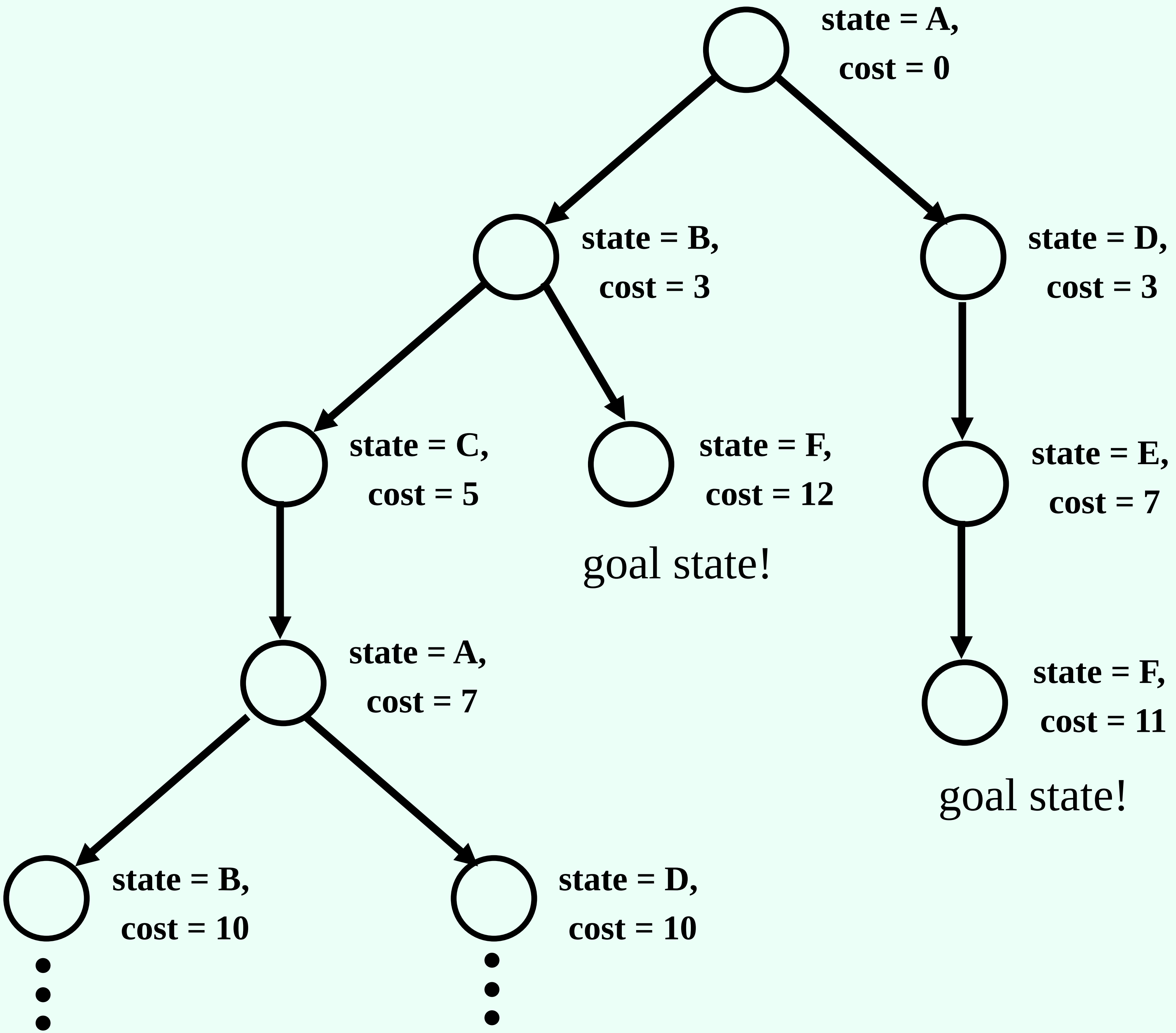


Search tree

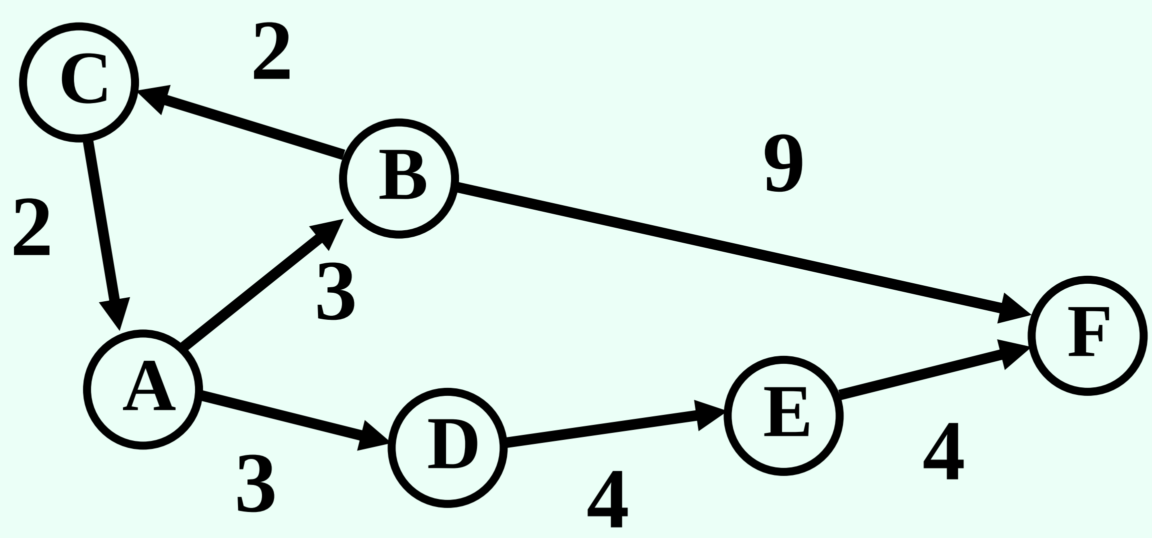


search tree nodes and states are not the same thing!

Full search tree



Changing the goal: want to visit all vertices on the graph



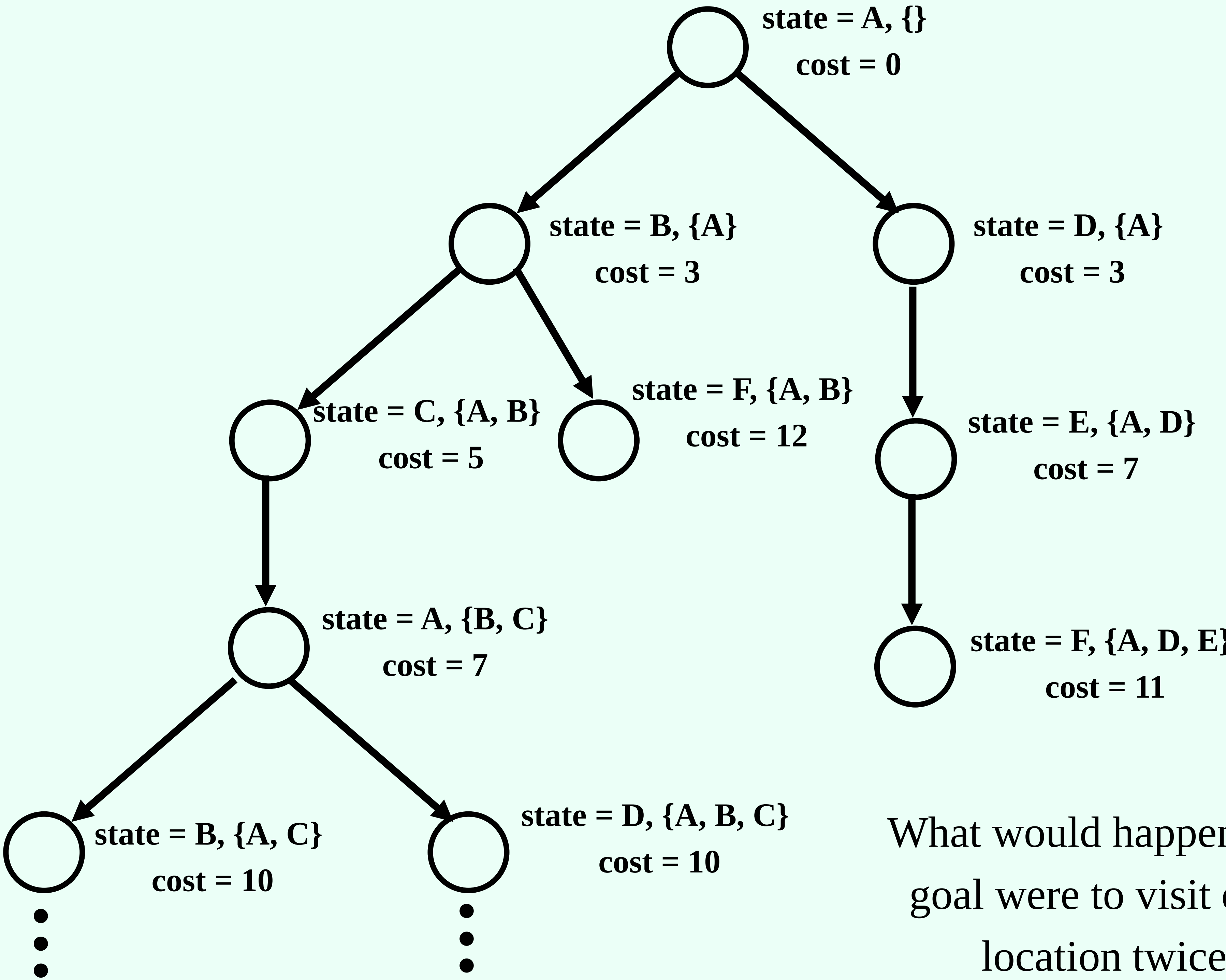
need a different definition of a state

“currently at A, also visited B, C already”

large number of states: $n*2^{n-1}$

could turn these into a graph, but…

Full search tree



What would happen if the goal were to visit every location twice?

Key concepts in search

- Set of states that we can be in
  - Including an initial state…
  - … and goal states (equivalently, a goal test)
- For every state, a set of actions that we can take
  - Each action results in a new state
  - Typically defined by successor function
    - Given a state, produces all states that can be reached from it
- Cost function that determines the cost of each action (or path = sequence of actions)
- Solution: path from initial state to a goal state
  - Optimal solution: solution with minimal cost

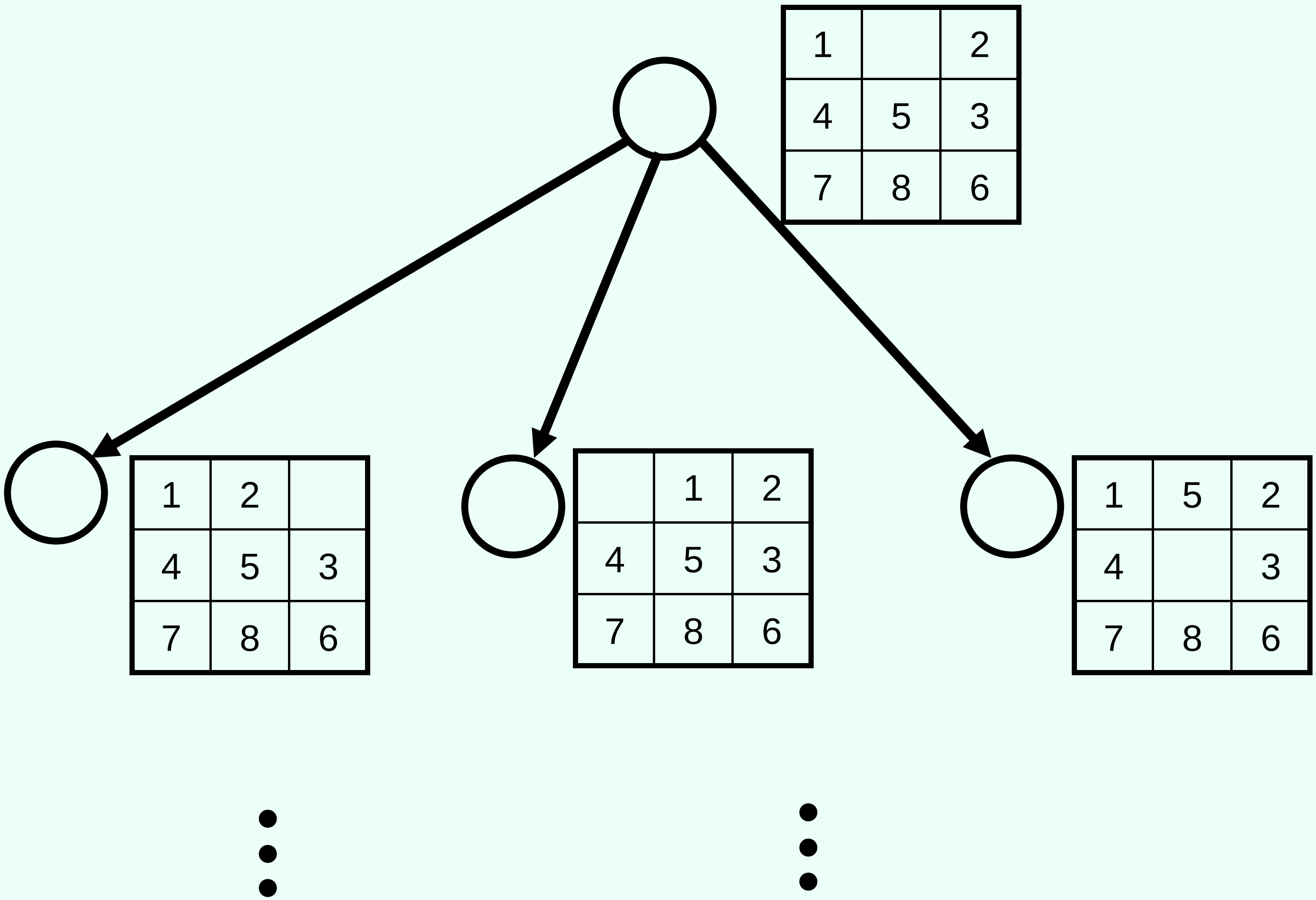
8-puzzle

1		2
4	5	3
7	8	6

1	2	3
4	5	6
7	8	

goal state

8-puzzle



Generic search algorithm

- Frontier = set of nodes generated but not expanded

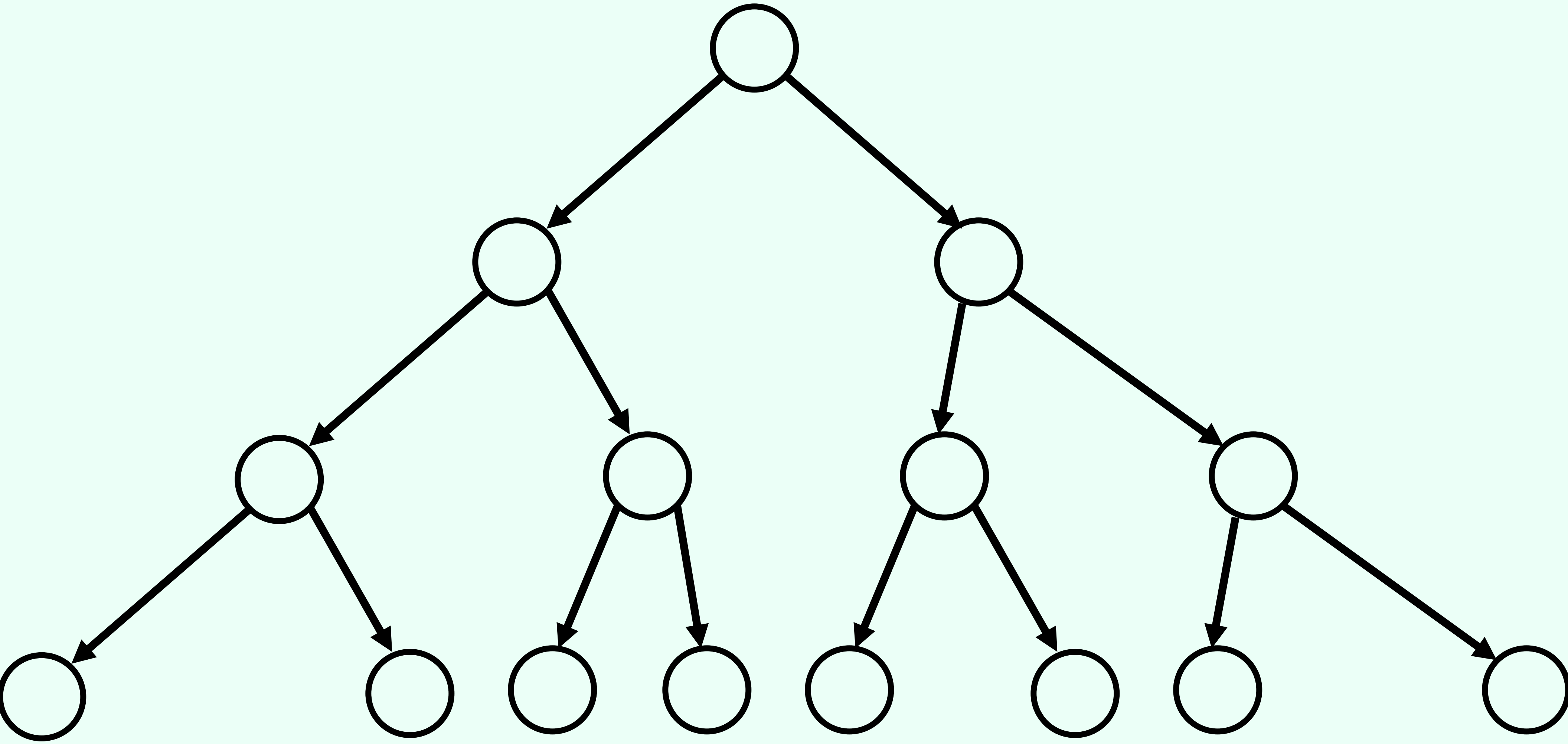
- frontier := {node with initial state}
- loop:
  - if frontier empty, declare failure
  - choose and remove a node v from frontier
  - check if v's state s is a goal state; if so, declare success
  - if not, expand v, insert resulting nodes into frontier

- Key question in search: Which of the generated nodes do we expand next?

Uninformed search

- Given a state, we only know whether it is a goal state or not
- Cannot say one nongoal state looks better than another nongoal state
- Can only traverse state space blindly in hope of somehow hitting a goal state at some point
  - Also called blind search
  - Blind does **not** imply unsystematic!

Breadth-first search



Properties of breadth-first search

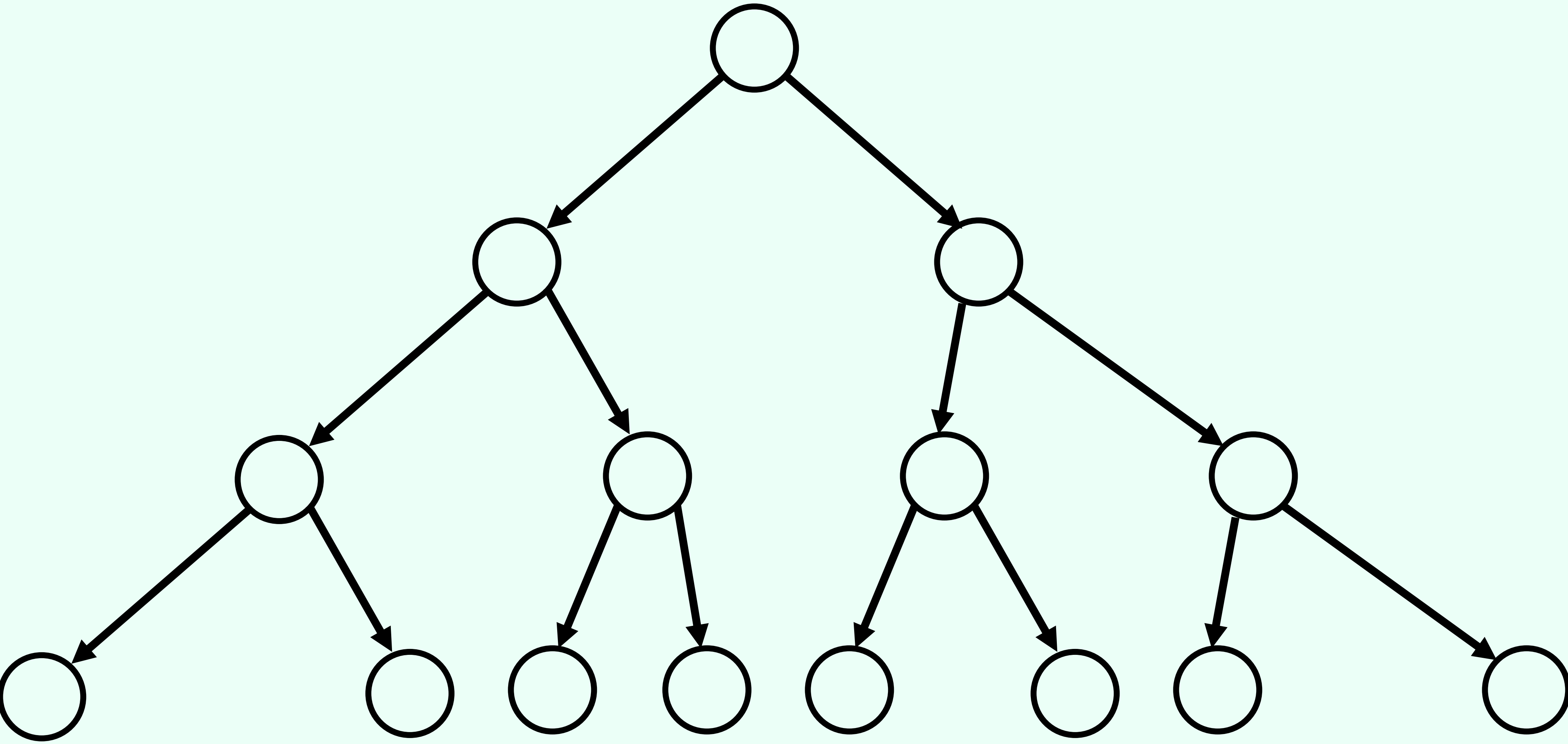
- Nodes are expanded in the same order in which they are generated
  - Frontier can be maintained as a First-In-First-Out (FIFO) queue
- BFS is complete: if a solution exists, one will be found
- BFS finds a shallowest solution
  - Not necessarily an optimal solution
- If every node has b successors (the branching factor), first solution is at depth d, then frontier size will be at least b^d at some point
  - This much space (and time) required ☹

Properties of breadth-first search

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Depth-first search



Implementing depth-first search

- Frontier can be maintained as a Last-In-First-Out (LIFO) queue (aka. a stack)
- Also easy to implement recursively:
- DFS(node)
  - If goal(node) return solution(node);
  - For each successor of node
    - Return DFS(successor) unless it is *failure*;
  - Return *failure;*

Properties of depth-first search

- Not complete (might cycle through nongoal states)
- If solution found, generally not optimal/shallowest
- If every node has b successors (the branching factor), and we search to at most depth m, frontier is at most bm
  - Much better space requirement ☺
  - Actually, generally don't even need to store all of frontier
- Time: still need to look at every node
  - $b^m + b^{m-1} + \ldots + 1$ (for $b>1$, $O(b^m)$)
  - Inevitable for uninformed search methods…

Combining good properties of BFS and DFS

- Limited depth DFS: just like DFS, except never go deeper than some depth d
- Iterative deepening DFS:
  - Call limited depth DFS with depth 0;
  - If unsuccessful, call with depth 1;
  - If unsuccessful, call with depth 2;
  - Etc.
- Complete, finds shallowest solution
- Space requirements of DFS
- May seem wasteful timewise because replicating effort
  - Really not that wasteful because **almost all effort at deepest level**
  - $db + (d-1)b^2 + (d-2)b^3 + ... + 1b^d$ is $O(b^d)$ for $b > 1$

Let's start thinking about cost

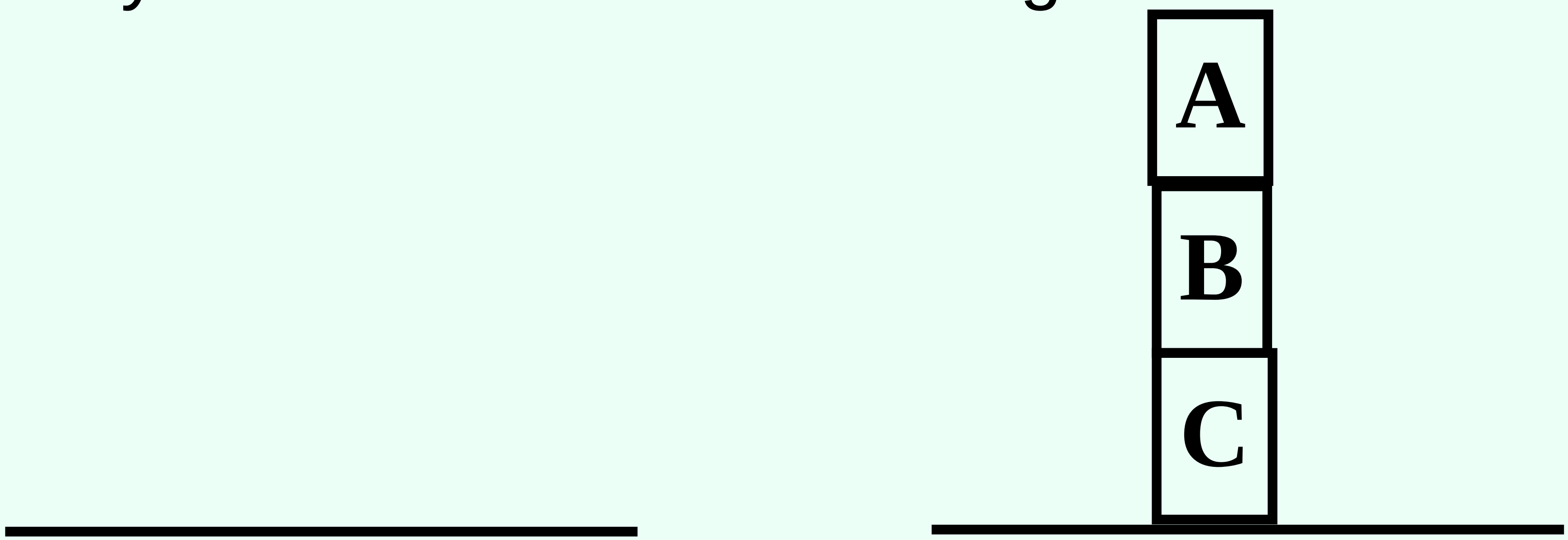
- BFS finds shallowest solution because always works on shallowest nodes first
- Similar idea: always work on the lowest-cost node first (uniform-cost search)
- Will find optimal solution (assuming costs increase by at least constant amount along path)
- Will often pursue lots of short steps first
- If optimal cost is C, and cost increases by at least L each step, we can go to depth C/L
- Similar memory problems as BFS

Searching backwards from the goal

- Sometimes can search backwards from the goal
  - Maze puzzles
  - Eights puzzle
  - Reaching location F
  - What about the goal of “having visited all locations”?
- Need to be able to compute predecessors instead of successors
- What’s the point?

Predecessor branching factor can be smaller than successor branching factor

- Stacking blocks:
  - only action is to add something to the stack



In hand: A, B, C

Start state

In hand: nothing

Goal state

We'll see more of this…

Bidirectional search

- Even better: search from both the start and the goal, in parallel!

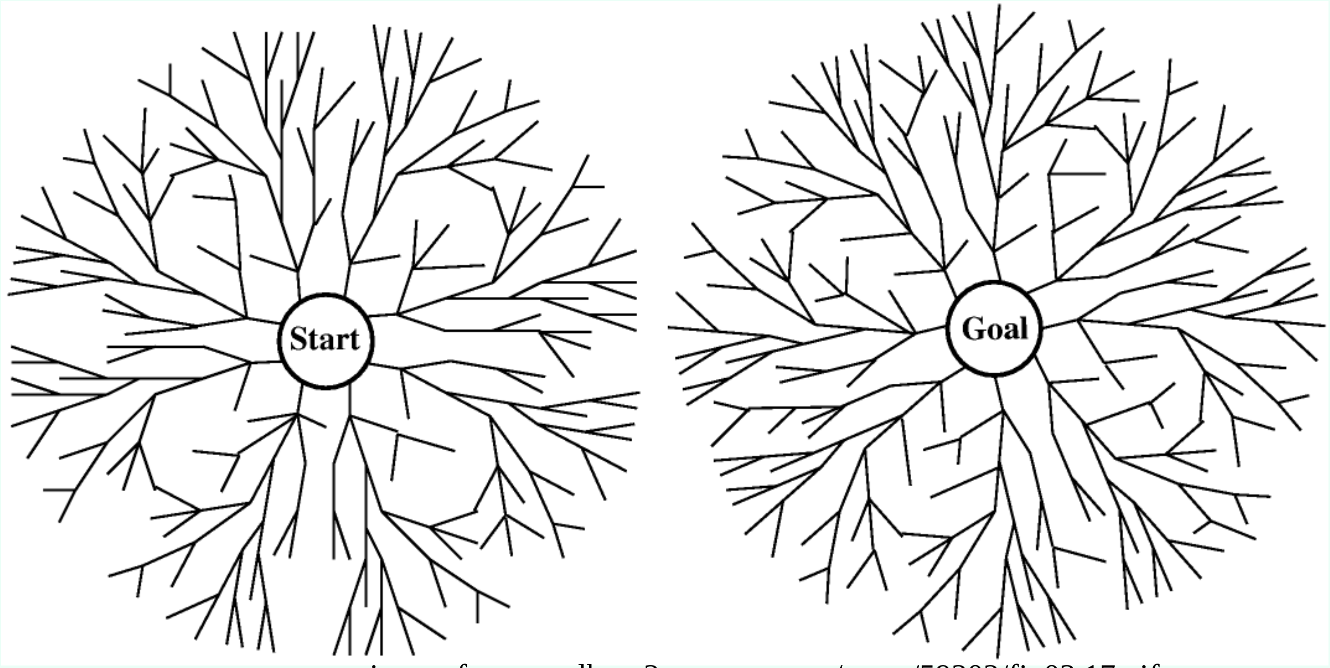


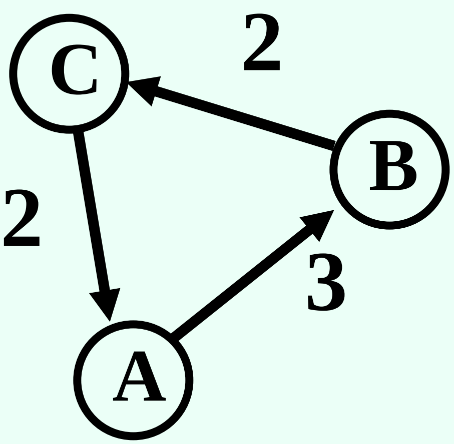
image from cs-alb-pc3.massey.ac.nz/notes/59302/fig03.17.gif

- If the shallowest solution has depth d and branching factor is b on both sides, requires only $O(b^{d/2})$ nodes to be explored!

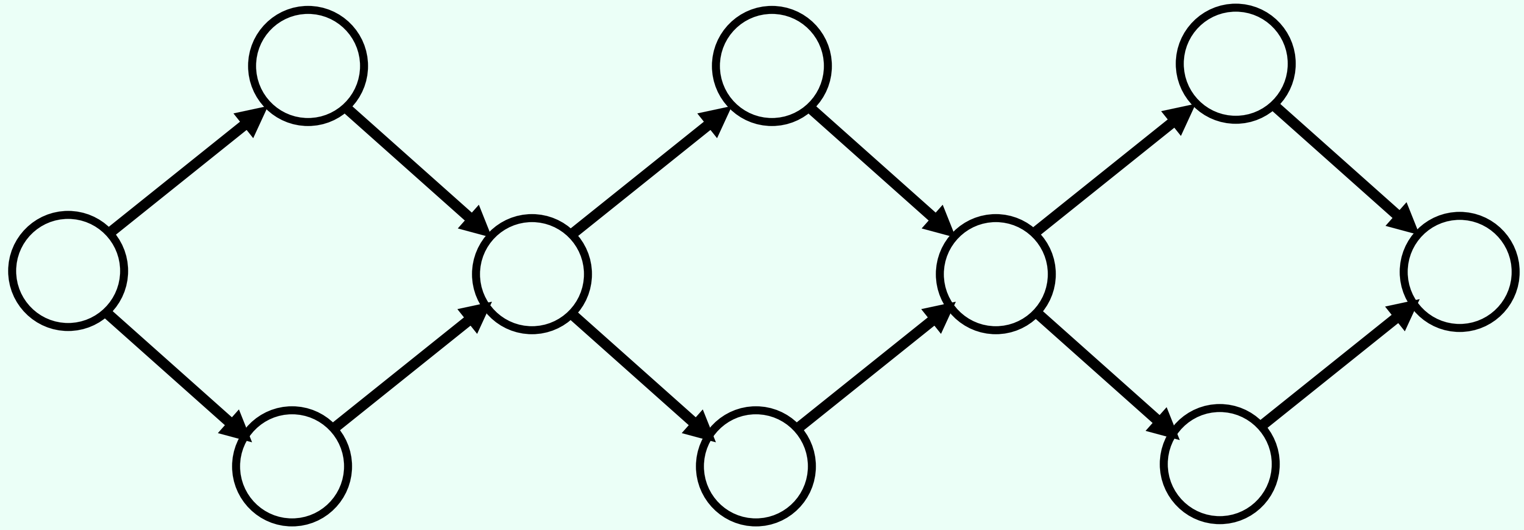
Making bidirectional search work

- Need to be able to figure out whether the frontier intersect
  - Need to keep at least one frontier in memory…
- Other than that, can do various kinds of search on either tree, and get the corresponding optimality etc. guarantees
- Not possible (feasible) if backwards search not possible (feasible)
  - Hard to compute predecessors
  - High predecessor branching factor
  - Too many goal states

Repeated states



cycles



exponentially large search trees (try it!)

- Repeated states can cause incompleteness or enormous runtimes
- Can maintain list of previously visited states to avoid this
  - If new path to the same state has greater cost, don’t pursue it further
  - Leads to time/space tradeoff
- “Algorithms that forget their history are doomed to repeat it” [Russell and Norvig]