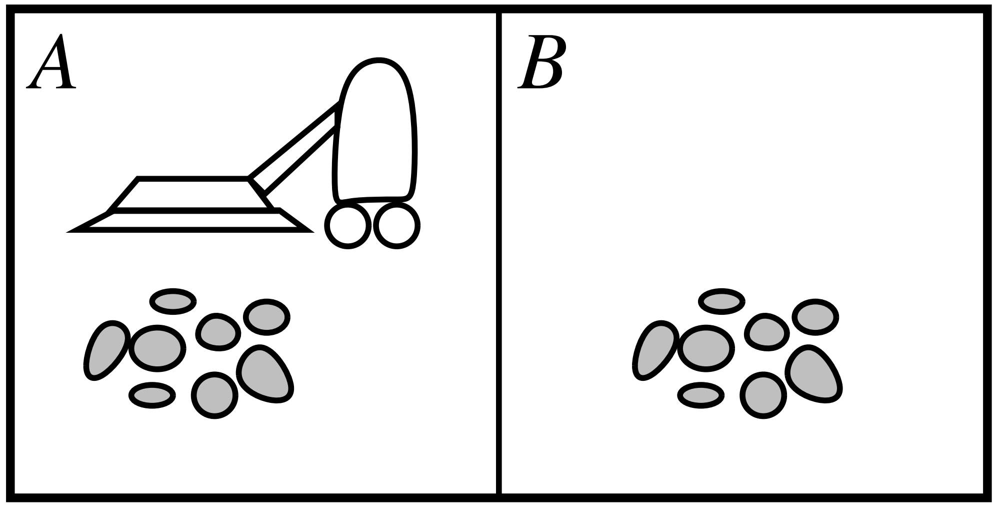
Chapter 3: Problem solving with search

Announcements:
- HW0 is up
- Tutorials will be announced soon

Recap:
- AI: rational/useful agents
- Components of an agent: PEAS
- Environment types determine which algorithms we can apply.

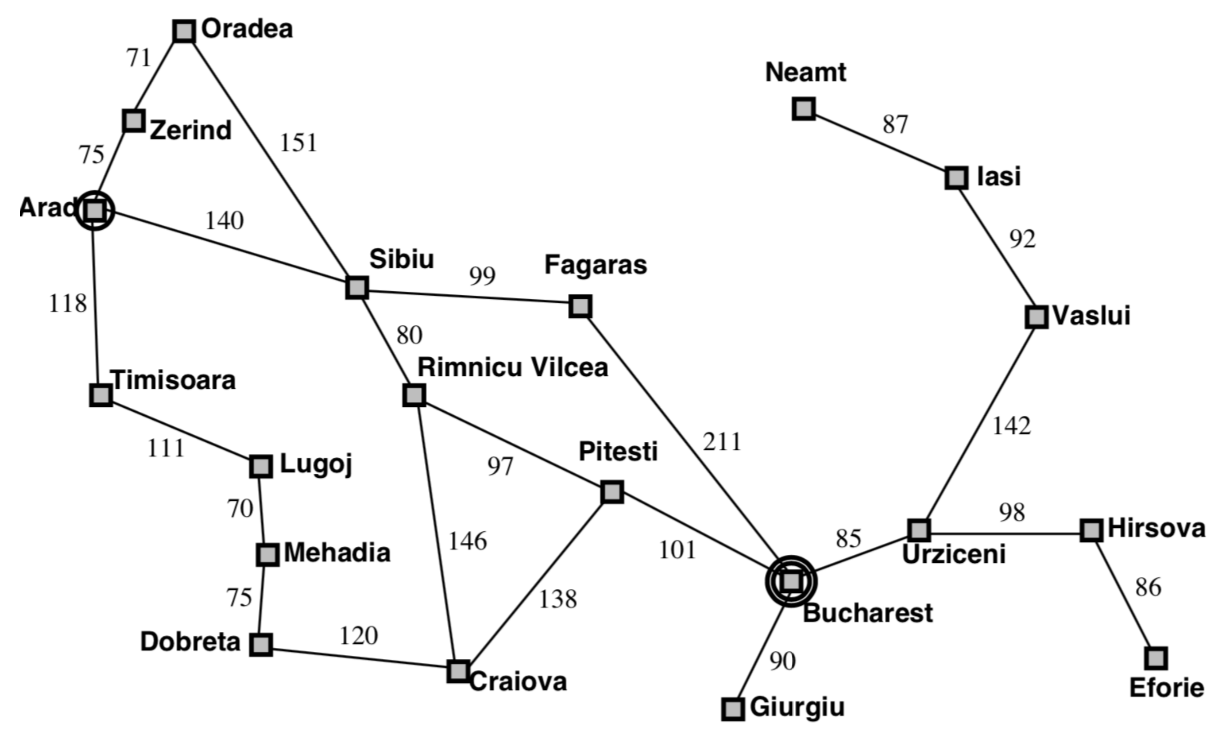
- Current focus: **observable** and **deterministic**.
- And **single-agent**; we will talk about multi-agent (games) next.
- Also **discrete**. We will assume this for most of this course.
- Offline problem solving. Note that we can execute "eyes closed".
- Problems like this are solvable with search, like breadth-first or depth-first (and more that we will discuss).

Example: Vacuum cleaner



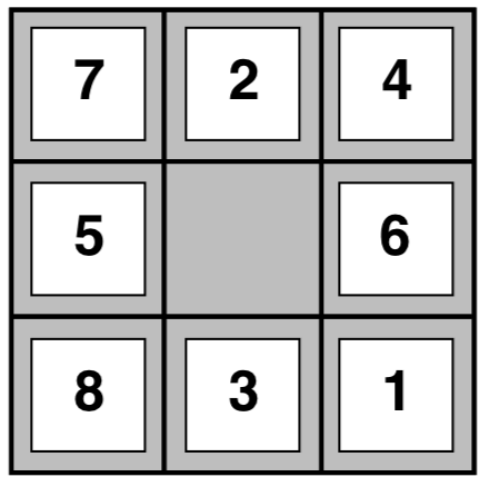
- First, a few examples
- Specifically, the observed, deterministic variant.
- Plan: Suck, Right, Suck

Example: Directions in a map

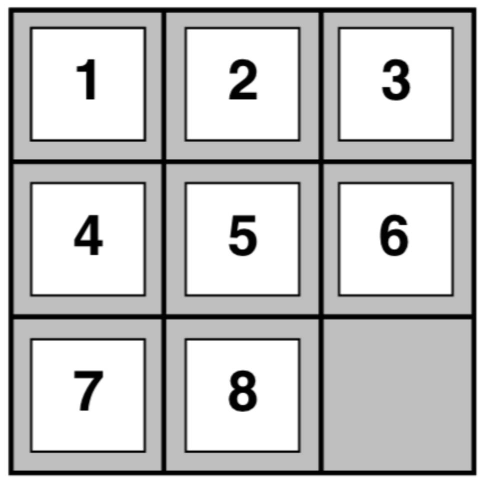


- Deterministic: From the perspective of Google Maps, the driver is nondeterministic (they sometimes choose a different route); it needs to reroute. We assume they are deterministic.

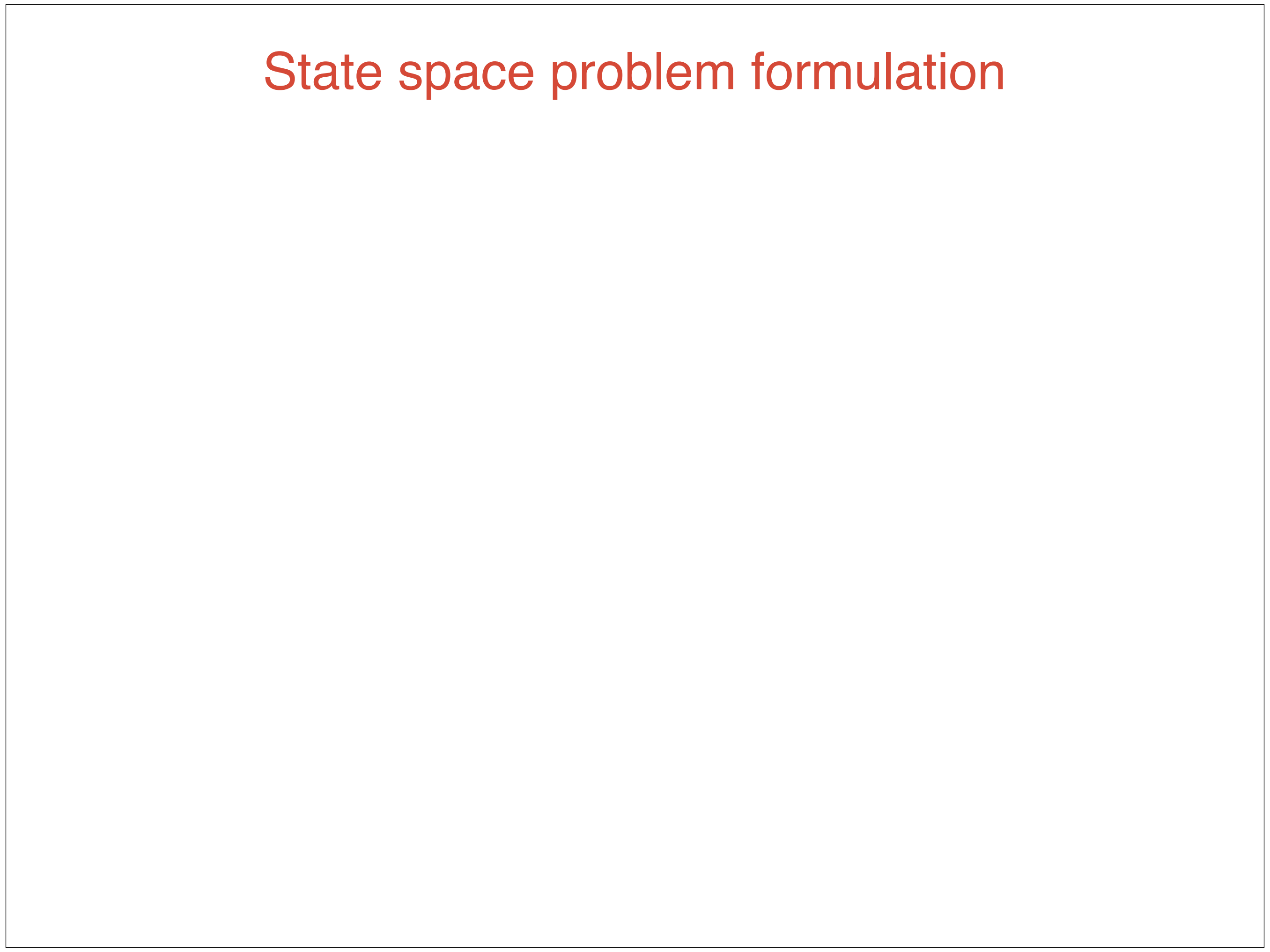
Example: 8-puzzle



Start State



Goal State



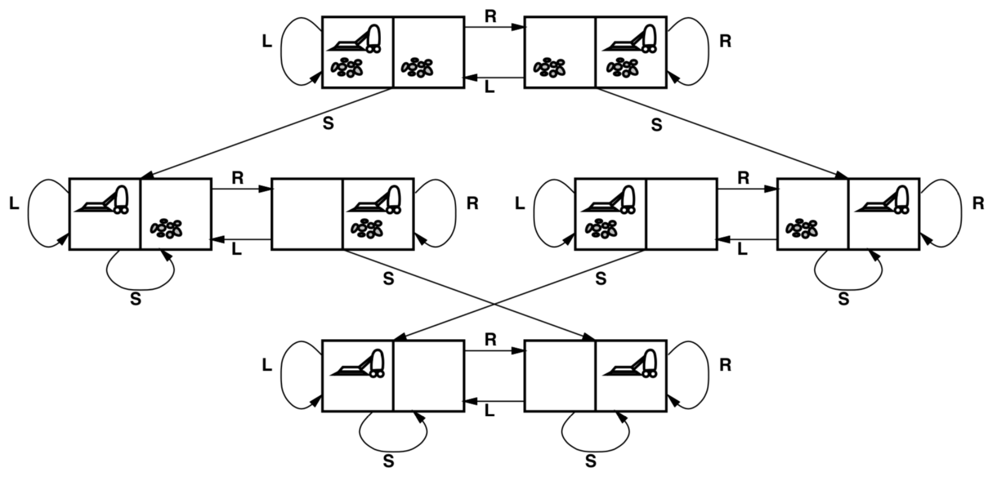
State space problem formulation

state = unique arrangement of the world

A problem is defined by four things:
Problem:

- **Initial state**: **Arad**
- **Successor function**: **S(Arad , Arad -> Zerind) = Zerind.** Defines how actions influence state.
- **Goal test: Bucharest.** Defines when we are done. Either a single goal state or a set. **Vacuum: no dirt, either R or L**. In general: a function that returns "goal" or "not goal".
- **Action cost: cost(Arad , Arad -> Zerind) = 75**. How we choose between different solutions. Cost, distance, etc. What if we just care about the total number of actions? cost = 1. What if we don't care and just want any solution that reaches the goal? cost = 0.

Vacuum world state space



States:

Actions:

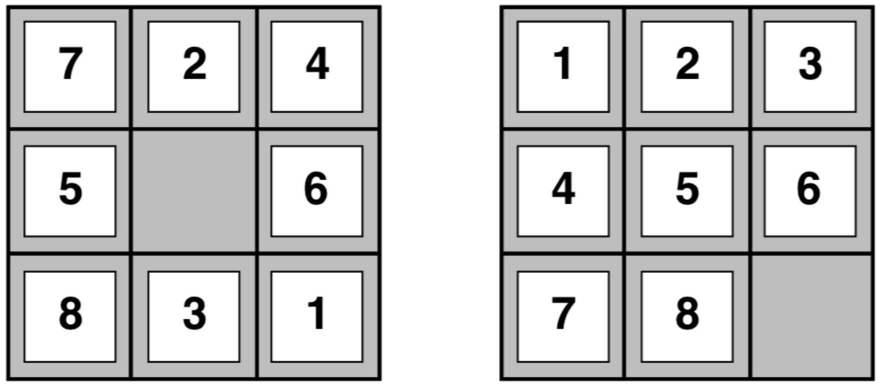
Initial:

Successor:

Goal test: No dirt

Cost: 1 per action

8-puzzle

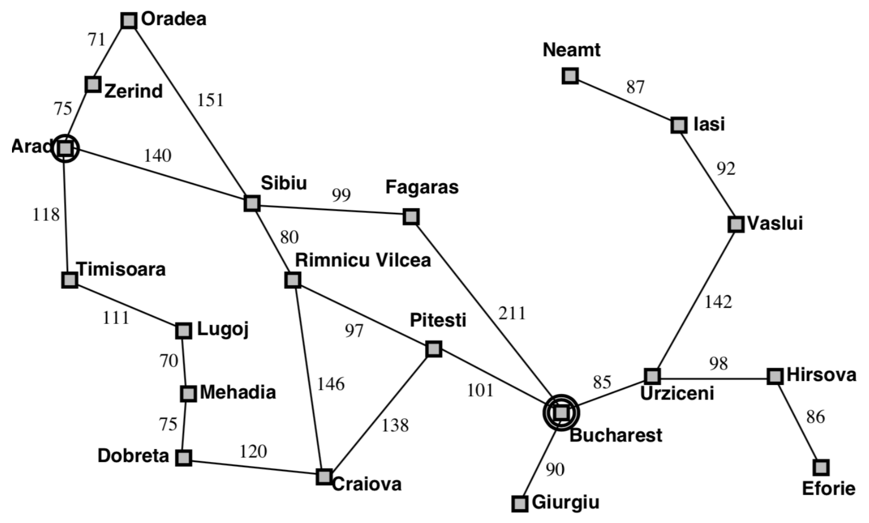


Start State **Goal State**

States: Positions
Actions: Move empty square

Initial:
Successor:
Goal: Ordered
Path: 1 per action

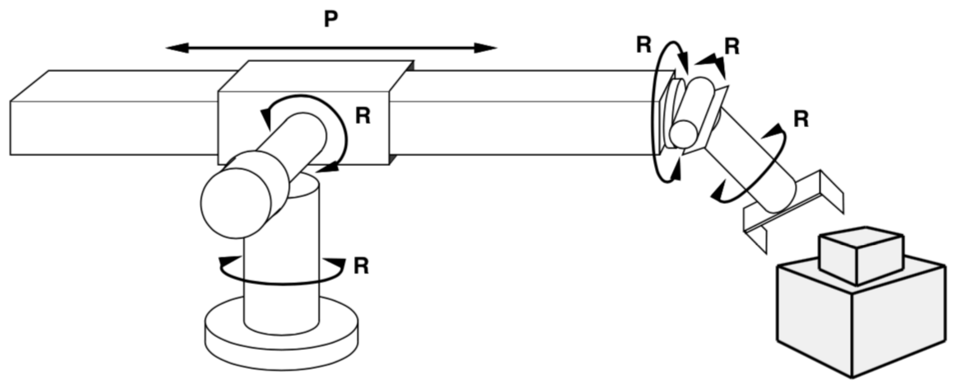
Directions in a map



States: Locations
Actions: Drive

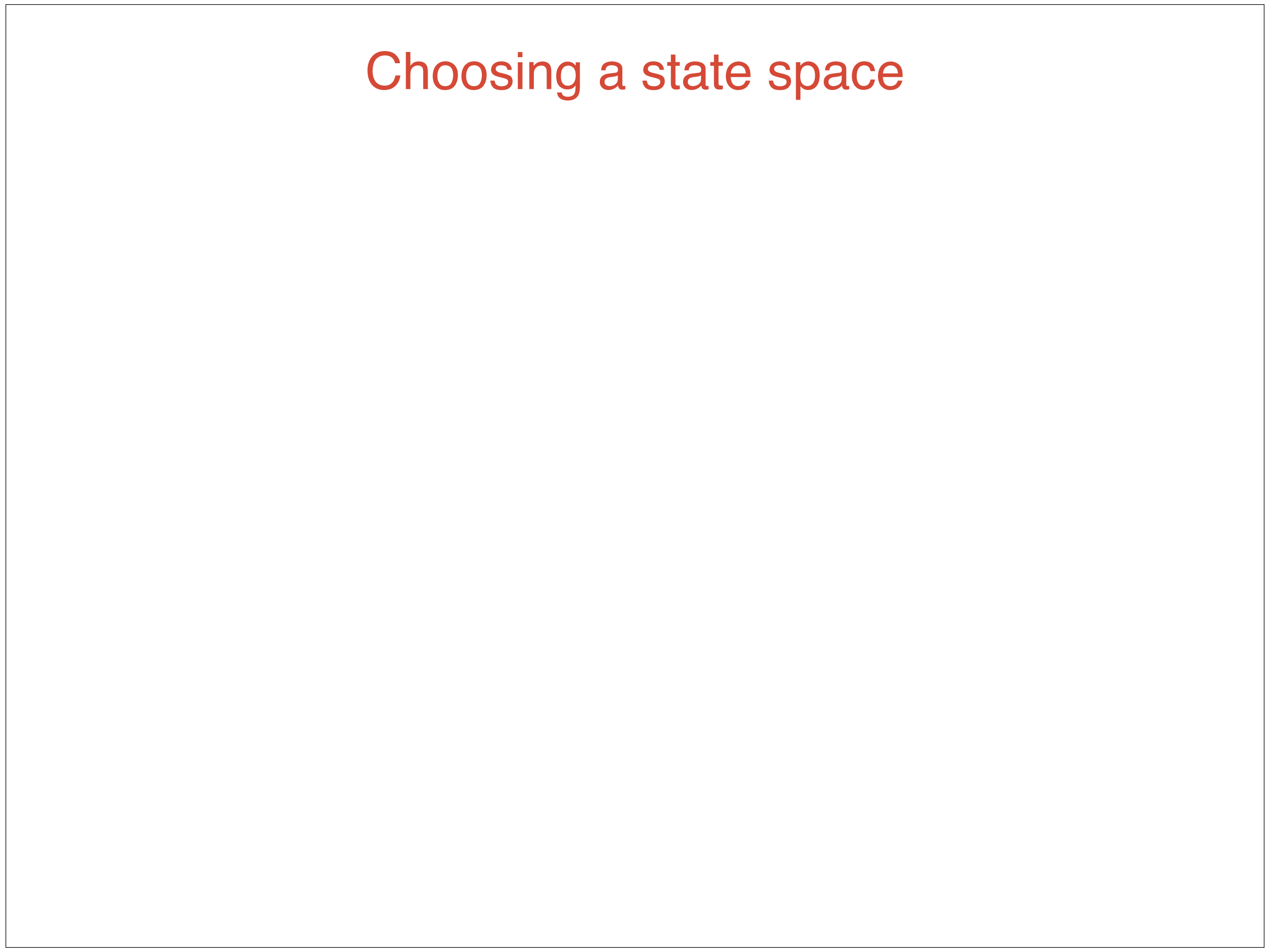
Initial: Arad
Successor:
Goal: Bucharest
Path: Distance

Robotic assembly



States: Real-value angles of joints, real-value location of parts.
Actions: Change joint angle.

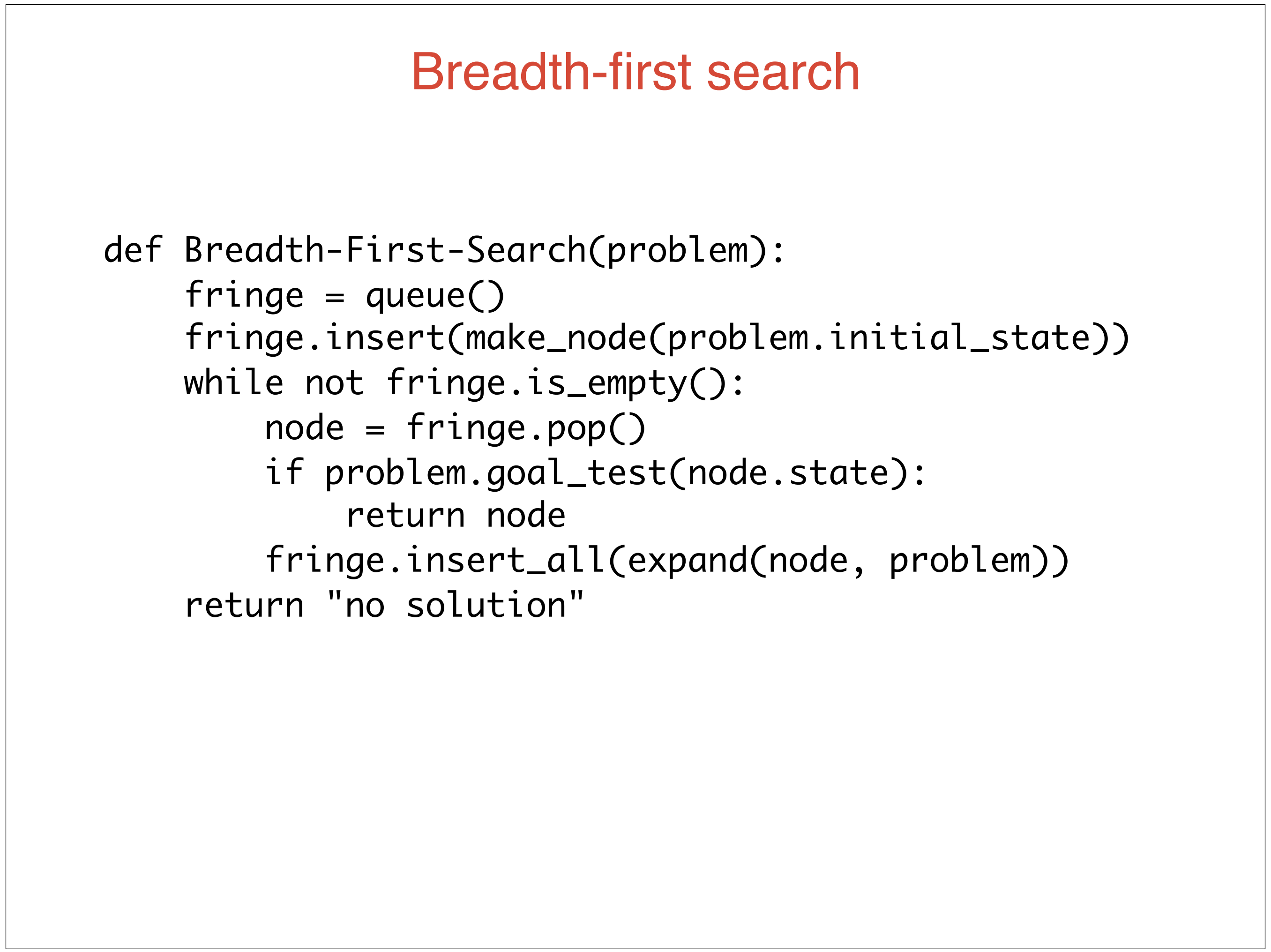
Initial:
Successor:
Goal: Correct arrangement of parts. (We don't care about arm)
Path: Time to execute



Choosing a state space

The real world is complex and continuous.
We always solve problems using an abstract state space.
Real state = location of all atoms, ...
Abstract states:

- **state = set of equivalent real states.** Equivalent for the purposes of our problem.
    - Map: we could be anywhere in the city.
    - 8-puzzle: The tile has "wiggle room" within the same spot.
- **action = set of equivalent real actions**. Equivalent for the purposes of our problem. All real actions should have about the same cost.
    - Map route: Many possible routes, rest spots, etc. Ignore turn directions?
    - 8-puzzle: Ignore intermediate positions. Ignore sticky tiles?
- Choosing a real action from an abstract one should be computationally easy! Otherwise it's a bad abstraction.



Breadth-first search

```
def Breadth-First-Search(problem):
    fringe = queue()
    fringe.insert(make_node(problem.initial_state))
    while not fringe.is_empty():
        node = fringe.pop()
        if problem.goal_test(node.state):
            return node
        fringe.insert_all(expand(node, problem))
    return "no solution"
```

- We will go through a bunch of algorithms for solving these problems.
- First uninformed. States are either goal or not goal -- no knowledge of which states are "better".
- Informed later ("better" = closer to the goal)

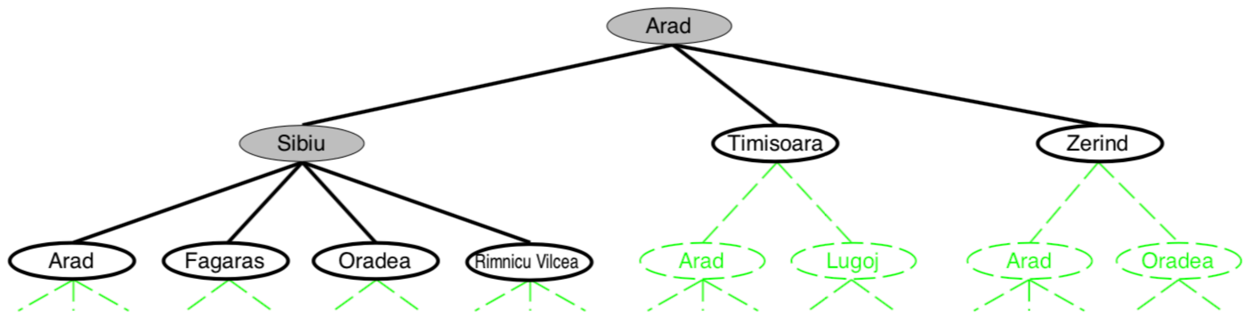
Properties

Expand

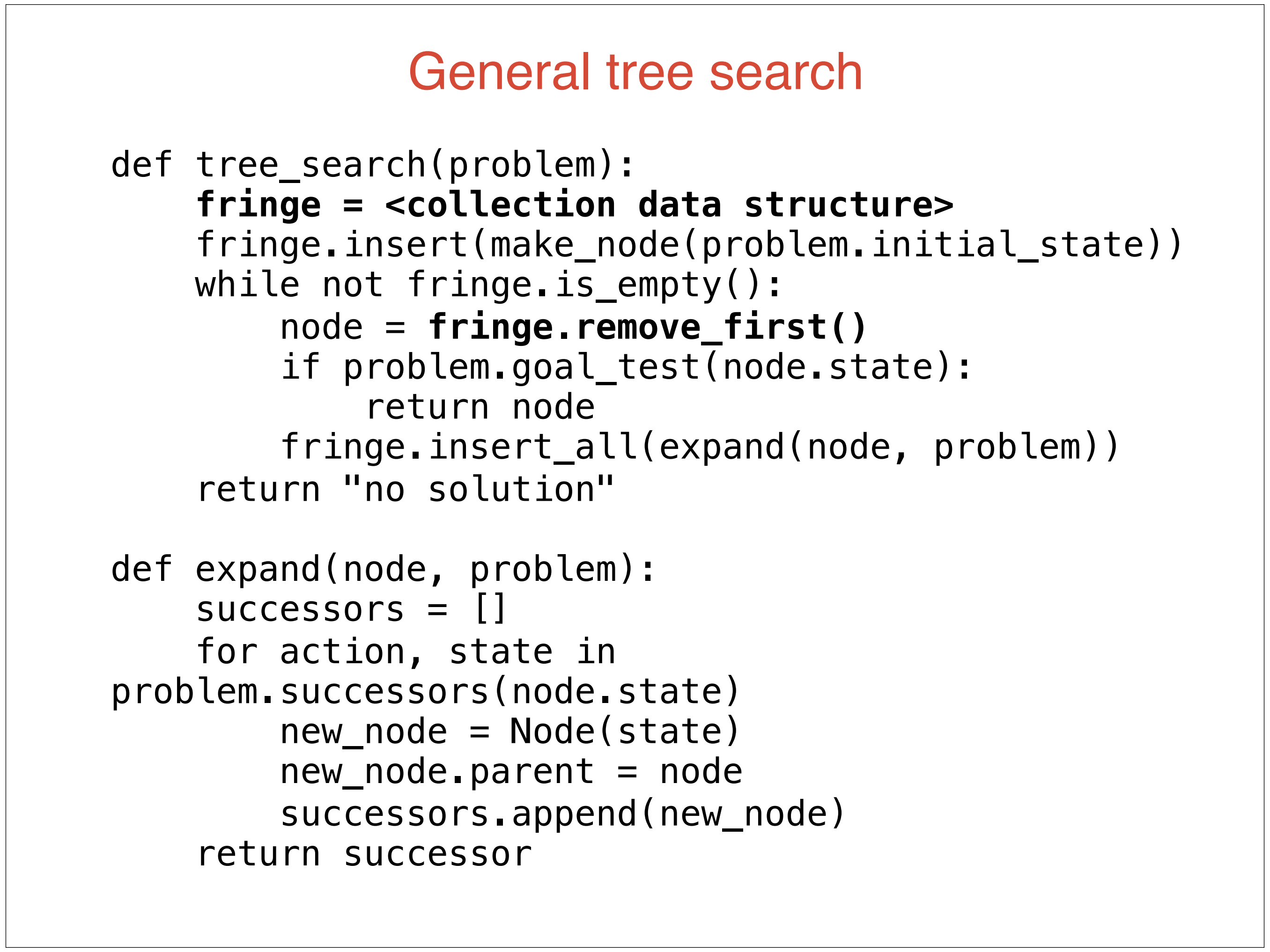
```
def expand(node, problem):
    successors = []
    for action, state in
problem.successors(node.state)
        new_node = Node(state)
        new_node.parent = node
        successors.append(new_node)
    return successors
```

Search tree

search node: state + parent node + action



- Search node = state + list of actions to get there
- Different than just a state! Can have two nodes with the same state.
- Search tree = formed as we search

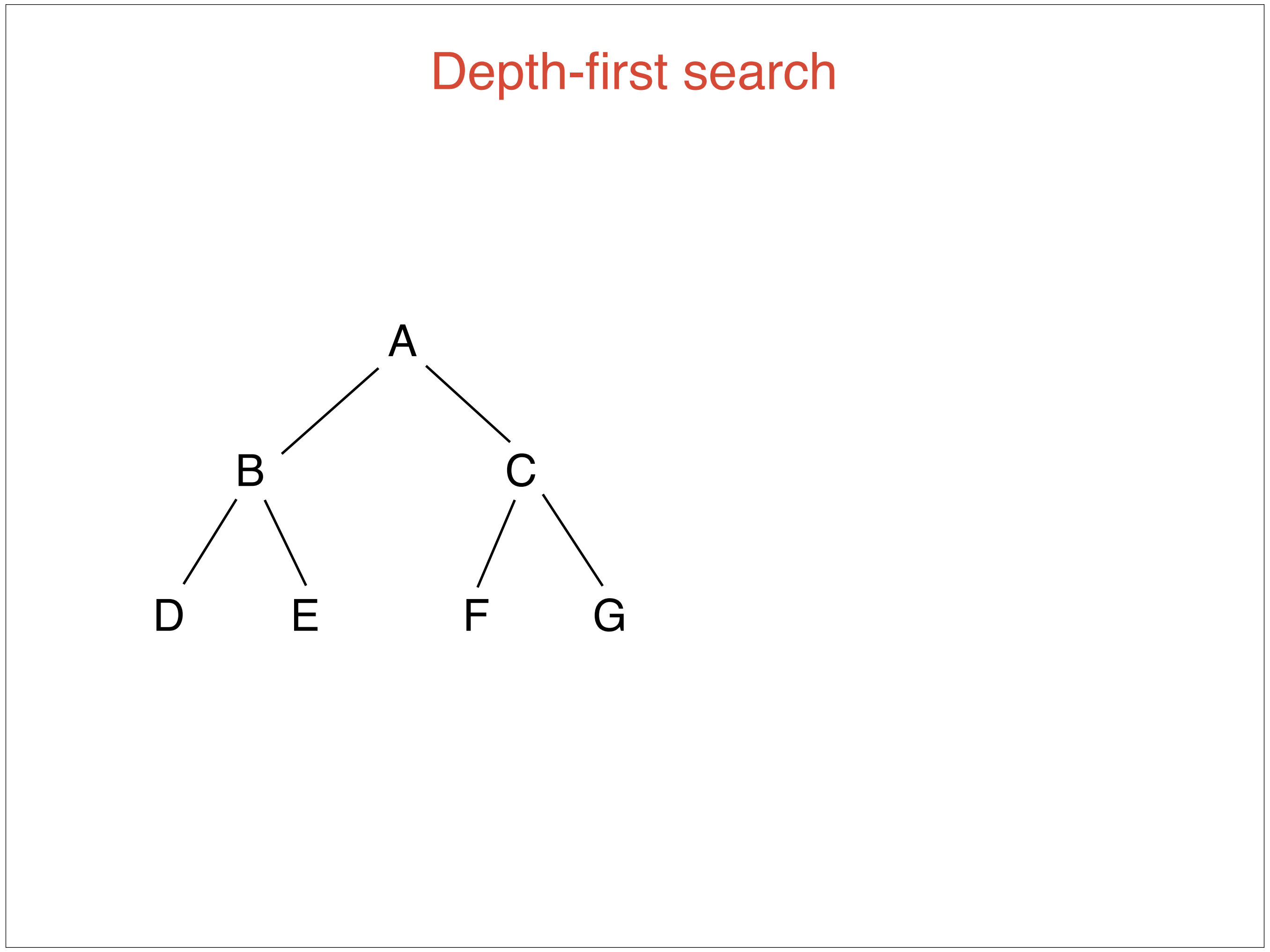


General tree search

```
def tree_search(problem):
    fringe = <collection data structure>
    fringe.insert(make_node(problem.initial_state))
    while not fringe.is_empty():
        node = fringe.remove_first()
        if problem.goal_test(node.state):
            return node
        fringe.insert_all(expand(node, problem))
    return "no solution"

def expand(node, problem):
    successors = []
    for action, state in
problem.successors(node.state)
        new_node = Node(state)
        new_node.parent = node
        successors.append(new_node)
    return successor
```

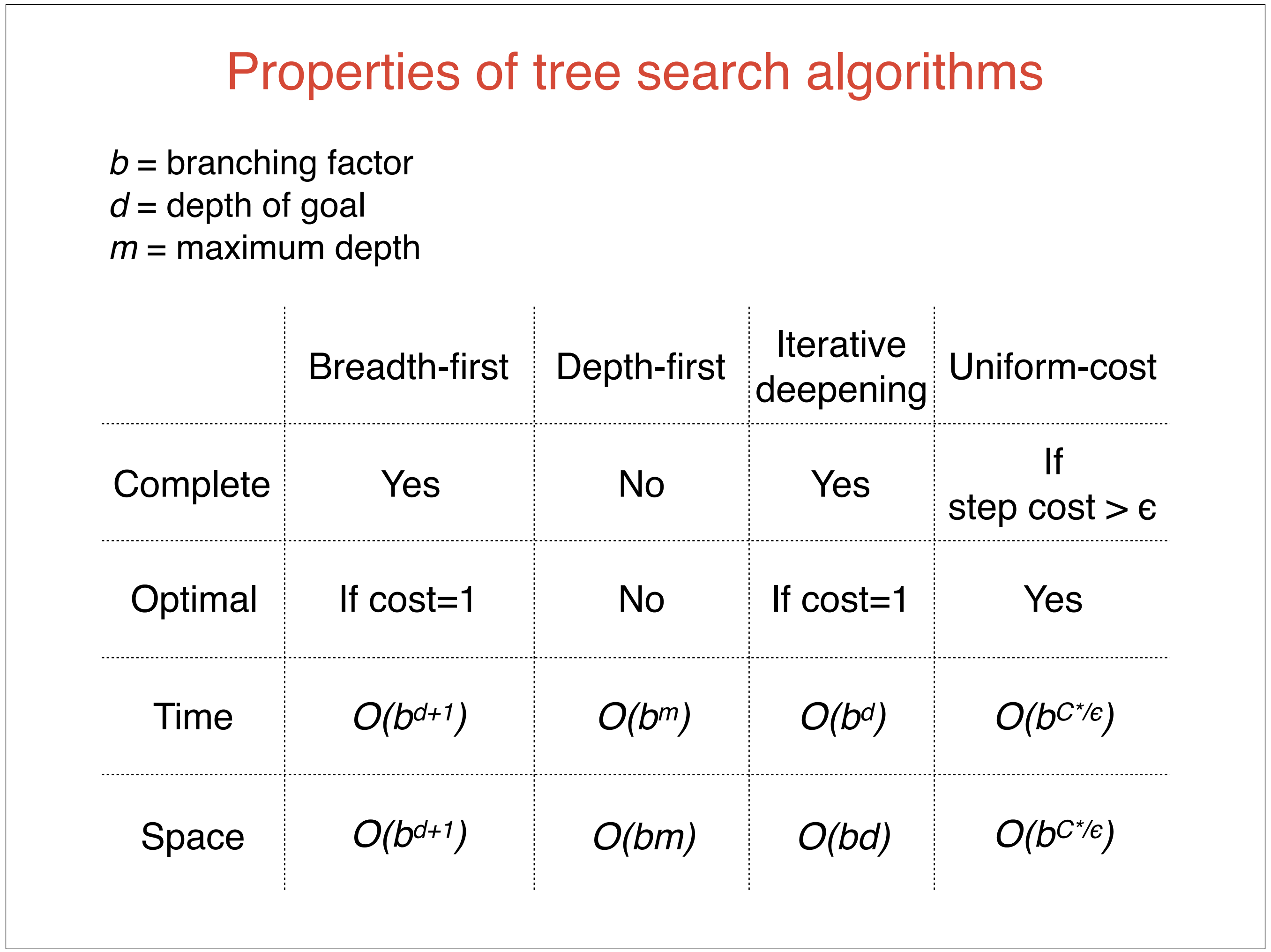
- Recap:
    - Properties, problem solving with search
    - State space. Difference between states and nodes.
    - BFS

- We can generalize BFS.
- This is a general algorithm that we can modify to be any of the algorithms we see
- We just modify fringe / remove_first
- **BFS: FIFO queue**



Depth-first search

- **fringe: stack (LIFO)**
- Excecute: track stack, searched nodes
- Another way to implement is using recursion. The function call stack is a stack.
- Properties.
    - If solution space is dense, can be very fast.



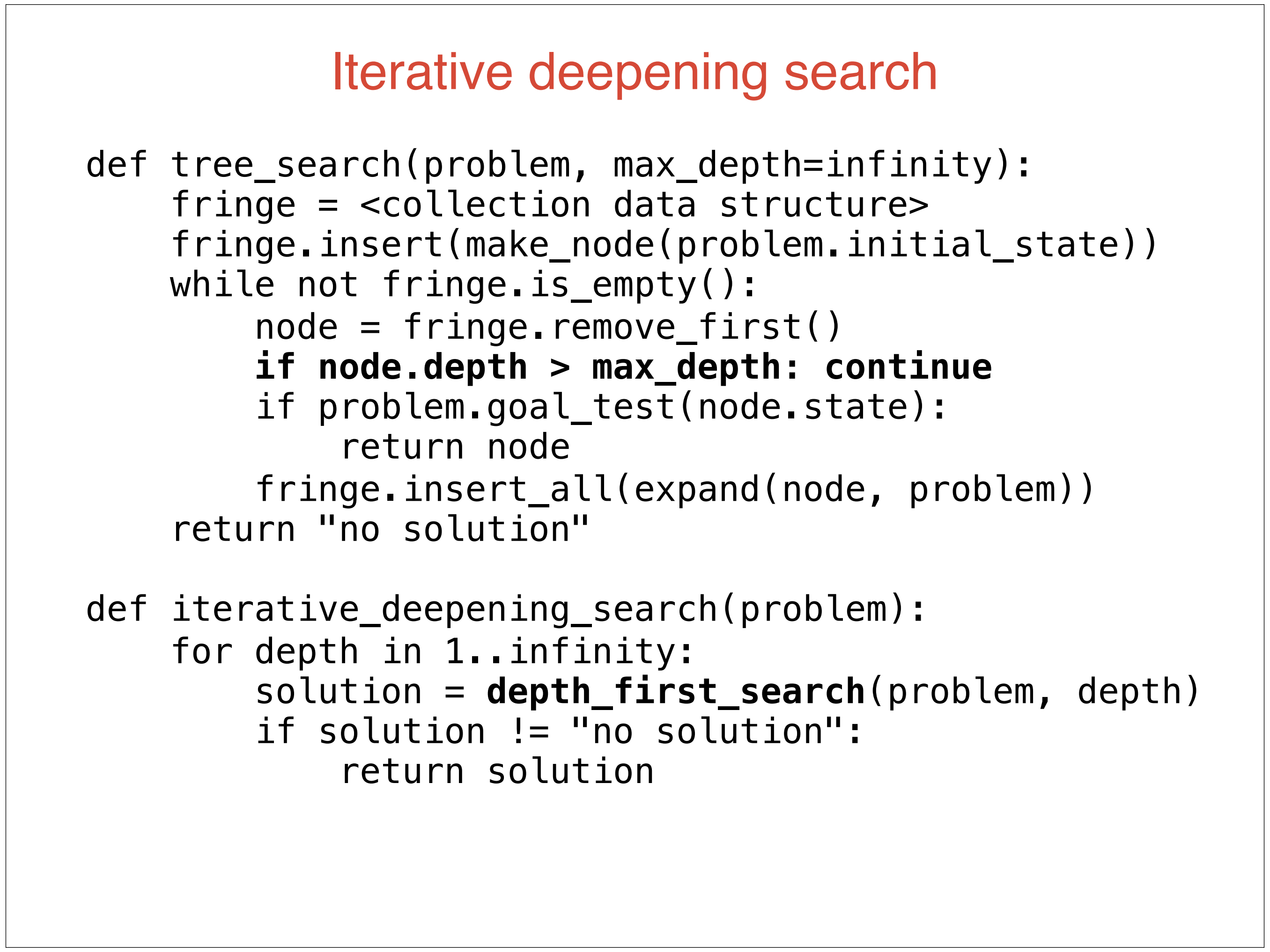
Properties of tree search algorithms

b = branching factor
d = depth of goal
m = maximum depth

	Breadth-first	Depth-first	Iterative deepening	Uniform-cost
Complete	Yes	No	Yes	If step cost > ϵ
Optimal	If cost=1	No	If cost=1	Yes
Time	$O(b^{d+1})$	$O(b^m)$	$O(b^d)$	$O(b^{C^*/\epsilon})$
Space	$O(b^{d+1})$	$O(bm)$	$O(bd)$	$O(b^{C^*/\epsilon})$

Complete = Finds a solution if it exists.
Optimal = Finds the lowest-cost solution.

b = number of actions
d = depth of goal
C* = cost of goal
epsilon = least-cost action

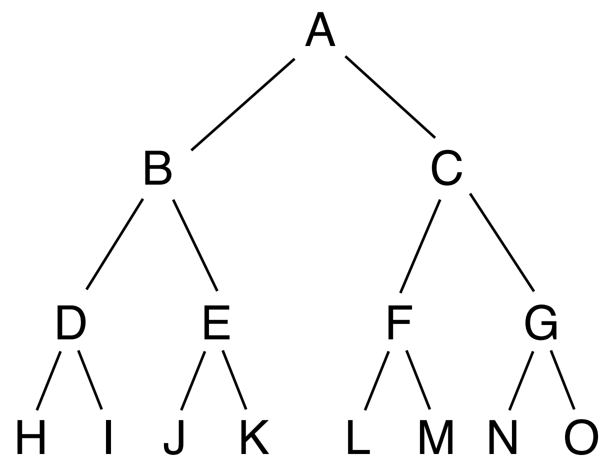


Iterative deepening search

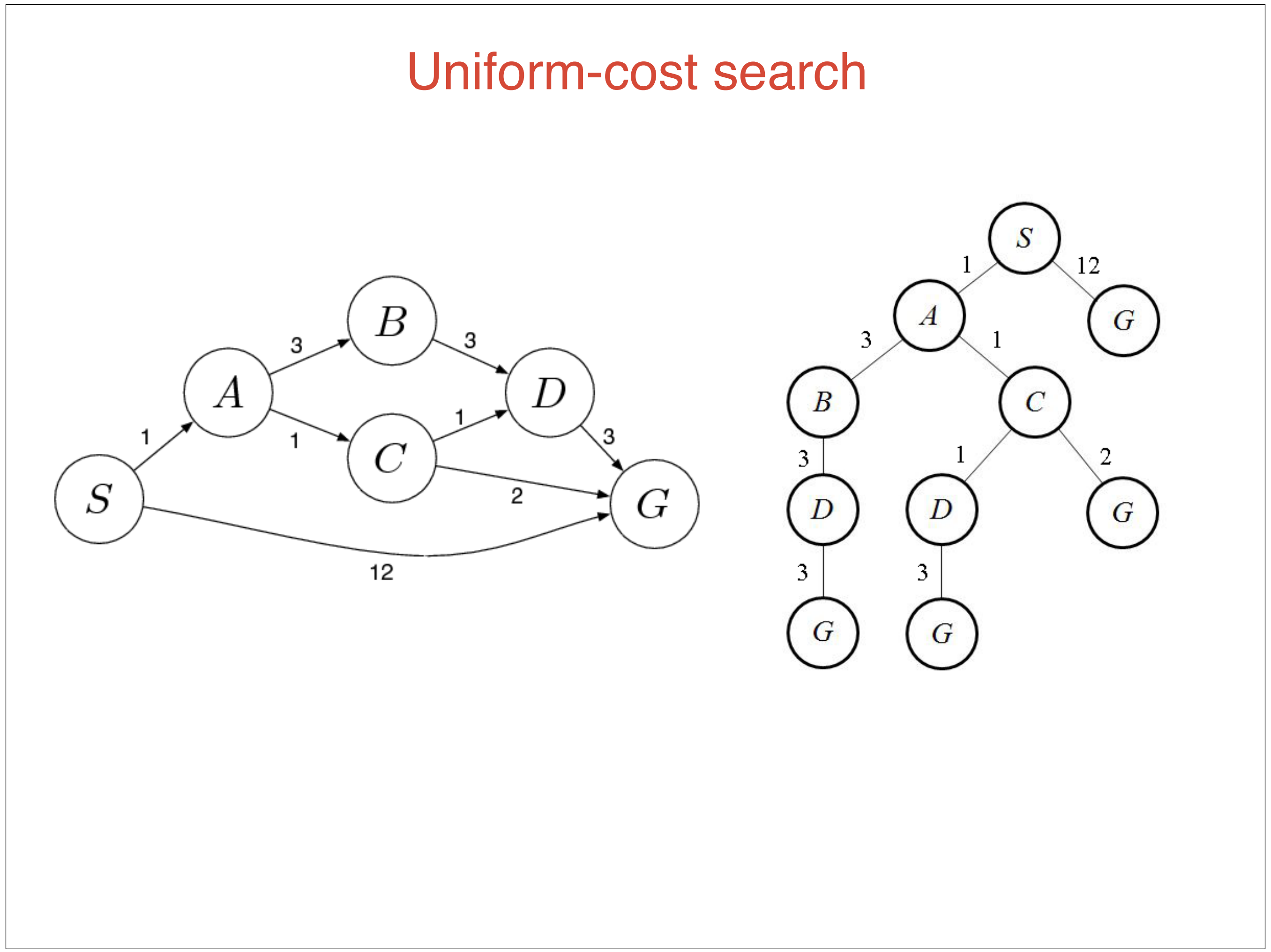
```
def tree_search(problem, max_depth=infinity):
    fringe = <collection data structure>
    fringe.insert(make_node(problem.initial_state))
    while not fringe.is_empty():
        node = fringe.remove_first()
        if node.depth > max_depth: continue
        if problem.goal_test(node.state):
            return node
        fringe.insert_all(expand(node, problem))
    return "no solution"

def iterative_deepening_search(problem):
    for depth in 1..infinity:
        solution = depth_first_search(problem, depth)
        if solution != "no solution":
            return solution
```

- Note that we use specifically DFS. Remember: how do we turn tree search into DFS? Stack
-

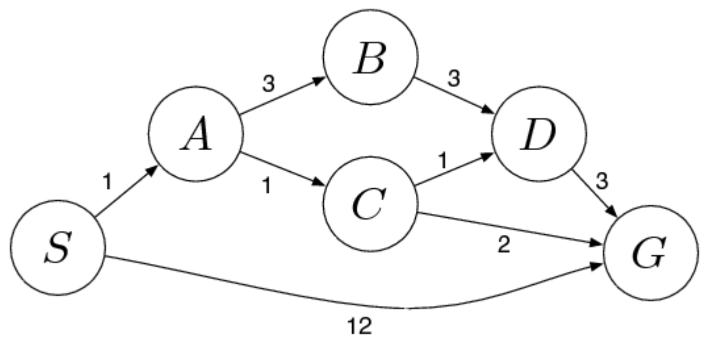
Iterative deepening search

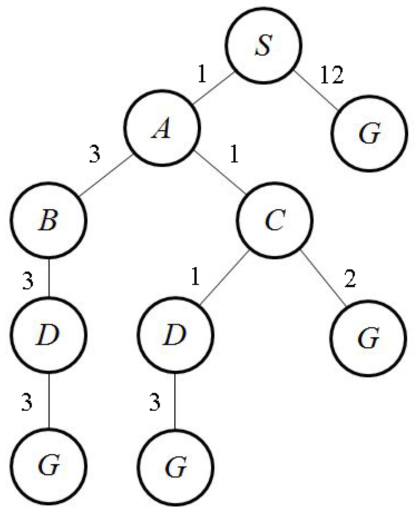


- Execute: track depth, explored nodes
- Properties
- Time: why O(b^d)?
- (d+1)b^0 +db^1 +(d–1)b^2 +...+b^d =O(bd)
- Numerical comparison for b = 10 and d = 5, solution at far right leaf:
- N(IDS) = 50+400+3,000+20,000+100,000=123,450
- N(BFS) = 10+100+1,000+10,000+100,000=111,100



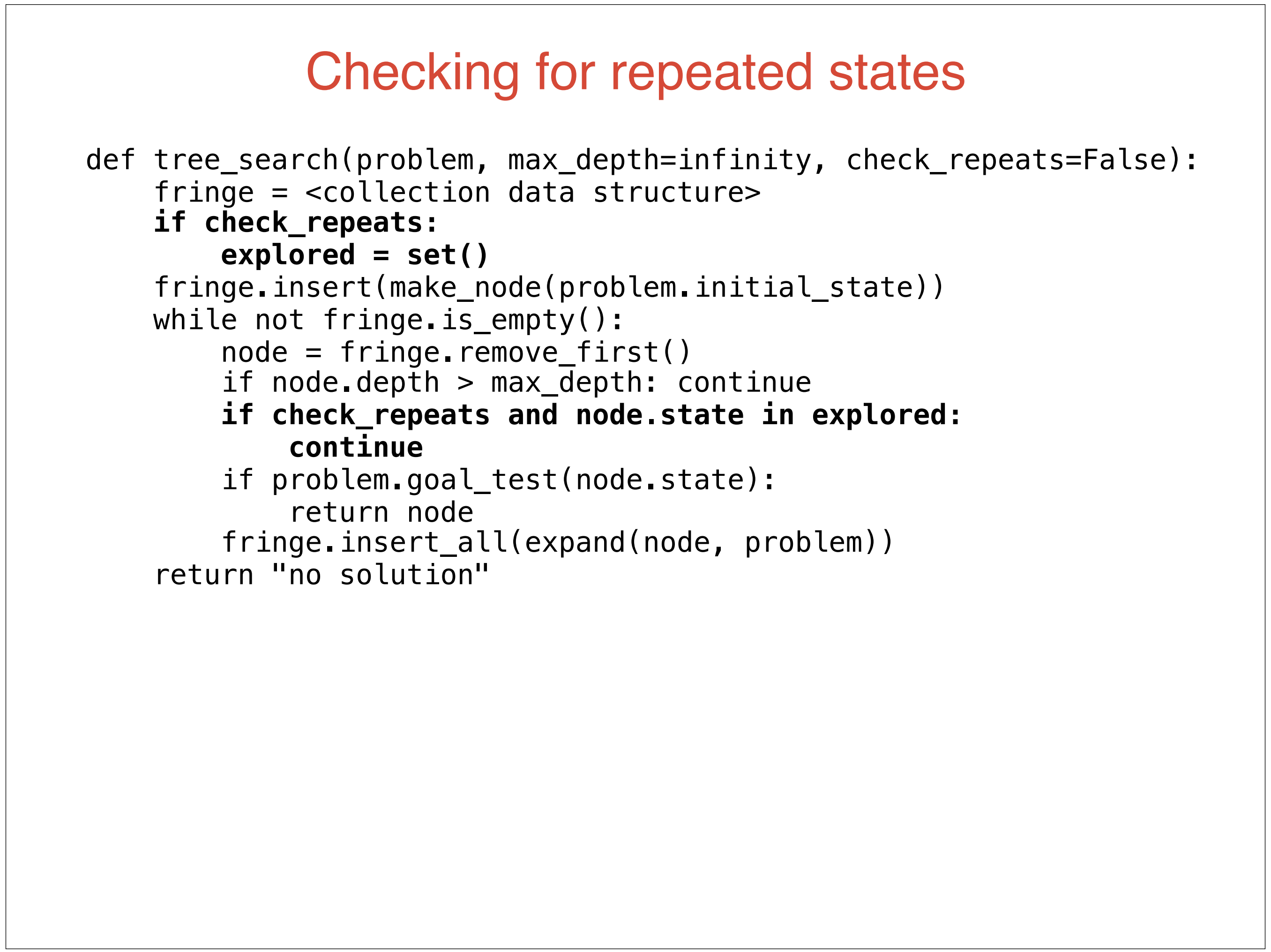
Uniform-cost search





- Recap:
  - Tree search algorithms
  - Properties of algorithms
  - Q about comparison

- What if step costs != 1?
- Uniform-cost search: **fringe = priority_queue.** Using cost to node.
- **search problem** ; **search tree**.
- Example: This is a simpler version of map problem.
- Execute, track pqueue, explored nodes.
- Properties.

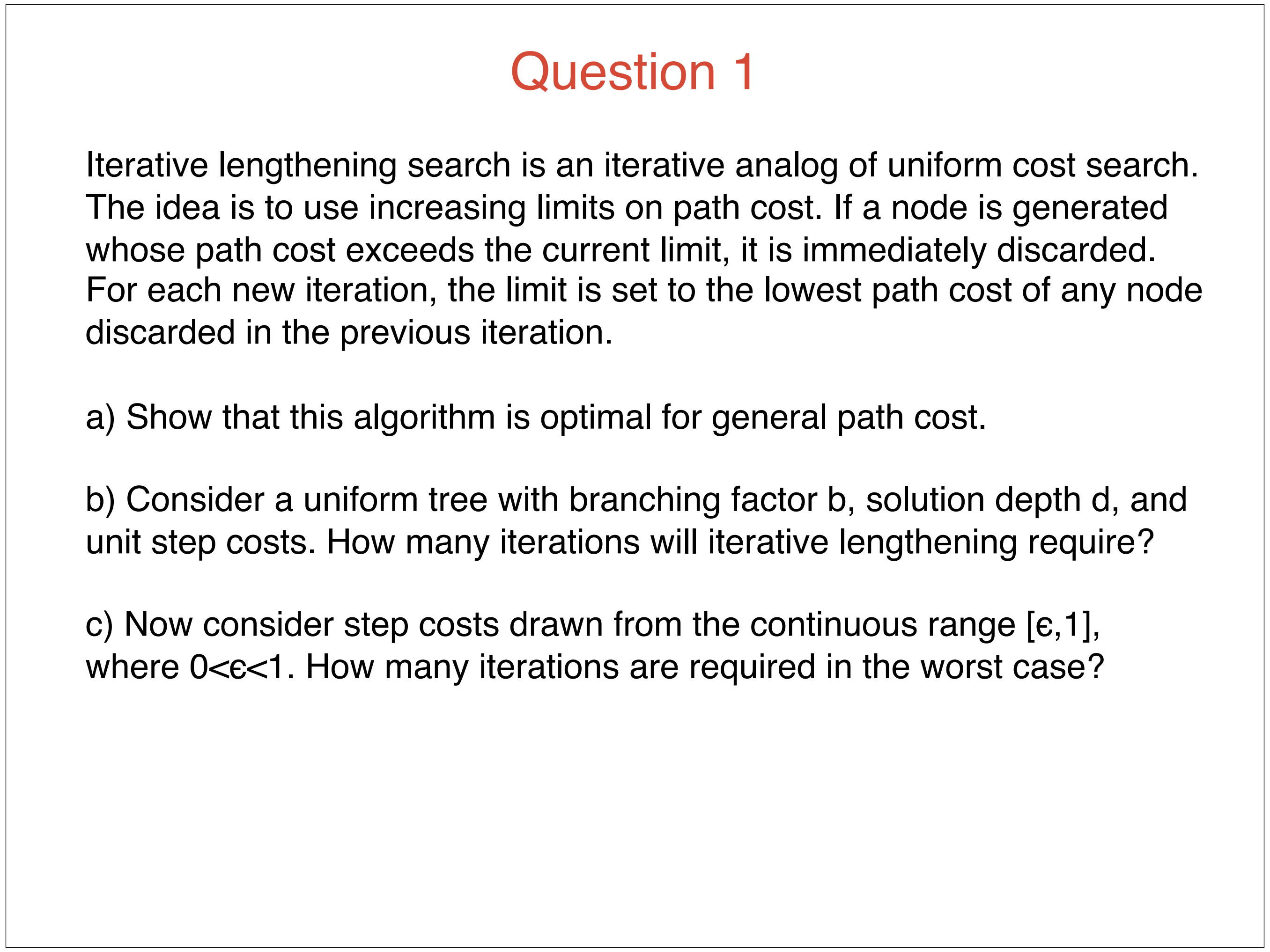


Checking for repeated states

```
def tree_search(problem, max_depth=infinity, check_repeats=False):
    fringe = <collection data structure>
    if check_repeats:
        explored = set()
    fringe.insert(make_node(problem.initial_state))
    while not fringe.is_empty():
        node = fringe.remove_first()
        if node.depth > max_depth: continue
        if check_repeats and node.state in explored:
            continue
        if problem.goal_test(node.state):
            return node
        fringe.insert_all(expand(node, problem))
    return "no solution"
```

- Works for any tree search algorithm. Textbook calls this graph search.
- Why would we want to do this?
    - Linear problem with multiple paths
- What does it do to memory? O(b^d)
- For BFS / UCS, we might as well do it. Would we ever want to do it with DFS?

BFS + repeated state checking.
Would we ever want to do this with DFS?



Question 1

Iterative lengthening search is an iterative analog of uniform cost search. The idea is to use increasing limits on path cost. If a node is generated whose path cost exceeds the current limit, it is immediately discarded. For each new iteration, the limit is set to the lowest path cost of any node discarded in the previous iteration.

a) Show that this algorithm is optimal for general path cost.

b) Consider a uniform tree with branching factor b, solution depth d, and unit step costs. How many iterations will iterative lengthening require?

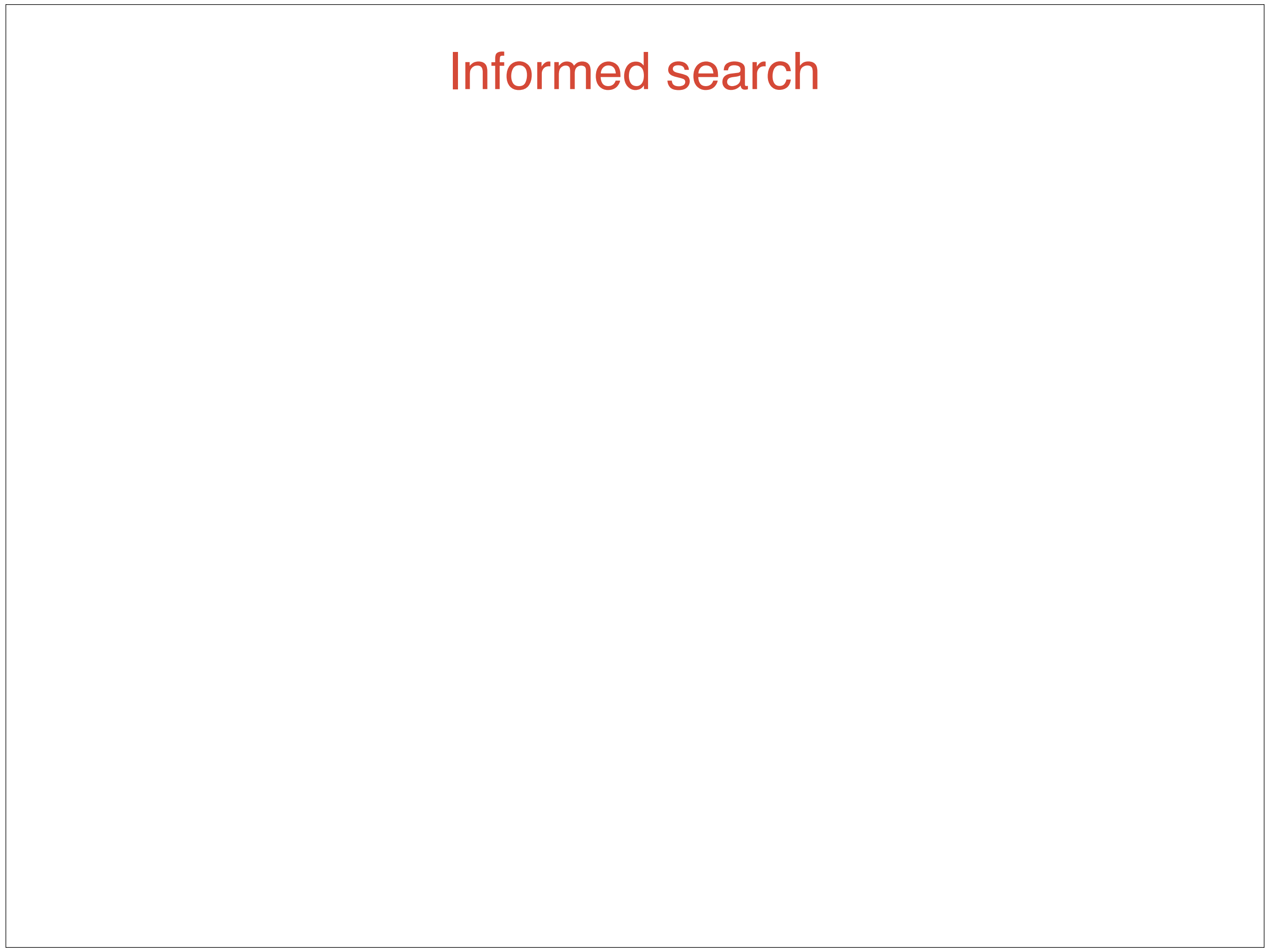
c) Now consider step costs drawn from the continuous range [ϵ,1], where 0<ϵ<1. How many iterations are required in the worst case?

a) Same as IDS. d
b) d/epsilon

Question 2

Describe a state space in which iterative deepening search performs much worse than depth-first search (for example, $O(n^2)$ vs. $O(n)$).

Single successor. 1+2+3+... = O(n^2)

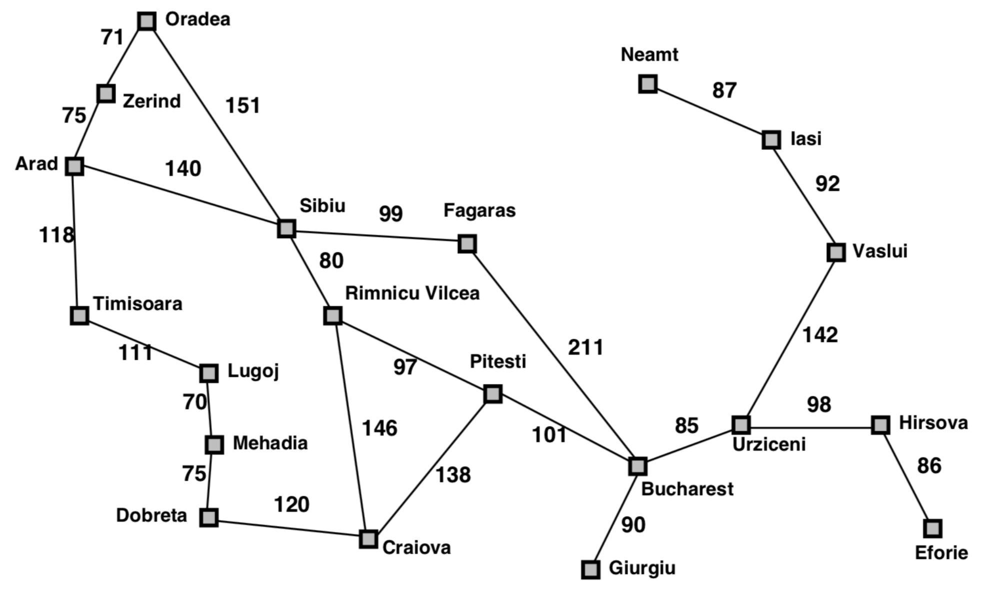


Informed search

- W3D1 Recap:
  - A0 due Friday at 11:59pm.
  - Python tutorials
  - TA office hours: weekly, A0.
  - Four search algorithms. Most practical so far are IDS (for unit step costs) and uniform-cost (with non-unit costs and when optimality is important).

- Previously uniformed: No idea which state is "better".
- Now informed: We'll use a heuristic for much faster search.
- **Heuristic h(n): Estimate cost to goal from n**
  - Examples: map and 8-puzzle.
  - It can be anything (with some restrictions later). The benefit depends on how good of an estimate h(x) is. We will say what it means to be good or bad later, and put restrictions so that it can't be bad.
- **Greedy search:**
  - **fringe: pqueue ordered by h(n)**
  - Example: map starting at Arad. Fast but not optimal.
  - Example: map Iasi->Faragas. Incomplete.
- **A*:**
  - **fringe pqueue ordered by f(n) = h(n) + g(n) = estimated total cost to goal through n**
  - **g(n) = cost of a path so far**

Map search with distance estimates

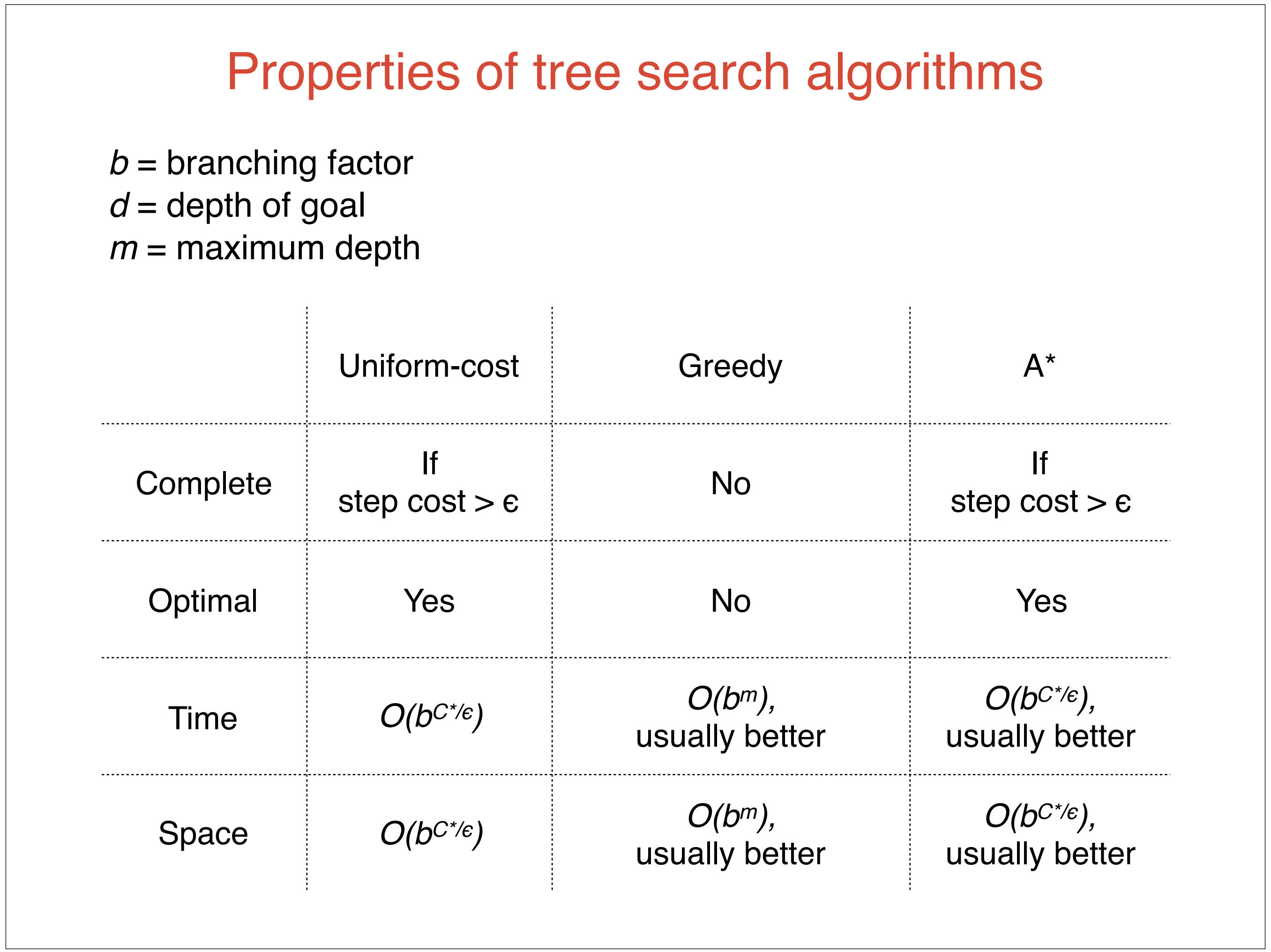


Straight−line distance to Bucharest

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

- Idea: we usually want to go towards the destination, not away from it.
- The straight-line distance (i.e. Euclidean distance) is an estimate of the cost.
- But doesn't it take time to compute straight-line distance? Why is that okay? It takes constant time, independent of the number of actions it would take to get there.

A* example

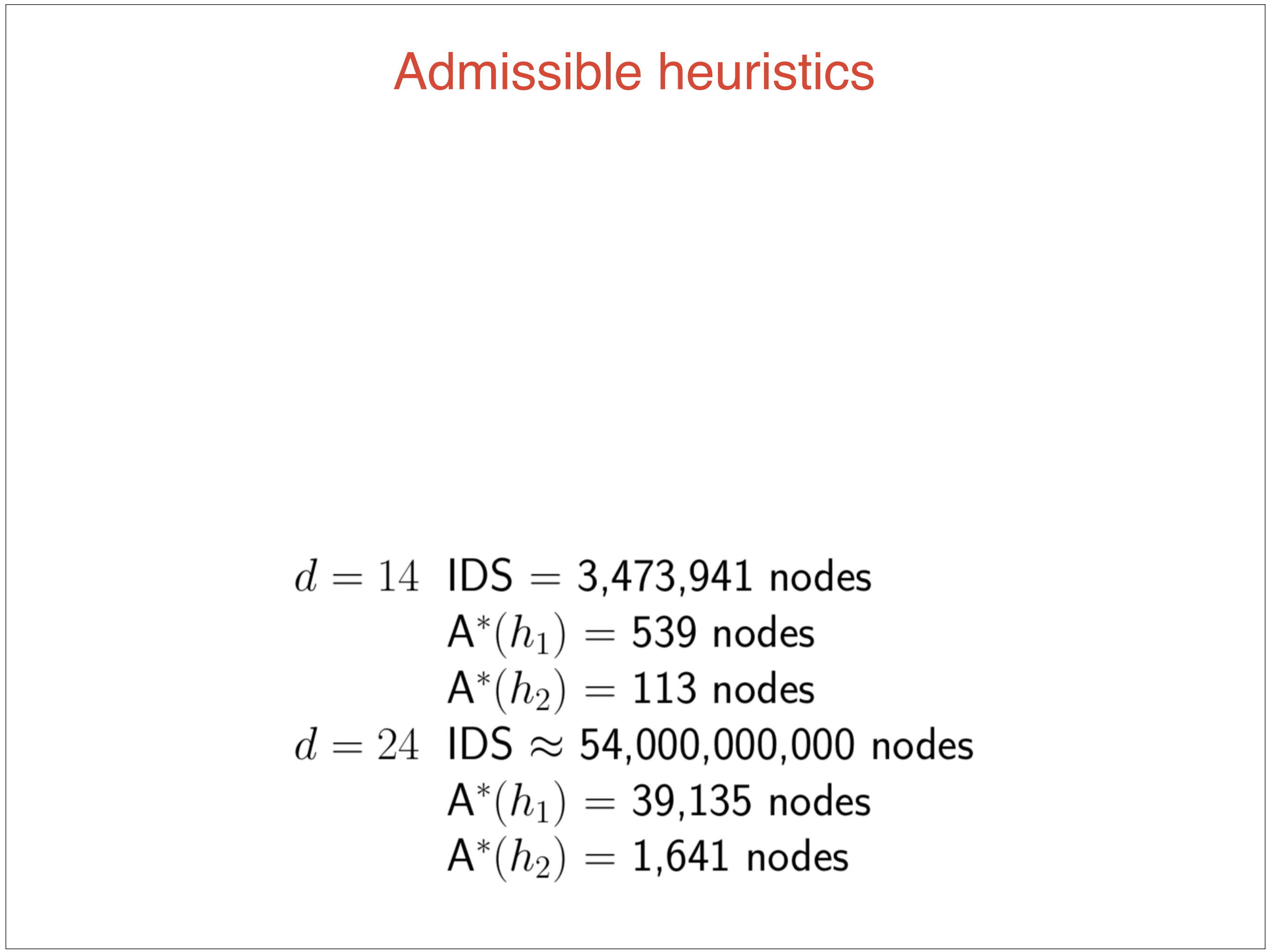


Properties of tree search algorithms

b = branching factor
d = depth of goal
m = maximum depth

	Uniform-cost	Greedy	A*
Complete	If step cost > ϵ	No	If step cost > ϵ
Optimal	Yes	No	Yes
Time	$O(b^{C^*/\epsilon})$	$O(b^m)$, usually better	$O(b^{C^*/\epsilon})$, usually better
Space	$O(b^{C^*/\epsilon})$	$O(b^m)$, usually better	$O(b^{C^*/\epsilon})$, usually better

- Usually better because of the heuristic. Analysis is for the worst case.
- A*: Optimal with some assumptions that we will talk about next.



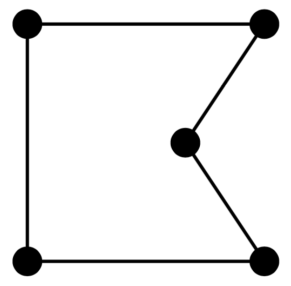
Admissible heuristics

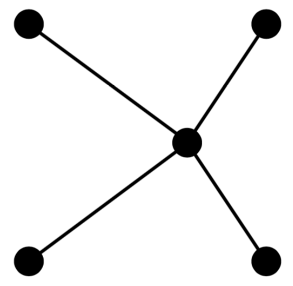
$$\begin{aligned} d = 14 \quad & \text{IDS} = 3{,}473{,}941 \text{ nodes} \\ & \text{A}^*(h_1) = 539 \text{ nodes} \\ & \text{A}^*(h_2) = 113 \text{ nodes} \\ d = 24 \quad & \text{IDS} \approx 54{,}000{,}000{,}000 \text{ nodes} \\ & \text{A}^*(h_1) = 39{,}135 \text{ nodes} \\ & \text{A}^*(h_2) = 1{,}641 \text{ nodes} \end{aligned}$$

- How do we choose a heuristic?
    - "Admissiblility" is a property that we want
    - **h*(n) = true cost to goal from n**
    - **h(n) is admissible iff: h(n) <= h*(n)**
    - **Also h(n) >= 0**. H(G) = 0
- That is, h(n) is optimistic: it is an underestimate, never an overestimate.
    - Intuition example: two paths; cheaper path has overestimated edge. We wouldn't explore that path.
- What is an admissible heuristic for map? Euclidean distance
- For 8-puzzle?
    - **h1(n) = Number of misplaced tiles.**
    - **h2(n) = Manhatten distance of each tile from its target.**
- Dumb admissible heuristic for 8-puzzle: h(n) = 0. h(n) = 1 if it's not the goal.
- Which is better? Intuitively h2
    - **h_a <u>dominates</u> h_b iff h_a(n) >= h_b(n)**. (And both are admissible).
    - The dominant heuristic is the better one.
- Running time example.

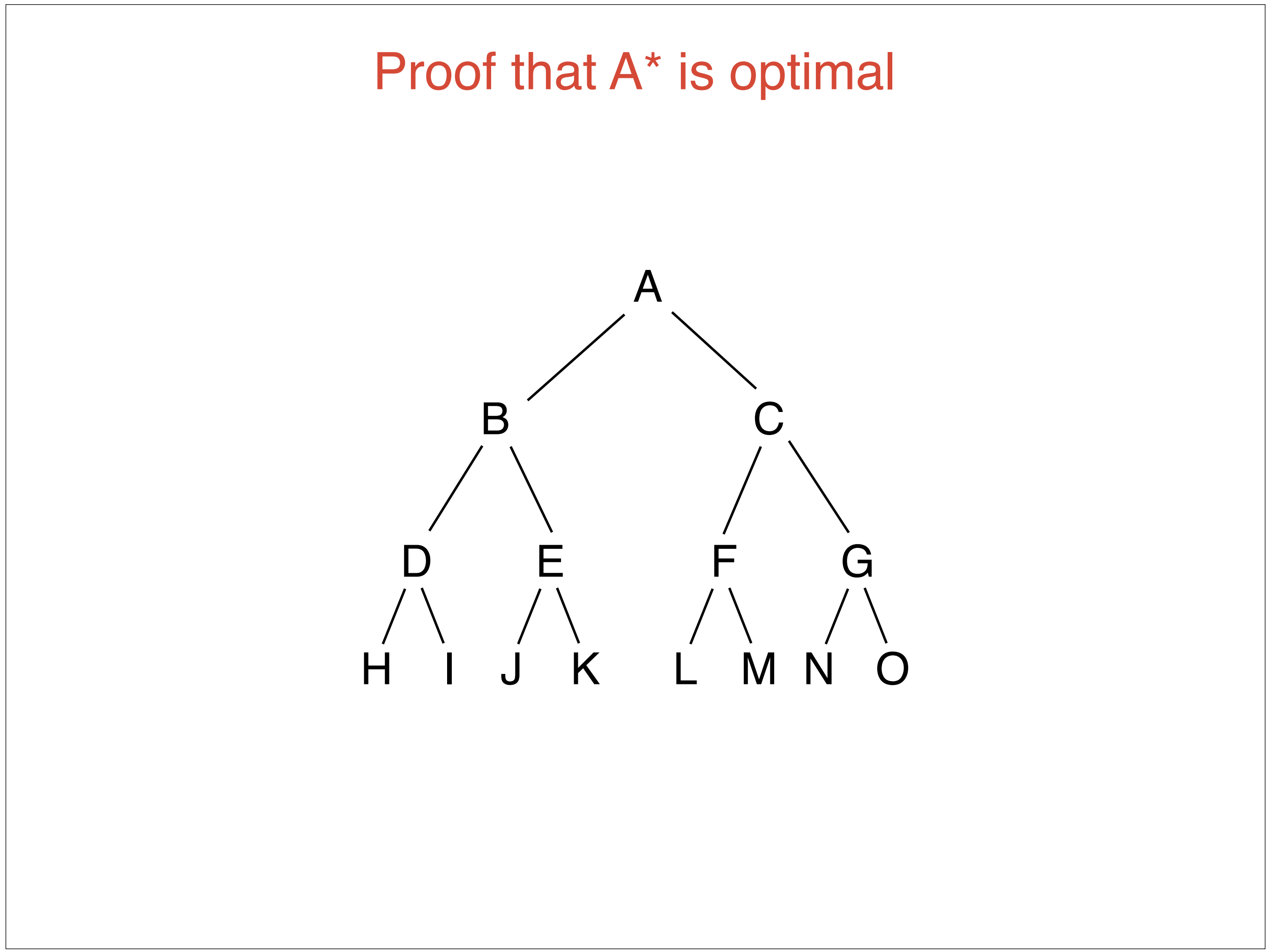
A way to create an admissible heuristic: relax the problem

Traveling salesperson problem (TSP)


Tour

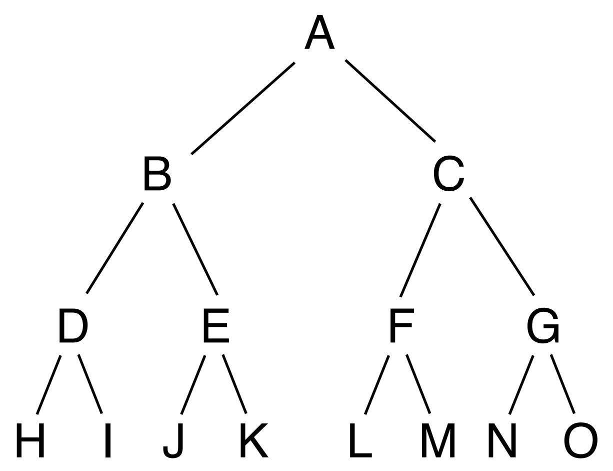

Minimum spanning tree

- h(n) = exact cost to goal in a relaxed problem
- Relaxed problem: one that is always has less cost than the real problem.
    - Hopefully one that is easy to solve (otherwise we can't compute h(x)).
    - 8-puzzle relaxation: You can teleport a tile.
- TSP (well-known hard problem):
    - MST can be computed in O(n^2)
    - MST is the shortest path if you're allowed to jump back to cities you've visited before.

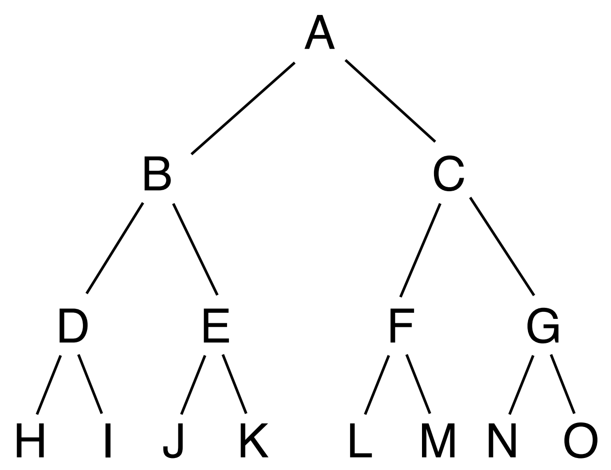


Proof that A* is optimal

- Warm up: BFS with unit step costs
    - We always expand nodes in increasing order of steps.
    - Note: If there are non-uniform costs, steps != cost.
- Uniform cost search:
    - Same thing: increasing order of cost.
    - Reminder: we do our check to consider returning a goal when we expand a node, not when it is created.
- A*
    - Increasing order of f().
    - Because f() is an underestimate, we will never expand a suboptimal G2
    - If f() were an overestimate, we might wait to long to expand a node.
    - There is a more rigorous proof in the book.


A
B
C
D
E
F
G
H
I
J
K
L
M
N
O

Proof that A* is optimal



Question 3

True or false:

1. Depth-first search always expands at least as many nodes as A* search with an admissible heuristic.
2. h(n) = 0 is an admissible heuristic for the 8-puzzle.
3. A* is of no use in robotics because percepts, states and actions are continuous.
4. BFS is complete even if zero step costs are allowed.
5. In chess, a rook can move any number of squares vertically or horizontally, but cannot jump over other pieces. Manhattan distance is an admissible heuristic for the problem of moving the rock from square A to square B in the smallest number of moves.

1. False. DFS can get lucky.
2. True. h(n) = 0 is always admissible.
3. False. A* is often used in robotics by discretizing the space.
4. True. Depth of the solution matters for breadth-first search, not cost.
5. False: a rook can move across the board in move one, although the Manhattan distance from start to finish is 8.

Question 4

The heuristic path algorithm is a search algorithm in which the evaluation function is $f(n) = (2 - w)g(n) + wh(n)$. For what values of w is this complete? For what values is it optimal, assuming that h is admissible? What kind of search does this perform for $w = 0$, $w = 1$, and $w = 2$?

w = 0 gives uniform-cost search. The factor of two doesn't make a difference in ordering nodes.
w = 1 gives A* search.
w = 2 gives greedy search.
It is complete when $0 <= w < 2$.
It is optimal when $w <= 1$.