DATA130008 Introduction to Artificial Intelligence

复旦大学大数据学院
School of Data Science, Fudan University

魏忠钰

Search I

September 14th, 2021

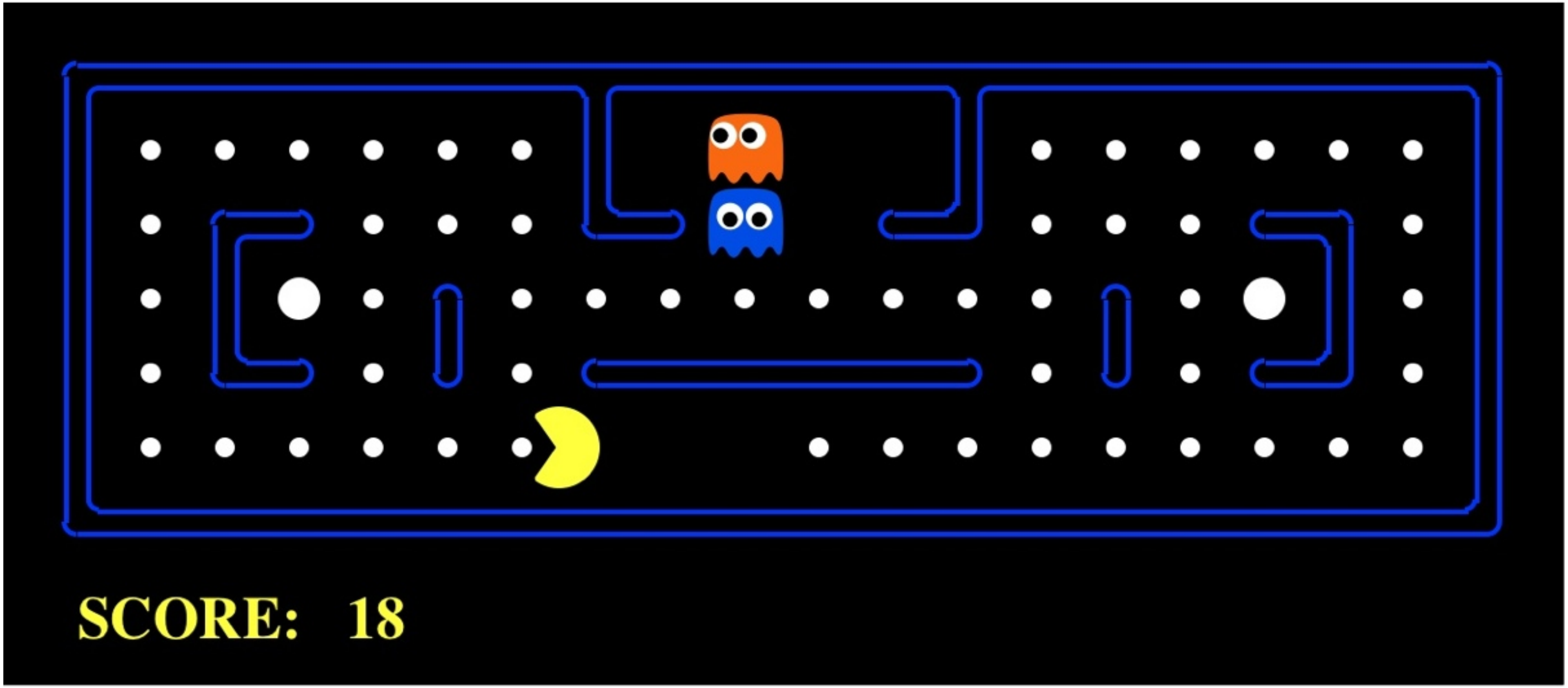
Outline

- **Search Problems**

Search Problems

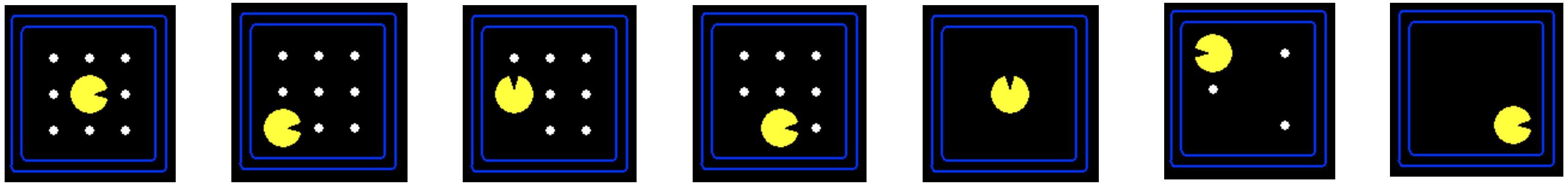
- **A search problem consists of:**
  - **A state space**
    - State describes the status of a piece of situation
  - **A successor/result function (with actions, costs)**
    - Actions are choices an agent could take in each state
  - **A start state**
    - Where the game starts
  - **A goal test**
    - When the game ends

Real world task - Pac-man

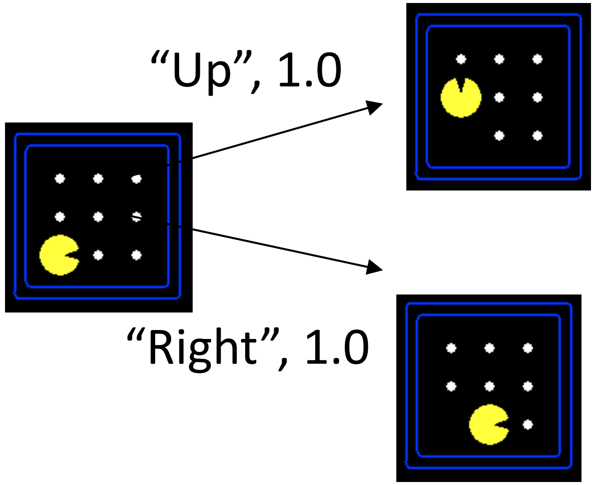


Search problem formulation - Pac-man

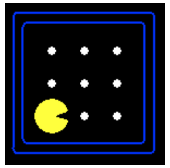
- A search problem consists of:
  - A state space



  - A successor/result function (with actions, costs)
    - Actions: Up, Down, Left, Right
    - Cost: 1 for each step



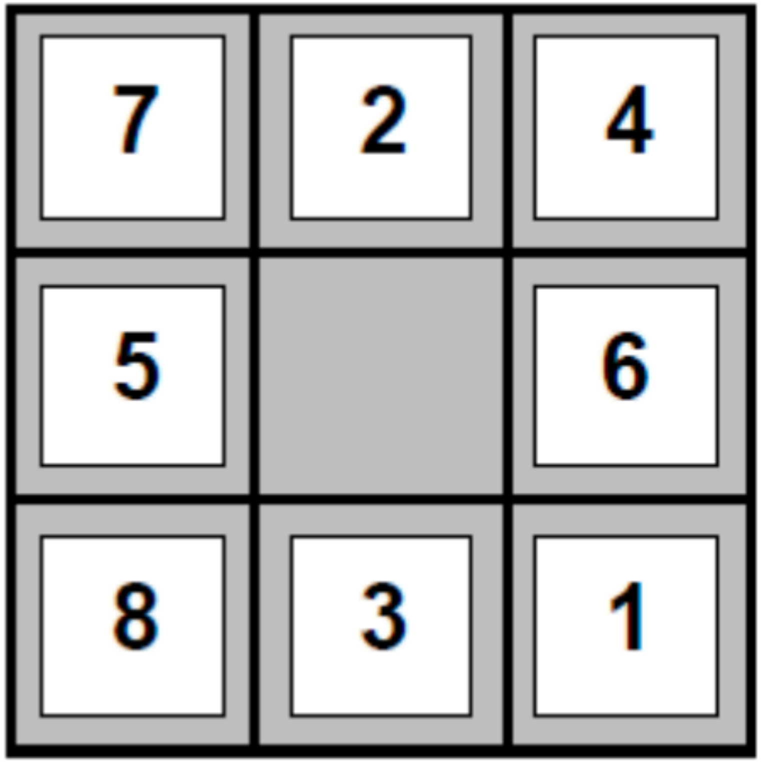
  - A start state



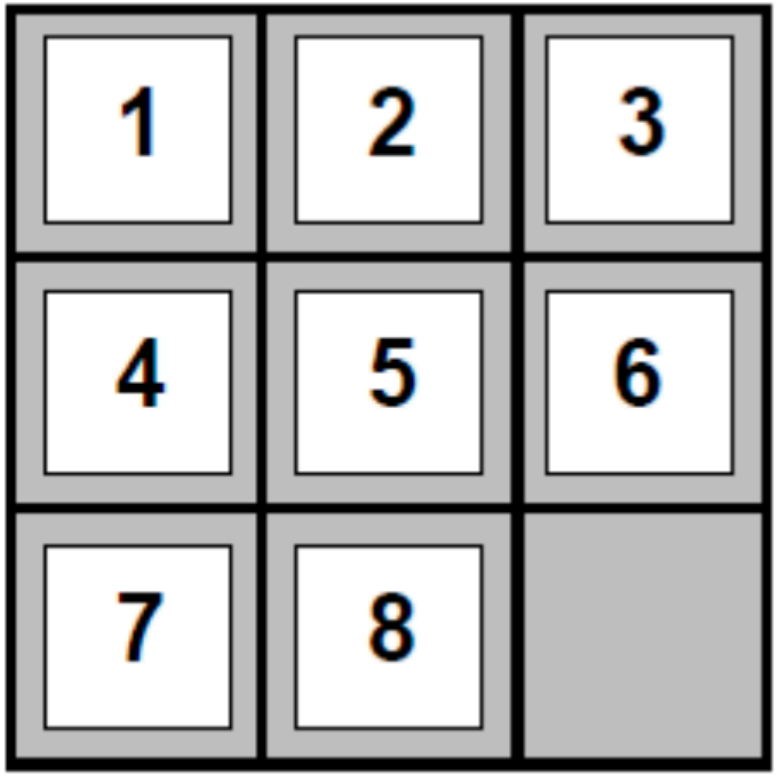
  - A goal test
    - Move to a specific position

Real world task : the 8-puzzle

You can move each tile to the neighbor position if it is blank

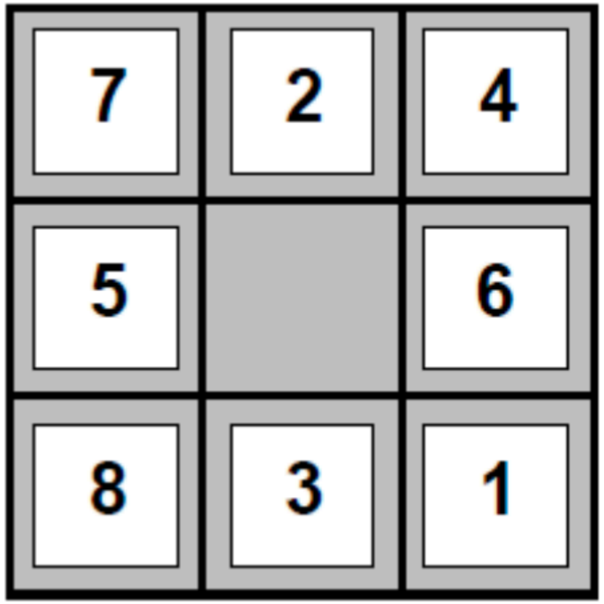


Start State

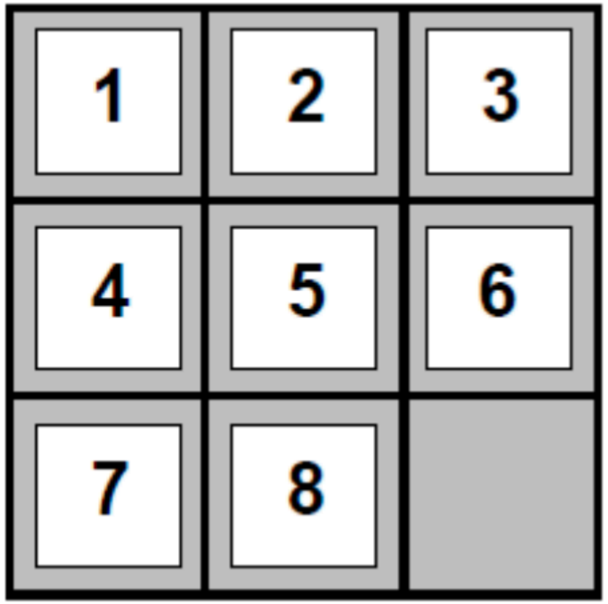


Goal State

Search problem formulation : the 8-puzzle



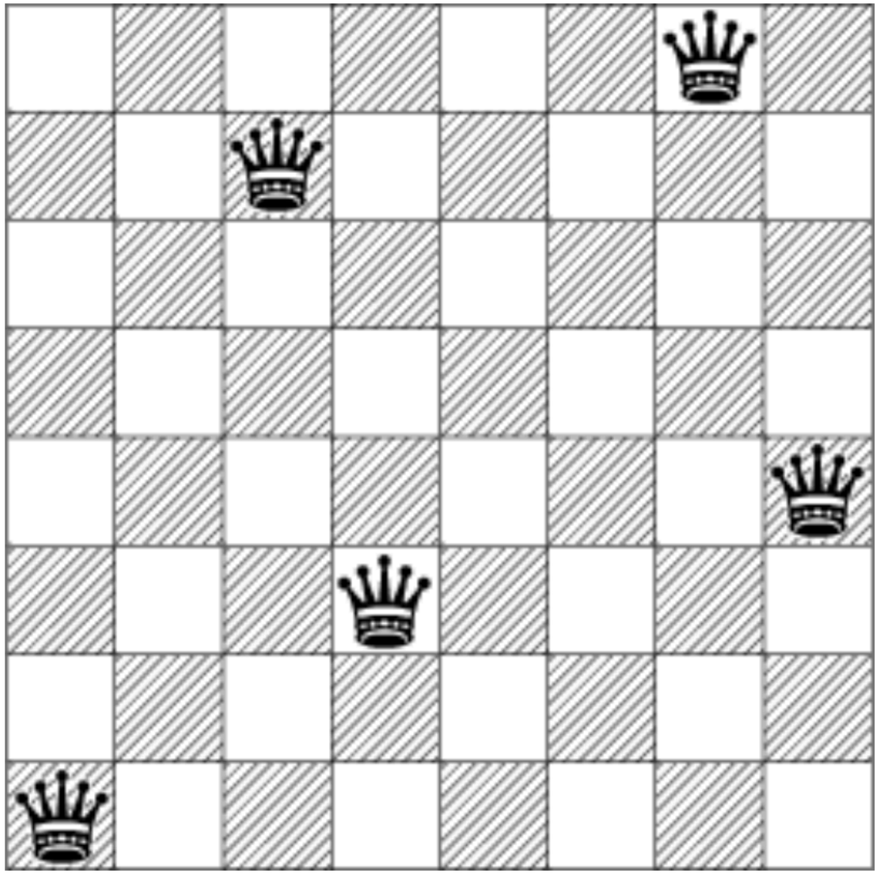
Start State



Goal State

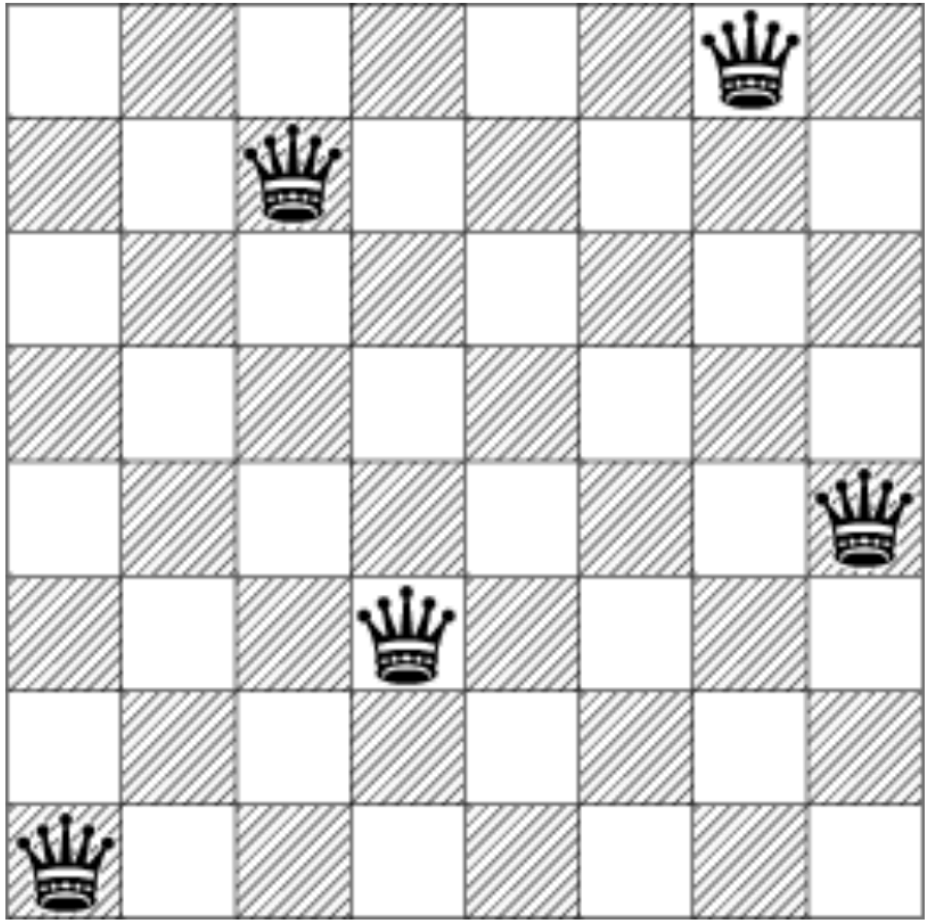
- State space
  - Locations of 8 tiles. Each element contains a (x,y).
- Successor function:
  - Move blank (up, down, right, left)

Real Task: 8-Queens Puzzle



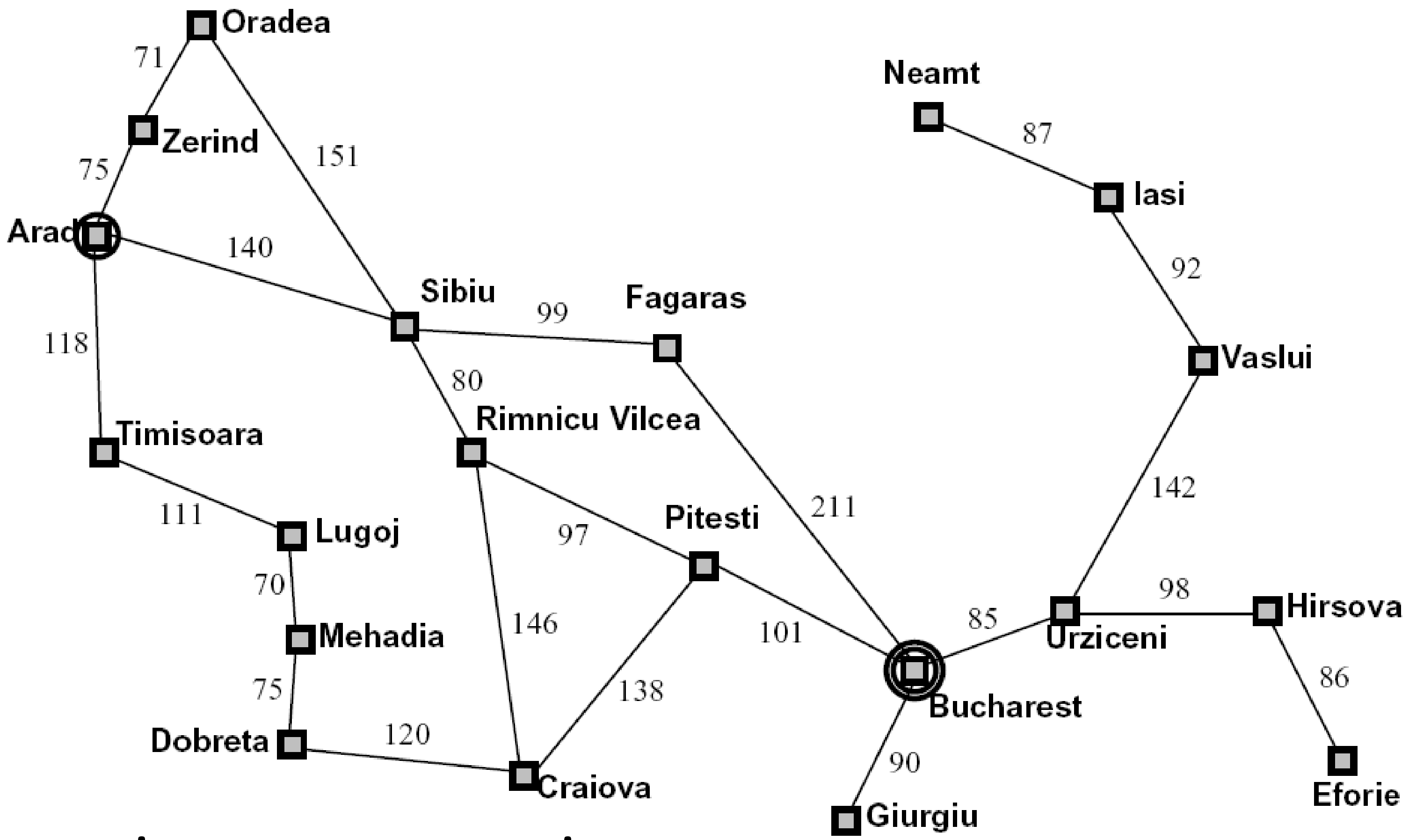
- Place 8 queens on a chessboard and ensure no two queens attack each other.
- A queen attacks any piece in the same row, column or diagonal.
- In this case, 3 more queens missing

Search problem formulation : 8-Queens Puzzle



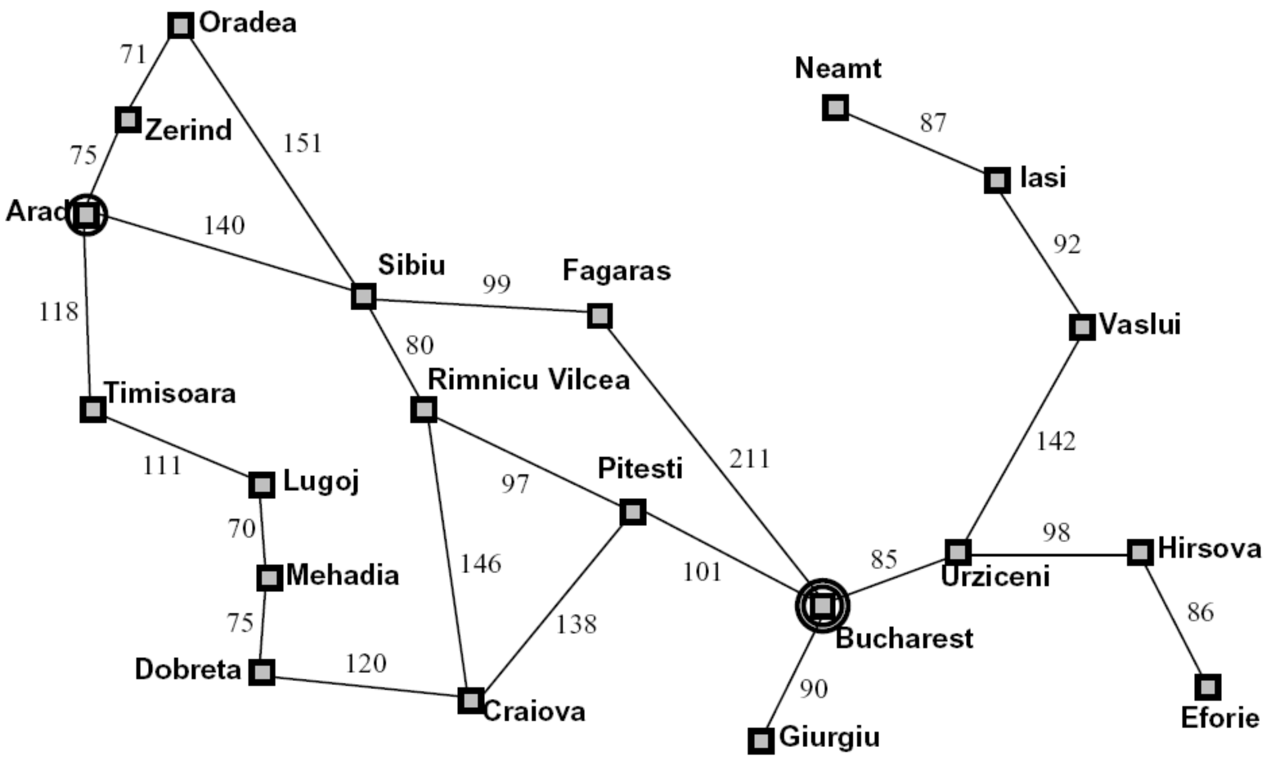
- **State space**
  - any arrangement of 0 to 8 queens on board. A vector a 8X8 elements, 0/1.
- **Successor function**
  - add a queen to any square
- **Start state**
  - blank board
- **Goal test**
  - 8 queens on board, non attacked

Real world task : Traveling in Romania



- Travel from one city to a target city.
- Each node stands for a city. Numbers on edges are distances.
- Identify a shortest path from origin city to target city.
- In this case from Arad to Bucharest.

Search problem formulation : Traveling in Romania



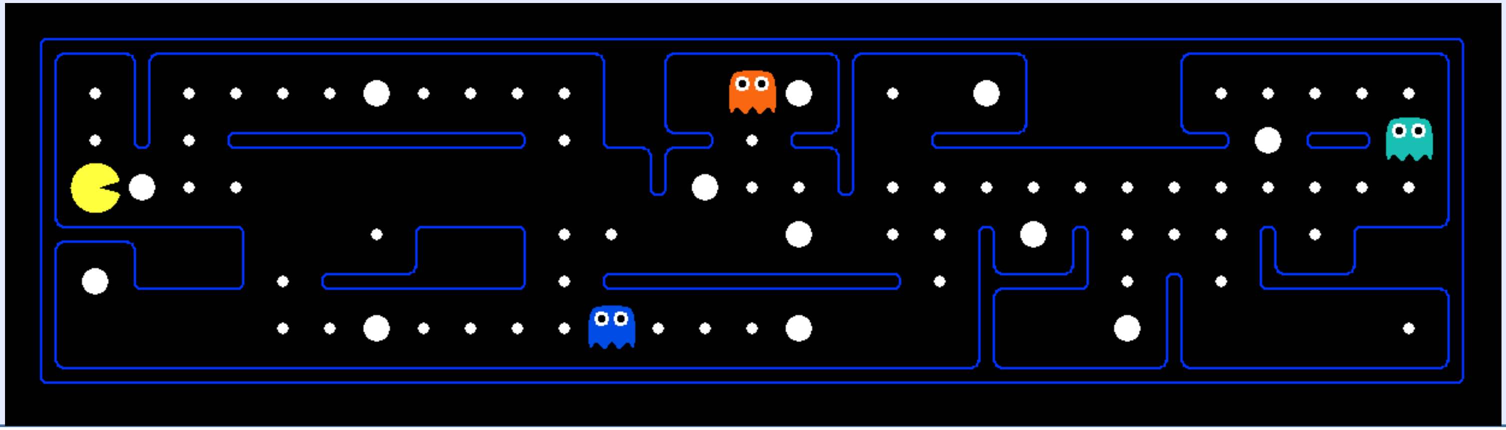
- State space
  - Cities
- Successor function:
  - Action: Go to adjacent city
  - Cost： distance
- Start state:
  - Arad
- Goal test:
  - Is state == Bucharest?

Outline

- Search Problems
- State Spaces

What is in a State Space?

The world state includes every last detail of the environment



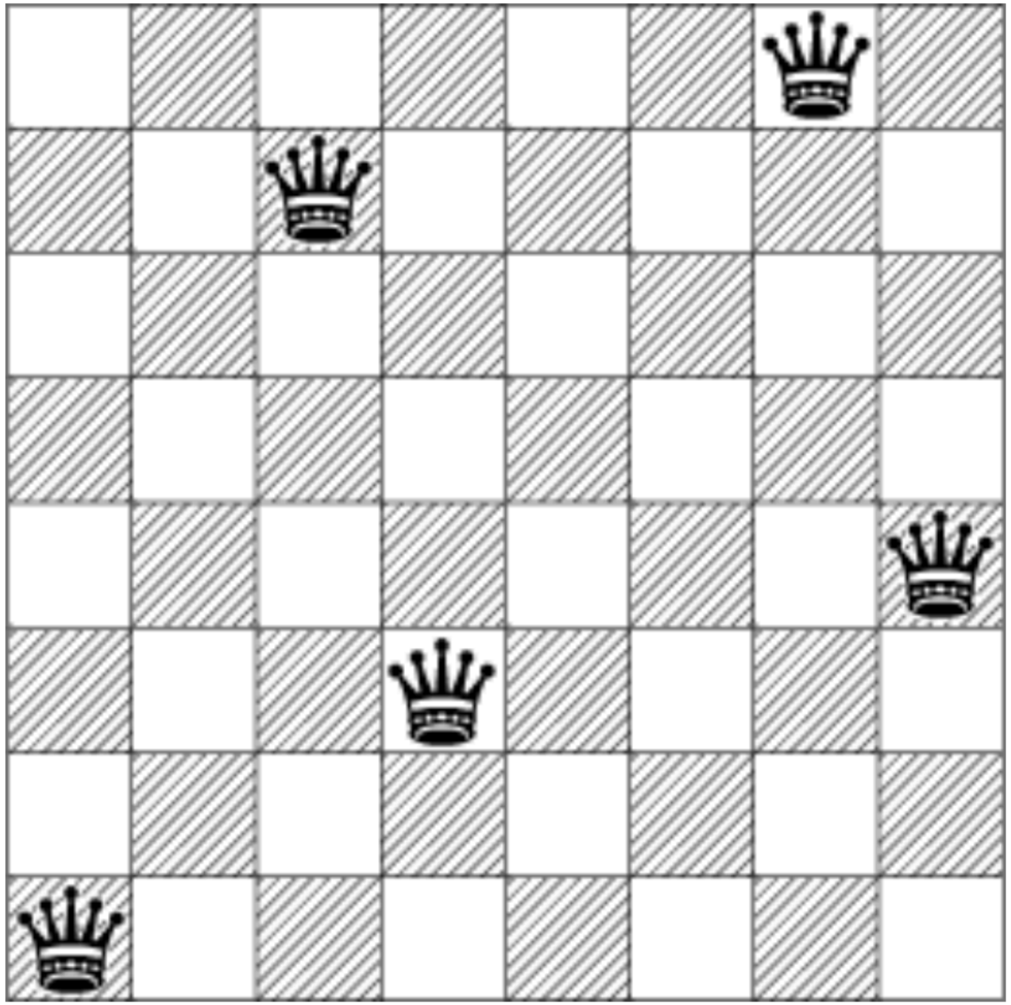
A search state keeps only the details needed for solving a specific problem

- Problem: Path-Finding
  - States: (x,y) *location*
  - Actions: NSEW
  - Successor: update location following an action
  - Goal test: is (x,y)=END
- Problem: Eat-All-Dots
  - States: { (x,y), dot map}
  - Actions: NSEW
  - Successor: update location and dot distribution
  - Goal test: dots all false

State Space Sizes?

- World state:
  - Board blanks: 64
  - Queen number: 8

- How many
  - World states?
  - 64^8



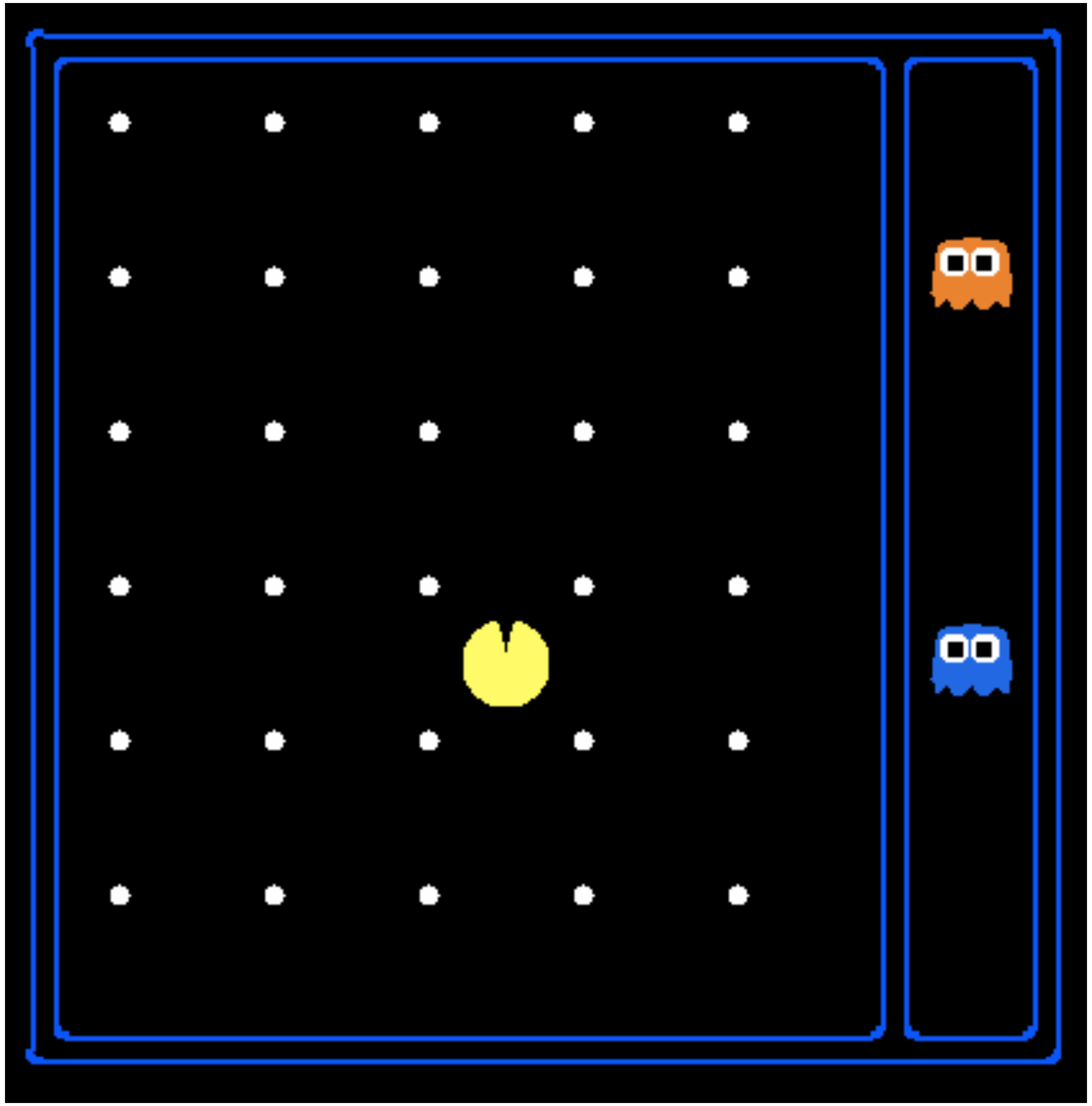
State Space Sizes?

- World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW
- How many
  - World states?

    $120 \times (2^{30}) \times (12^{2}) \times 4$
  - States for path-finding?

    120
  - States for eat-all-dots?

    $120 \times (2^{30})$

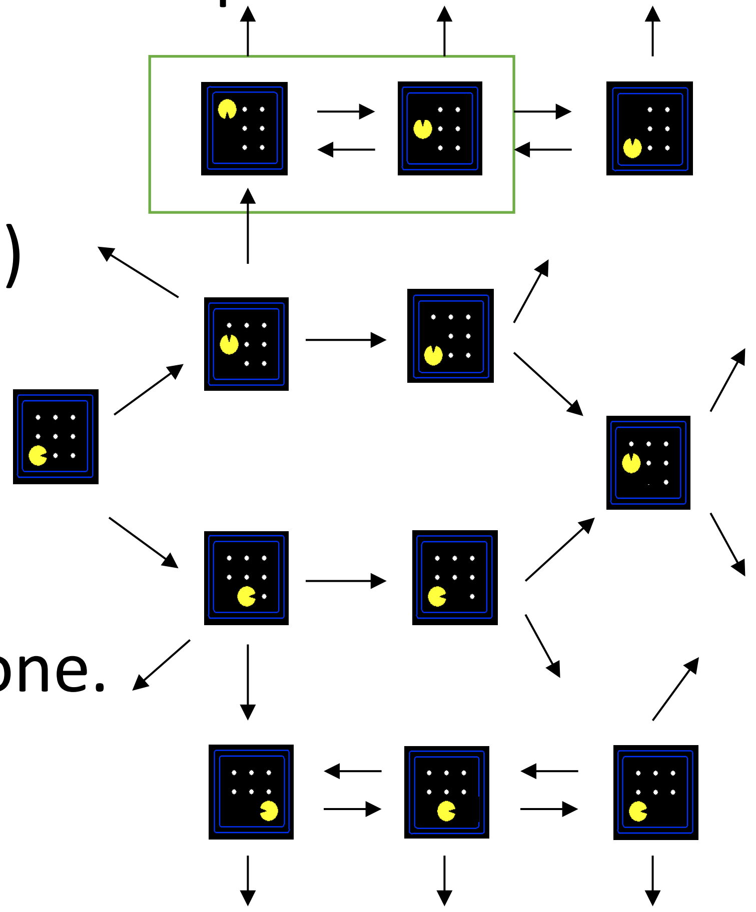


Outline

- **Search Problems**
- **State Spaces**
- **State Space Representation**

State Space Graph

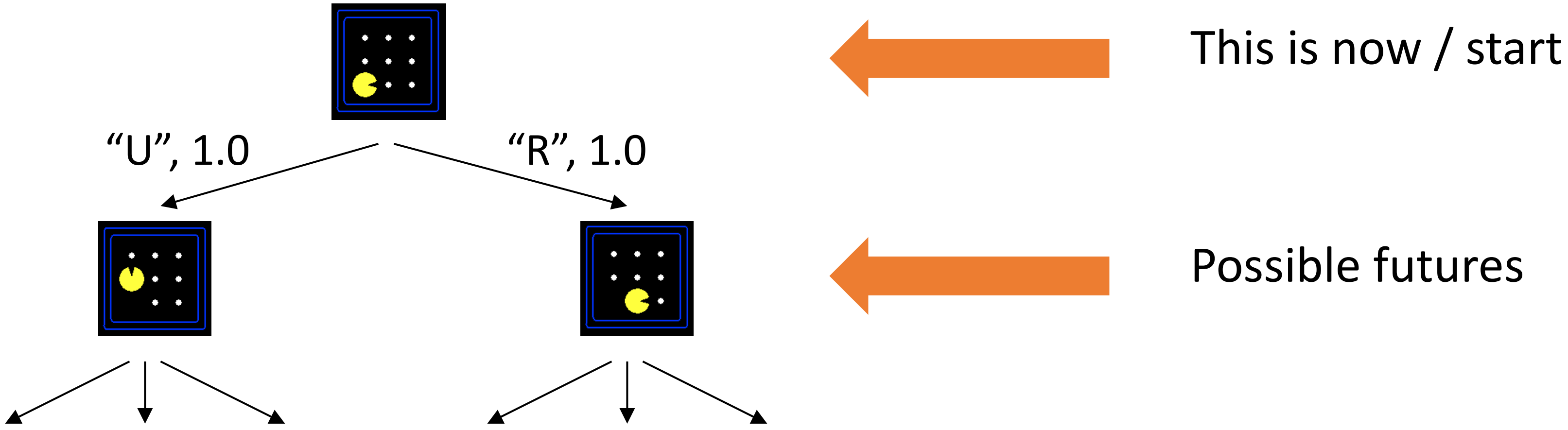
- State space graph: A mathematical representation of a search problem
  - **Nodes** are (abstracted) **states**
  - **Arcs** represent **successors** (action results)
  - The goal test is **a set of goal nodes** (maybe only one)



- In a state space graph, each state occurs only once.
- Nodes corresponds to states in the state space one to one.

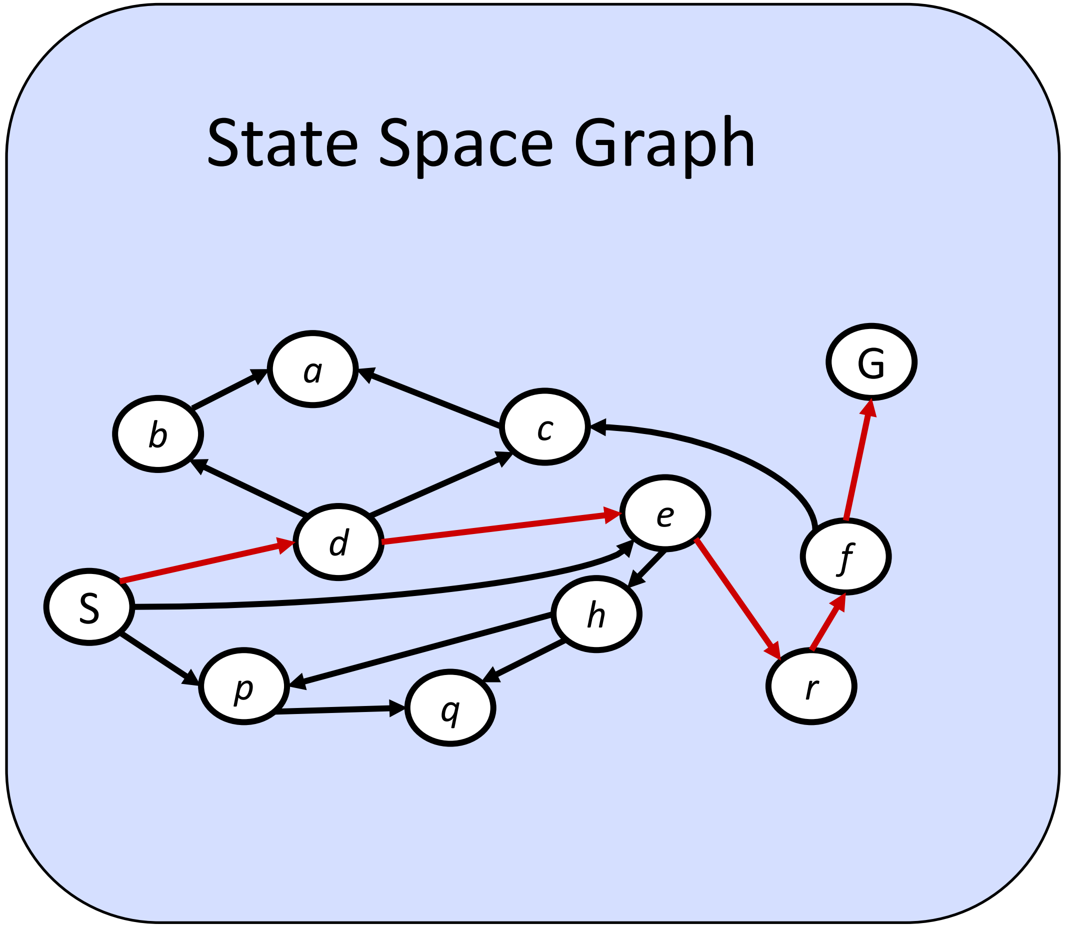
Search Trees

- The start state is the root node
- Children correspond to successors
- Nodes show states, but correspond to PLANS that achieve those states



- PLAN means a series of actions.

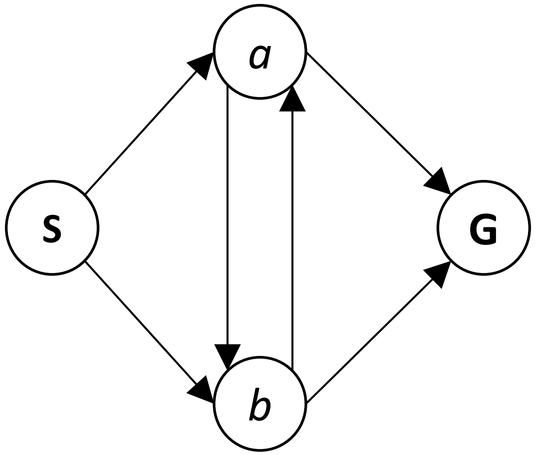
Transforms this State Space Graph to a Search Tree



- *Each NODE in the search tree is an entire PATH in the state space graph.*
- *Each state can can be corresponded to multiple nodes in the tree.*

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!

For most problems, we can never actually build the whole tree.

Outline

- Search Problems
- State Spaces
- State Space Representation
- Solve a Search Problem

Solving a Search Problem

- A search problem consists of:
  - A state space
  - A successor/result function (with actions, costs)
  - A start state
  - A goal test
- Problem Representation
  - State Space Graph
  - State Search Tree
- **Solving a search problem is to identify a sequence of actions from start state to goal state.**
- **Also known as planning**

Standard Search Setup

- Process
  - Start at the initial state from the search tree
  - Expand an un-visited but observed node
  - Generate neighbors of current state (observe new nodes)
  - Test whether the state matches the goal configuration

- Implementation Components
  - Fringe: store those nodes expanded but not visited
    - Expand： numerate neighbors of a node
    - Generate: add a node into fringe
  - Search strategy: choose a node from fringe to visit

Pseudo-Code for Tree Search Paradigm

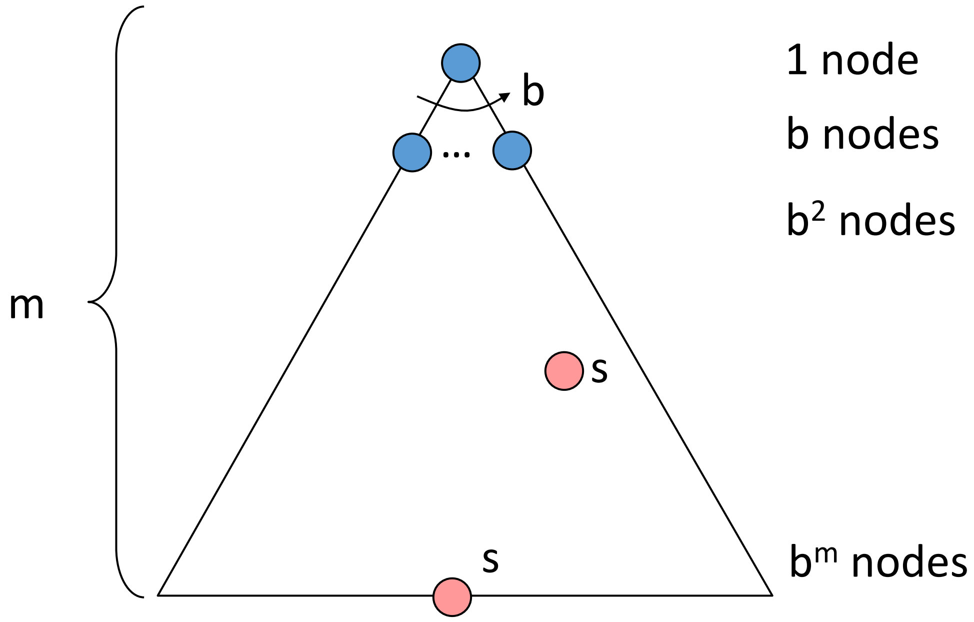
```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

- Fringe: store those nodes expanded but not visited
  - Expand： EXPAND
  - Generate: INSERT
- Search strategy: REMOVE-FRONT

Goal test is performed when the node is removed from the fringe.

Evaluation of Search Algorithm

- Some symbolic of search tree:
  - b is the branching factor
  - m is the maximum depth
  - s are the solutions at various depths



- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots . bm = \frac{1-b^{m+1}}{1-b} = O(b^m)$
- **Complete**: guaranteed to find a solution if one exists?
- **Optimal**: guaranteed to find the least cost path?
- **Time complexity**: # of nodes generated
- **Space complexity**: # of nodes stored (size of fringe)

Outline

- Search Problems
- State Spaces
- State Space Representation
- Solve a Search Problem
- Uninformed Search Algorithms
  - Depth-First Search
  - Breadth-First Search
  - **Iterative Deepening**
  - Uniform-Cost Search

Depth-First Search

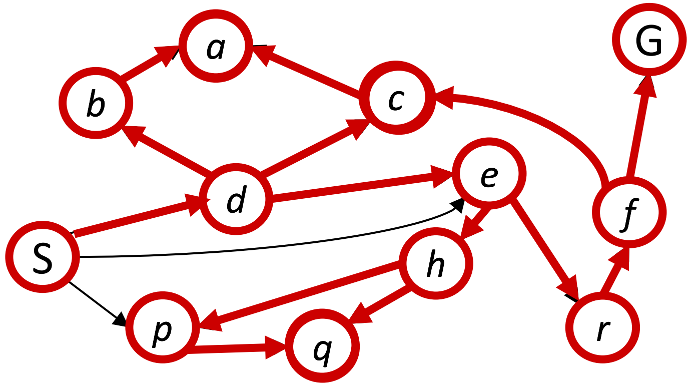
Fringe: *A data structure to store nodes observed but not visited*

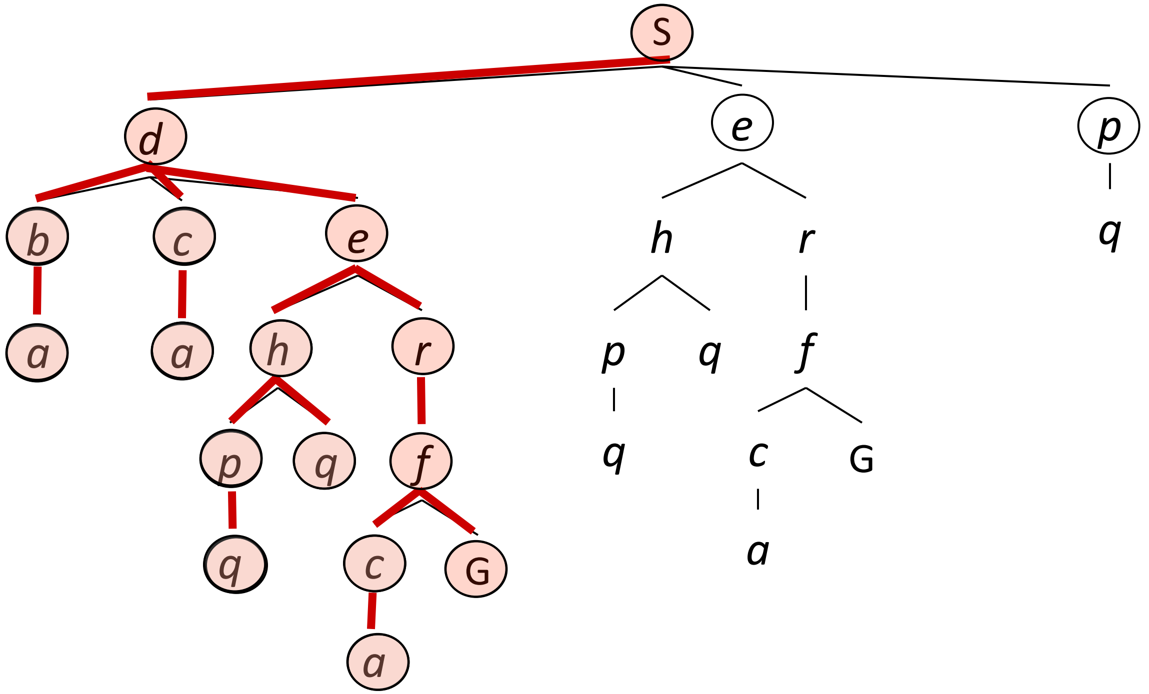
Expand*: pop a node from the fringe*

Generate*: get neighbors of the expanded node and insert them into the fringe*

Strategy*: expand a deepest node first (number of steps to the root)*

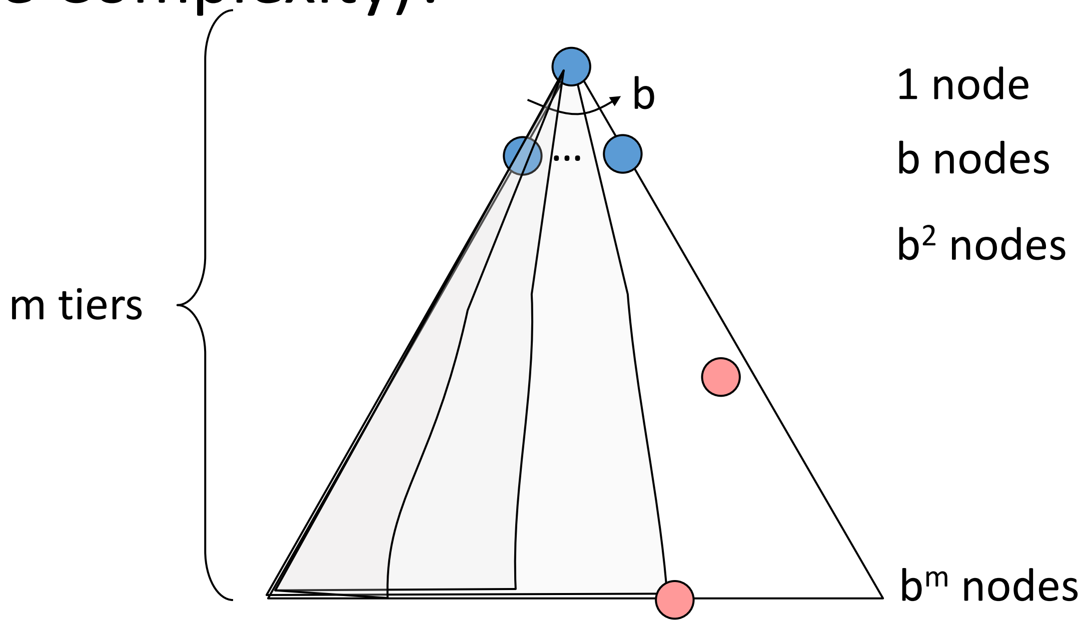
Implementation*: Fringe is a LIFO stack*





Depth-First Search (DFS) Properties

- How many nodes does DFS generate (Time Complexity)?
  - If m is finite, takes time $O(b^m)$
- How much space does the fringe take (Space Complexity)?
  - b nodes in each layer, O(bm)
- Is it complete?
  - No, m can be infinite
- Is it optimal?
  - No, it finds the “leftmost” solution, regardless of depth or cost



Breadth-First Search (BFS)

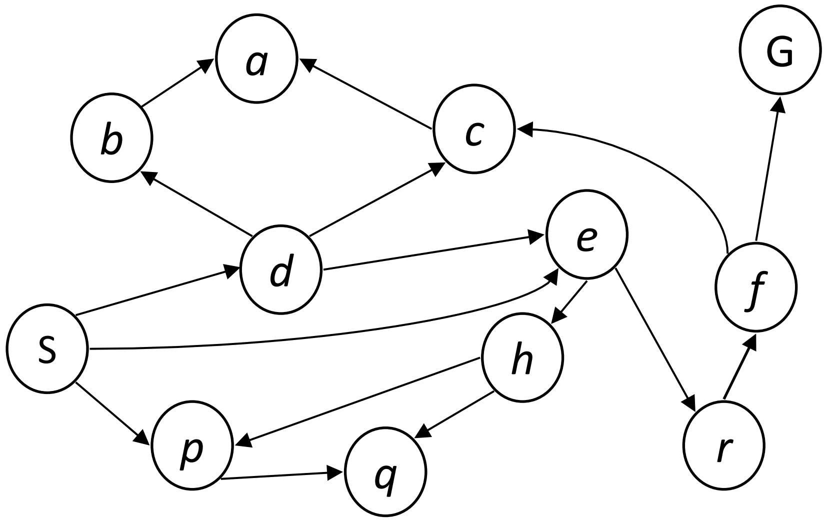
Fringe: *A data structure to store nodes observed but not visited*

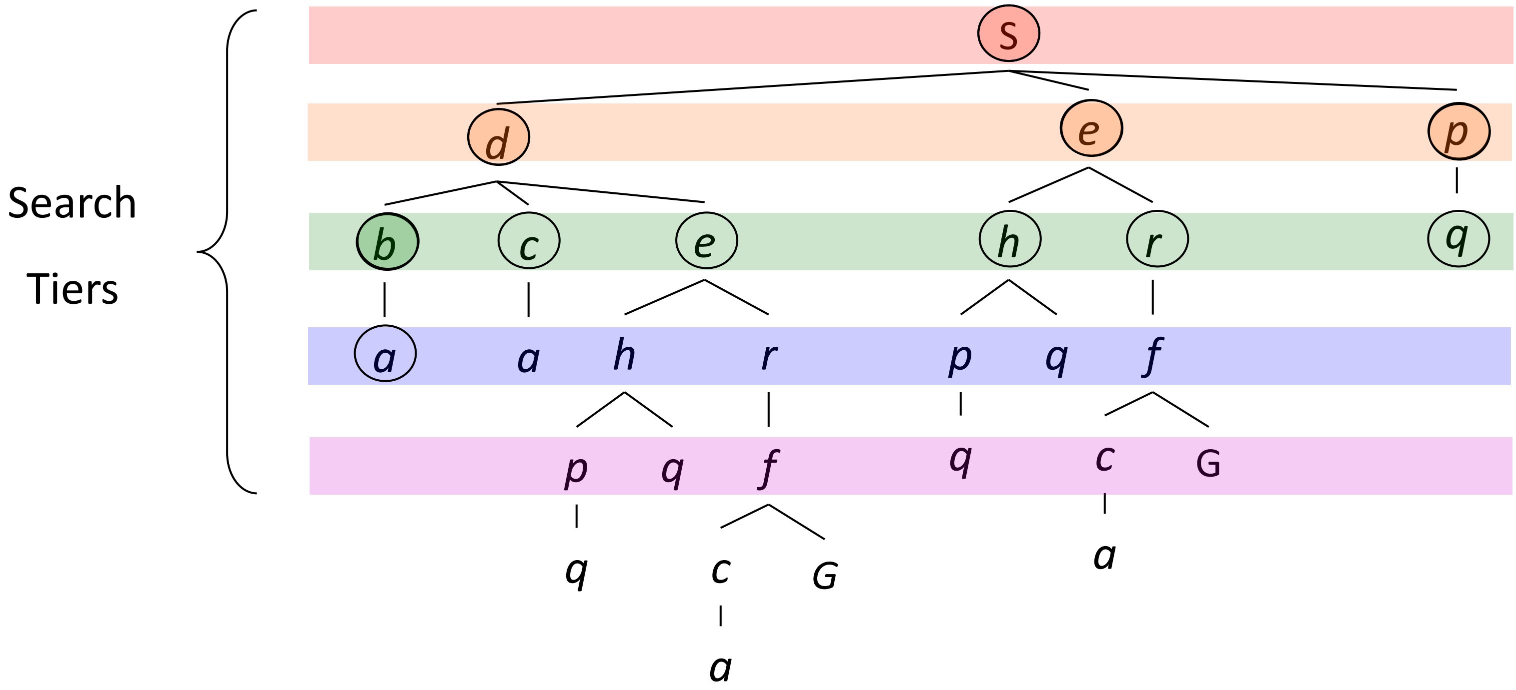
Expand*: pop a node from the fringe*

Generate*: get neighbors of the expanded node and insert them into the fringe*

Strategy*: expand a shallowest node first (number of steps to the root)*

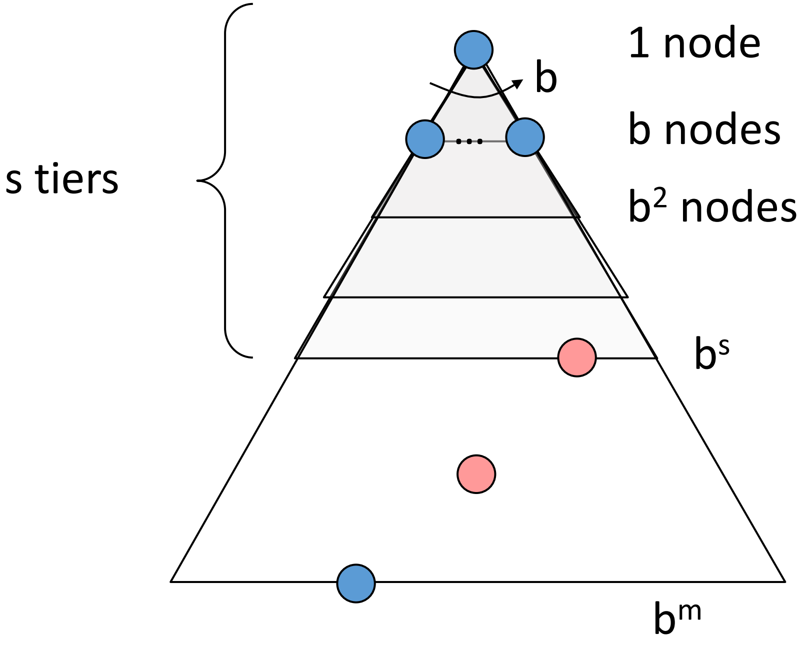
Implementation*: Fringe is a FIFO queue*





Breadth-First Search (BFS) Properties

- How many nodes does BFS expand (Time Complexity)?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^{s+1})$. Why s+ 1?
- How much space does the fringe take?
  - Has roughly the last tier, so $O(b^{s+1})$
- Is it complete?
  - yes
- Is it optimal?
  - Yes (if the cost is equal per step)



DFS vs BFS

- DFS requires smaller storage

- BFS is robust (complete and optimal) but needs more space for storage

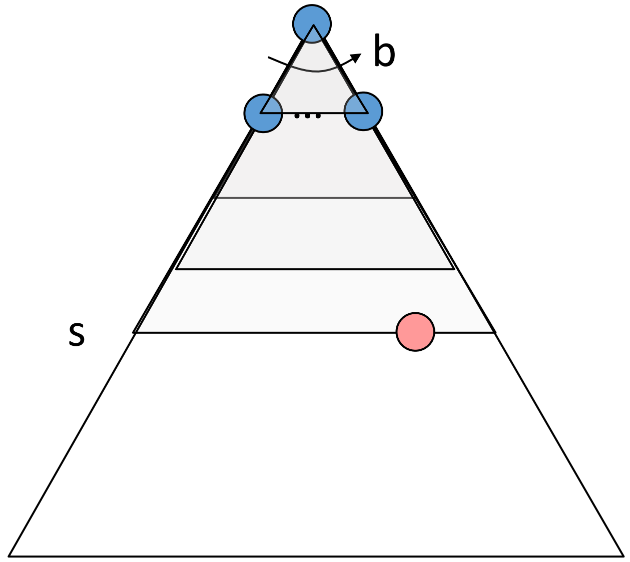
How bad is BFS?

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.

Iterative Deepening

- 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. …..
- How many nodes does BFS generate?
  - $O(b^{s+1})$
- How much space does the fringe take?
  - $O(bs)$
- Although the time complexity of IDS is in the same order as BFS, it generates many more nodes.

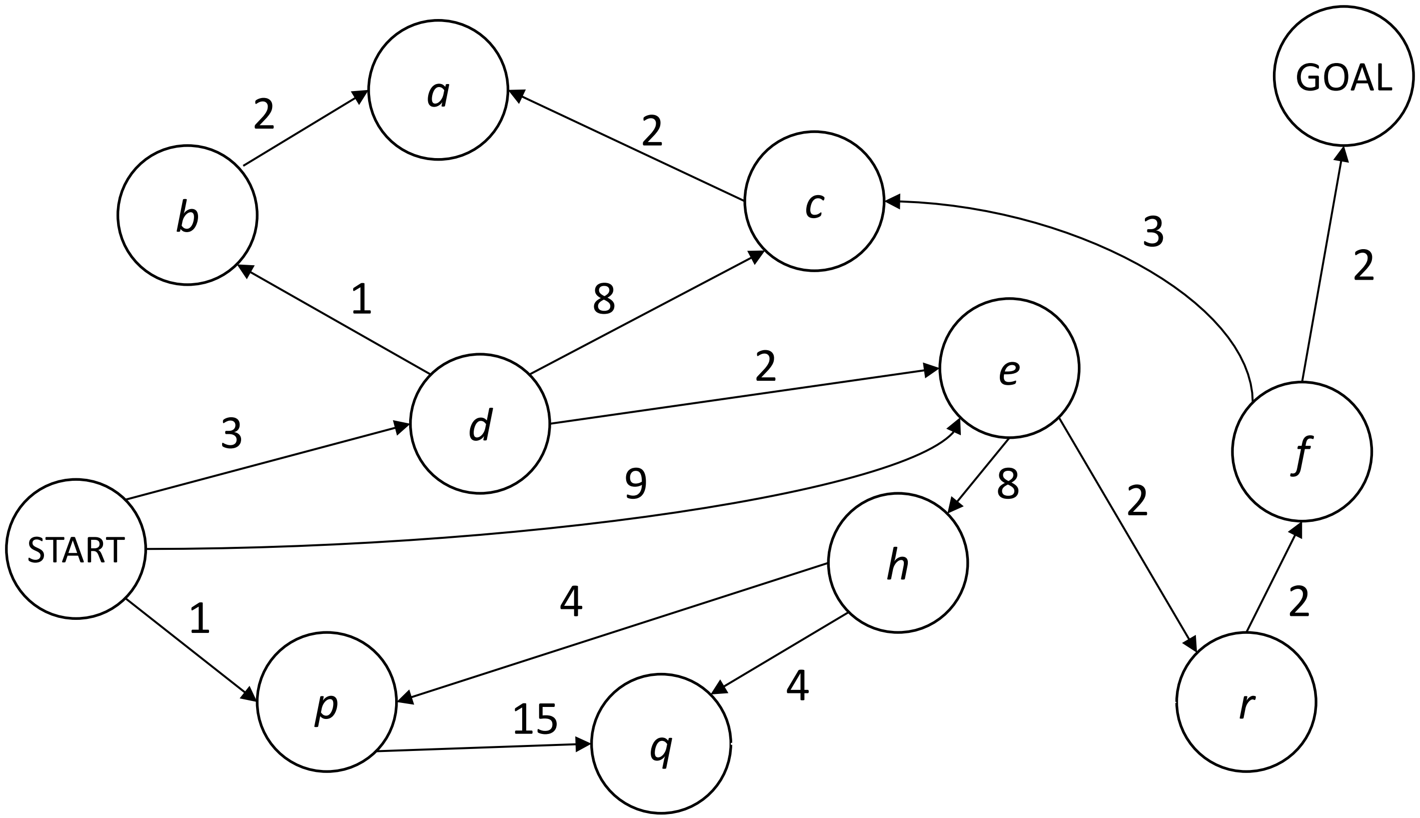


$b + (b + b^2) + (b+ b^2+b^3)\dots$

$= sb^1+ (s-1)b^2+ (s-2)b^3+ (s-3)b^4\dots+b^{s+1}$

$= O(sb^1+ (s-1)b^2+ (s-2)b^3+ (s-3)b^4\dots+b^{s+1})= O(b^{s+1})$

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

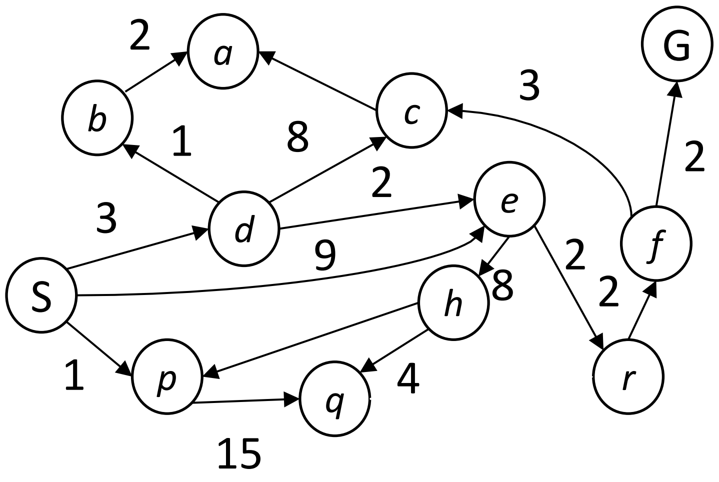
Fringe: *A data structure to store nodes observed but not visited*

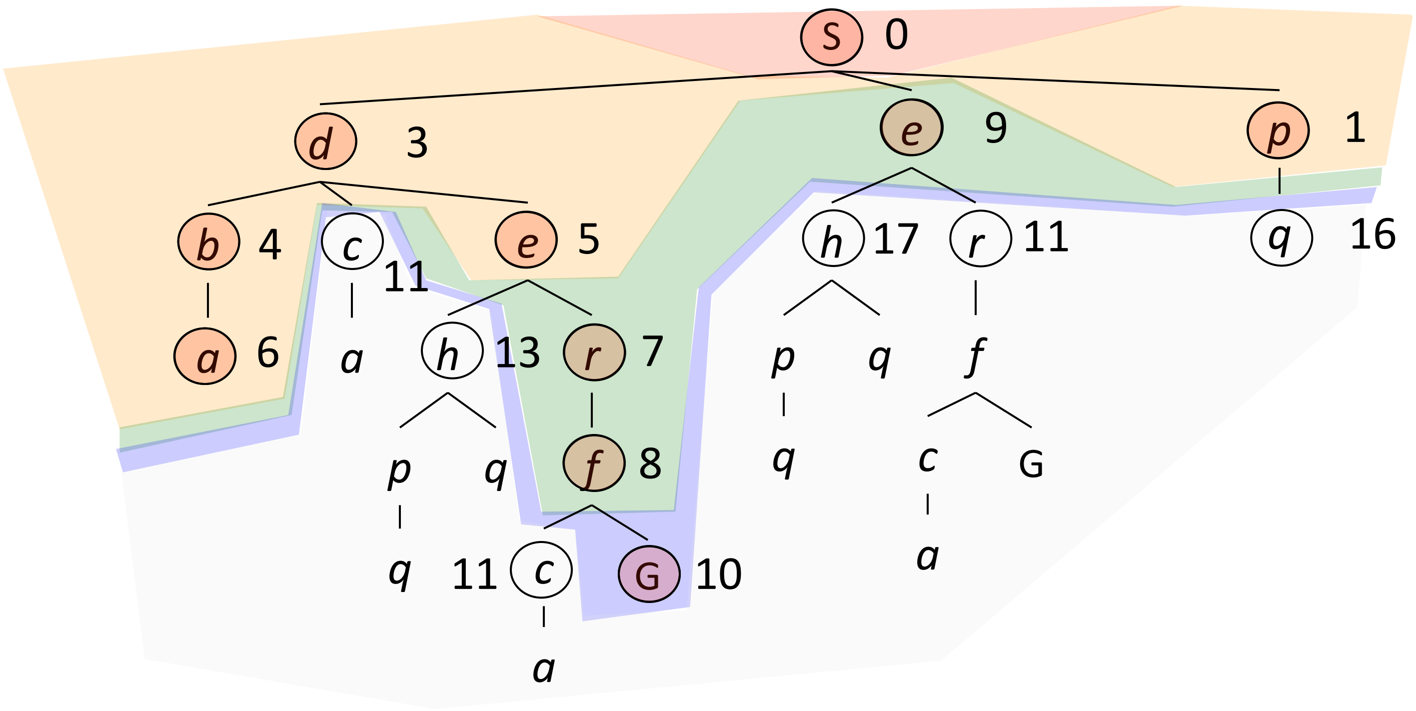
Expand*: pop a node from the fringe*

Generate*: get neighbors of the expanded node and insert them into the fringe*

Strategy*: expand a node with least cost (accumulative cost from root to this node)*

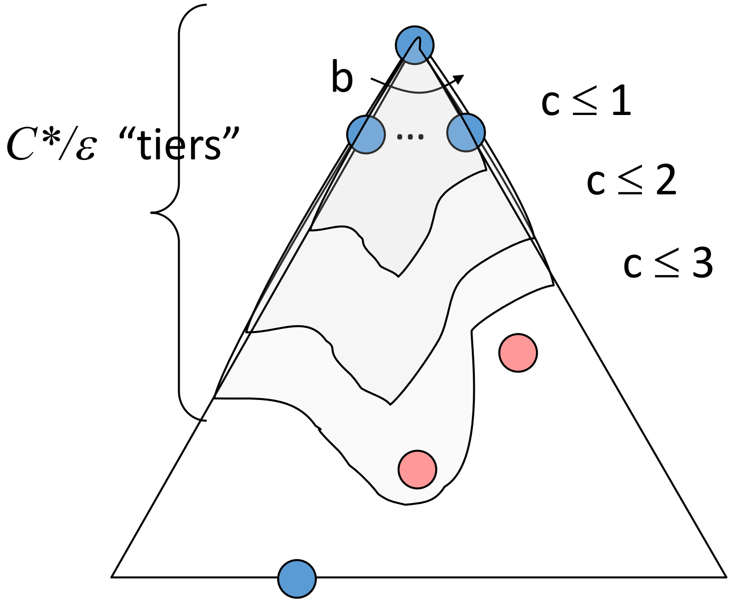
Implementation*: Fringe is a Priority queue*





Uniform Cost Search (UCS) Properties

- Time Complexity
  - Takes time O(b$^{C^*/\varepsilon+1}$) (exponential in effective depth)
- Space Complexity
  - Has roughly the last tier, so O(b$^{C^*/\varepsilon+1}$)
- Is it complete?
  - yes!
  - best solution has a finite cost
  - minimum arc cost is positive.
- Is it optimal?
  - Yes! minimum arc cost is positive.



C* : cost for the lowest solution
ε : the cost of least arc in the search tree
C*/ε: effective depth, most steps from root to the solution

Comparison

Algorithm	Complete?	Optimal?	Time?	Space?
DFS	N	N	O(b^m)	O(bm)
BFS	Y	Y	O(b^{s+1})	O(b^{s+1})
IDS	Y	Y	O(b^{s+1})	O(bs)
UCS	Y	Y	O($b^{C^*/\varepsilon+1}$)	O($b^{C^*/\varepsilon+1}$)

b is branching factor
s is the depth of the optimal solution
m is the maximum depth of the tree
C* is cost for the lowest solution
ε is the cost of least arc in the search tree

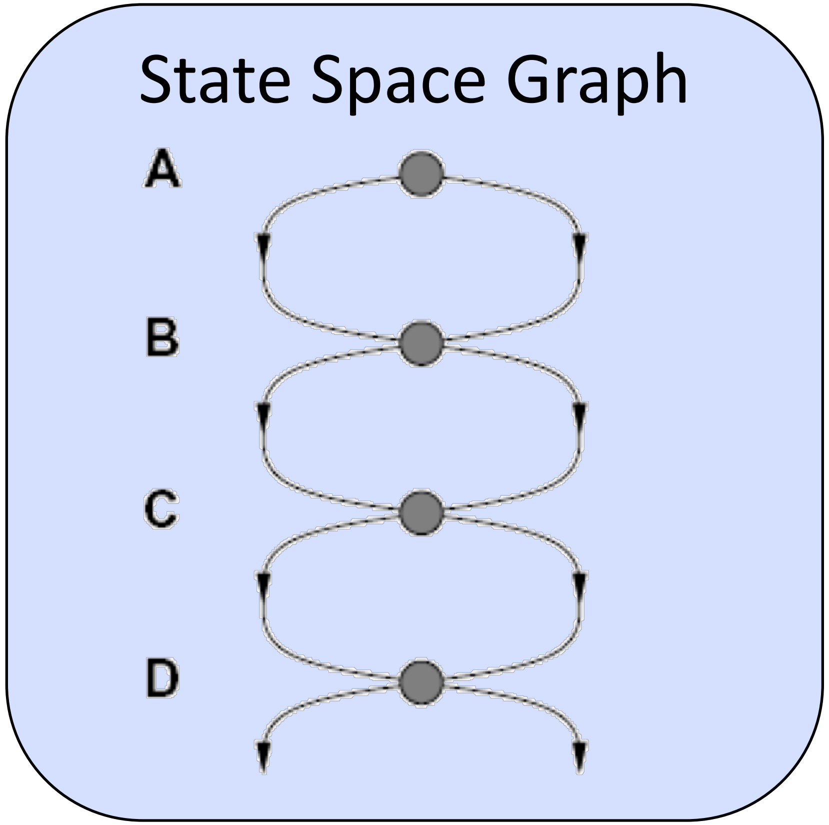
For BFS, Suppose the branching factor *b* is finite and step costs are identical

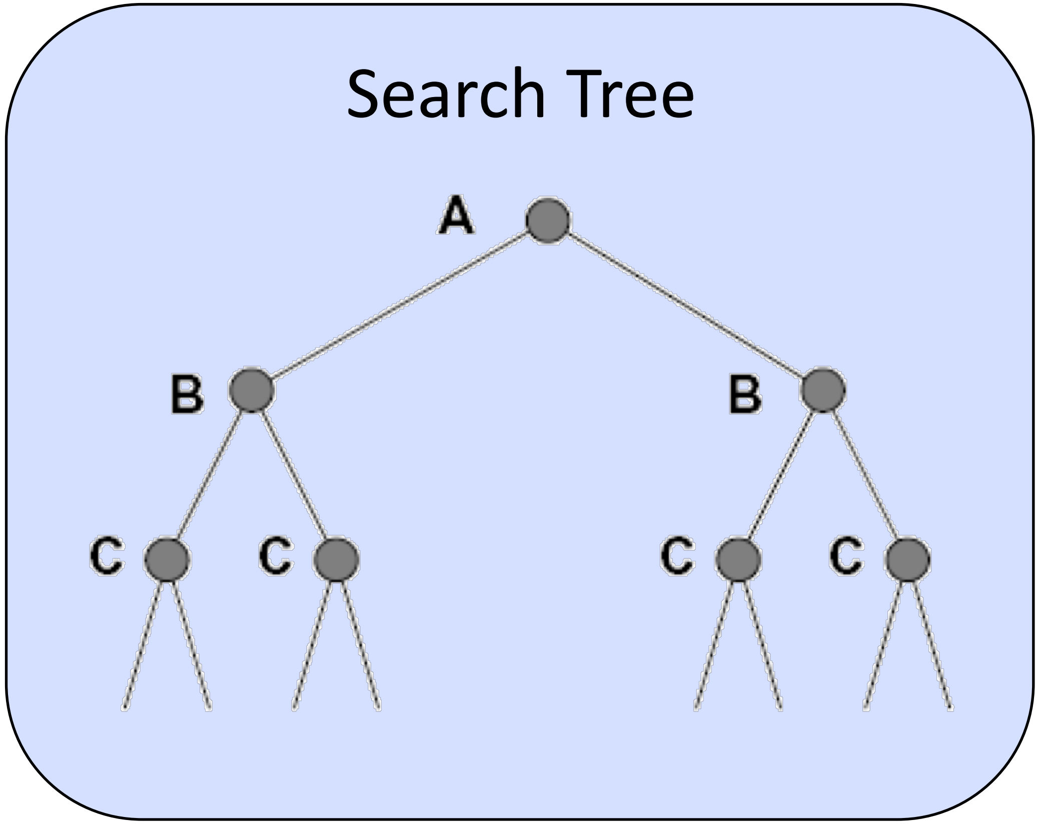
Outline

- Search Problems
- State Spaces
- State Space Representation
- Solve a Search Problem
- Uninformed Search Algorithms
  - Depth-First Search
  - Breadth-First Search
  - **Iterative Deepening**
  - Uniform-Cost Search
- Tree Search VS Graph Search

Tree Search: Extra Work!

- Failure to detect repeated states can cause exponentially more work.

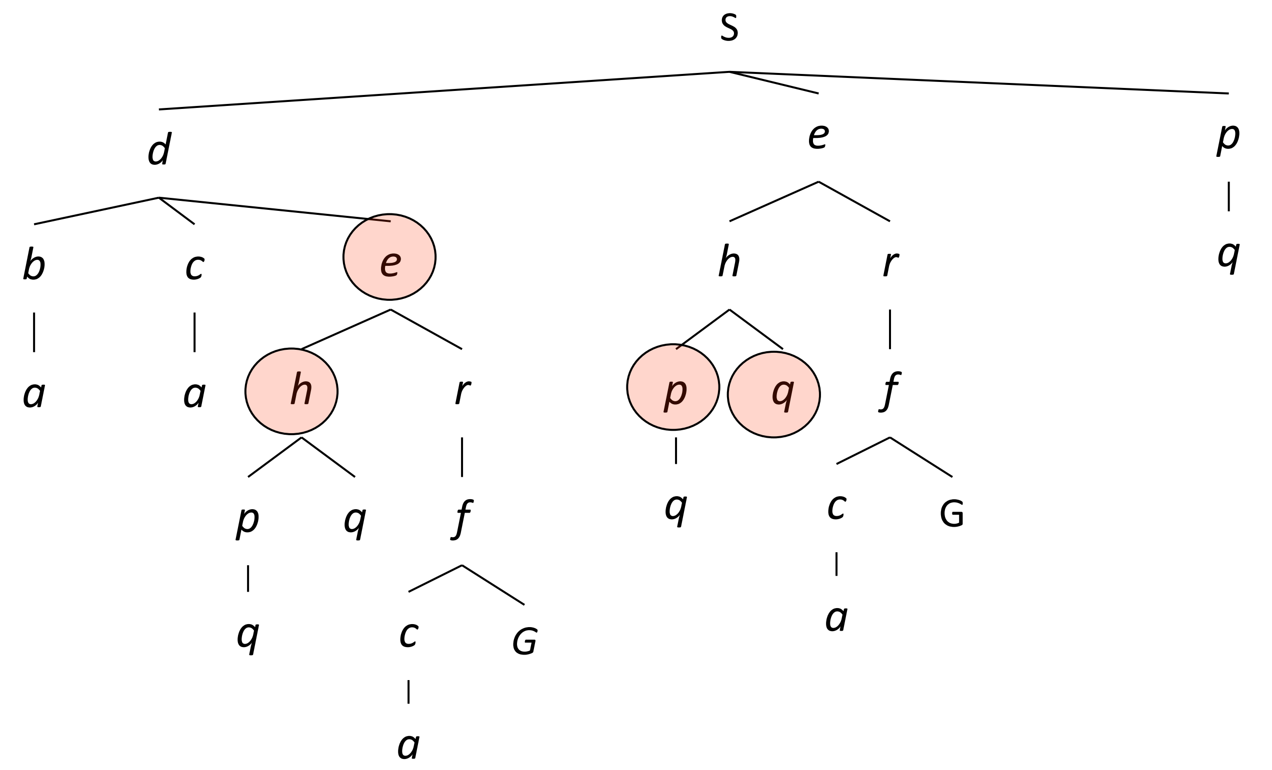




Motivation of Graph Search

- We shouldn't bother expanding the circled nodes (why?)

- We have visited their related states earlier with less cost



Graph Search

- Idea: never expand a state twice
- Tree search + set of expanded states (“closed set”)
- Update the expanded states set after expanding a state
- Check the expanded states set before expanding a state, if it has been added into the set, skip it.

Pseudo-Code for Search Paradigm

```
function TREE-SEARCH(problem, fringe) return a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        for child-node in EXPAND(STATE[node], problem) do
            fringe ← INSERT(child-node, fringe)
        end
    end
```

```
function GRAPH-SEARCH(problem, fringe) return a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            for child-node in EXPAND(STATE[node], problem) do
                fringe ← INSERT(child-node, fringe)
            end
    end
```