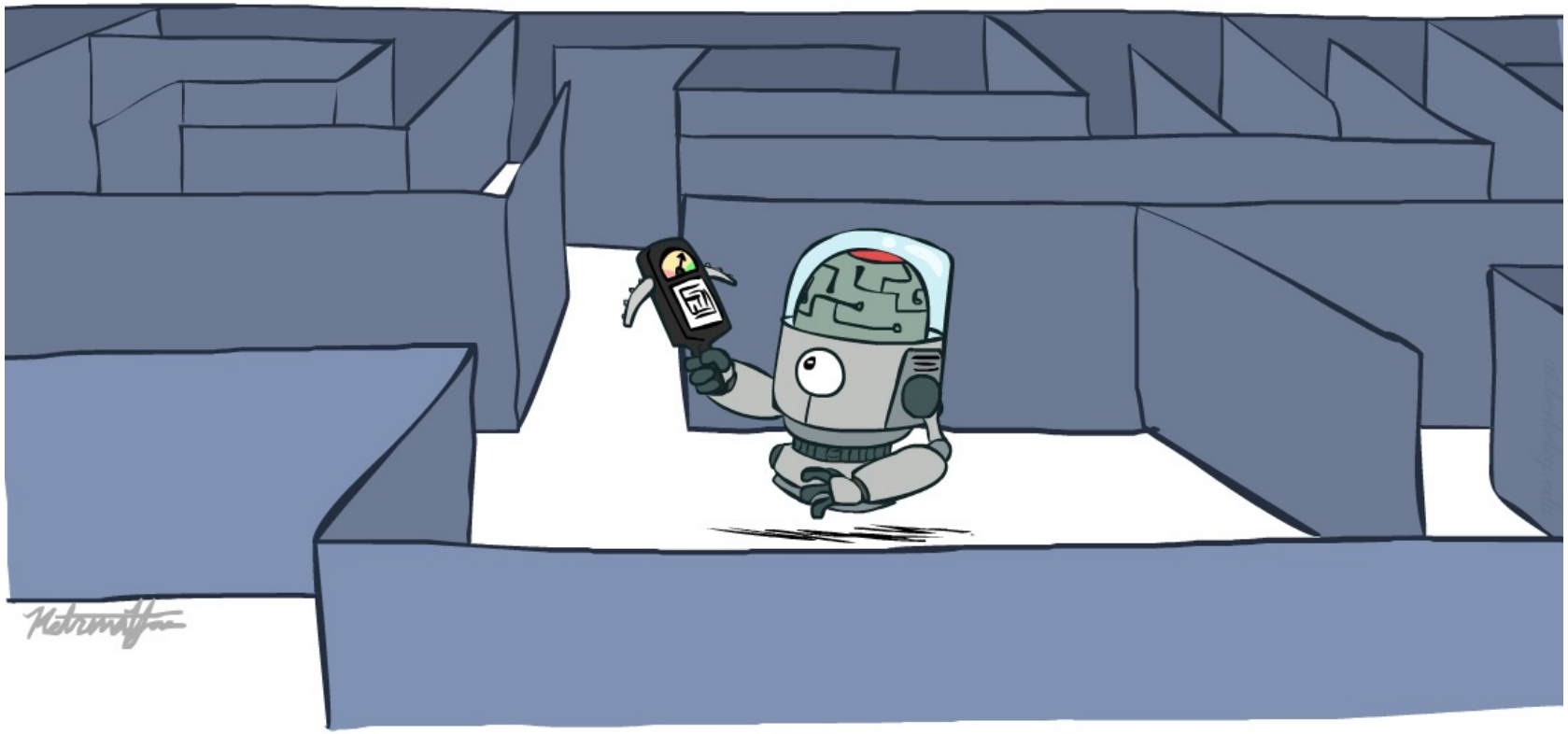
Solving problems by searching: Uninformed Search

CE417: Introduction to Artificial Intelligence
Sharif University of Technology
Fall 2023

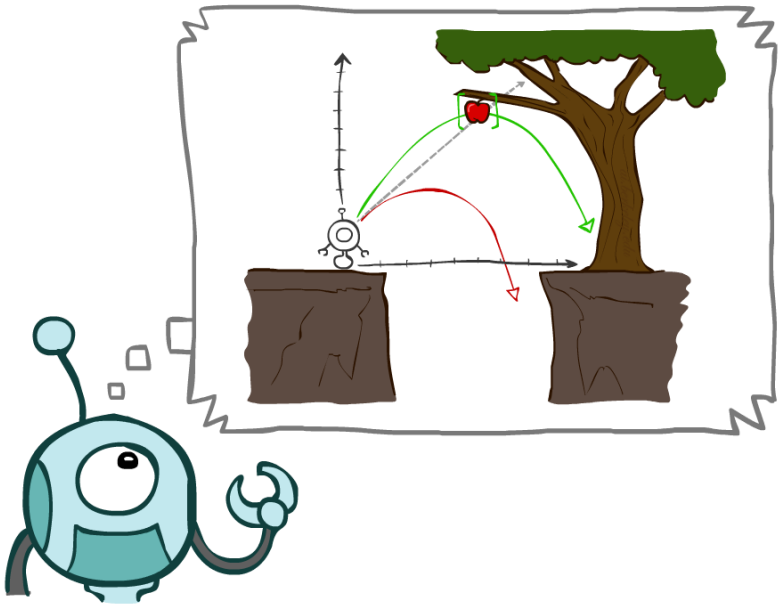
Soleymani

Most slides have been adopted from Klein and Abdeel, CS188, UC Berkeley.

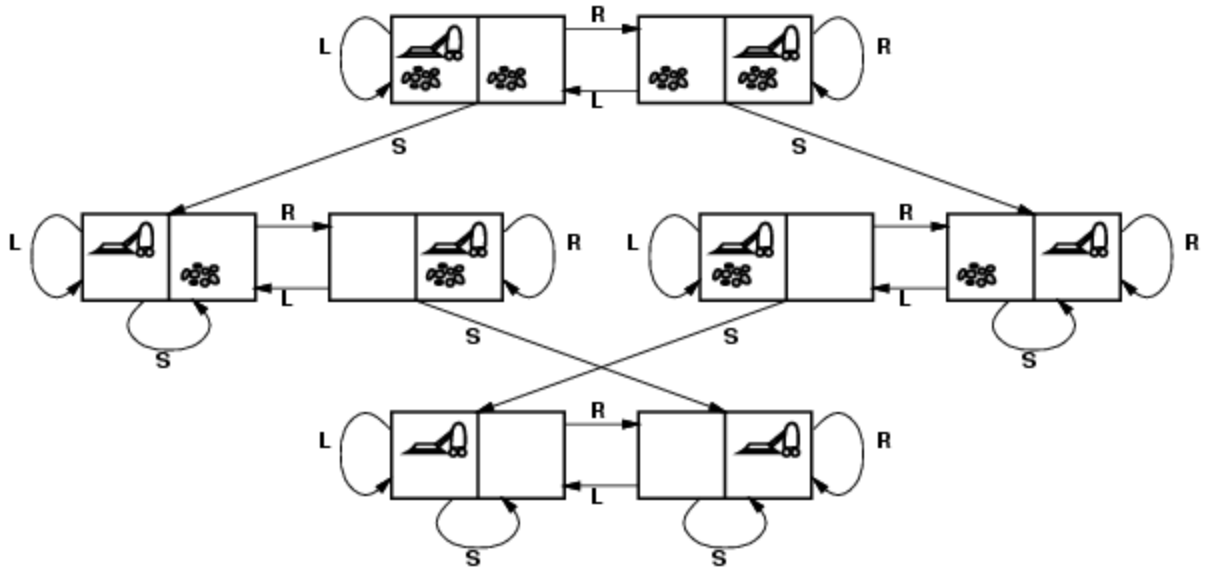


Outline

- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search



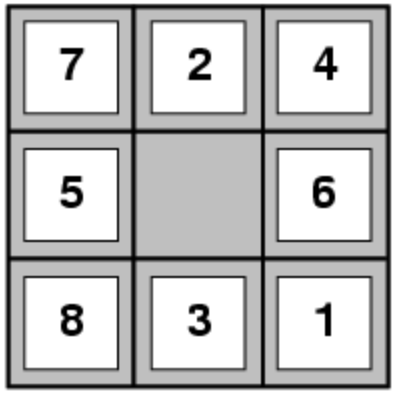
Vacuum world state space graph



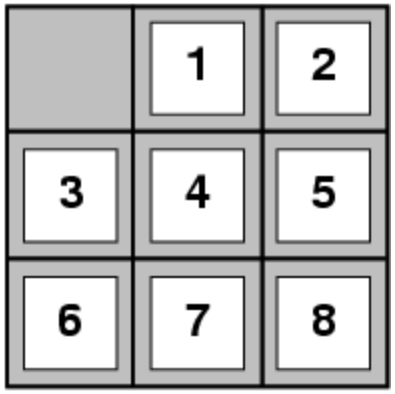
$2 \times 2^2 = 8$
States

- States? dirt locations & robot location
- Actions? Left, Right, Suck
- Goal test? no dirt at all locations
- Path cost? one per action

Example: 8-puzzle



Start State



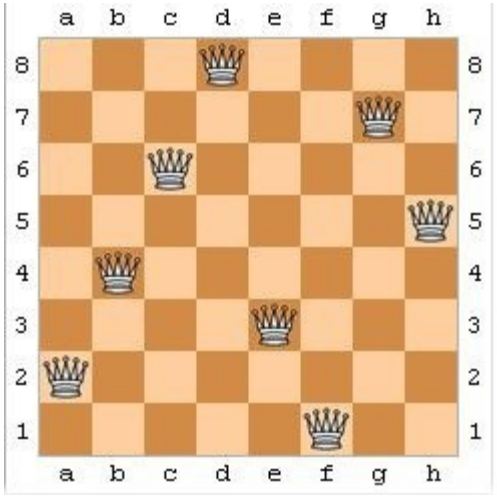
Goal State

$9!/2 = 181{,}440$ States

- States? locations of eight tiles and blank in 9 squares
- Actions? move blank left, right, up, down (within the board)
- Goal test? e.g., the above goal state
- Path cost? one per move

Note: the family of sliding-block puzzles is known to be NP-complete and optimal solution of *n*-Puzzle family is NP-hard.

Example: 8-queens problem

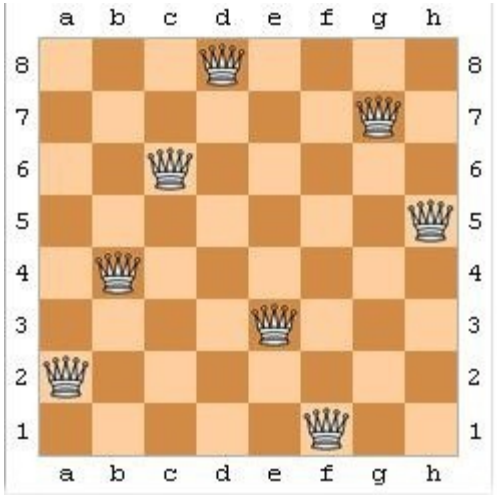


$64 \times 63 \times \cdots \times 57 \simeq 1.8 \times 10^{14}$ States

- Initial State? no queens on the board
- States? any arrangement of 0-8 queens on the board is a state
- Actions? add a queen to the state (any empty square)
- Goal test? 8 queens are on the board, none attacked
- Path cost? of no interest

  search cost vs. solution path cost

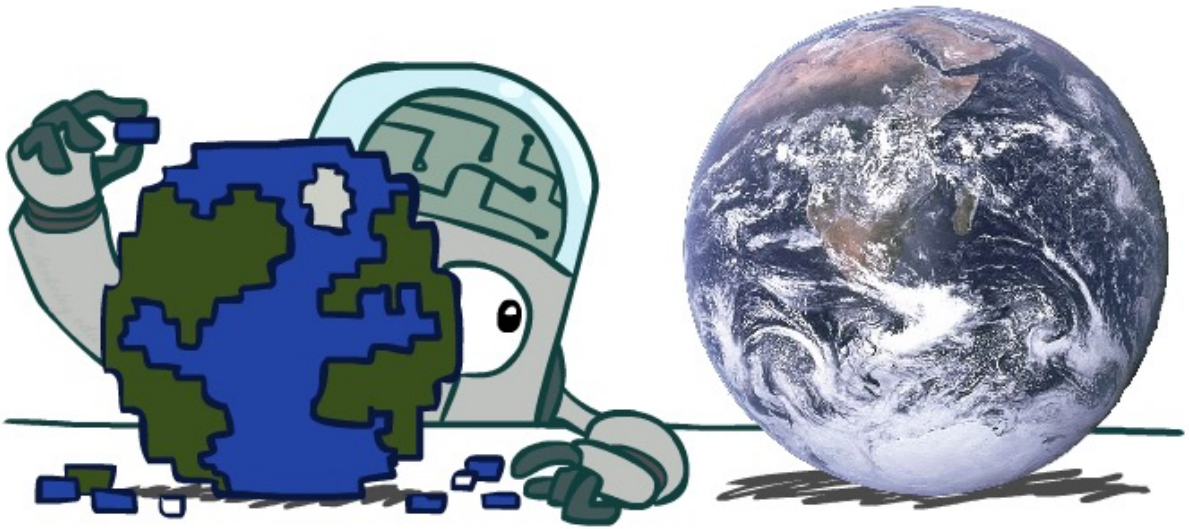
Example: 8-queens problem
(other formulation)



2,057 States

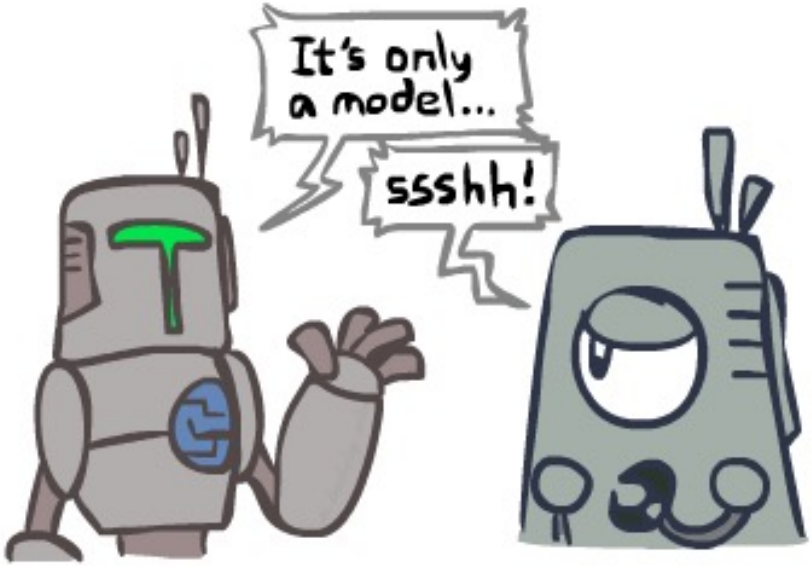
- Initial state? no queens on the board
- States? any arrangement of *k* queens one per column in the leftmost *k* columns with no queen attacking another
- Actions? add a queen to any square in the leftmost empty column such that it is not attacked by any other queen
- Goal test? 8 queens are on the board
- Path cost? of no interest

Search problems are models



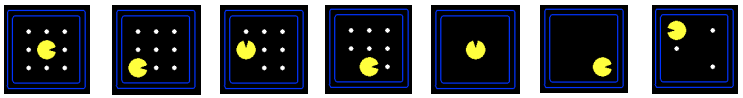
Search and models

- Search operates over models of the world
  - The agent doesn't actually try all the plans out in the real world!
  - Planning is all "in simulation"
  - Your search is only as good as your models…

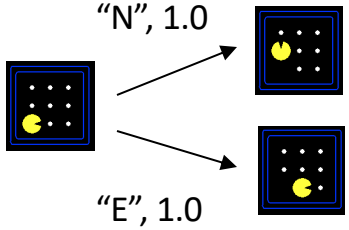


Search problems

- A search problem consists of:
  - A state space



  - A successor function
    (with actions, costs)

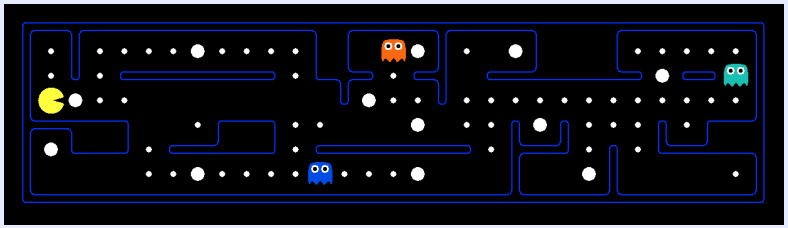


  - A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

What's in a state space?

The world state includes every last detail of the environment



A search state keeps only the details needed for planning (abstraction)

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Successor: update location only
  - Goal test: is (x,y)=END

- Problem: Eat-All-Dots
  - States: {(x,y), dot booleans}
  - Actions: NSEW
  - Successor: update location and possibly a dot boolean
  - Goal test: dots all false

State space sizes?

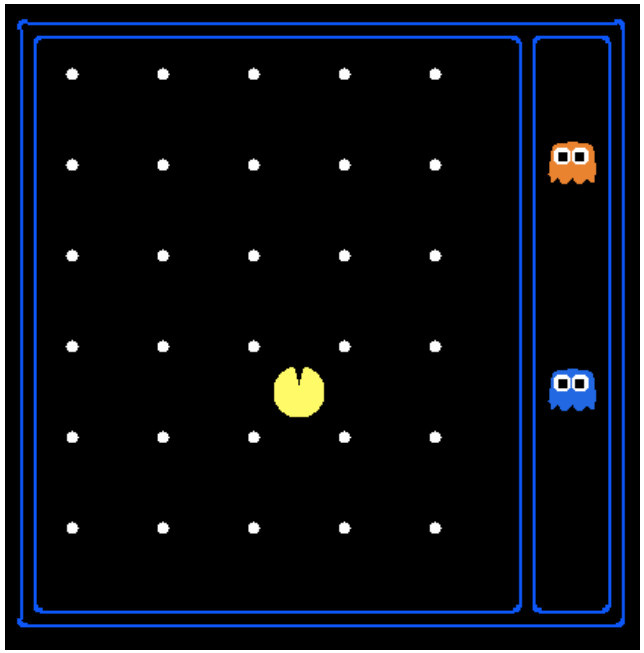
- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- How many
  - World states?

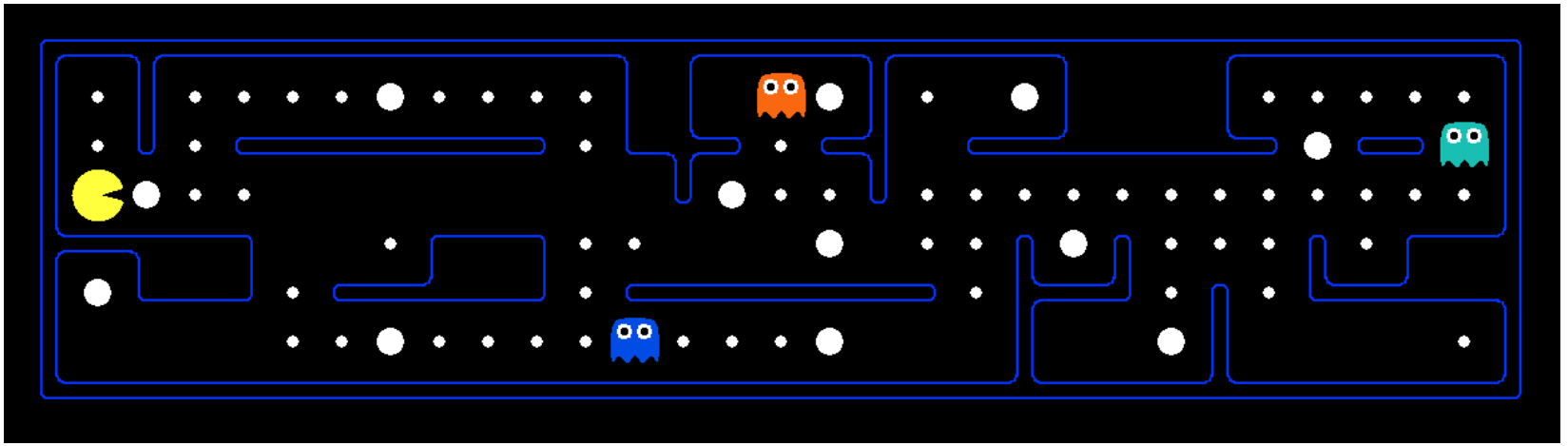
    $120x(2^{30})x(12^2)x4$
  - States for pathing?

    120
  - States for eat-all-dots?

    $120x(2^{30})$



Quiz: Safe passage



- Problem: eat all dots while keeping the ghosts perma-scared
- What does the state space have to specify?
  - (agent position, dot booleans, power pellet booleans, remaining scared time)

State space

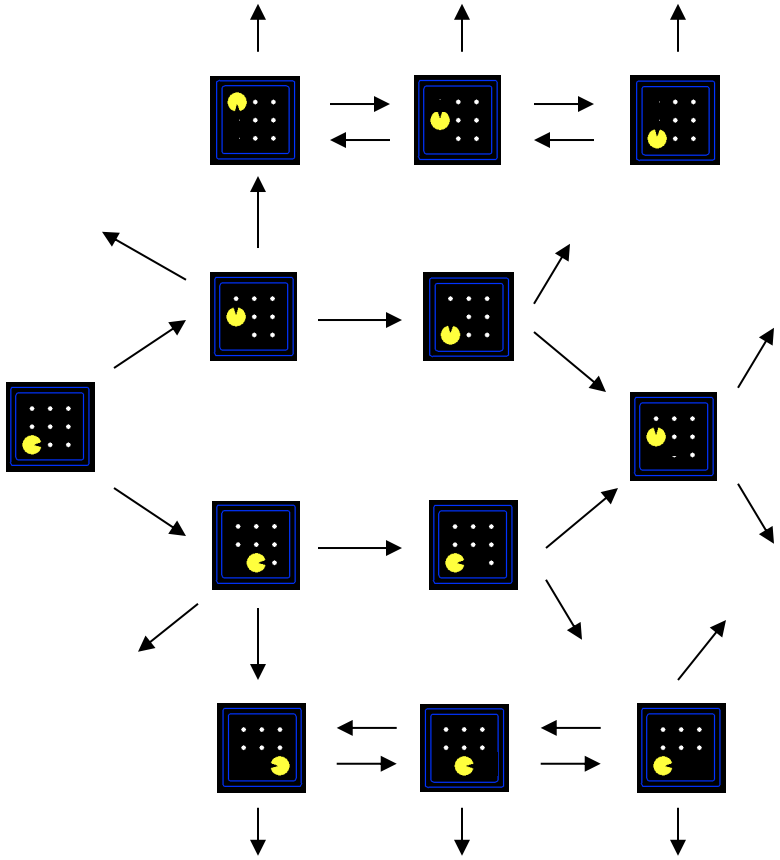
- State space: set of all reachable states from initial state
  - Initial state, actions, and transition model together define it

- It forms a directed graph
  - Nodes: states
  - Links: actions

- Constructing this graph on demand

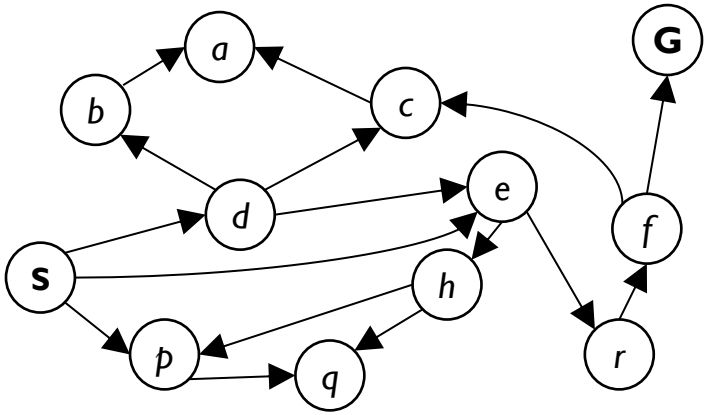
State space graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



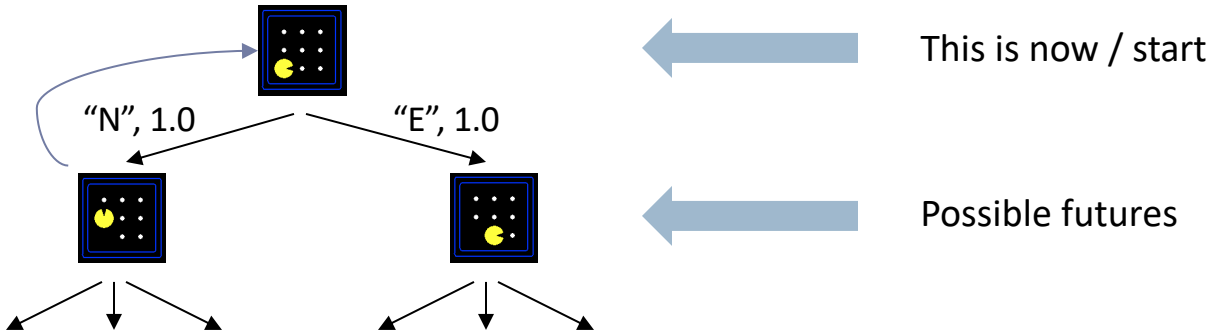
State space graphs

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a search graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



Tiny state space for a tiny search problem

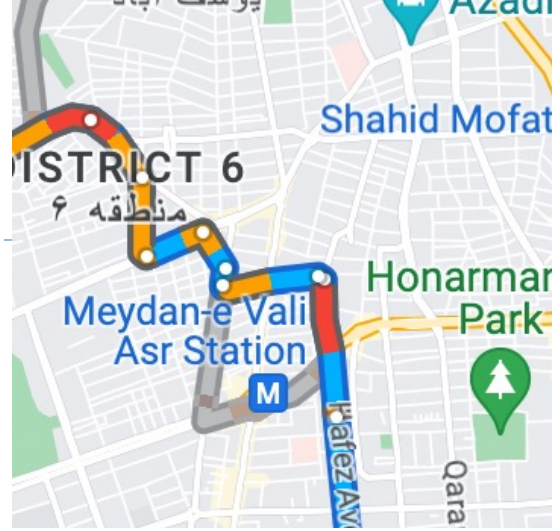
Search trees

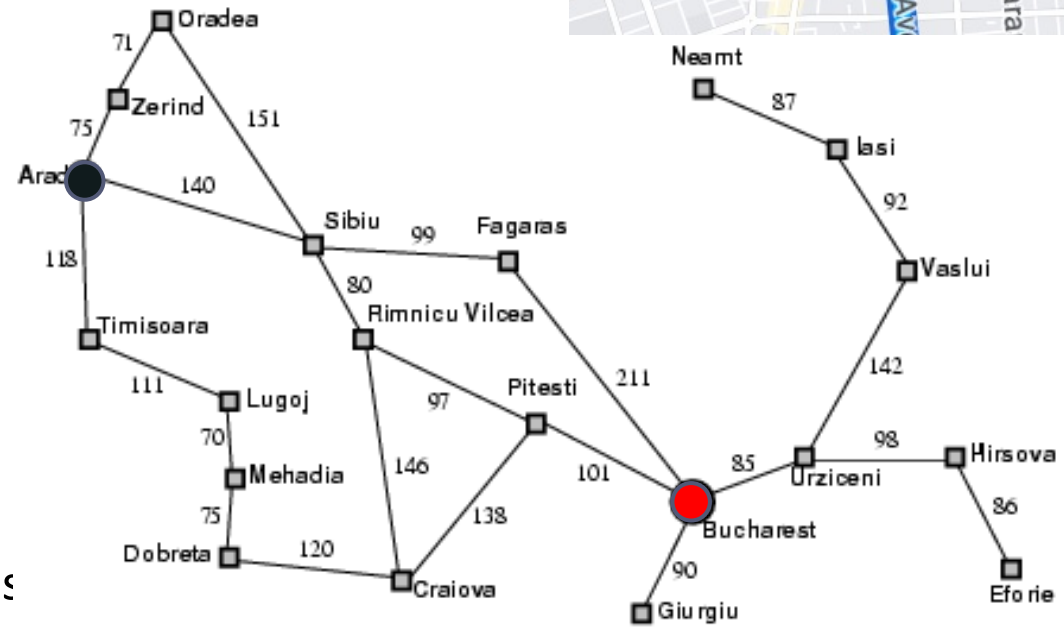


- A search tree:
  - A "what if" tree of plans and their outcomes
  - The start state is the root node
  - Children correspond to successors
  - Nodes show states, but correspond to PLANS that achieve those states
    - Nodes contain **problem state,** parent, path length, a depth, and a cost
  - For most problems, we can never actually build the whole tree

Routing example

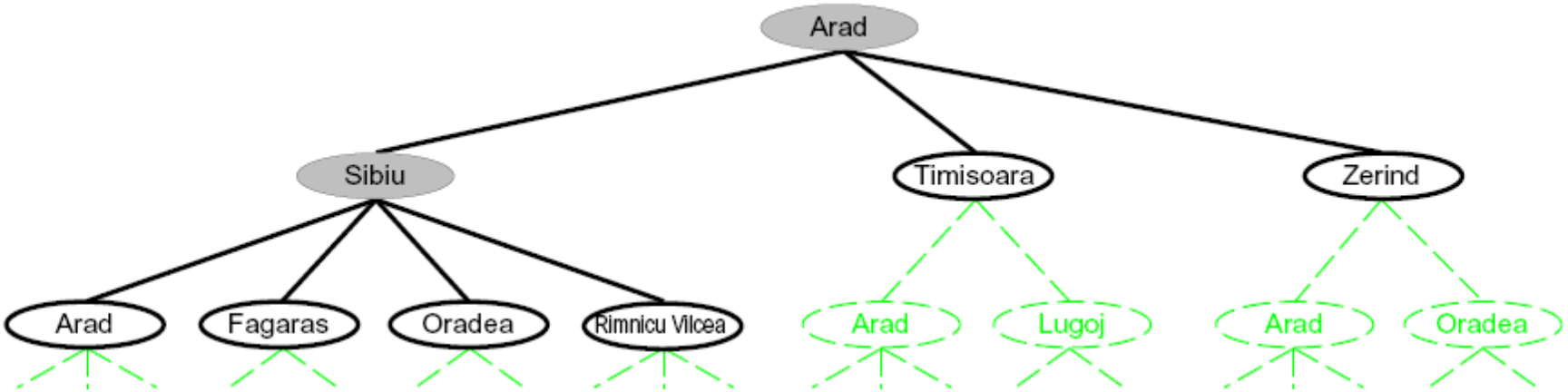
- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
- Initial state
  - currently in Arad
- Formulate goal
  - be in Bucharest
- Formulate problem
  - states: various cities
  - actions: drive between cities
- Solution
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest





Map of Romania

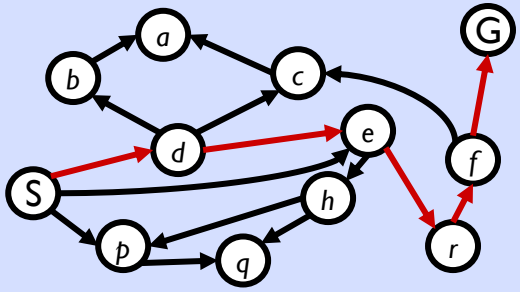
Searching with a search tree



- Search:
  - Expand out potential plans (tree nodes)
  - Maintain a frontier of partial plans under consideration
  - Try to expand as few tree nodes as possible

State space graphs vs. search trees

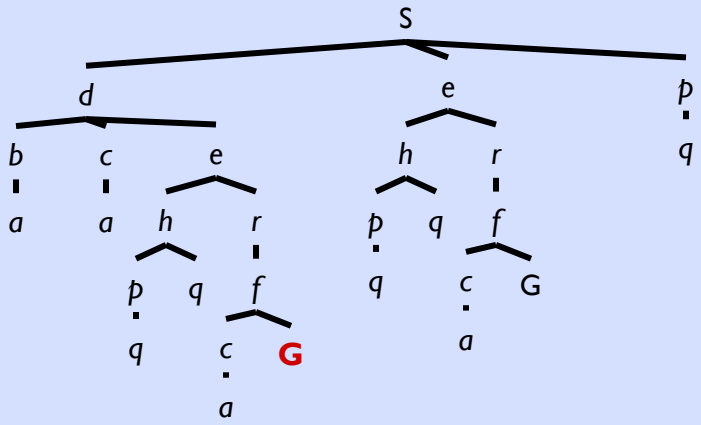
State Space Graph



Each NODE in the search tree is an entire PATH in the state space graph.

We construct both on demand – and we construct as little as possible.

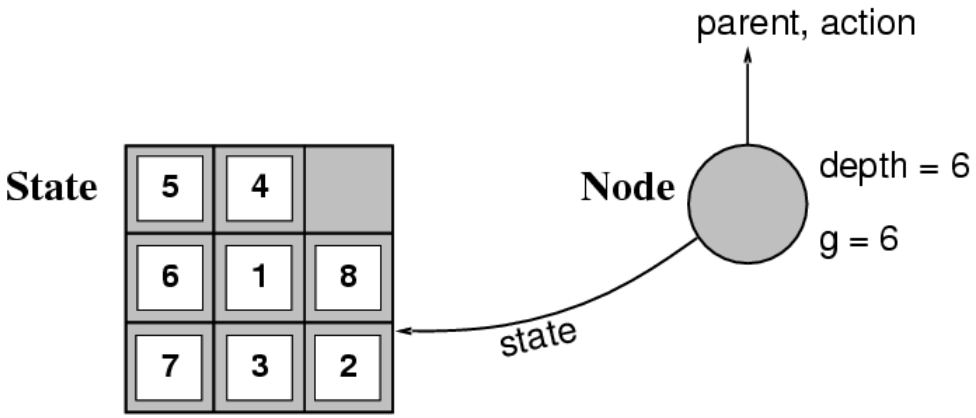
Search Tree



Implementation: States vs. nodes

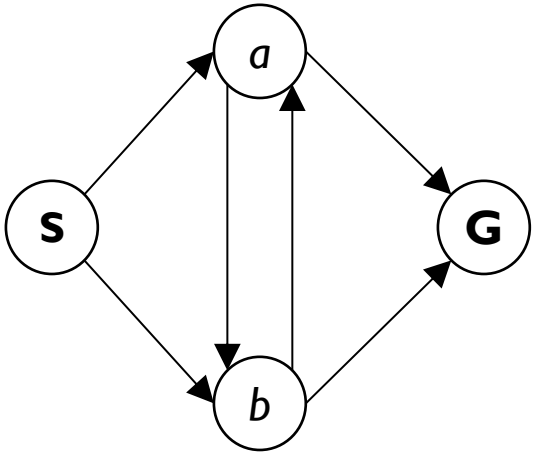
- A state is a representation of a physical configuration

- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost *g(x)*, depth

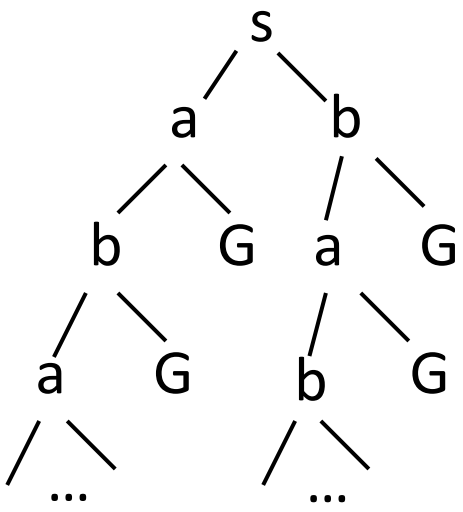


Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:



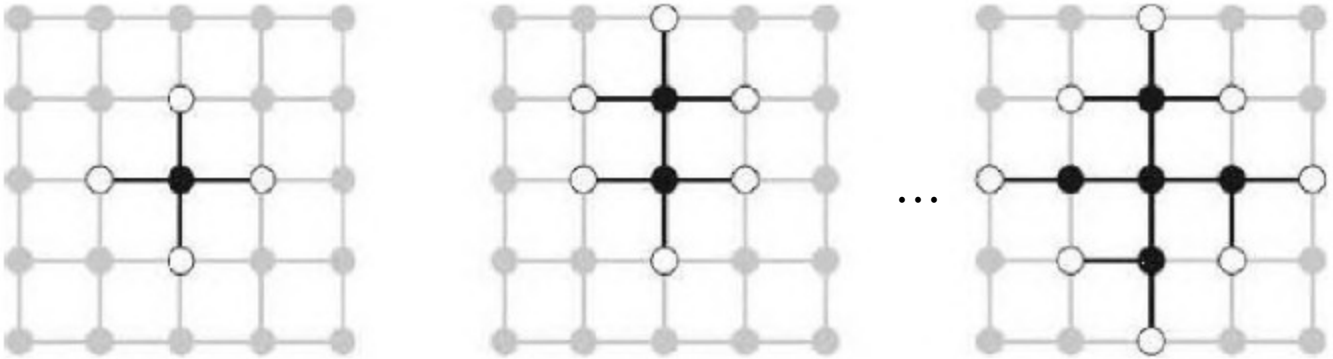
How big is its search tree (from s)?



- Important: Lots of repeated structure in the search tree!

Graph Search

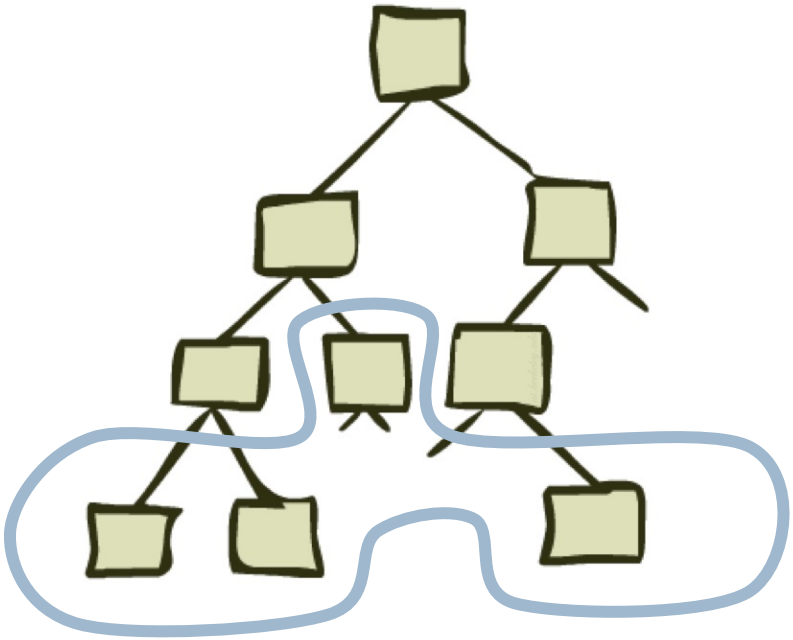
- Example: rectangular grid



explored

frontier

Tree Search



Tree search algorithm

- Basic idea
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Frontier: all leaf nodes available for expansion at any given point

Different data structures (e.g, FIFO, LIFO) for **frontier** can cause different orders of node expansion and thus produce different search algorithms.

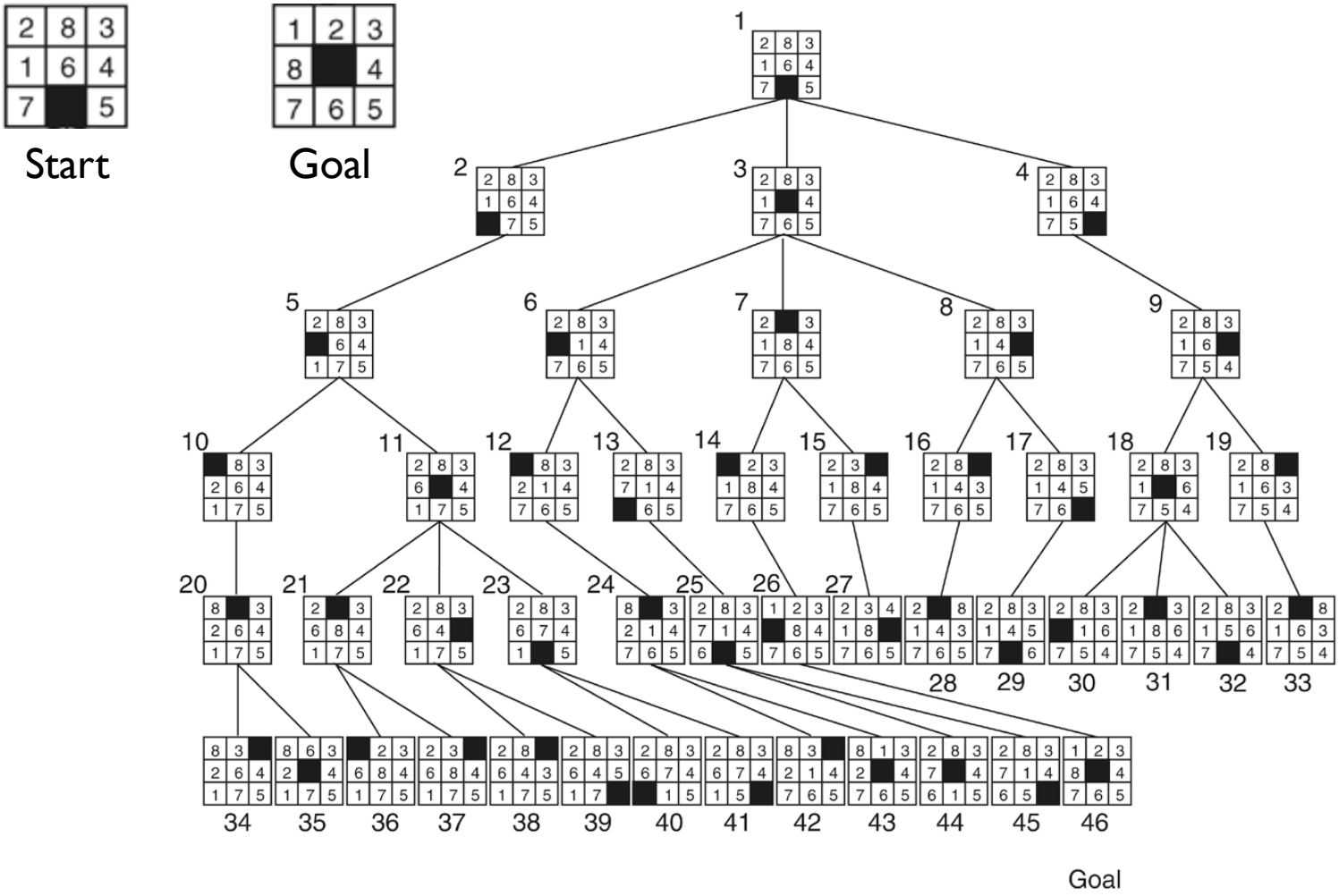
Graph search

- Redundant paths in tree search: more than one way to get from one state to another
  - may be due to a bad problem definition or the essence of the problem
  - can cause a tractable problem to become intractable

function GRAPH-SEARCH(*problem)* **returns** a solution, or failure
    initialize the frontier using the initial state of *problem*
    loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set

explored set: remembered every explored node

Search for 8-puzzle Problem



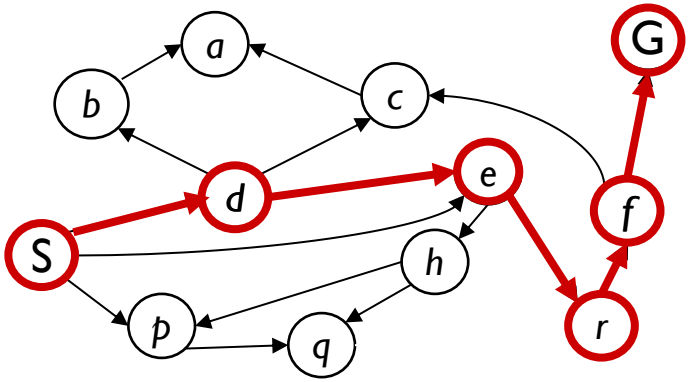
source: iis.kaist.ac.kr/es

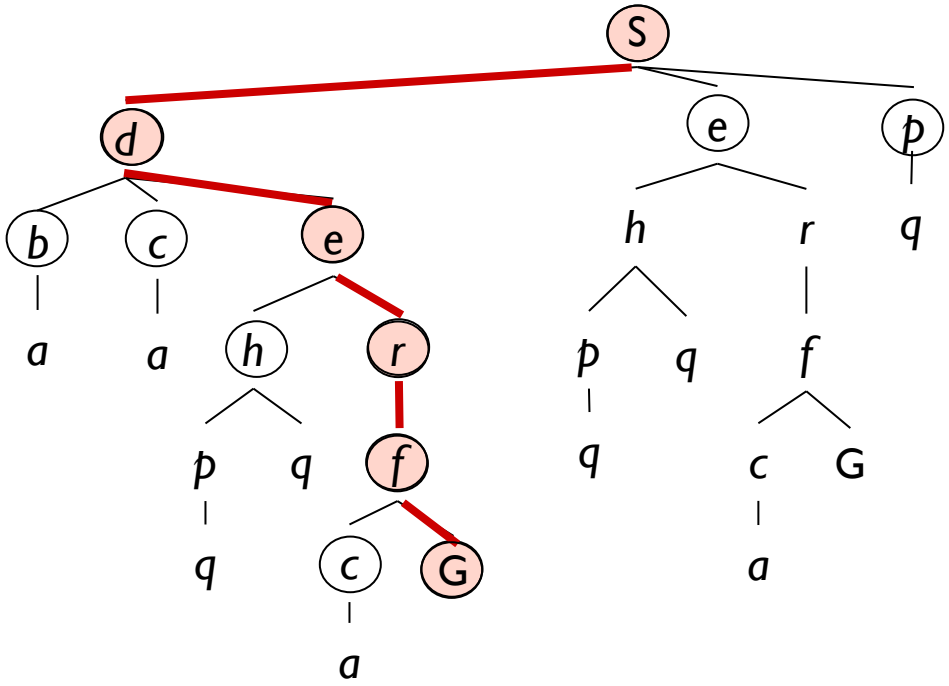
General tree search

- Important ideas:
  - Frontier
  - Expansion
  - Exploration strategy

- Main question: which frontier nodes to explore?

Example: Tree search





~~s~~

~~s → d~~

s → e

s → p

s → d → b

s → d → c

~~s → d → e~~

s → d → e → h

~~s → d → e → r~~

~~s → d → e → r → f~~

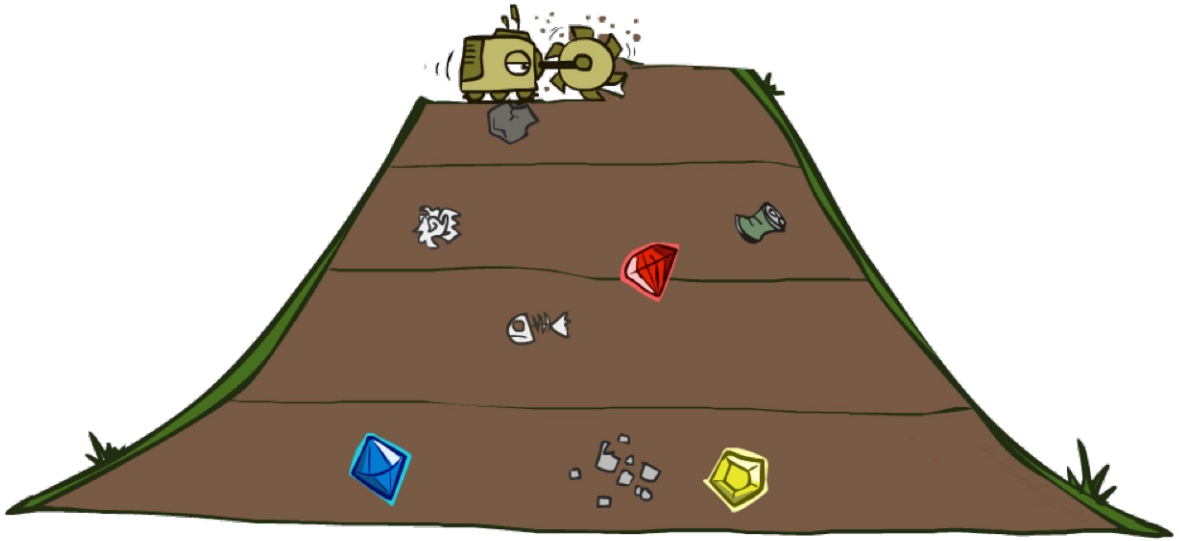
s → d → e → r → f → c

~~s → d → e → r → f → G~~

Uninformed (blind) search strategies

- No additional information beyond the problem definition
  - Breadth-First Search (BFS)
  - Uniform-Cost Search (UCS)
  - Depth-First Search (DFS)
  - Depth-Limited Search (DLS)
  - Iterative Deepening Search (IDS)

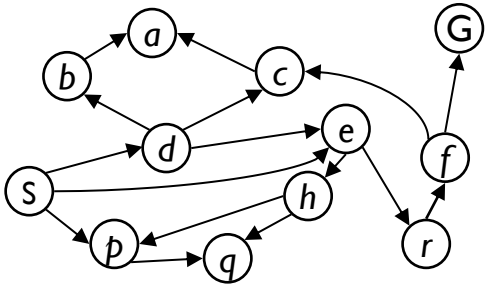
Breadth-First Search

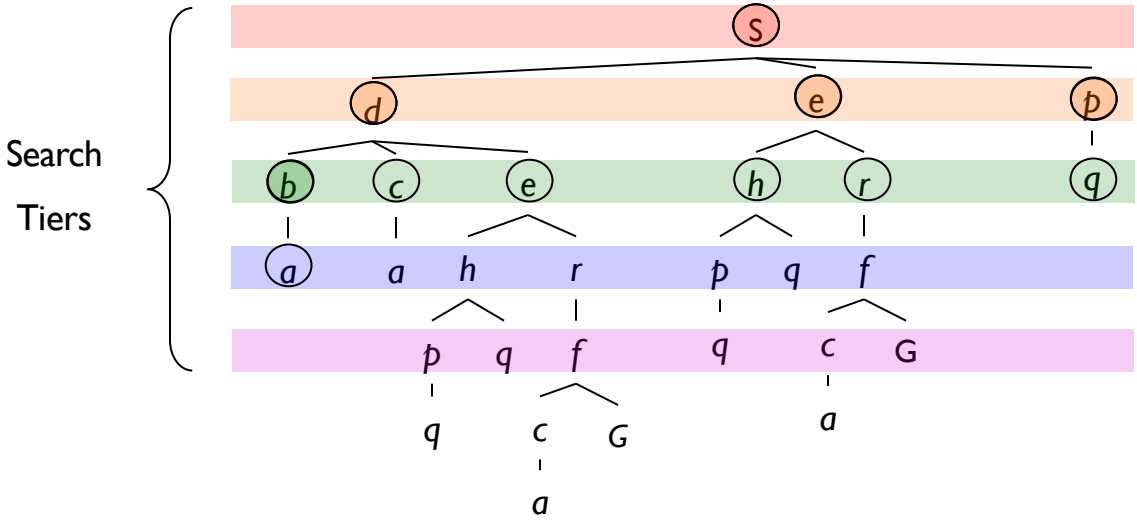


Breadth-First Search

Strategy: expand a shallowest node first

Implementation: frontier is a FIFO queue

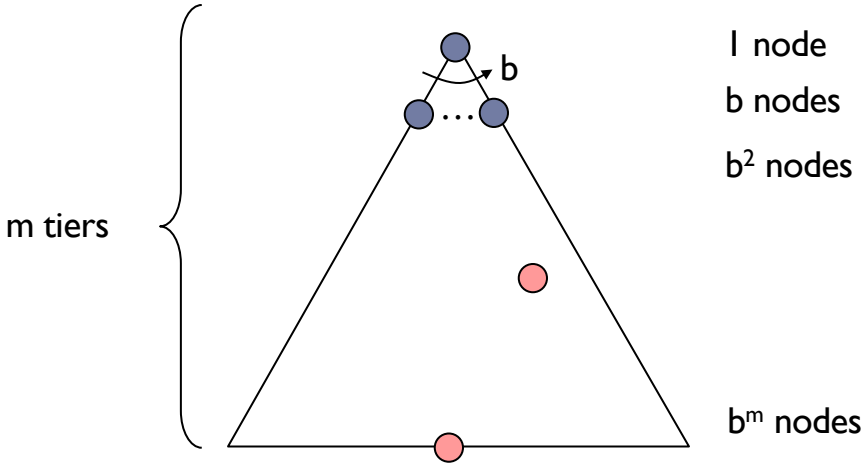




Search algorithm properties

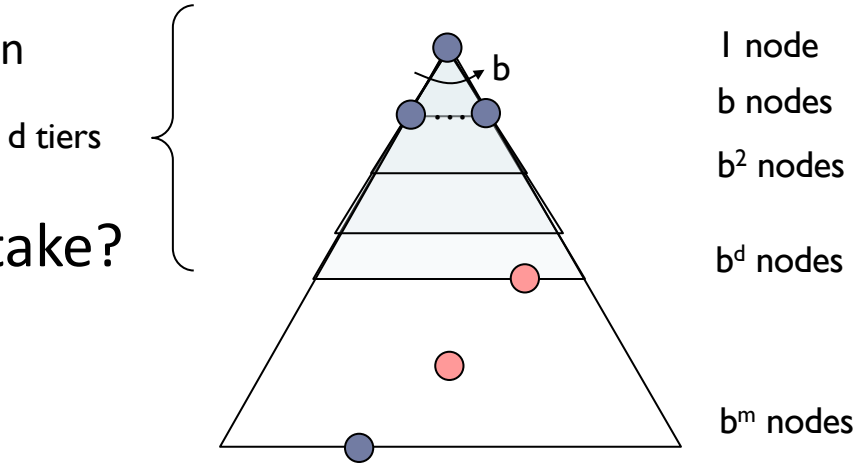
- Complete: Guaranteed to find a solution if one exists?
- Optimal: Guaranteed to find the least cost path?
- Time complexity?
- Space complexity?

- Cartoon of search tree:
  - b is the branching factor
  - m is the maximum depth
  - d is the depth of the shallowest goal
  - solutions at various depths



Breadth-First Search (BFS) properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be d
- How much space does the frontier take?
- Is it complete?
- Is it optimal?



Properties of breadth-first search

- Complete?
  - Yes (for finite b and d)
- Time
  - $b + b^2 + b^3 + \cdots + b^d = O(b^d)$ total number of generated nodes
    - goal test has been applied to each node when it is generated
- Space
  - explored frontier
  - $O(b^{d-1}) + O(b^d) = O(b^d)$
    - Tree search does not save much space while may cause a great time excess
- Optimal?
  - Yes, if path cost is a non-decreasing function of *d*
    - *e.g.* all actions having the same cost

Properties of breadth-first search

- Space complexity is a bigger problem than time complexity
- Time is also prohibitive
  - Exponential-complexity search problems cannot be solved by uninformed methods (only the smallest instances)

1 million node/sec, 1kb/node $b = 10$

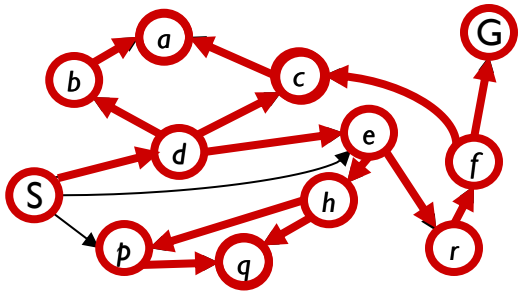
d	Time	Memory
6	1.1 secs	1 gigabytes
8	2 minutes	103 gigabytes
10	3 hours	10 terabytes
12	13 days	1 pentabyte
14	3.5 years	99 pentabytes
16	350 years	10 exabytes

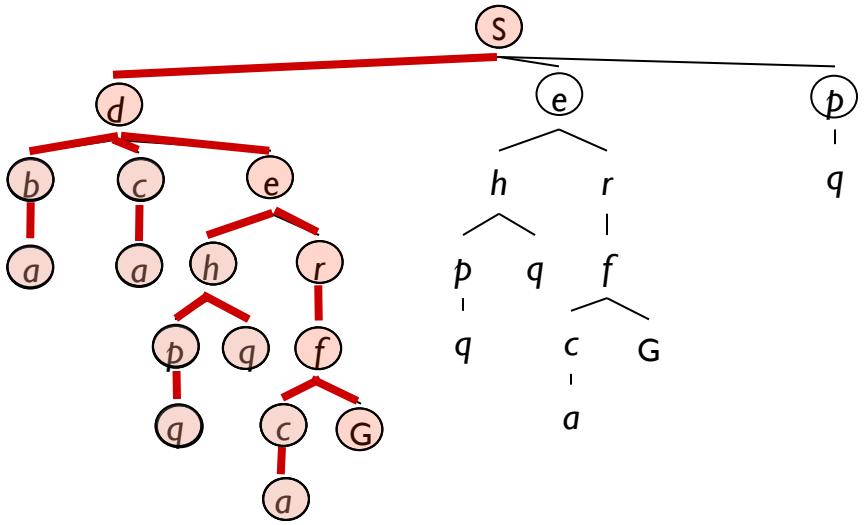
Depth-First Search

Depth-First Search

Strategy: expand a deepest node first

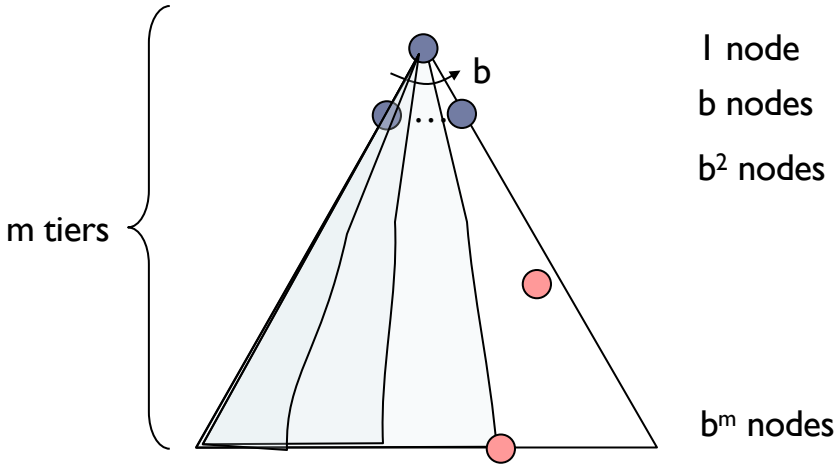
Implementation: frontier is a LIFO stack





Depth-First Search (DFS) Properties

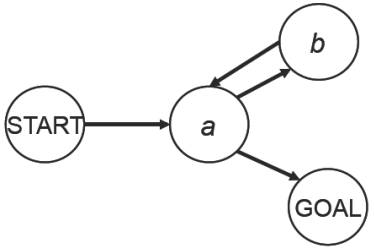
- What nodes DFS expand?
  - Some left prefix of the tree.
  - Could process the whole tree!
  - If m is finite, can take time O(b^m)



- How much space does the frontier take?
  - Only has siblings on path to root, so O(bm)
- Is it complete?
  - m could be infinite, so only if we prevent cycles (more later)
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost

Properties of DFS

- Complete?
  - Not complete (repeated states & redundant paths)



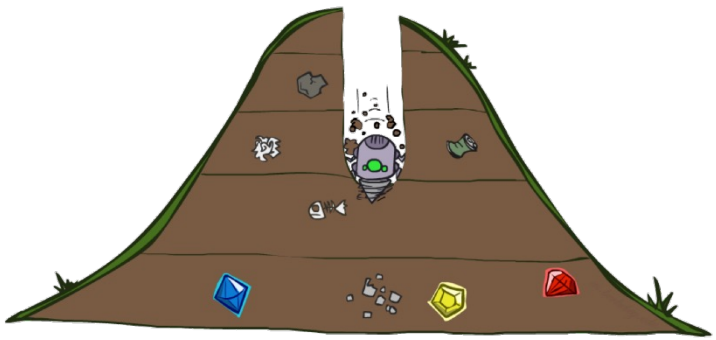
- Time
  - $O(b^m)$: terrible if m is much larger than d
    - In tree-version, m can be much larger than the size of the state space
- Space
  - $O(bm)$, i.e., linear space complexity for tree search
    - So depth first tree search as the base of many AI areas
  - Recursive version called backtracking search can be implemented in $O(m)$ space
- Optimal?
  - No

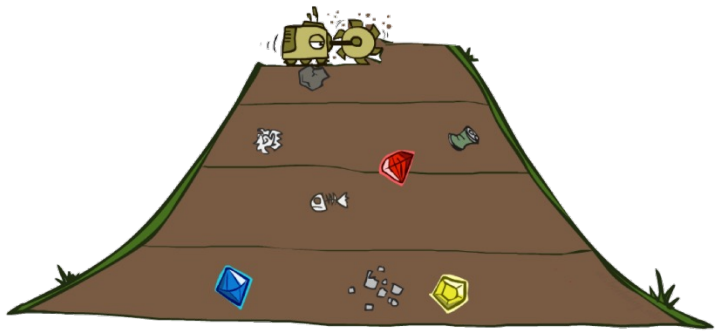
DFS: tree-search version

Video of Demo Maze Water DFS/BFS (part 1)

Video of Demo Maze Water DFS/BFS (part 2)

Quiz: DFS vs. BFS





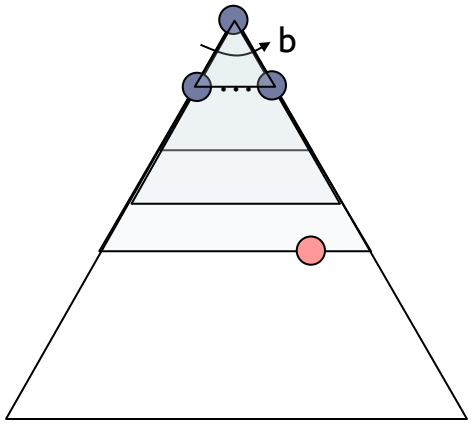
Quiz: DFS vs. BFS

- When will BFS outperform DFS?

- When will DFS outperform BFS?

Iterative Deepening Search

- Idea: get DFS's space advantage with BFS's time / shallow-solution advantages
  - Run a DFS with depth limit 1. If no solution…
  - Run a DFS with depth limit 2. If no solution…
  - Run a DFS with depth limit 3. …..
- Isn't that wastefully redundant?
  - Generally most work happens in the lowest level searched, so not so bad!

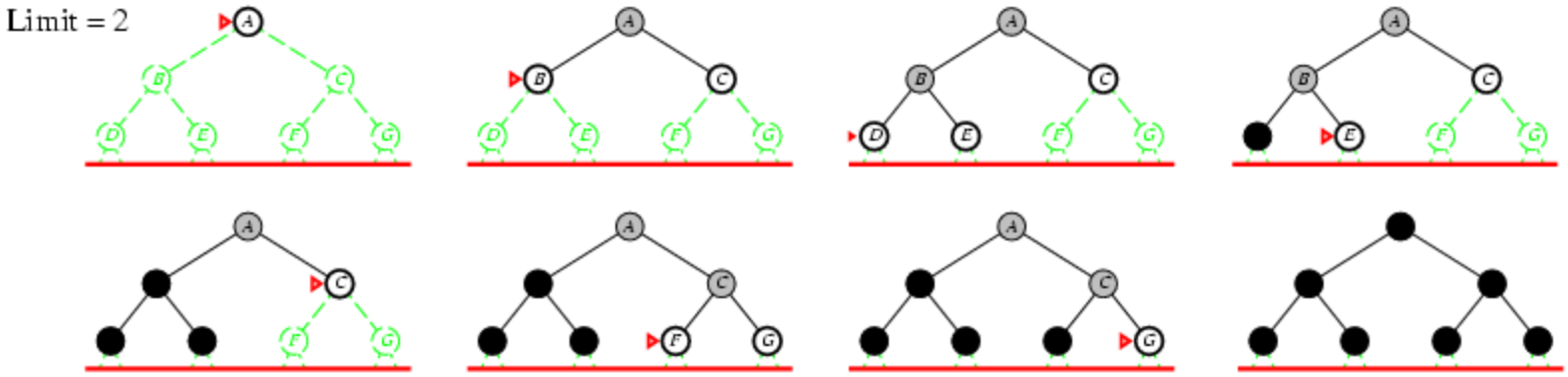


IDS: Example l =0

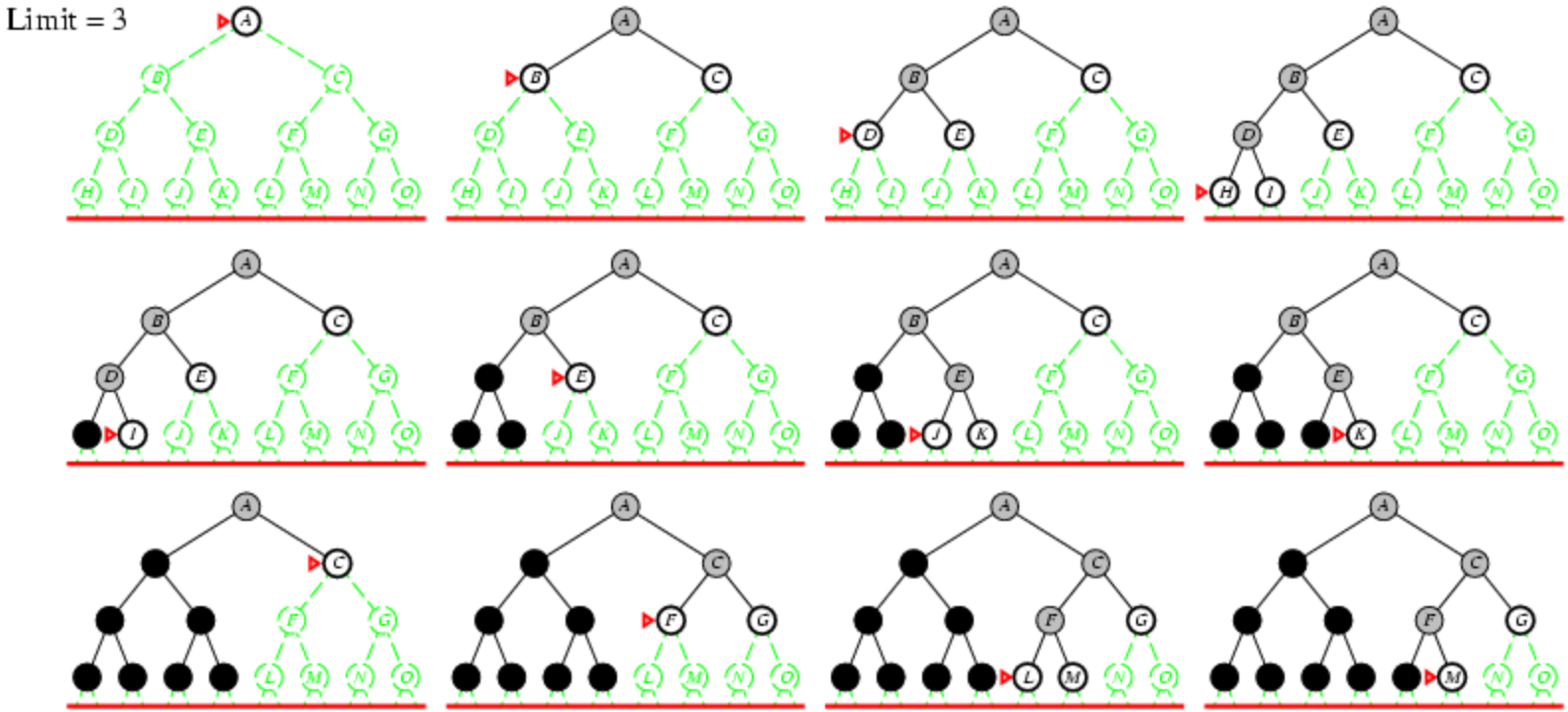
Limit = 0

IDS: Example $l = 1$

IDS: Example $l = 2$



IDS: Example $l = 3$



Iterative Deepening Search (IDS)

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or failure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

- Combines benefits of DFS & BFS
  - DFS: low memory requirement
  - BFS: completeness & also optimality for special path cost functions
- Not such wasteful (most of the nodes are in the bottom level)

Properties of iterative deepening search

- Complete?
  - Yes (for finite b and d)
- Time
  - $d \times b^1 + (d-1) \times b^2 + \cdots + 2 \times b^{d-1} + 1 \times b^d = O(b^d)$
- Space
  - $O(bd)$
- Optimal?
  - Yes, if path cost is a non-decreasing function of the node depth
- IDS is the **preferred method** when search space is large and the depth of solution is unknown

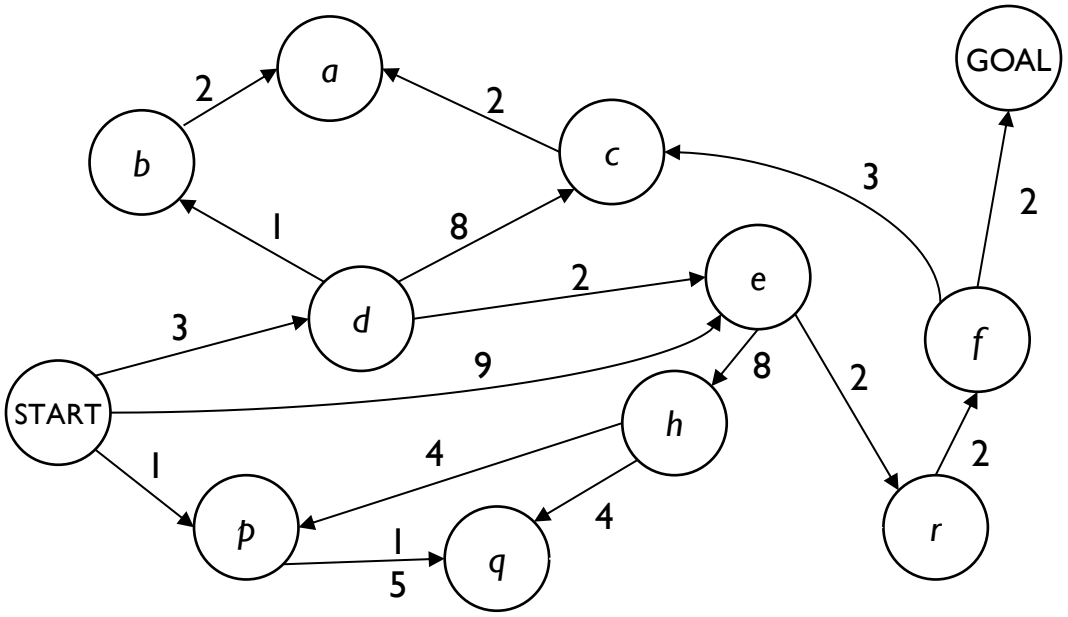
Iterative deepening search

- Number of nodes generated to depth *d*:

$$N_{IDS} = d \times b^1 + (d-1) \times b^2 + \ \ldots \ + 2 \times b^{d-1} + 1 \times b^d$$
$$= O(b^d)$$

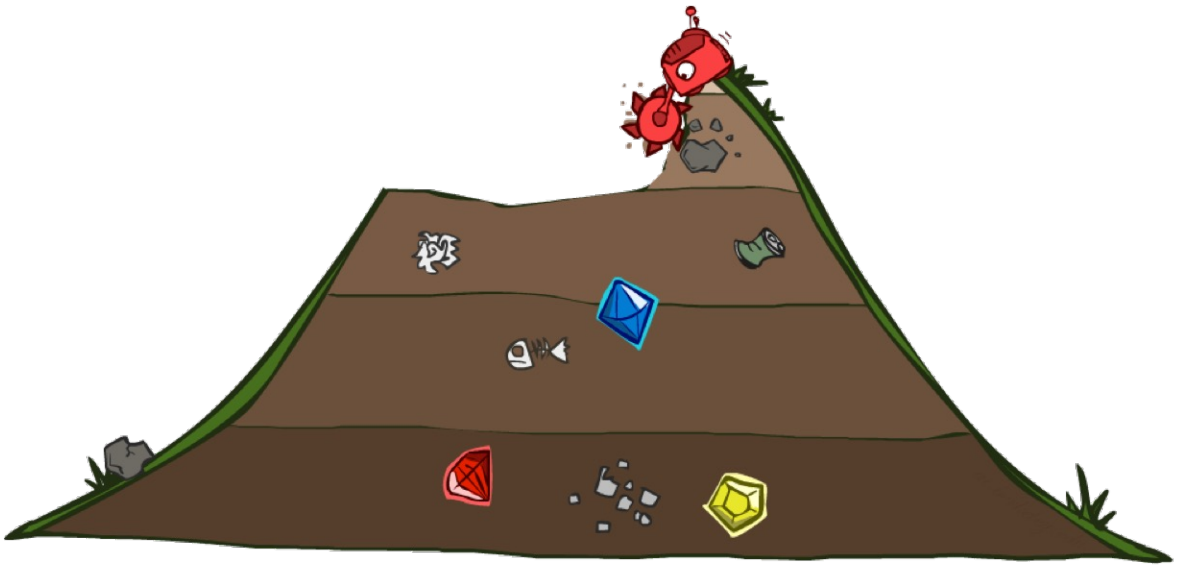
- For $b = 10, d = 5$, we compute number of generated nodes:
  - N_{BFS} = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110
  - N_{IDS} = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
  - Overhead of IDS = (123,450 - 111,110)/111,110 = 11%

Cost-Sensitive Search



- BFS finds the shortest path in terms of number of actions.
- It does not find the least-cost path.
- We will now cover a similar algorithm which does find the least-cost path.

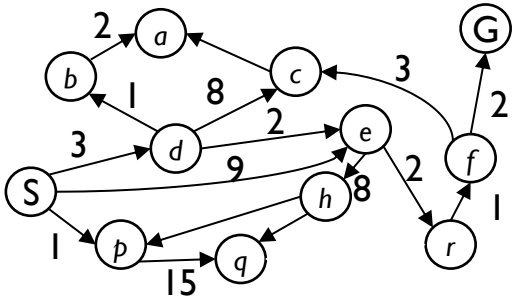
Uniform Cost Search

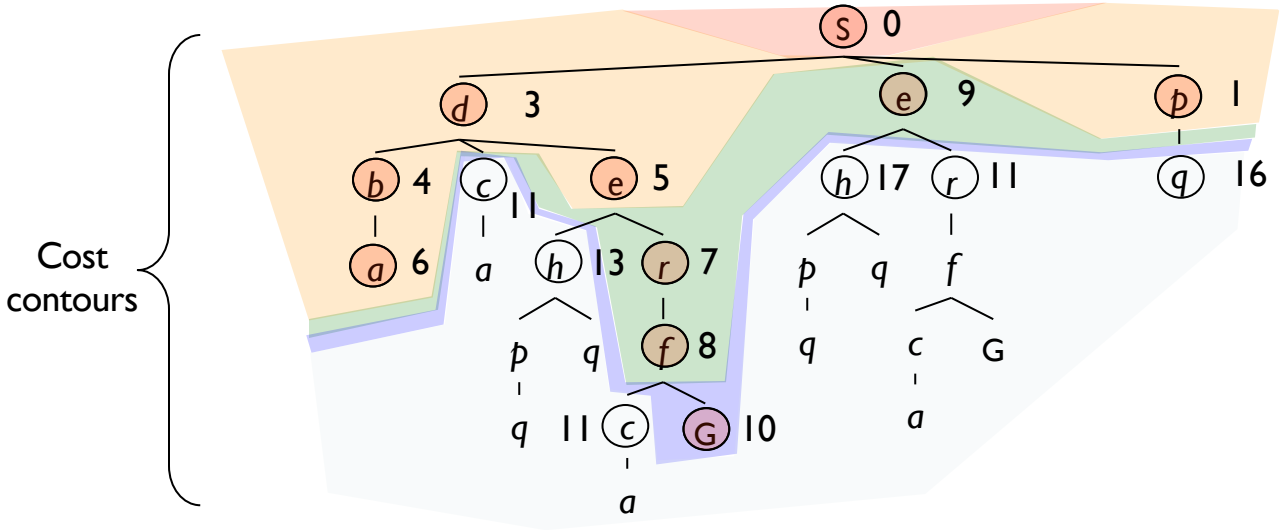


Uniform Cost Search

Strategy: expand a cheapest node first:

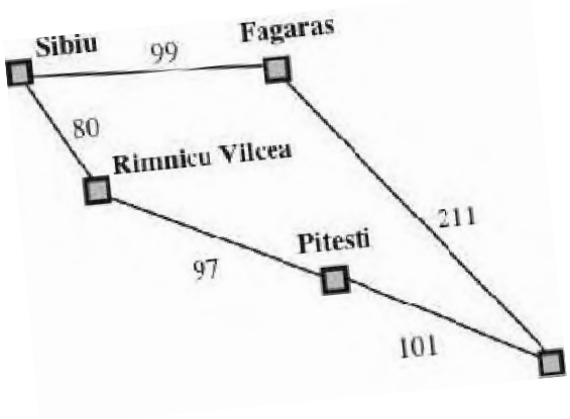
frontier is a priority queue (priority: cumulative cost)





Uniform-Cost Search (UCS)

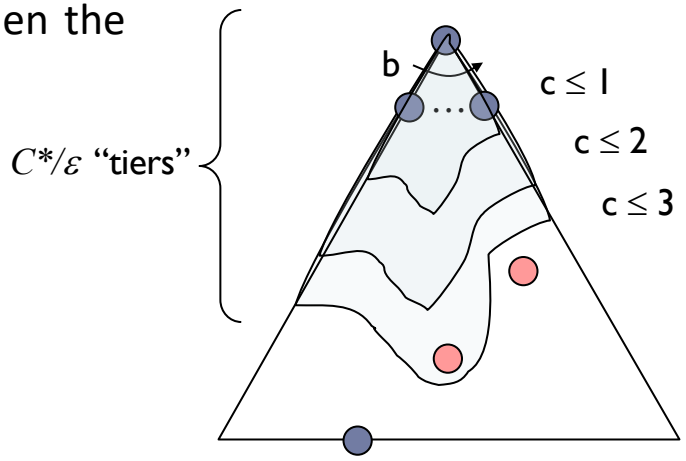
- Expand node n (in the frontier) with the lowest path cost $g(n)$
  - Extension of BFS that is proper for any step cost function
- Implementation: Priority queue (ordered by path cost) for frontier
- Equivalent to breadth-first if all step costs are equal
  - Two differences
    - Goal test is applied when a node is selected for expansion
    - A test is added when a better path is found to a node currently on the frontier



$$80 + 97 + 101 < 99 + 211$$

Uniform-Cost Search (UCS) Properties

- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε, then the "effective depth" is roughly C^*/ε
  - Takes time O($b^{C^*/\varepsilon}$) (exponential in effective depth)
- How much space does the frontier take?
  - Has roughly the last tier, so O($b^{C^*/\varepsilon}$)
- Is it complete?
  - Assuming best solution has a finite cost and minimum arc cost is positive, yes!
- Is it optimal?
  - Yes!



Uniform Cost search (proof of optimality)

- **Lemma**: If UCS selects a node n for expansion, the optimal solution to that node has been found.

  Proof by contradiction: Another frontier node n' must exist on the optimal path from initial node to n (using graph separation property). Moreover, based on definition of path cost (due to non-negative step costs, paths never get shorter as nodes are added), we have $g(n') \leq g(n)$ and thus n' would have been selected first.

⇒ Nodes are expanded in order of their optimal path cost.

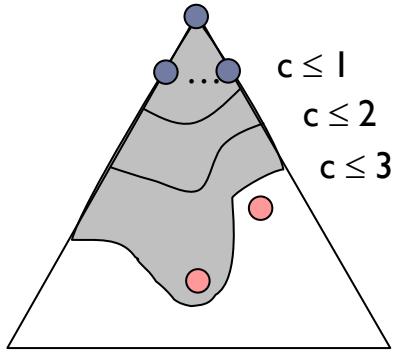
Properties of uniform-cost search

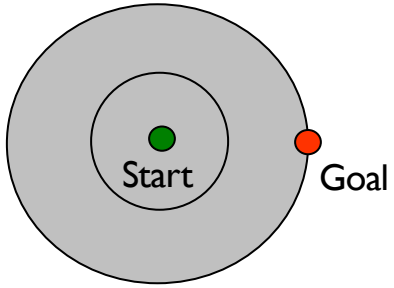
- Complete?
  - Yes, if step cost $\geq \varepsilon > 0$ (to avoid infinite sequence of zero-cost actions)
- Time
  - Number of nodes with "$g \leq$ cost of optimal solution", $O(b^{1+C^*/\varepsilon})$ where C^* is the optimal solution cost
    - $O(b^{d+1})$ when all step costs are equal
- Space
  - Number of nodes with $g \leq$ cost of optimal solution, $O(b^{1+C^*/\varepsilon})$
- Optimal?
  - Yes – nodes expanded in increasing order of $g(n)$

Difficulty: many long paths of actions may exist with cost $\leq C^*$

Uniform cost issues

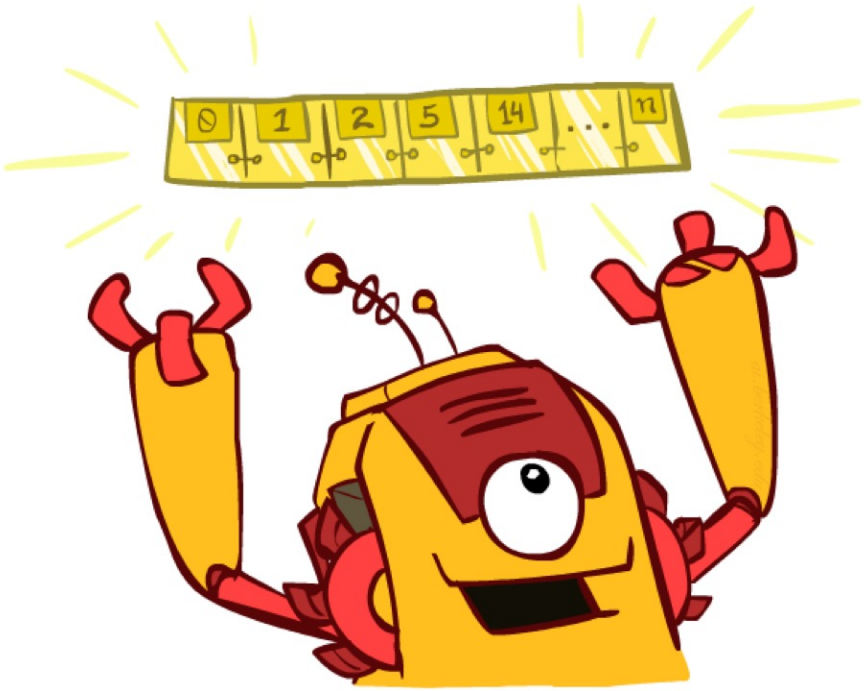
- Remember: UCS explores increasing cost contours
- The good: UCS is complete and optimal!
- The bad:
  - Explores options in every "direction"
  - No information about goal location
- We'll fix that soon!





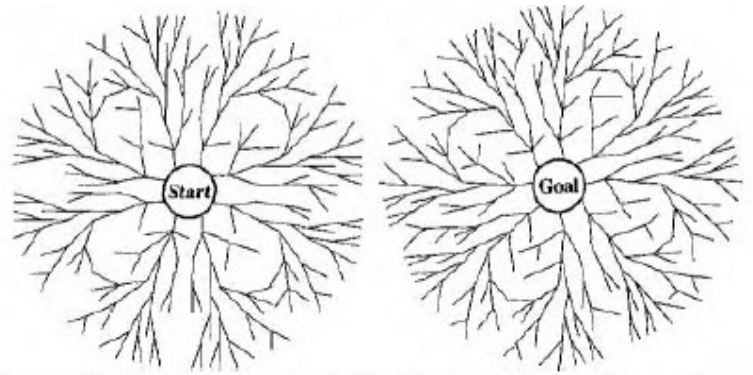
The one queue

- All these search algorithms are the same except for frontier strategies
  - Conceptually, all frontiers are priority queues (i.e. collections of nodes with attached priorities)
  - Practically, for DFS and BFS, you can avoid the log(n) overhead from an actual priority queue, by using stacks and queues
  - Can even code one implementation that takes a variable queuing object



Bidirectional search

- Simultaneous forward and backward search (hoping that they meet in the middle)
  - Idea: $b^{d/2} + b^{d/2}$ is much less than b^d
  - "Do the frontiers of two searches intersect?" instead of goal test
    - First solution may not be optimal
- Implementation
  - Hash table for frontiers in one of these two searches
    - Space requirement: most significant weakness
  - Computing predecessors?
    - May be difficult
  - List of goals? a new dummy goal
    - Abstract goal (checkmate)?!



Summary of algorithms (tree search)

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

a	Complete if *b* is finite
b	Complete if step cost ≥ ε>0
c	Optimal if step costs are equal
d	If both directions use BFS