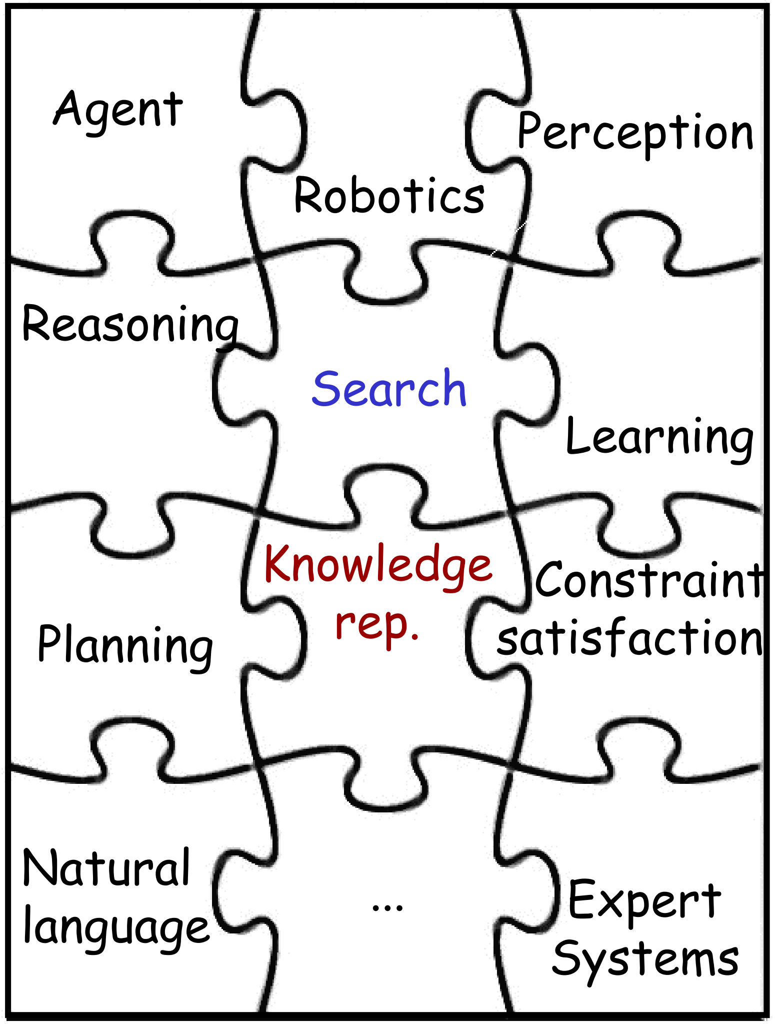
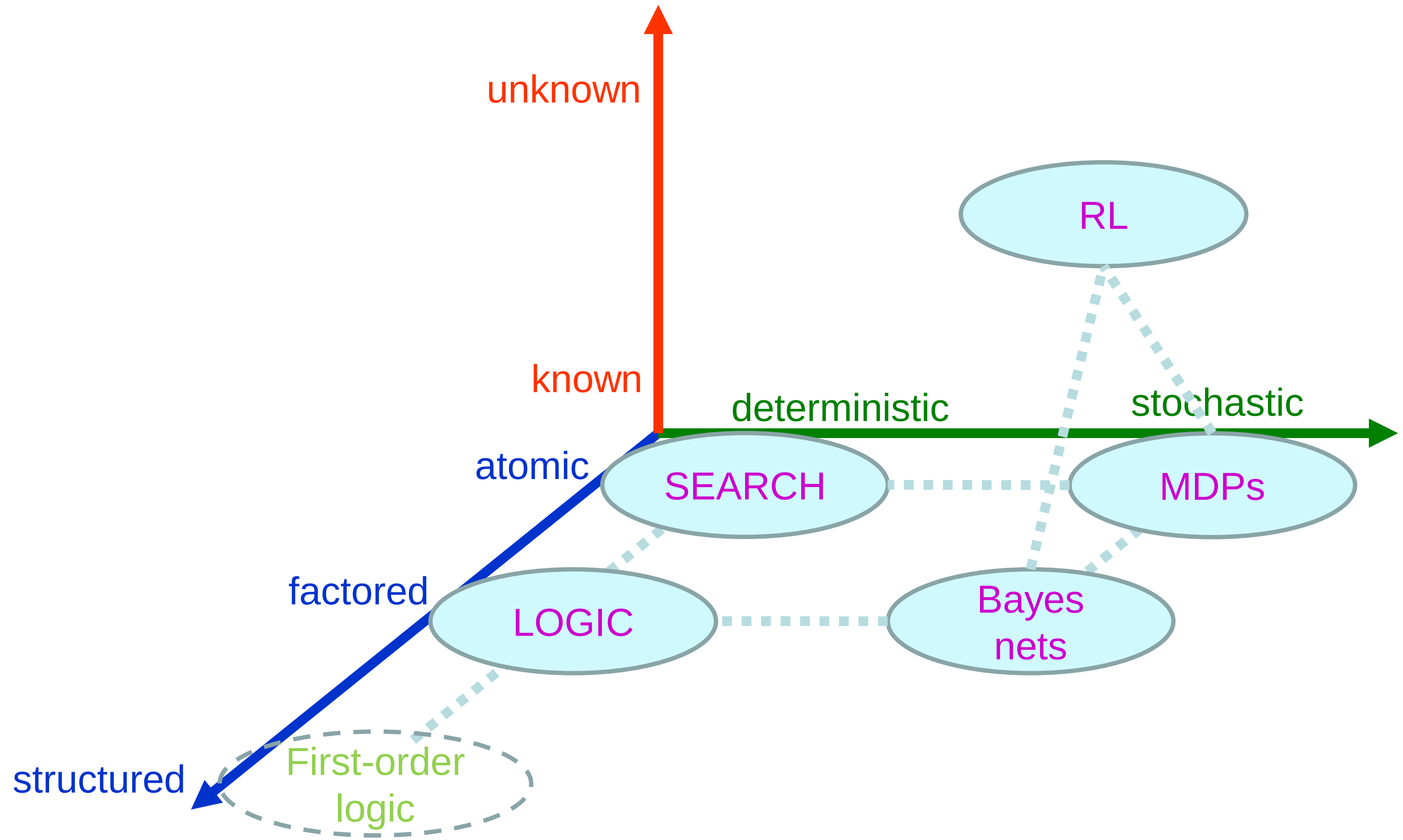
Artificial Intelligence



Outline of AI Strategies

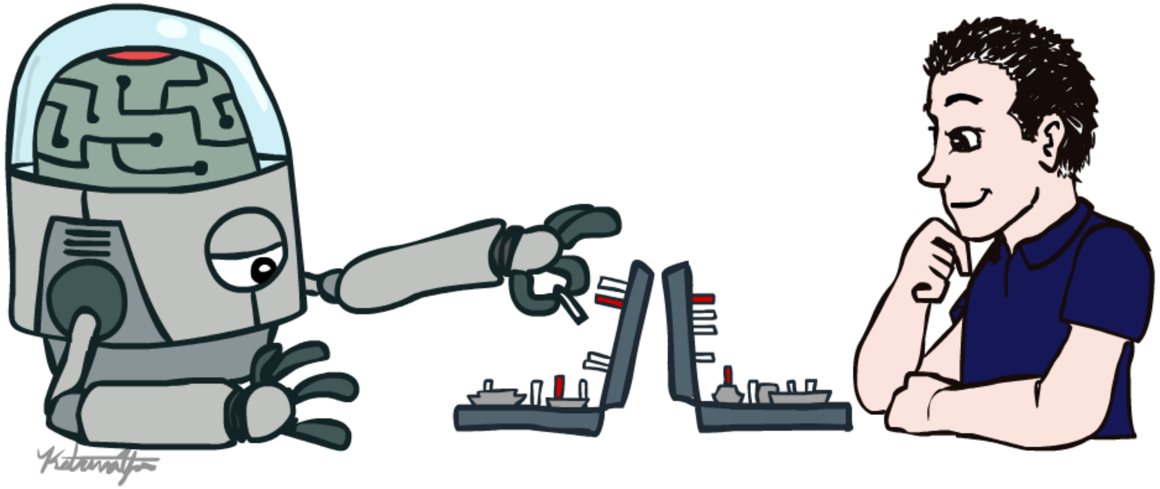


Why Game Examples

- Across history, puzzles and games requiring the exploration of alternatives have been considered a challenge for human intelligence:

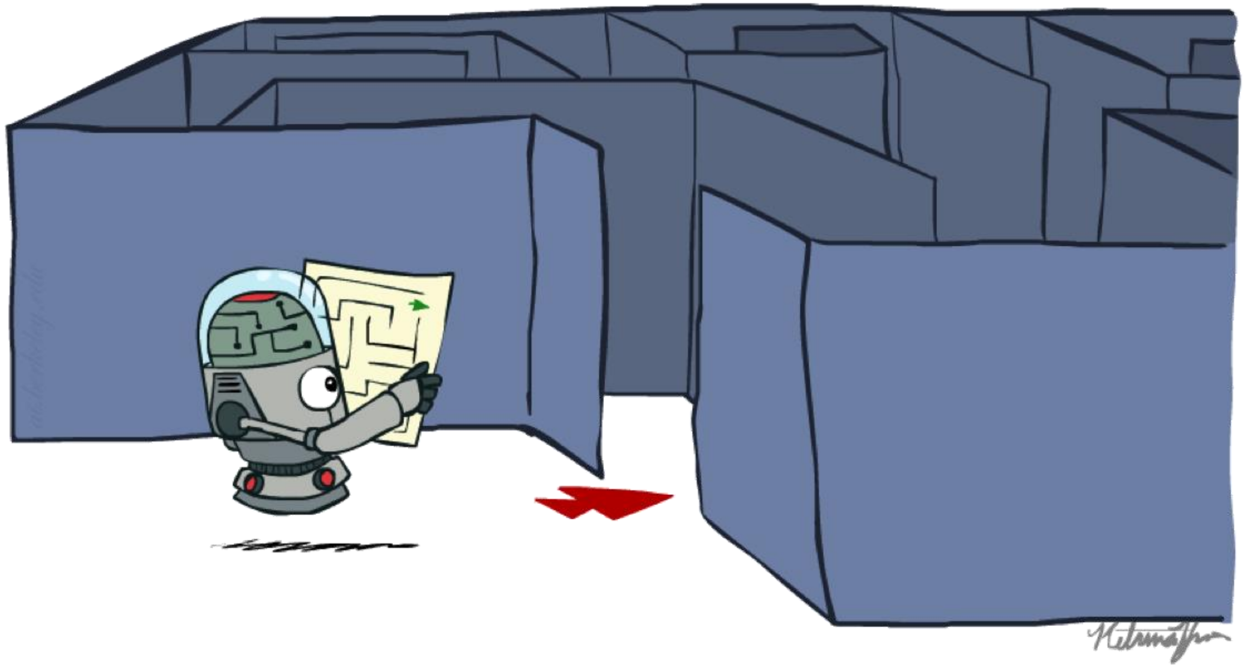
- Chess originated in Persia and India about 4000 years ago
- Checkers appear in 3600-year-old Egyptian paintings
- Go originated in China over 3000 years ago

- So, it's not surprising that AI uses games to design and test algorithms



Artificial Intelligence

Search

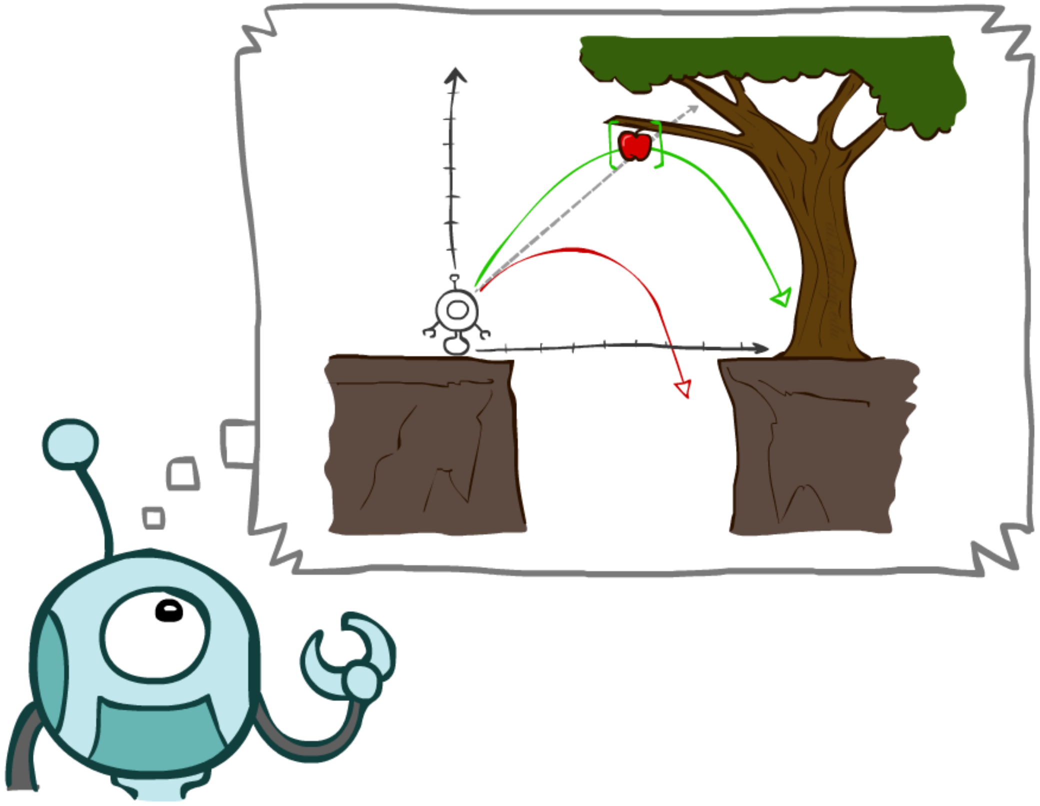


Motivation

- One of the major goals of AI is to help humans in solving complex tasks
  - How can I fill my container with pallets?
  - Which is the *shortest* way from Pune to Delhi?
  - Which is the *fastest* way from Pune to Delhi?
  - How can I *optimize* the load of my cargo to *maximize* my revenue?
  - How can I solve my Sudoku game?
  - What is the sequence of actions I should apply to win a game?
- Sometimes **finding a solution is not enough**, you want the **optimal solution** according to some “cost” criteria
- All the example presented involve looking for a **plan**
- **Plan:** the **set of operations** to be performed of an **initial state**, to reach a final state that is considered the **goal state**
- Thus, we **need efficient techniques to *search* for paths**, or sequences of actions, that can enable us to reach the goal state, i.e., to find a plan
- Such techniques are commonly called ***Search Methods***

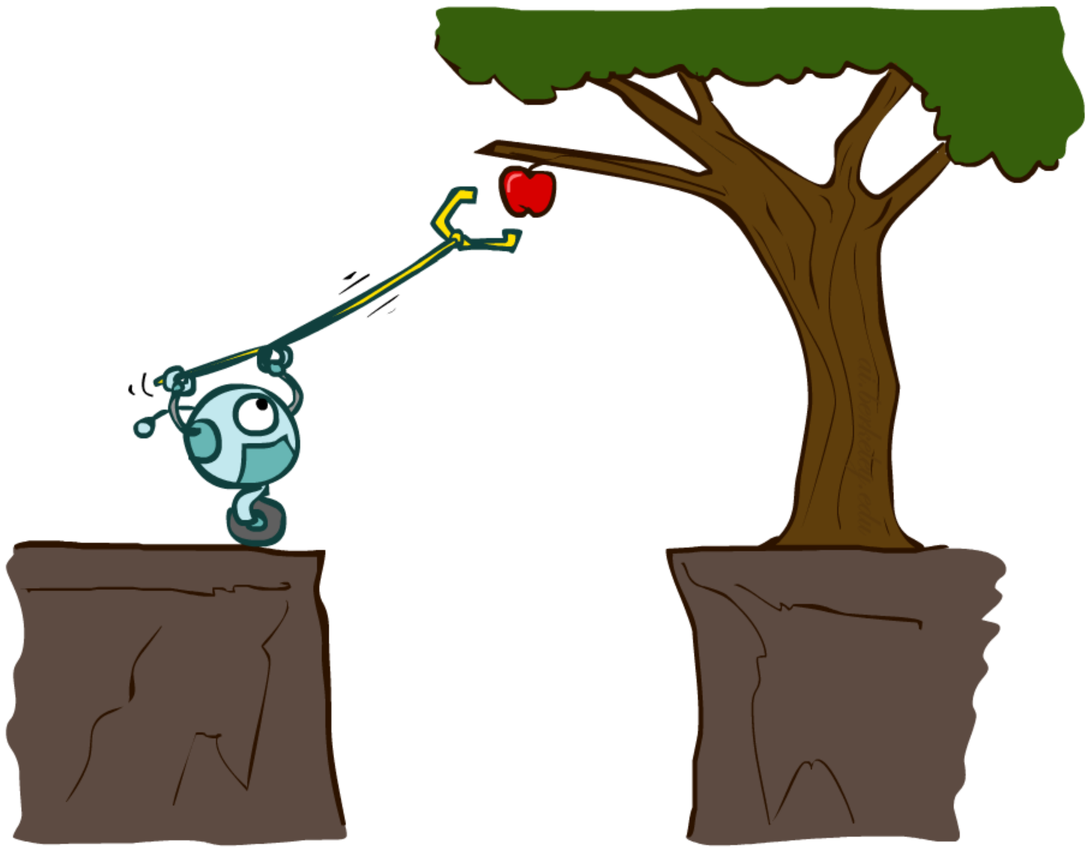
Today

- Agents that Plan Ahead
- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search



Planning Agents

- Planning agents decide based on evaluating future action sequences
- Must have a model of how the world evolves in response to actions
- Usually have a definite goal
- Optimal: Achieve goal at least cost



Problem solving

(using intelligence similar to human intelligence or using Intelligent Agents)

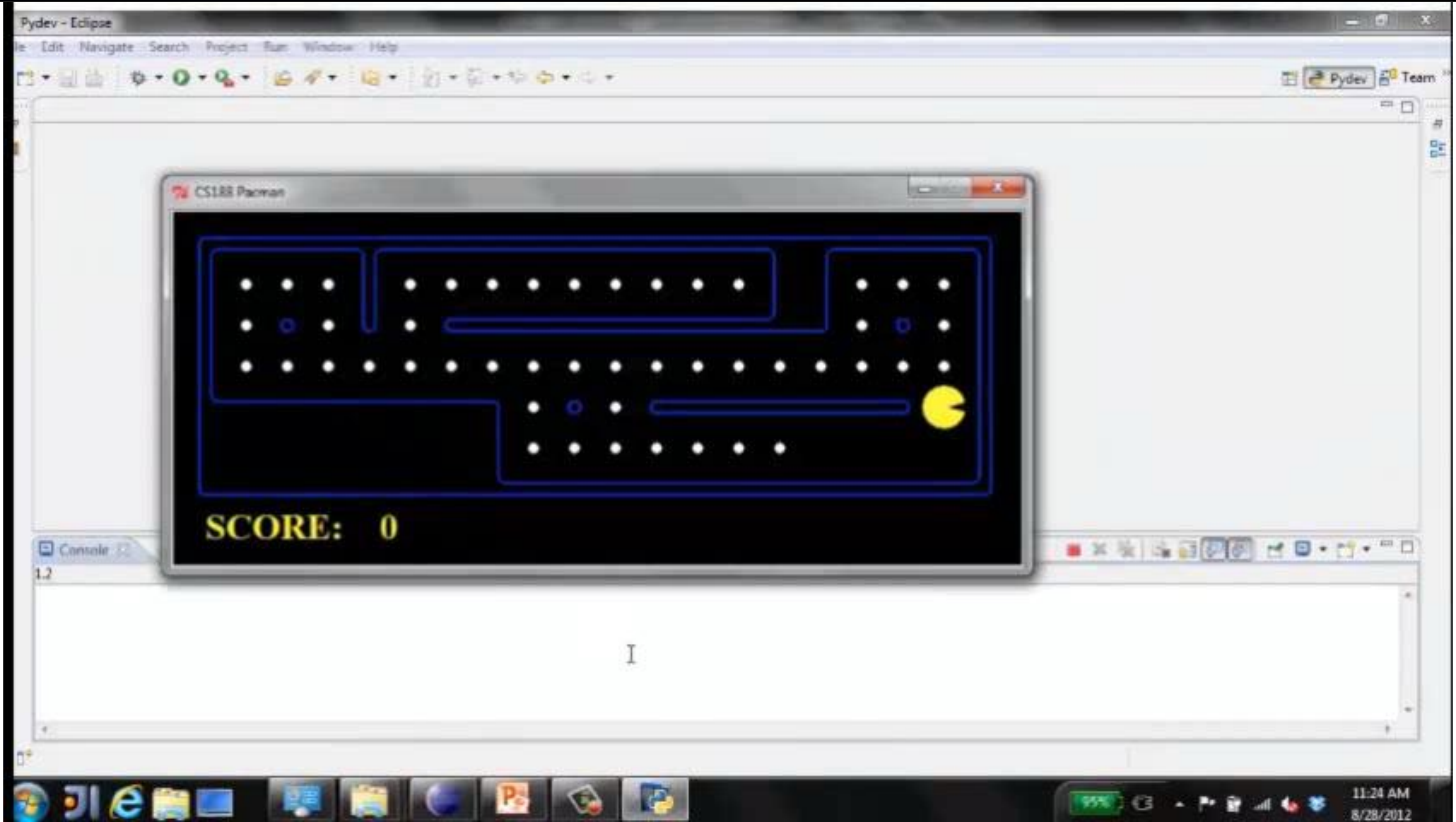
- We want:
  - To automatically solve a problem.

Formalization

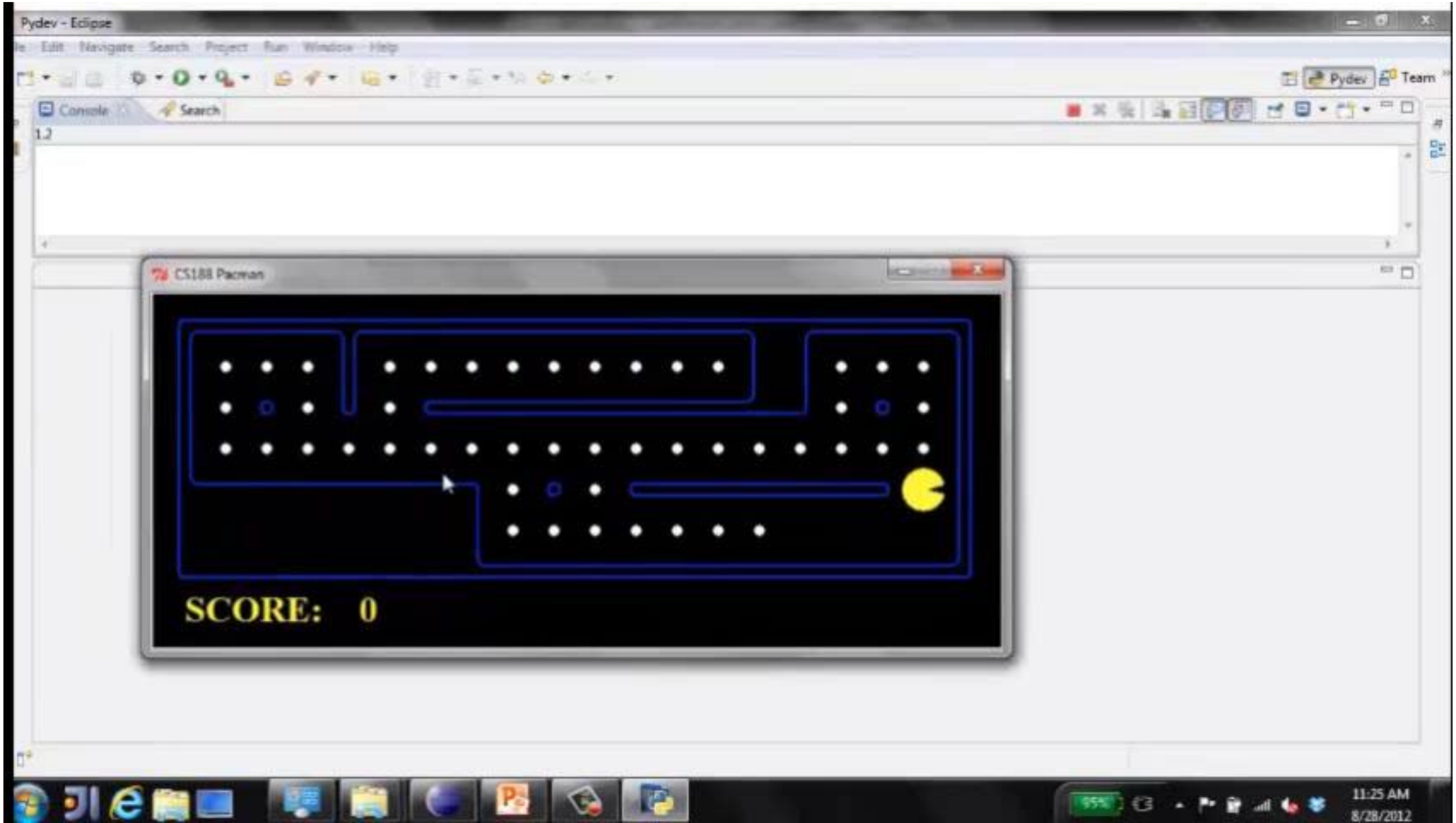
Searching technique

- We need:
  - A representation of the problem.
  - Algorithms that use some strategy to solve the problem defined in that representation.

Move to nearest dot and eat it



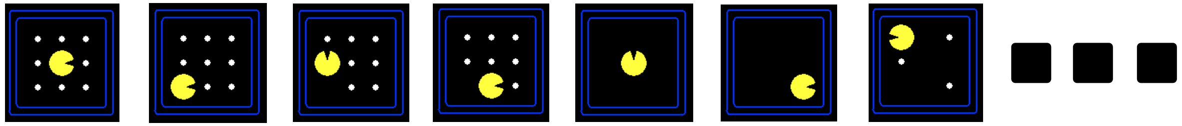
Precompute optimal plan, execute it

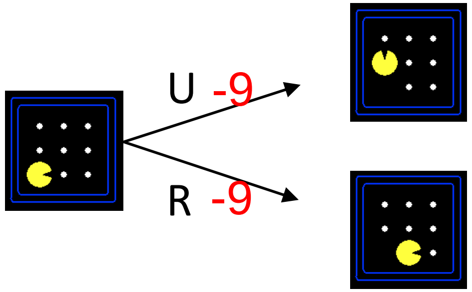


Search Problems

Search Problems

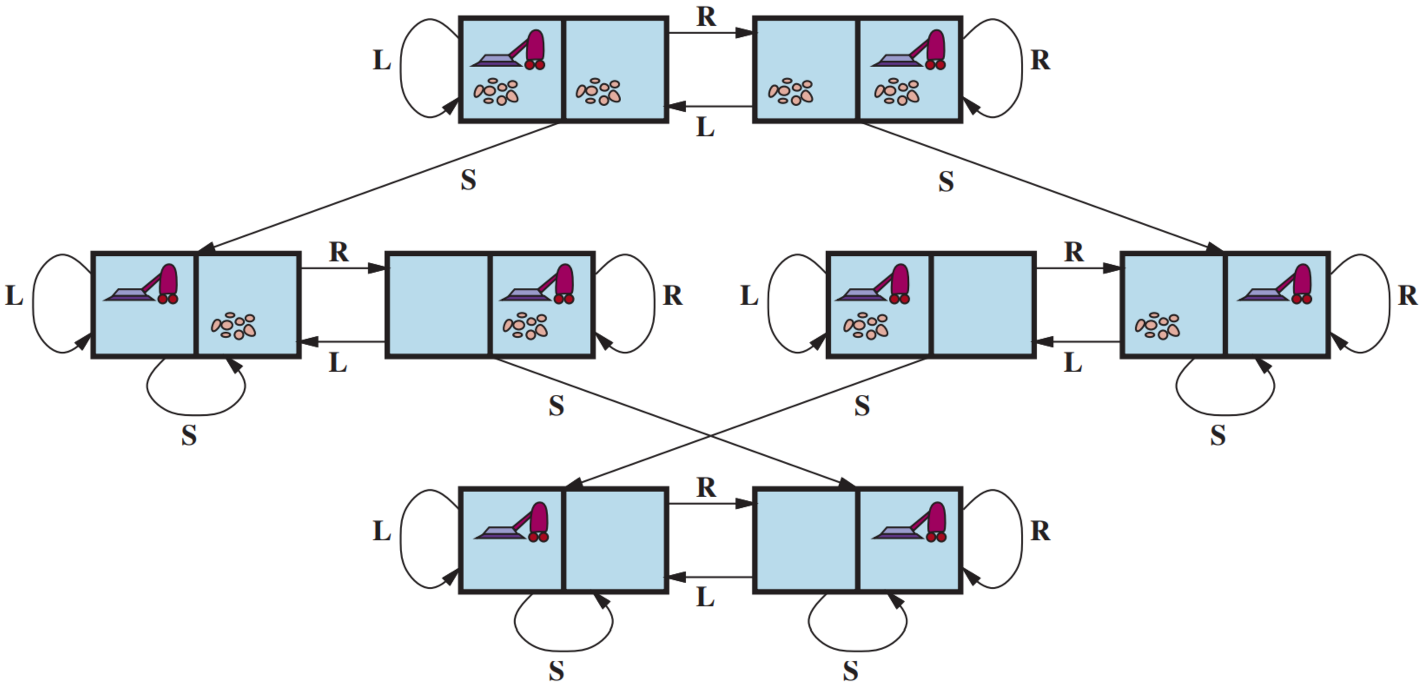
- A search problem consists of:
  - A state space $\mathcal{S}$
  - An initial state s_0
  - Actions $\mathcal{A}(s)$ in each state
  - Transition model $Result(s,a)$
  - A goal test $G(s)$
    - S has no dots left
  - Action cost $c(s,a,s')$
    - -1 per step; +10 food; +500 win; -500 die; +200 eat ghost





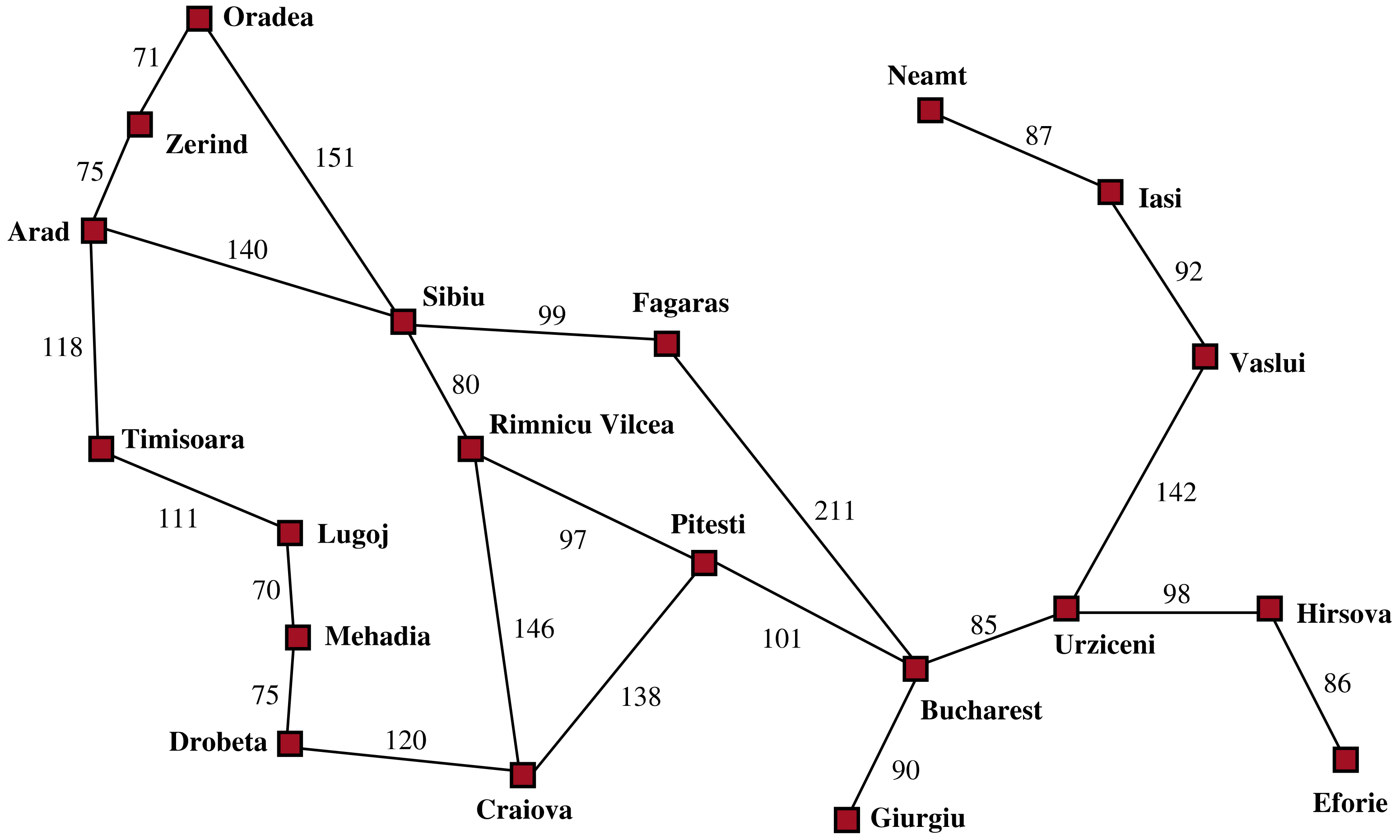
- A solution is an action sequence that reaches a goal state
- An optimal solution has least cost among all solutions

Example-1: Vacuum Cleaner world



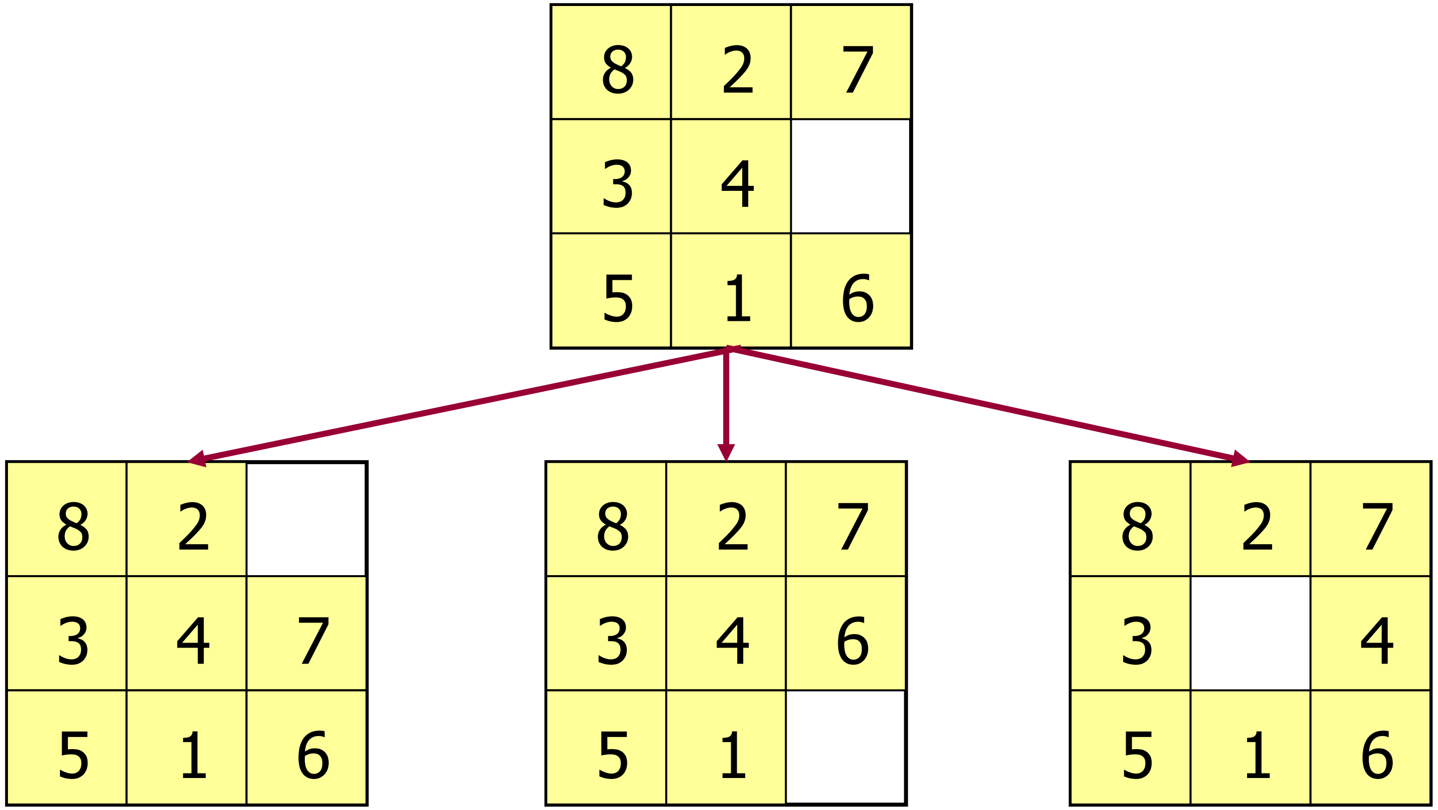
- States: Discrete: dirt and robot location
- Initial state: Any
- Actions: *Left, right, suck*
- Goal test: No dirt at all locations
- Path cost: 1 per action
- Solution Optimal sequence of operations(actions)

Example-2: Traveling in Romania



- State space:
  - Cities
- Initial state:
  - Arad
- Actions:
  - Go to adjacent city
- Transition model:
  - Reach adjacent city
- Goal test:
  - *s* = Bucharest?
- Action cost:
  - Road distance from *s* to *s'*
- Solution?

Example-3: 8-Puzzle



Search is about the exploration of alternatives

Example-3: 8-Puzzle

State: Any arrangement of 8 numbered tiles and an empty tile on a 3x3 board

8	2	
3	4	7
5	1	6

Initial state

1	2	3
4	5	6
7	8	

Goal state

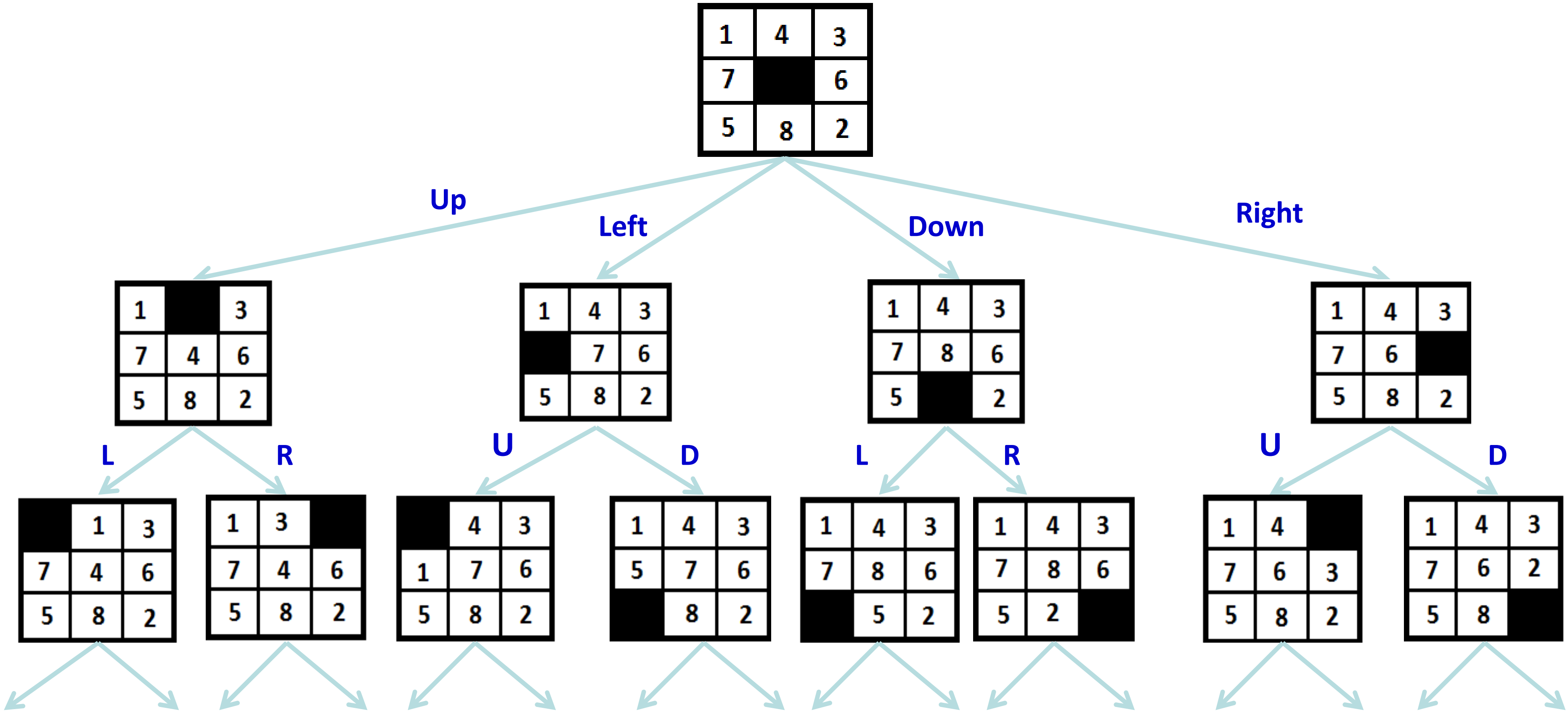
- States: Locations of tiles
- Initial State: Given
- Actions: Move blank left, right, up, down
- Goal test: Goal state (given)
- Path cost: 1 per move
- Solution: Optimal sequence of operators

How big is the state space of the $(n^2\text{-}1)$-puzzle?

- 8-puzzle → ?? states

How big is the state space of the (n^2-1)-puzzle?

- 8-puzzle → ?? states



How big is the state space of the (n^2-1)-puzzle?

- 8-puzzle → 9! = 362,880 states
- 15-puzzle → 16! ~ 2.09×10^{13} states
- 24-puzzle → 25! ~ 10^{25} states

But only half of these states are reachable from any given state

(but you may not know that in advance)

8-, 15-, 24-Puzzles

8-puzzle → 362,880 states

0.036 sec

15-puzzle → 2.09 x 10^{13} states

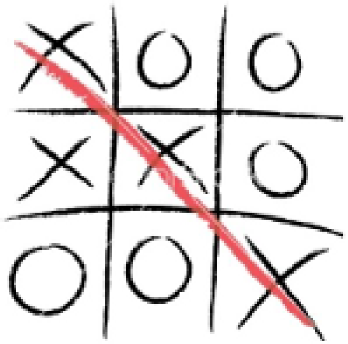
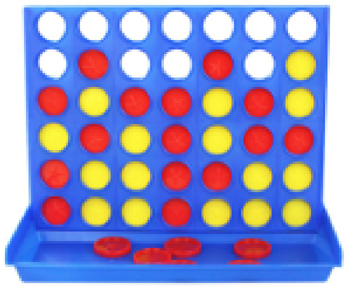
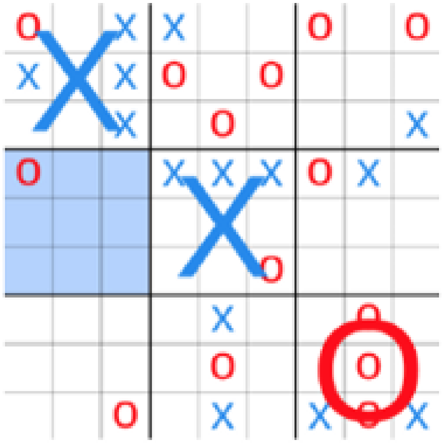
~ 55 hours

24-puzzle → 10^{25} states

> 10^9 years

100 millions states/sec

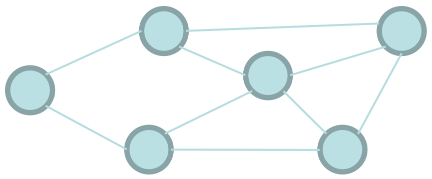
How big is the state space?

Tic-Tac-Toe	Connect Four	Ultimate Tic-Tac-Toe	Chess
			
<9 possible moves/turn 255,168 possible games	~7 possible moves/turn ~4.5 x 10^{12} possible games	~9 possible moves/turn ~4.4 x 10^{38} possible games	~37 possible moves/turn ~10^{120} possible games

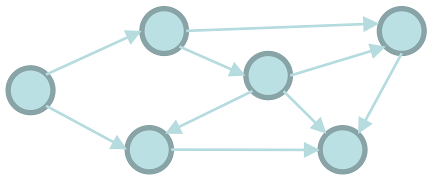
Search Space Representation

- A graph is *undirected* if arcs do not imply a direction, *direct* otherwise
- A graph is *connected* if every pair of nodes is connected by a path
- A connected graph with no loop is called *tree*
- A *weighted graph*, is a graph for which a value is associated to each arc

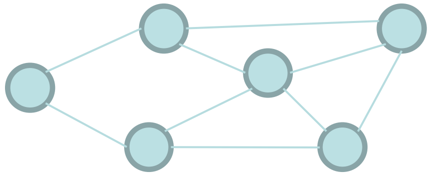
undirect



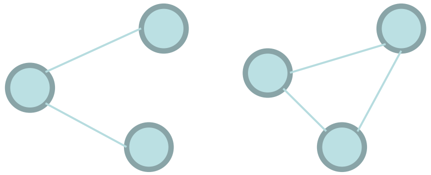
direct



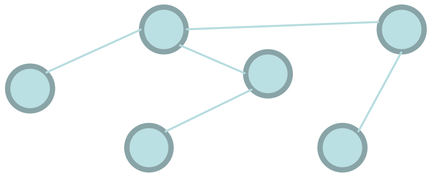
connected



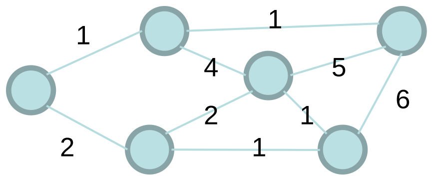
disconnected



tree

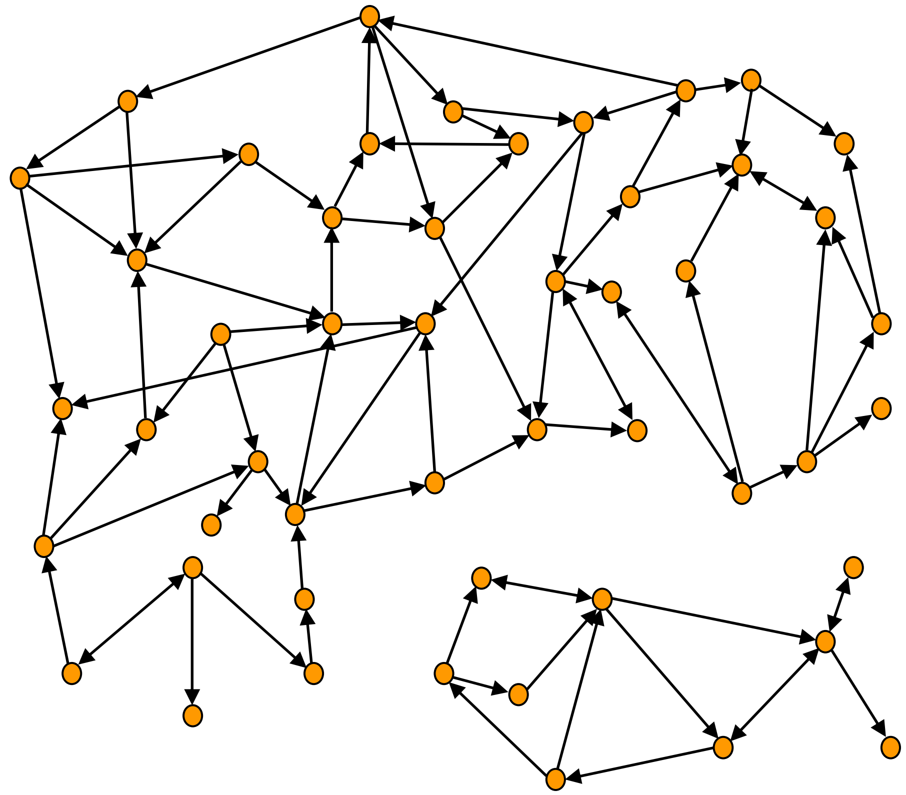


weighted



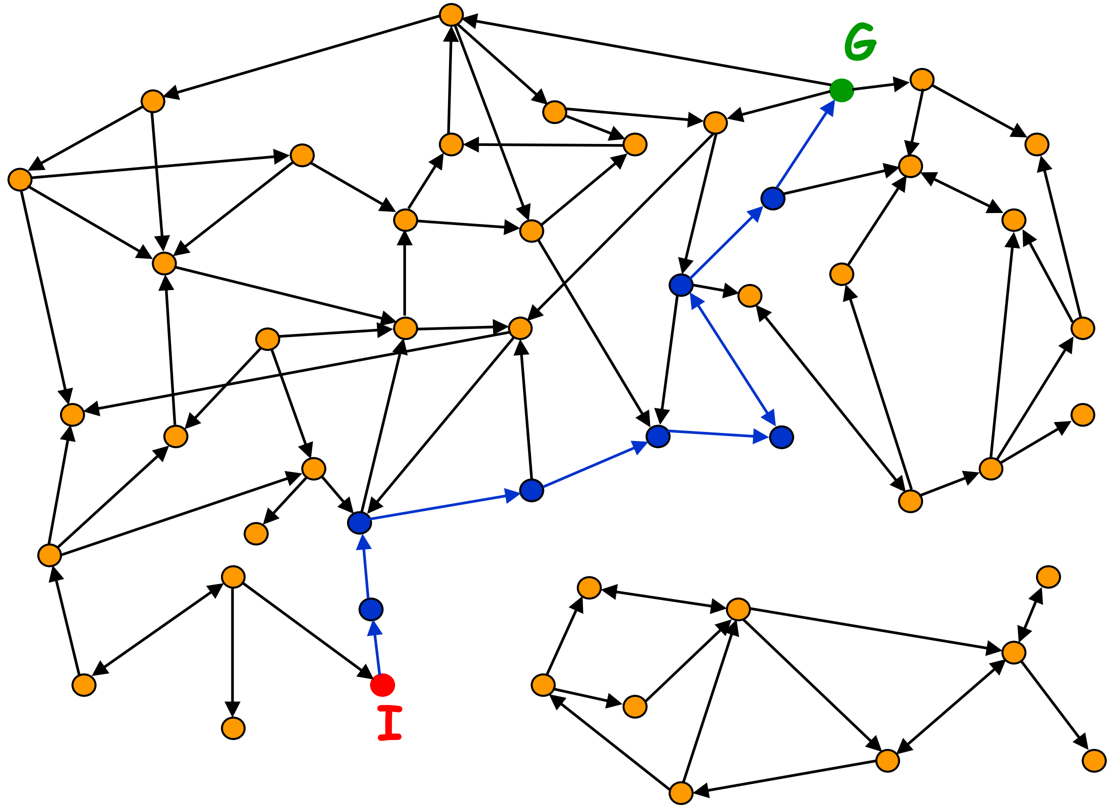
State Graph

- Each state is represented by a distinct node
- An arc (or edge) connects a node s to a node s’ if s’ $\in$ SUCCESSORS(s)
- The state graph may contain more than one connected component



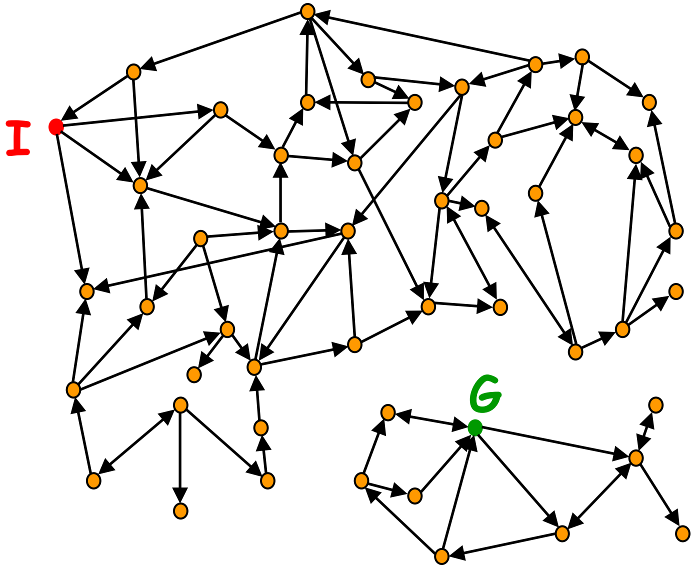
Solution to the Search Problem

- A solution is a path connecting the initial node to a goal node (any one)

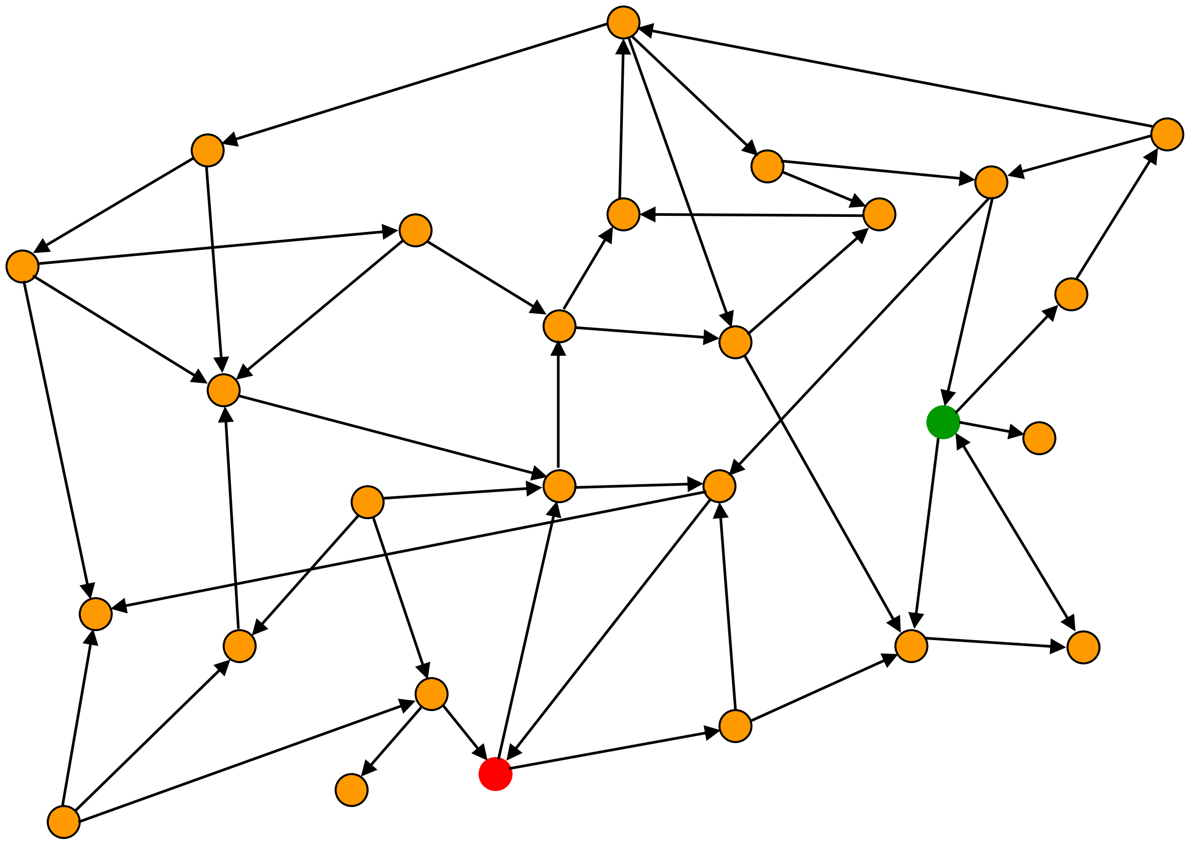


Solution to the Search Problem

- A solution is a path connecting the initial node to a goal node (any one)
- The cost of a path is the sum of the arc costs along this path
- An optimal solution is a solution path of minimum cost
- There might be no solution !

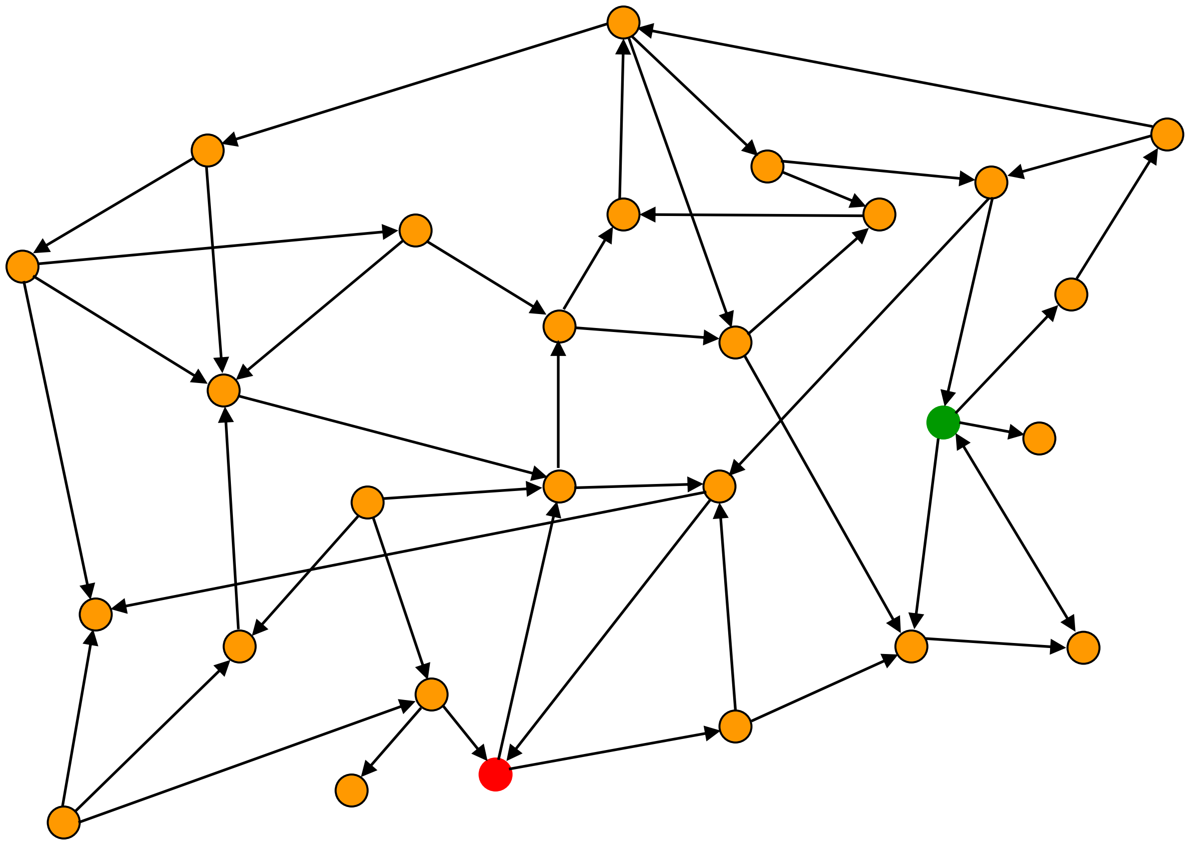


Searching the State Space



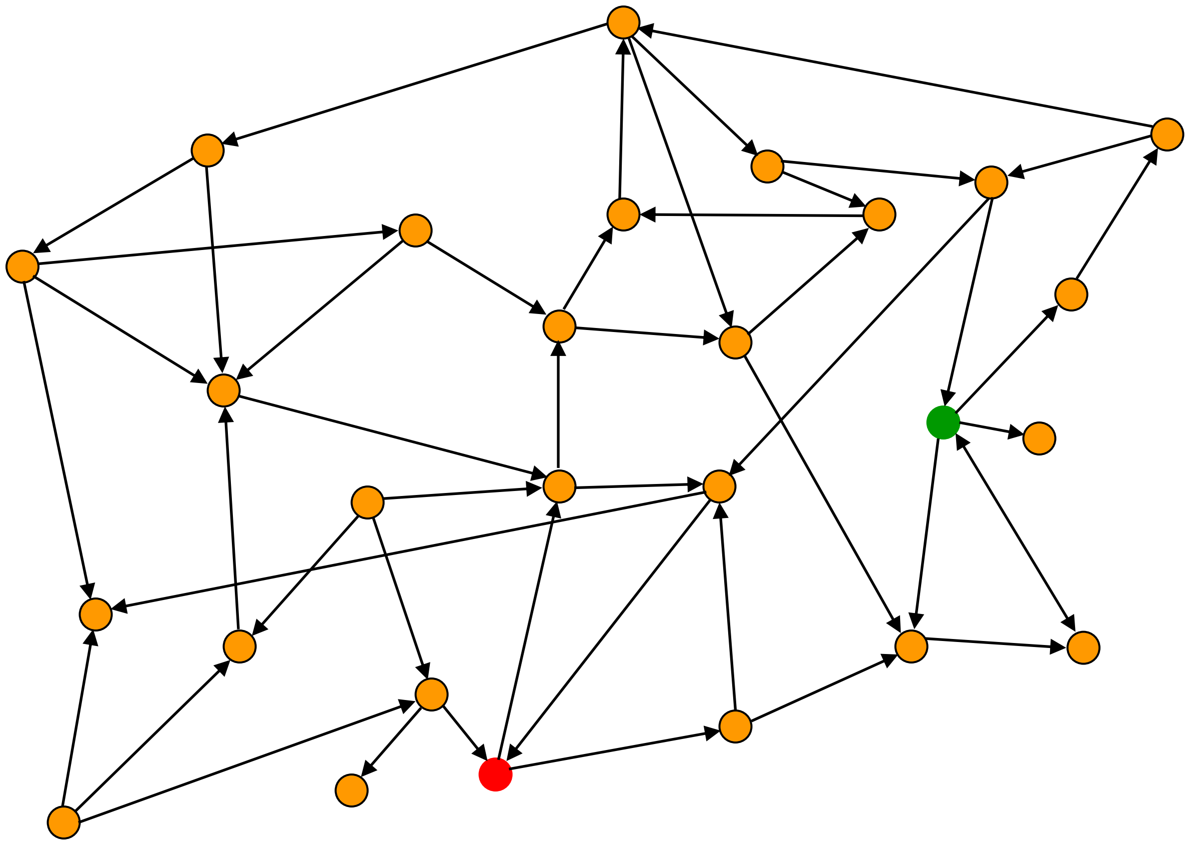
- It is often not feasible (or too expensive) to build a complete representation of the state graph

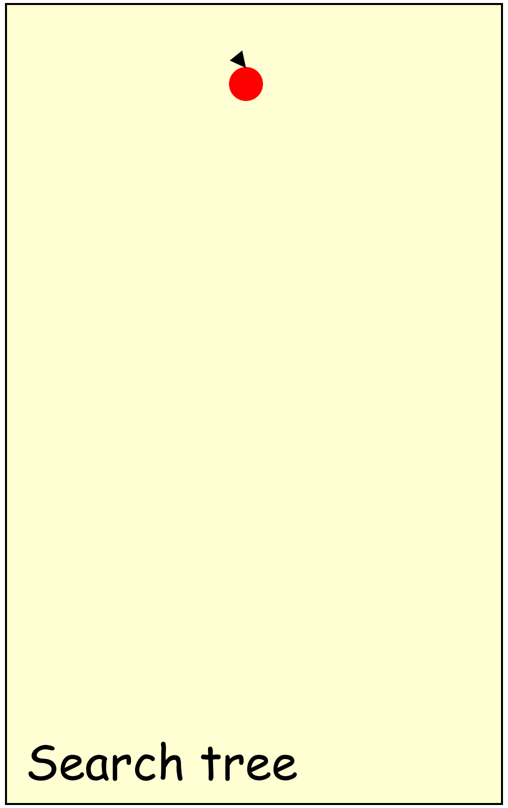
Searching the State Space



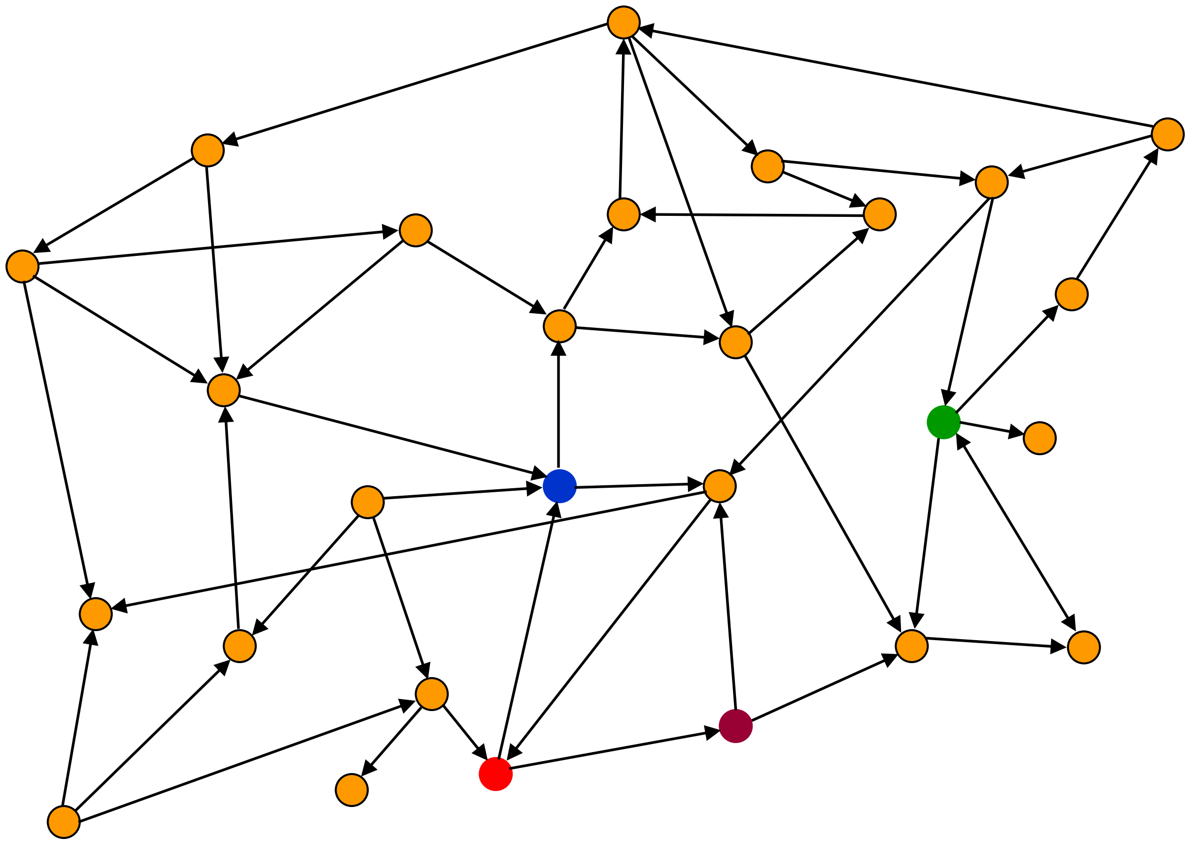
- Often it is not feasible (or too expensive) to build a complete representation of the state graph
- A problem solver must construct a solution by exploring a small portion of the graph

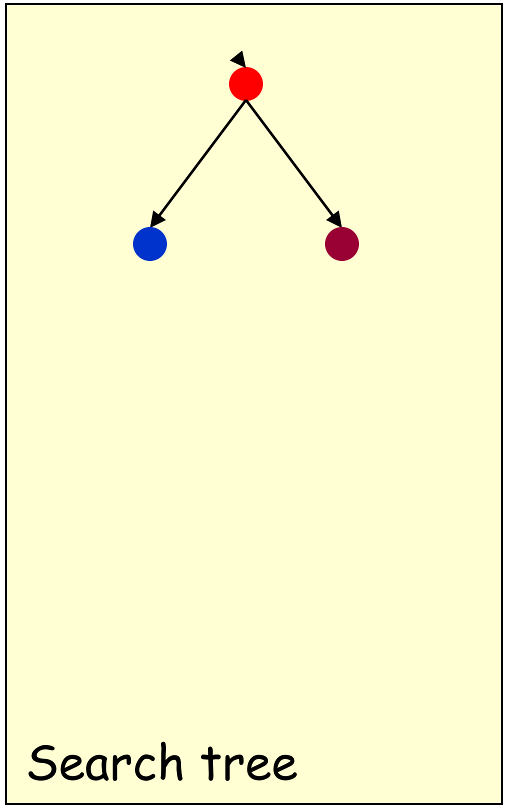
Searching the State Space



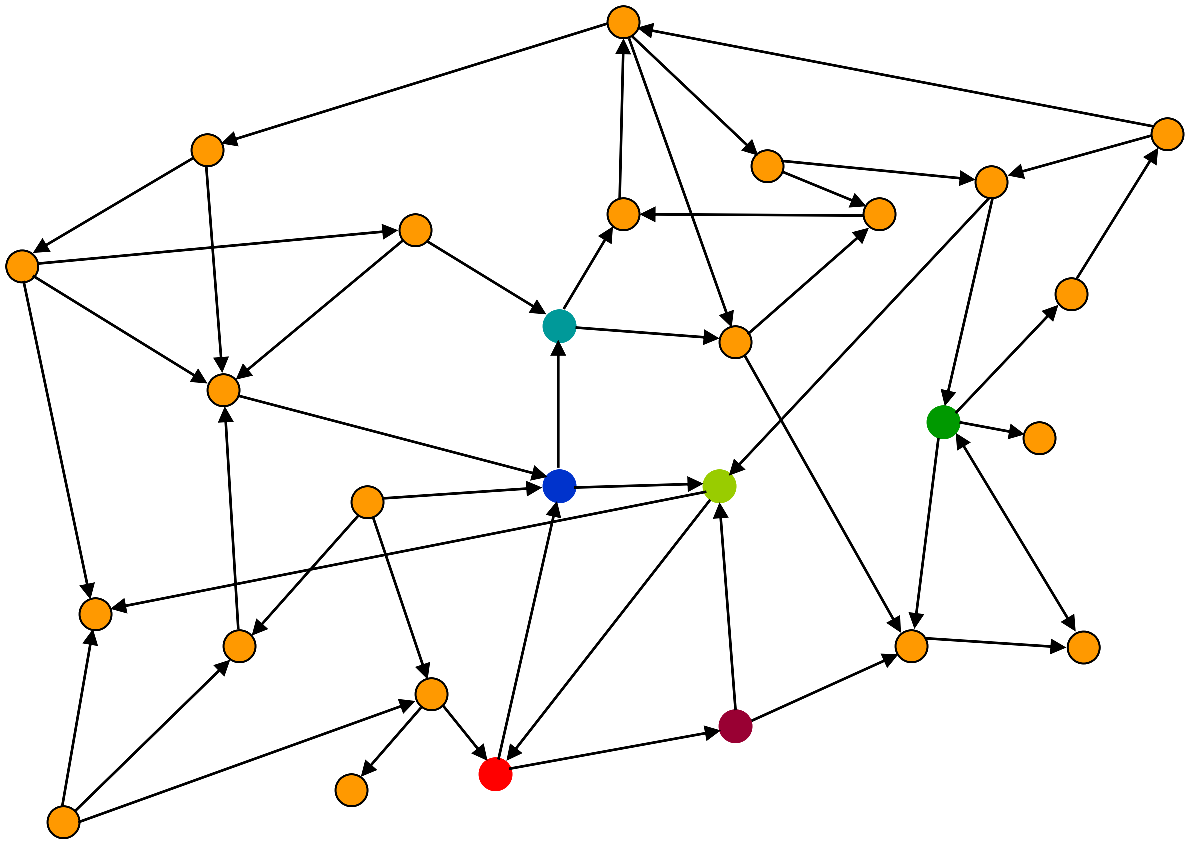


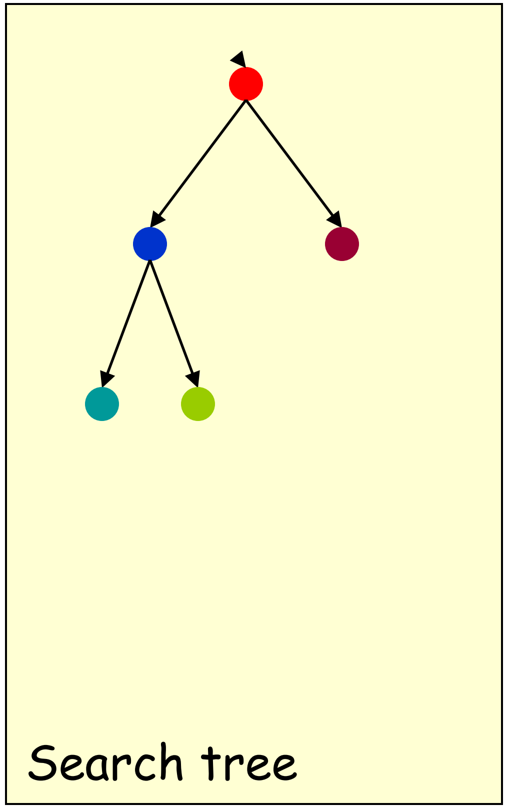
Searching the State Space



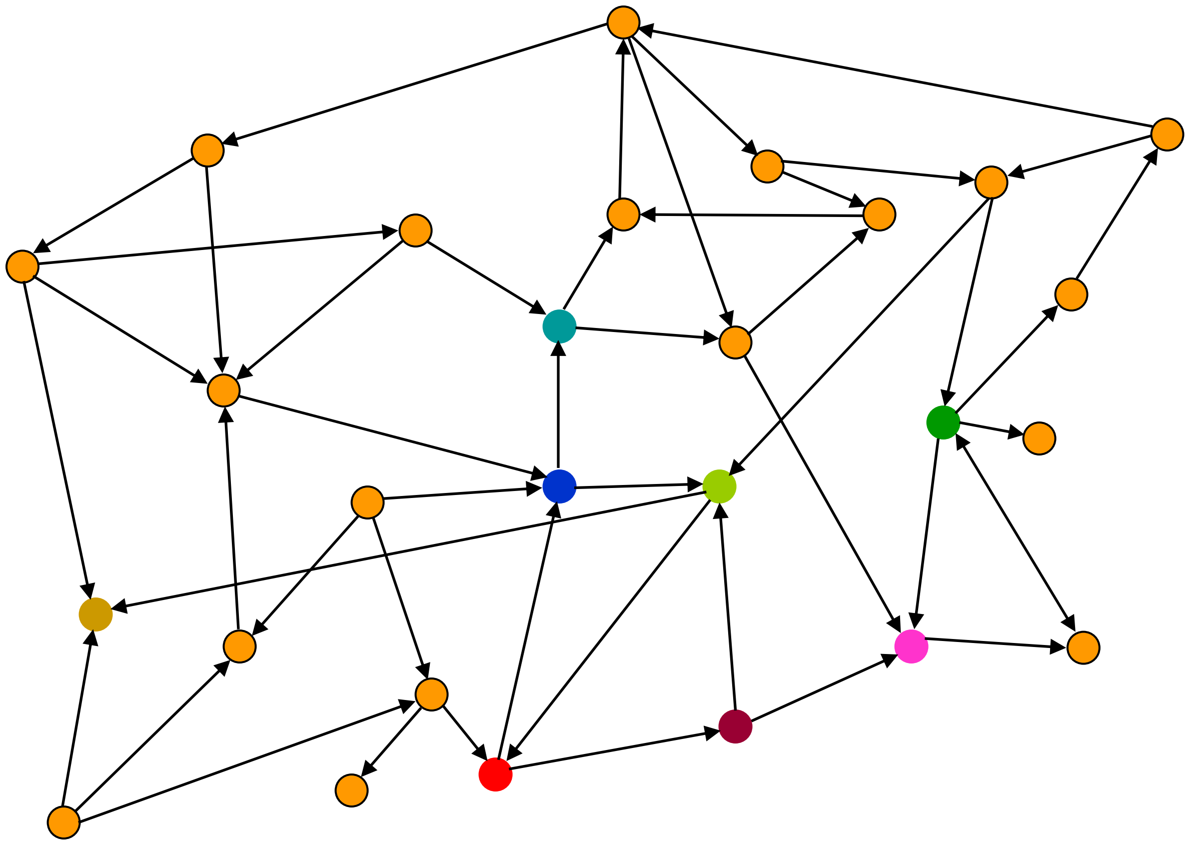


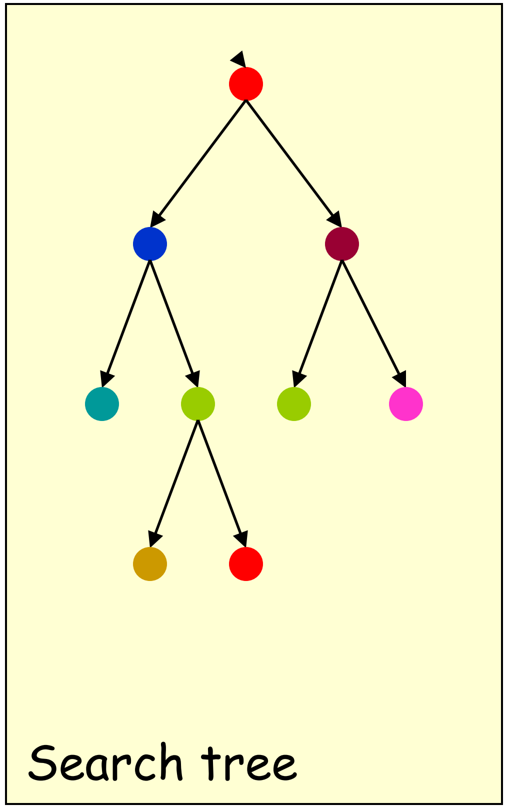
Searching the State Space



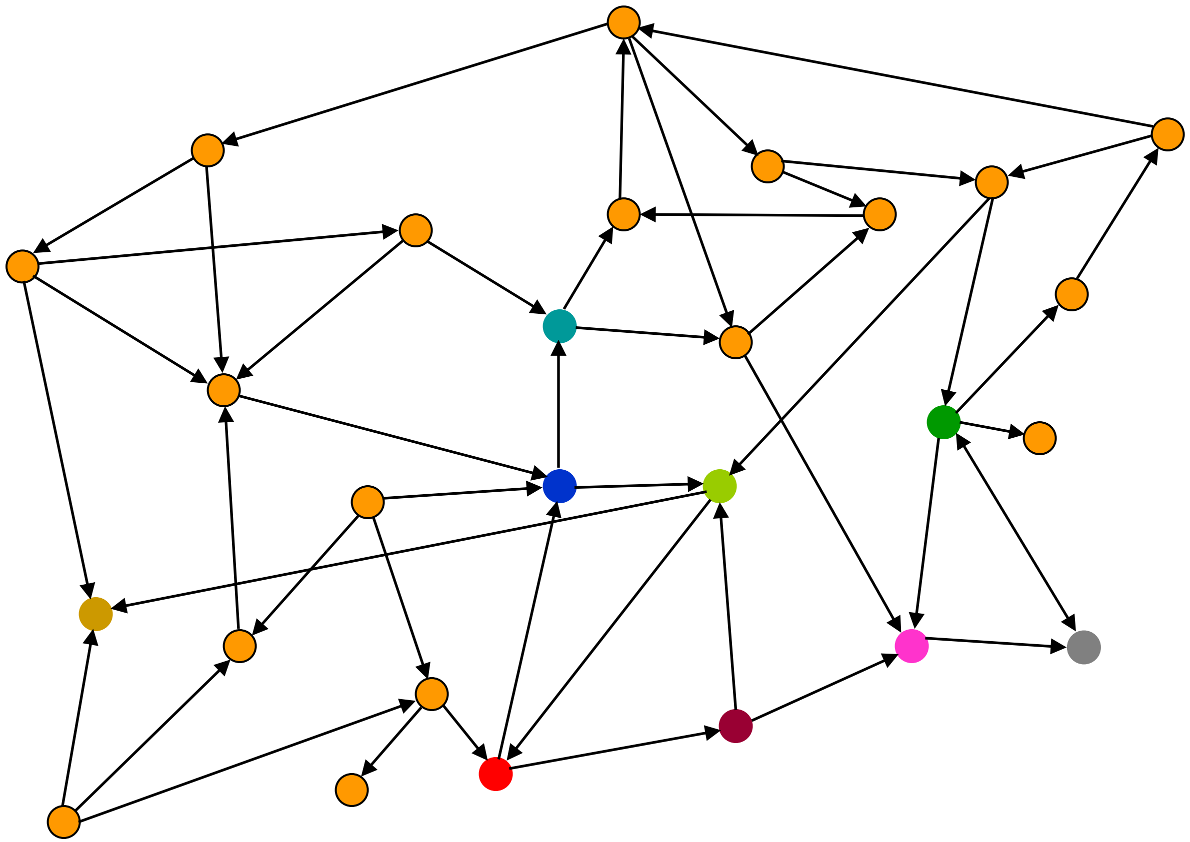


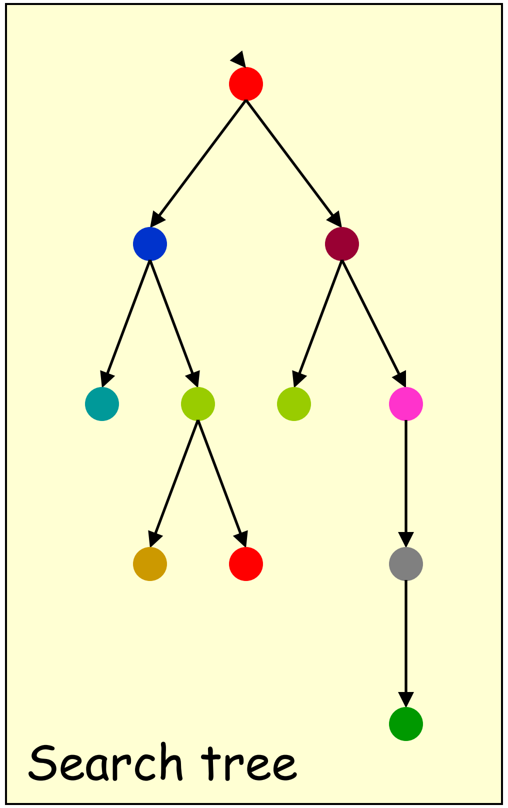
Searching the State Space



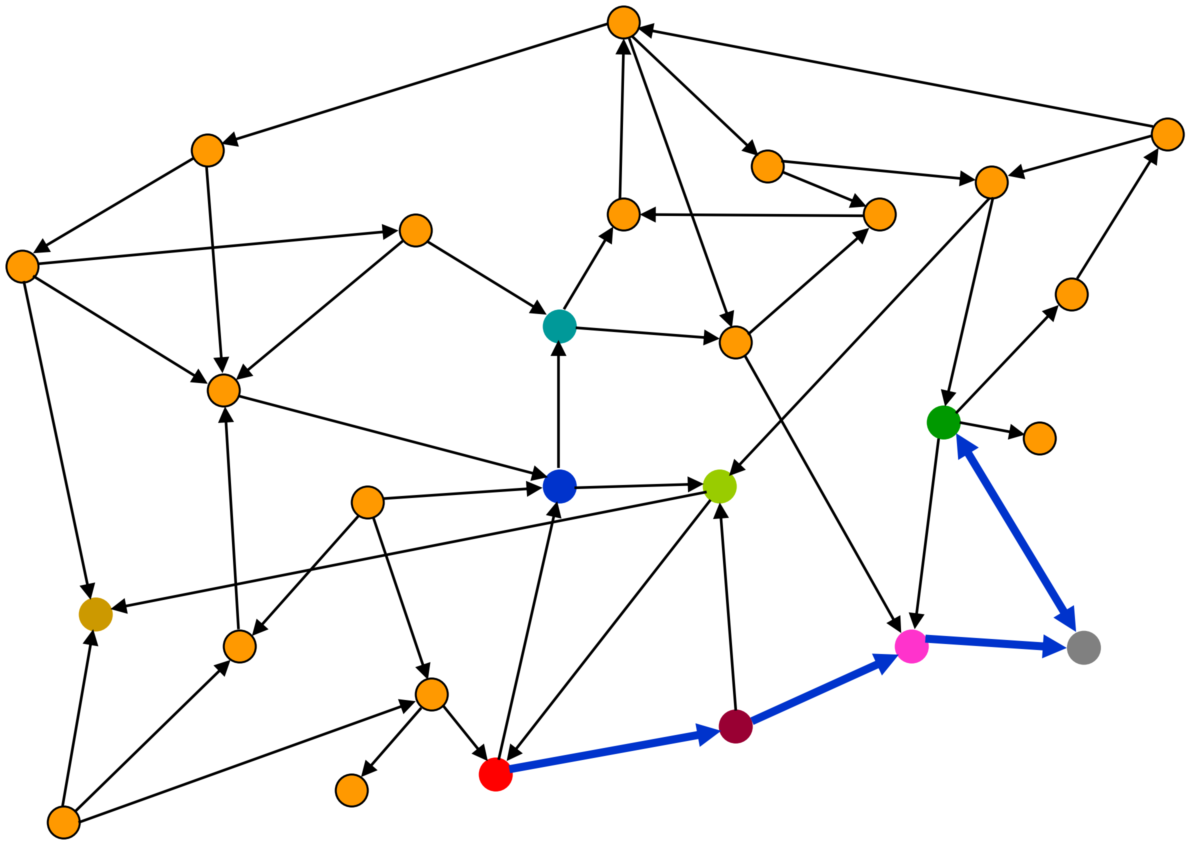


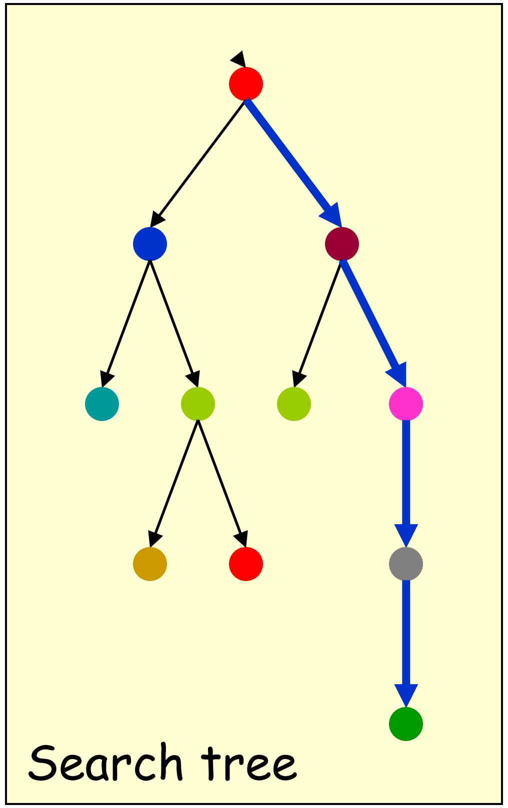
Searching the State Space



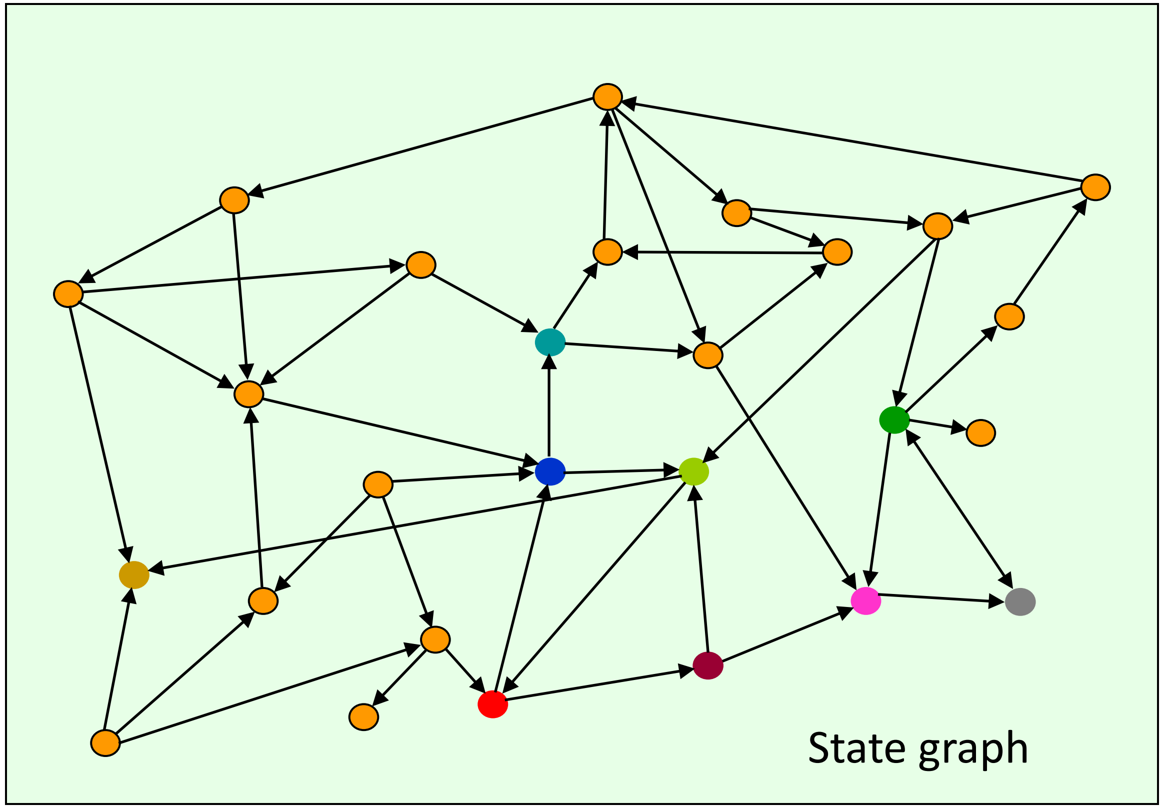


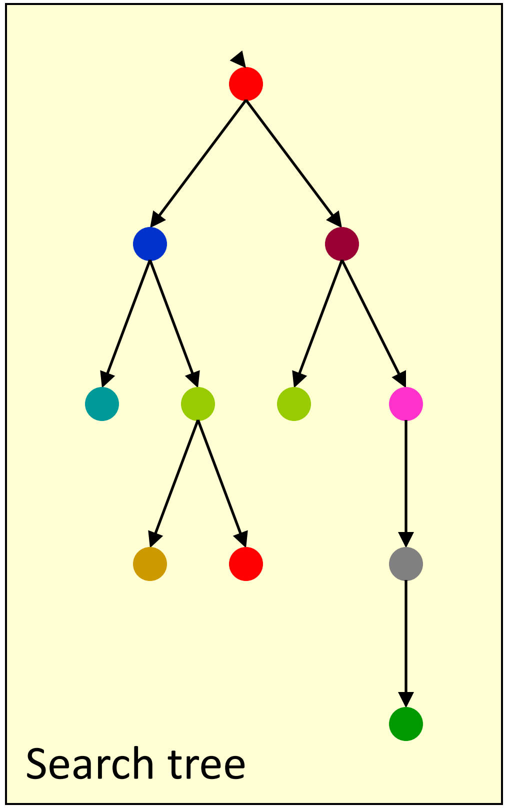
Searching the State Space





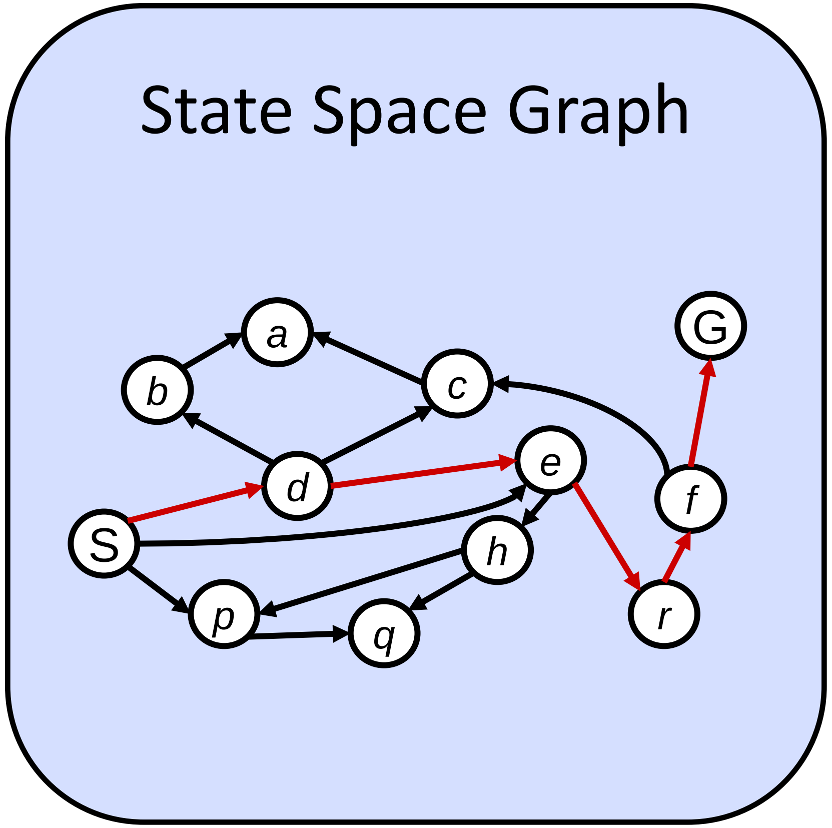
Search Tree





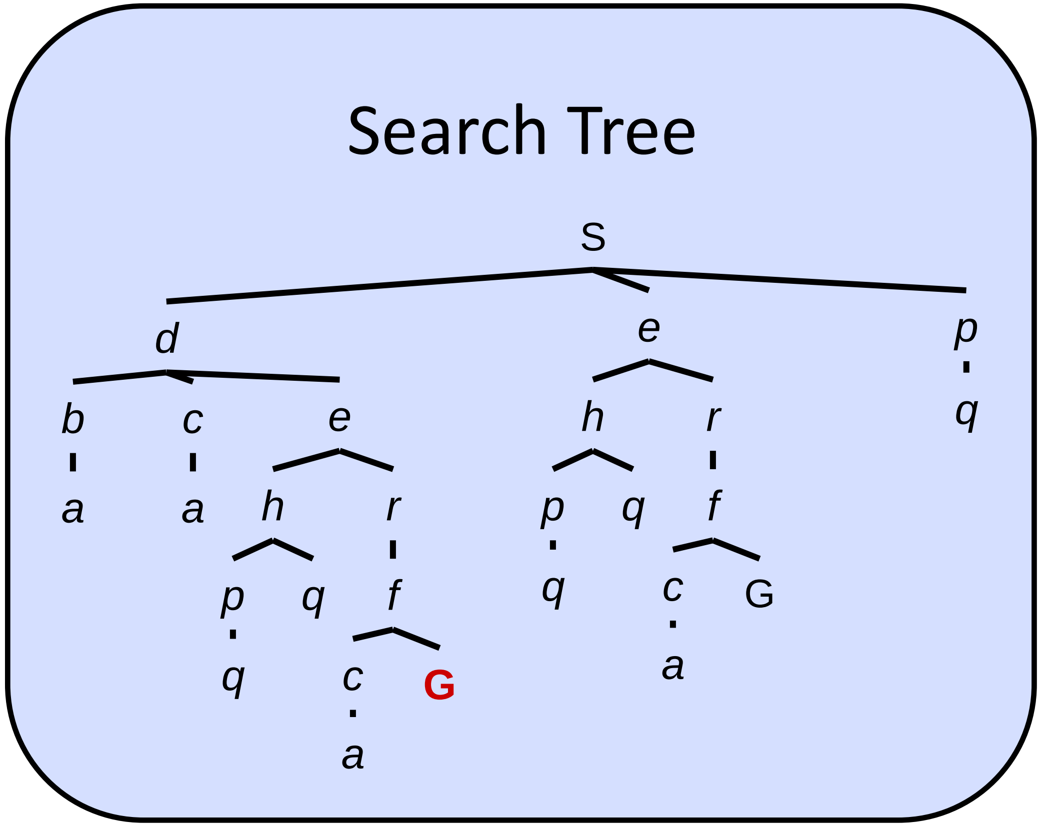
Note that some states may be visited multiple times

State Space Graphs vs. Search Trees



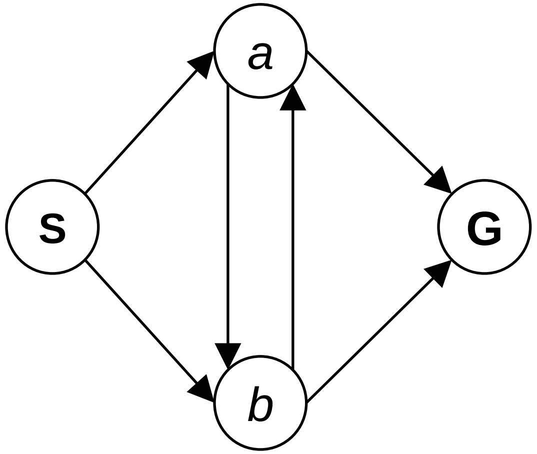
Each NODE in in the search tree is an entire PATH in the state space graph.

We construct the tree on demand – and we construct as little as possible.



Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:

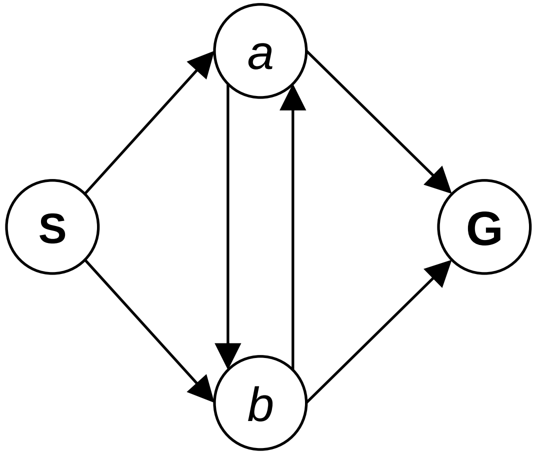


How big is its search tree (from S)?

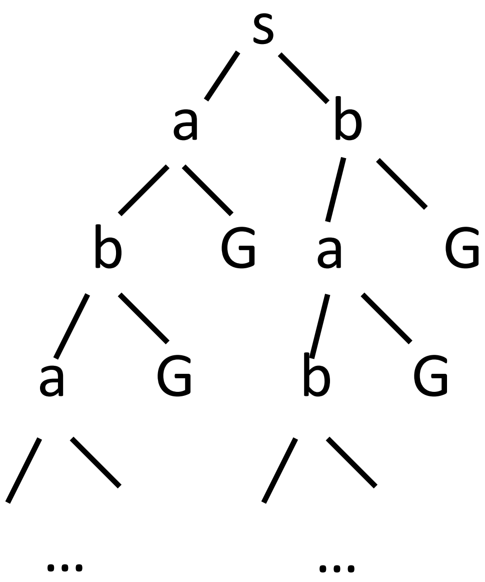
∞

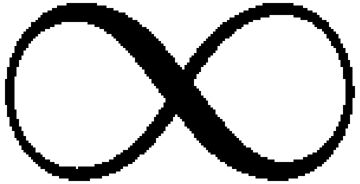
Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:



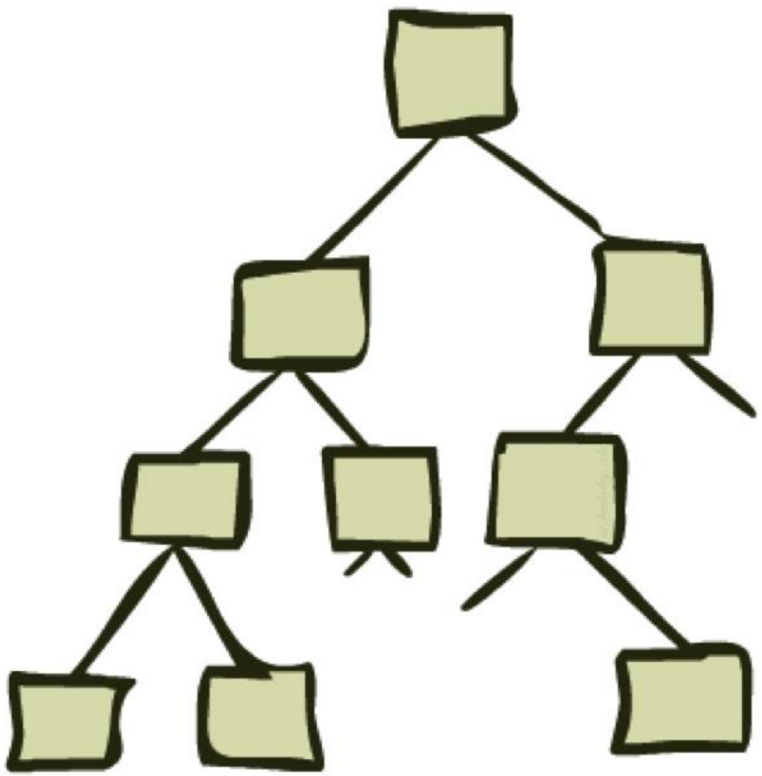
How big is its search tree (from S)?





Important: Those who don’t know history are doomed to repeat it!

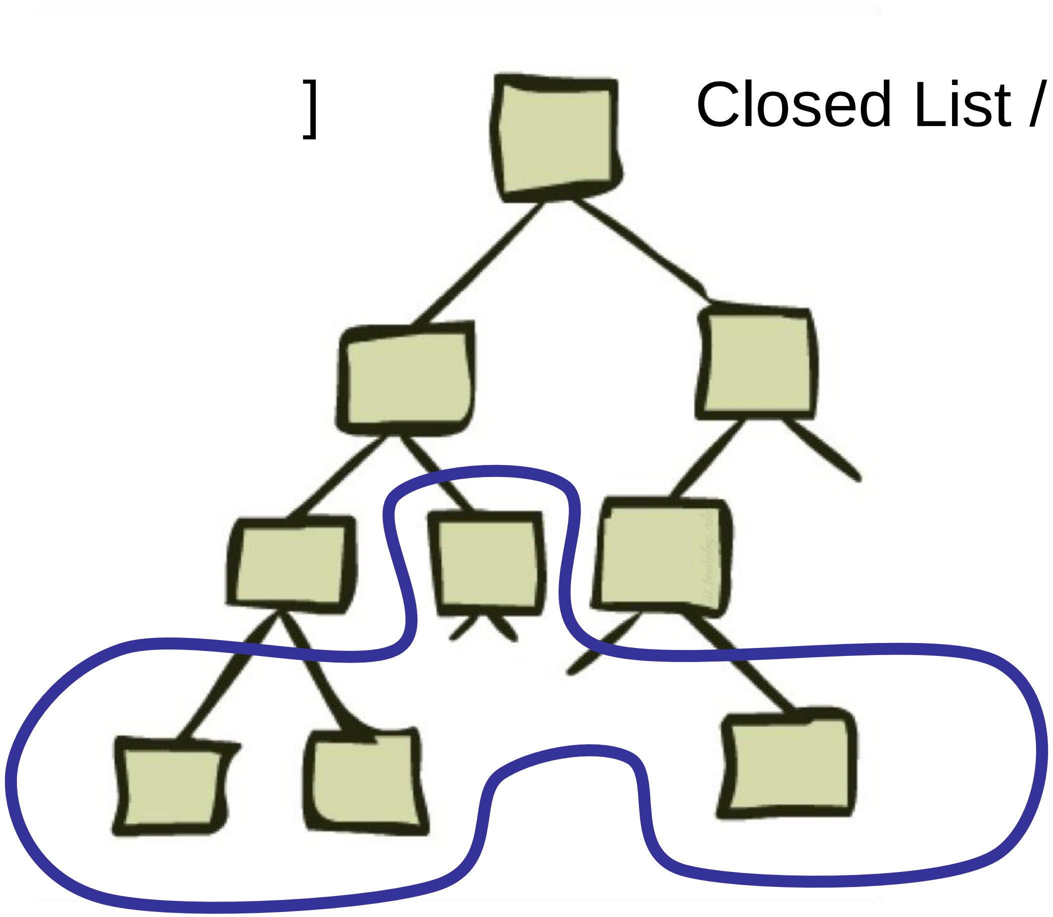
Search Trees



Search Trees

Open List / Fringe: [ ]

Closed List / Visited: [ ]



Search Problems Are Models

Problem solving

- We want:
  - To automatically solve a problem.

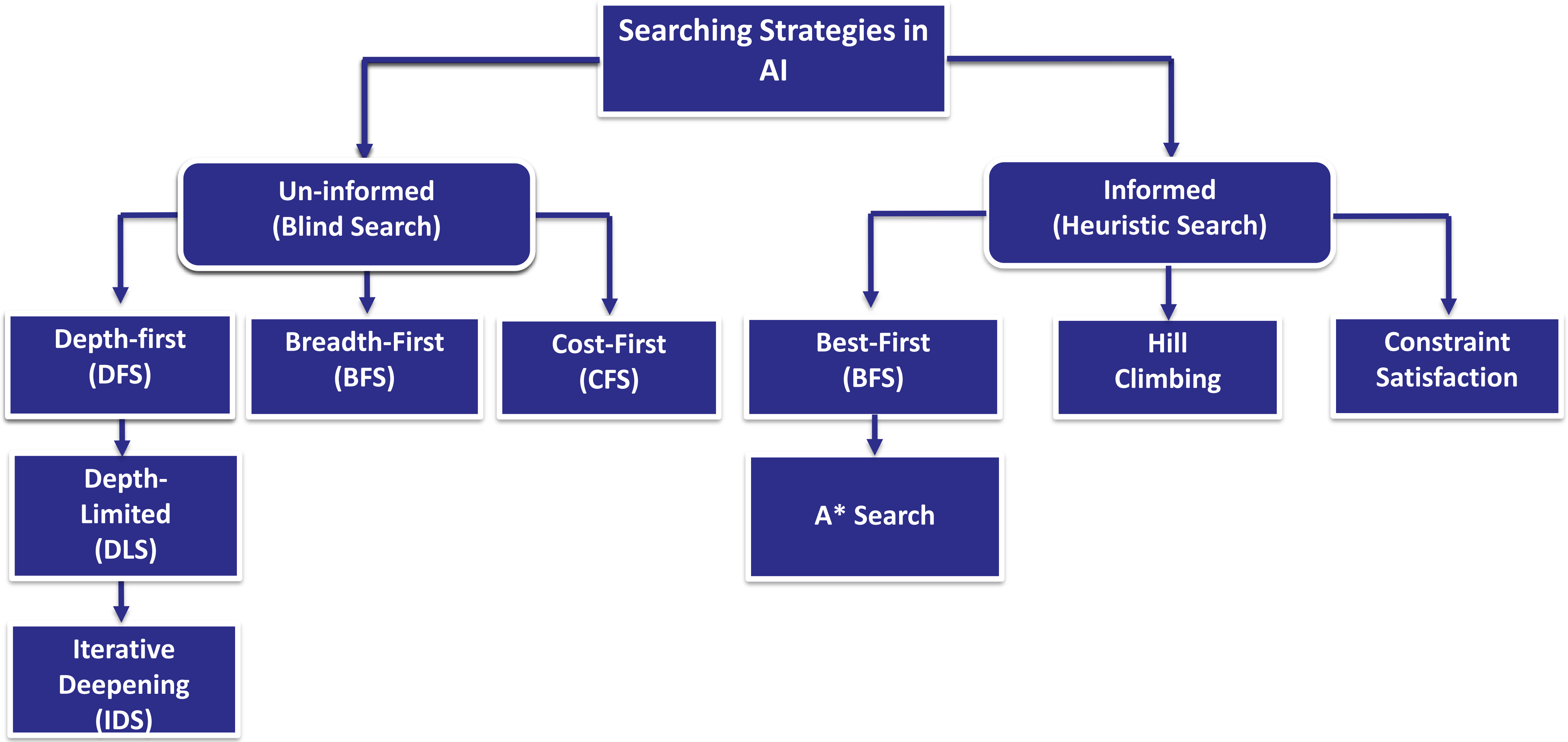
Formalization

- We need:

Searching technique

  - A representation of the problem.
  - Algorithms that use some strategy to solve the problem defined in that representation.

AI searching Strategies

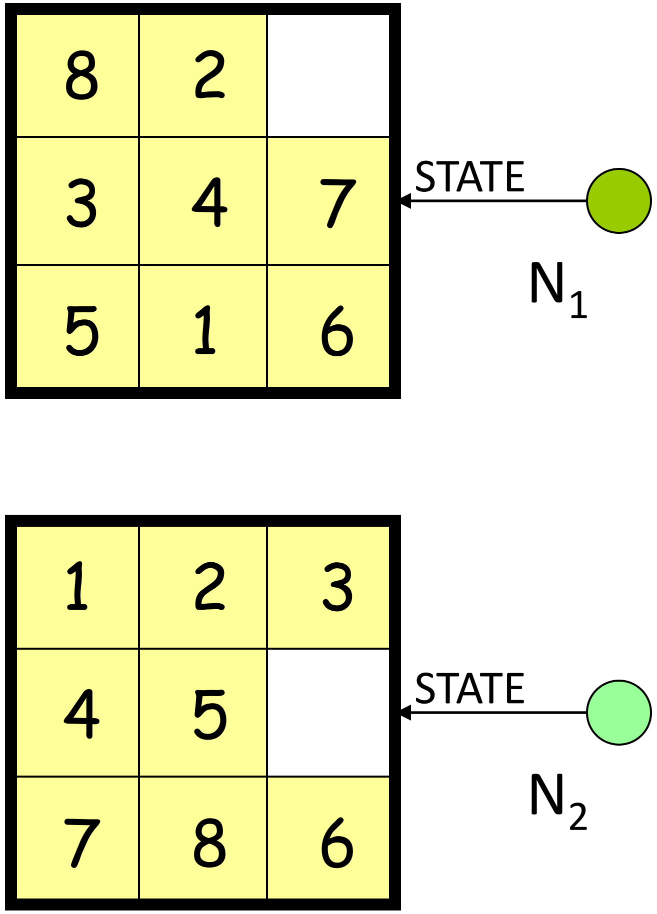


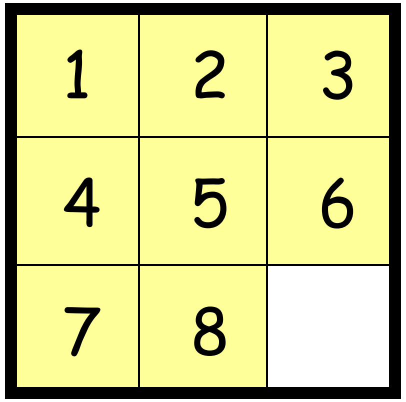
A very large number of AI problems are formulated as search problems.

Example

For a blind strategy, N_1 and N_2 are just two nodes (at some position in the search tree)

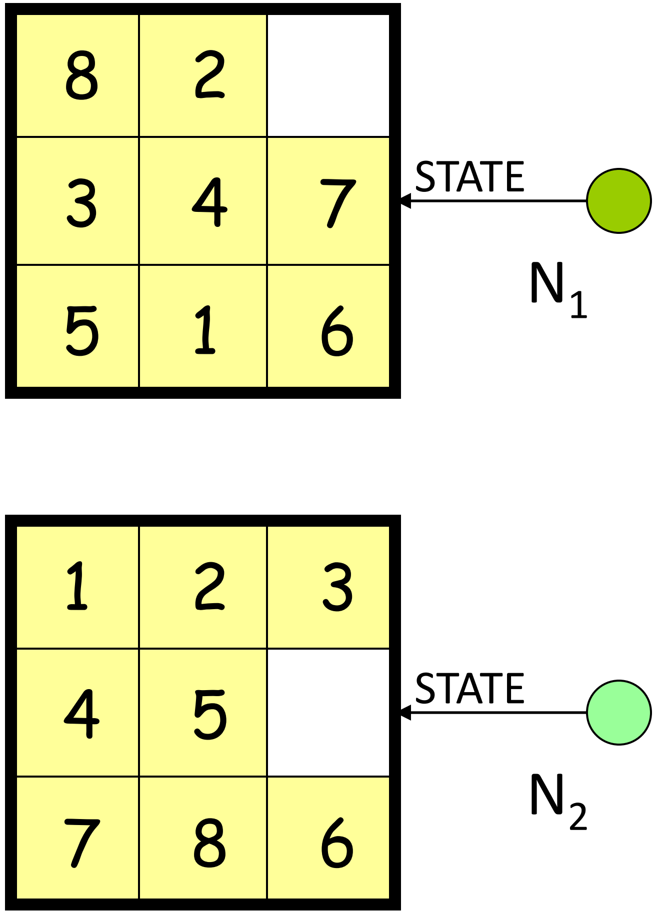
Strategies do not exploit state descriptions to order FRINGE





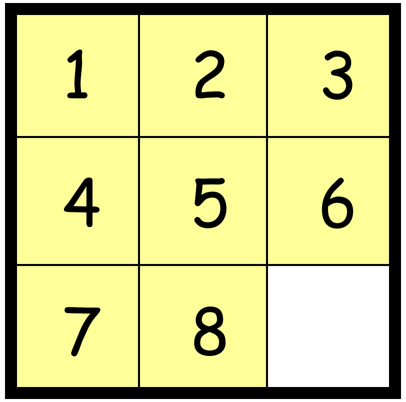
Goal state

Example



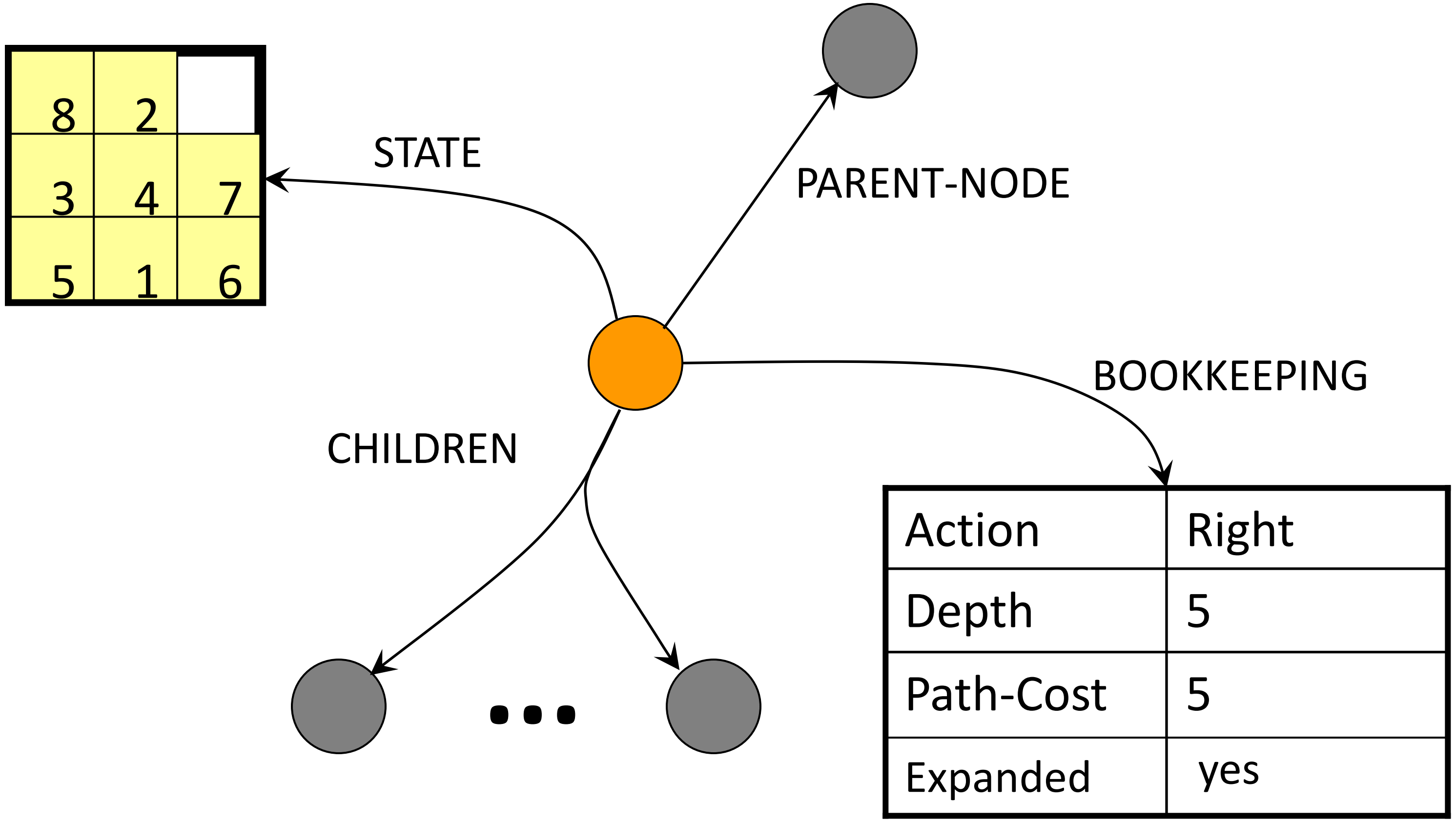
For a heuristic strategy counting the number of misplaced tiles, N_2 is more promising than N_1

Strategies exploit state descriptions to order FRINGE



Goal state

Data Structure of a Node



Depth of a node N = length of path from root to N

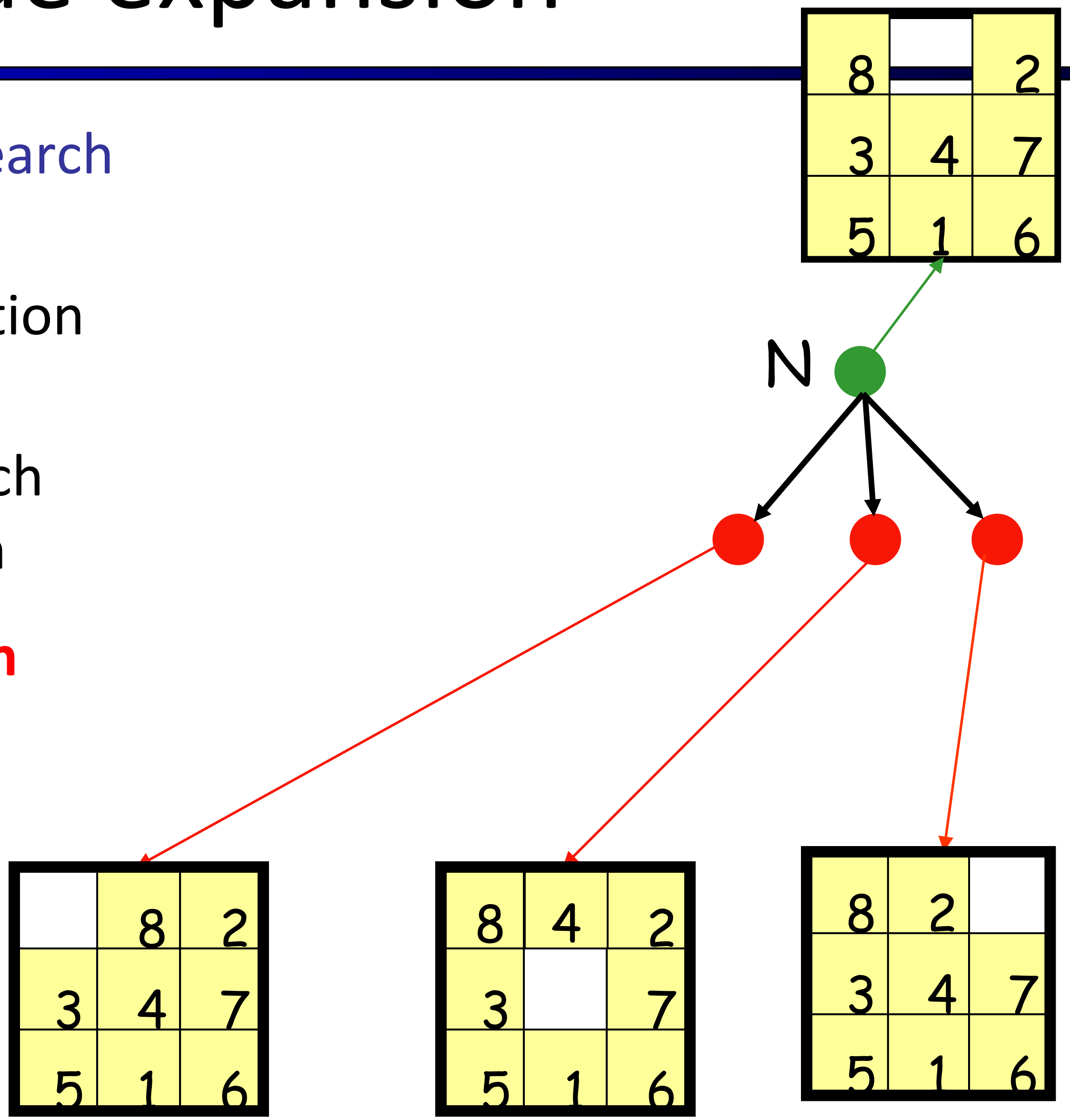
(depth of the root = 0)

Node expansion

The expansion of a node N of the search tree consists of:

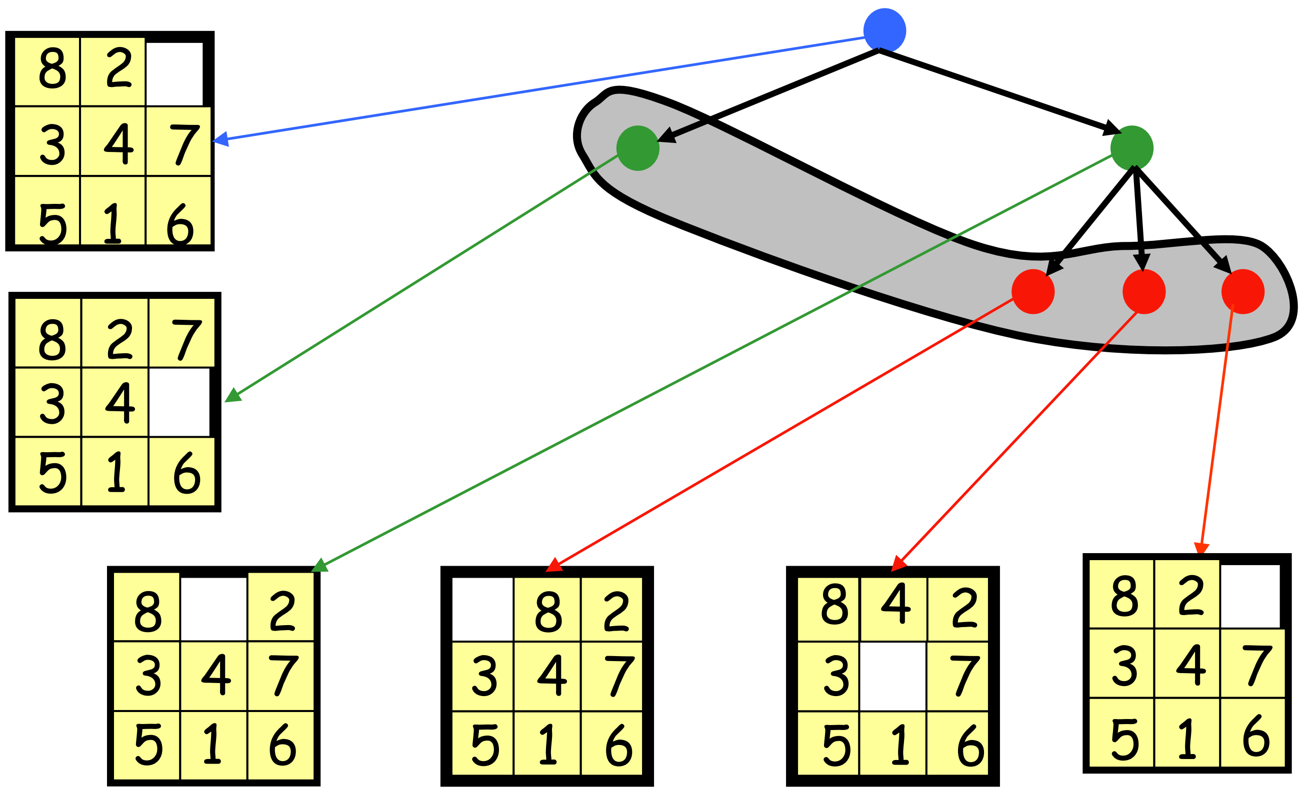
1) Evaluating the successor function on STATE(N)
2) Generating a child of N for each state returned by the function

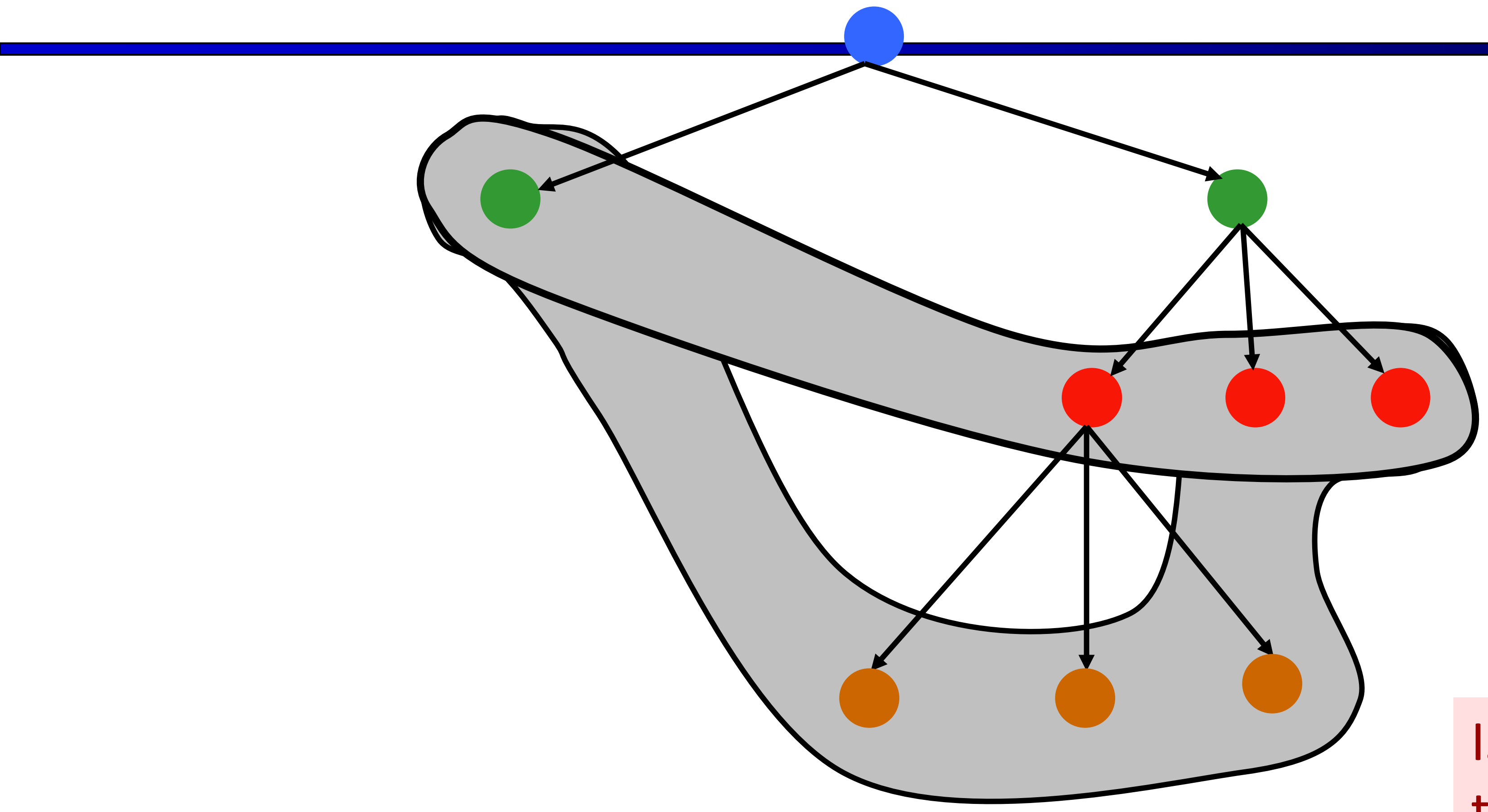
node **generation** ≠ node **expansion**



Fringe of Search Tree

- The fringe is the set of all search nodes that haven't been expanded yet





Is it identical to the set of leaves?

Search Strategy

- Fringe:

  Set of all search nodes that haven't been expanded yet

- Implemented as a priority queue FRINGE
  - INSERT(node, FRINGE)
  - REMOVE(FRINGE)
- The ordering of the nodes in FRINGE defines the search strategy

Performance Measures

- Completeness:

  A search algorithm is complete if it finds a solution whenever one exists
  [What about the case when no solution exists?]

- Optimality:

  A search algorithm is optimal if it returns a minimum-cost path whenever a solution exists

- Complexity:

  It measures the time and amount of memory required by the algorithm

General Graph Search

```
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
```

- Main variations:
  - Which leaf node to expand next
  - Whether to check for repeated states
  - Data structures for frontier, expanded nodes

General Graph Search (Python)

INPUT: Start = S, Goal = G, SearchSpace = *dictionary*

OUTPUT: Status (True/False), Path, Goal

```
GraphSearch(SearchSpace, Start, Goal) :
    OpenList = [S]
    ClosedList = [ ]
    while len(OpenList) :
        N = remove_first(OpenList)
        if N not in ClosedList : insert(N, ClosedList)
        if N == Goal : return True, N
        else:
            childrens = SearchSpace [N]
            for child in childrens :
                if child not in ClosedList and child not in OpenList :
                    OpenList = append(OpenList, child)
    return False
```

Note: this version is not saving the path for simplicity

Blind Strategies

- Breadth-first
  - Bidirectional
- Depth-first
  - Depth-limited
  - Iterative deepening

Arc cost = 1

- Uniform-Cost
  (variant of breadth-first)

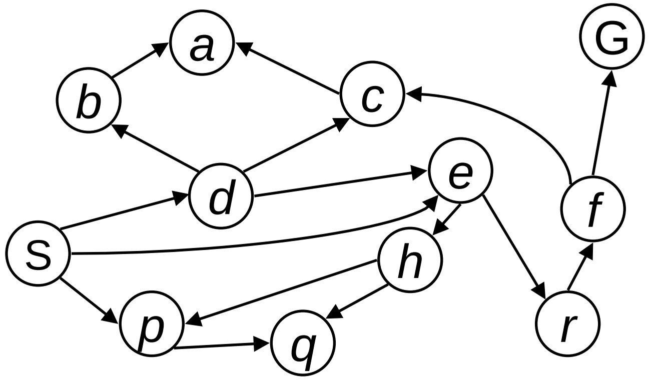
Arc cost
$= c(\text{action}) \geq \varepsilon > 0$

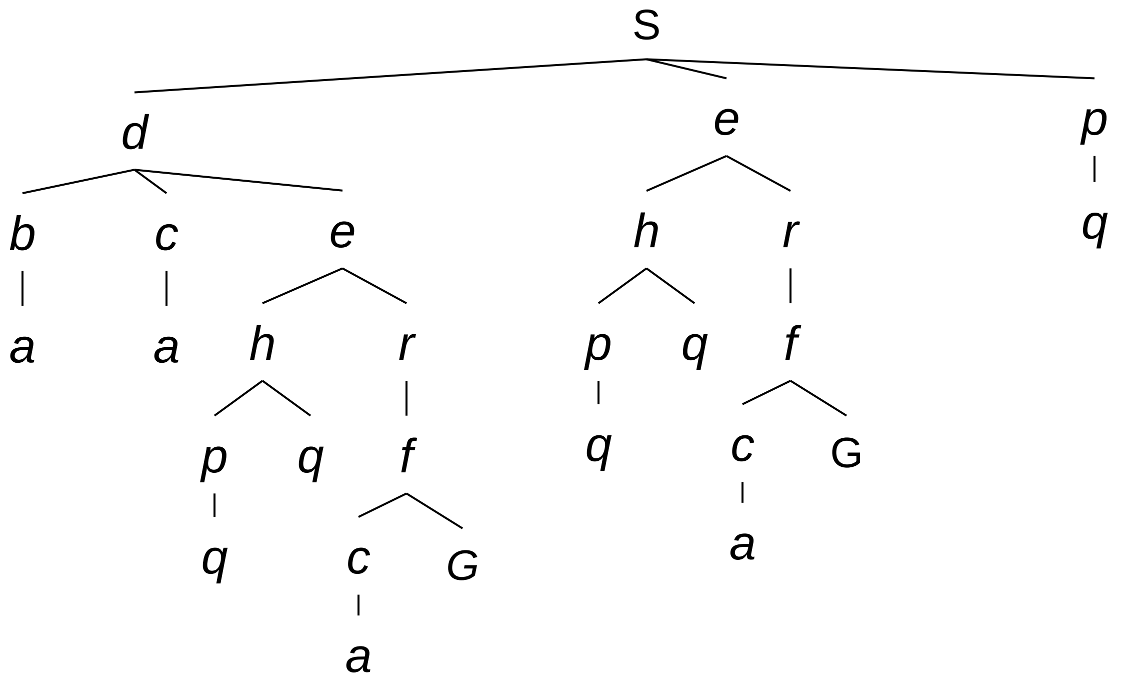
Depth-First Search

Depth-First Search

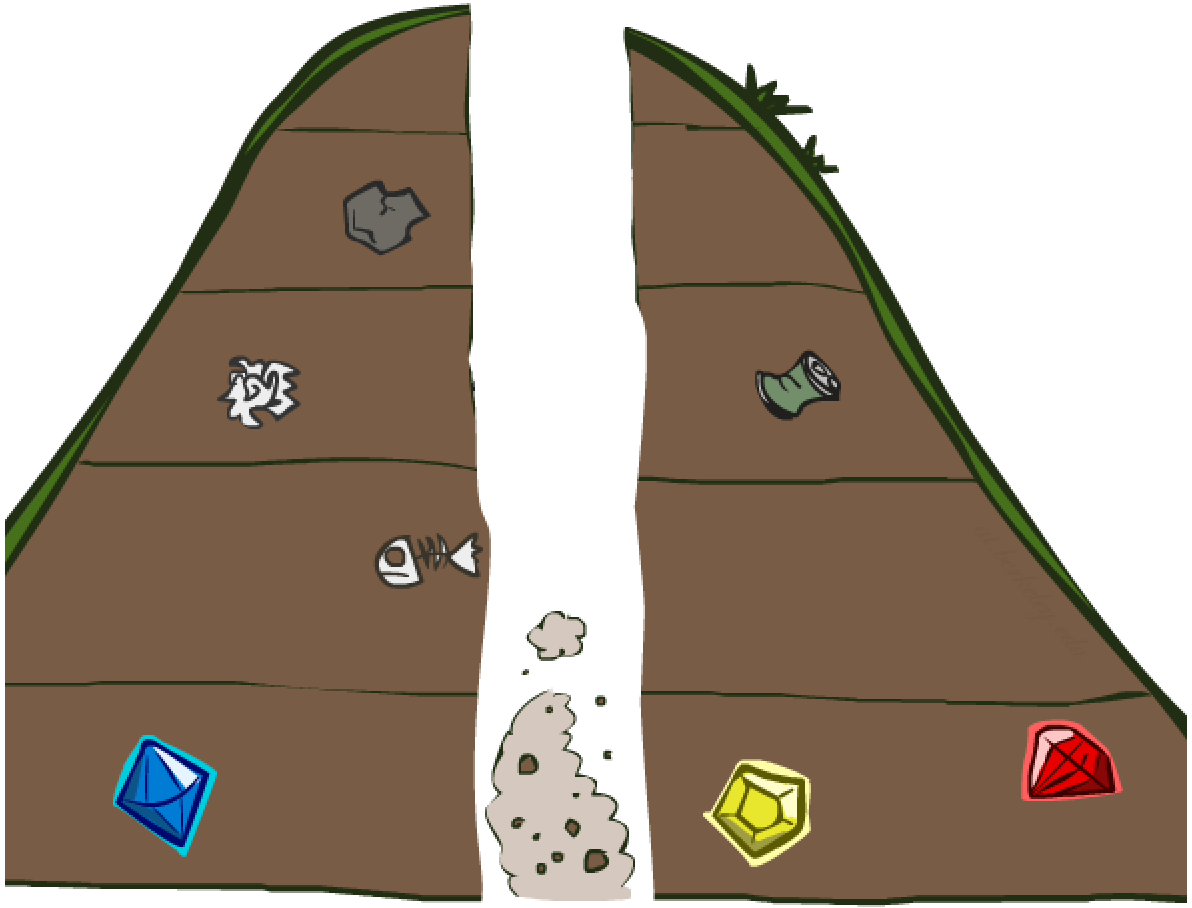
Strategy: *expand a deepest node first*

Implementation: *Fringe is a LIFO stack*



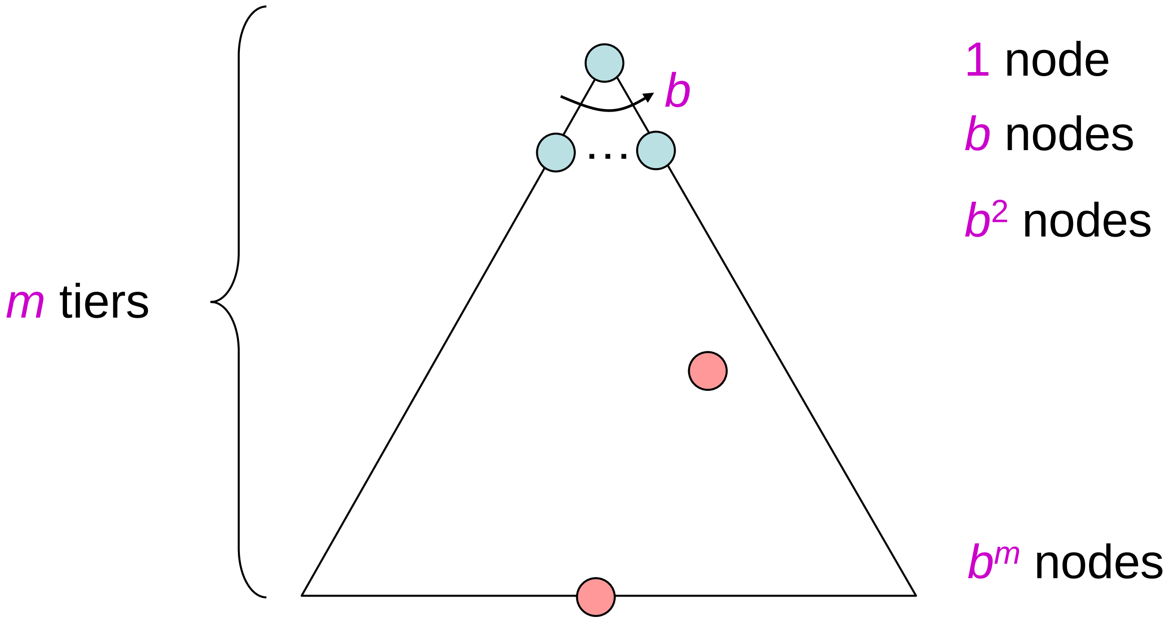


Search Algorithm Properties



Search Algorithm Properties

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

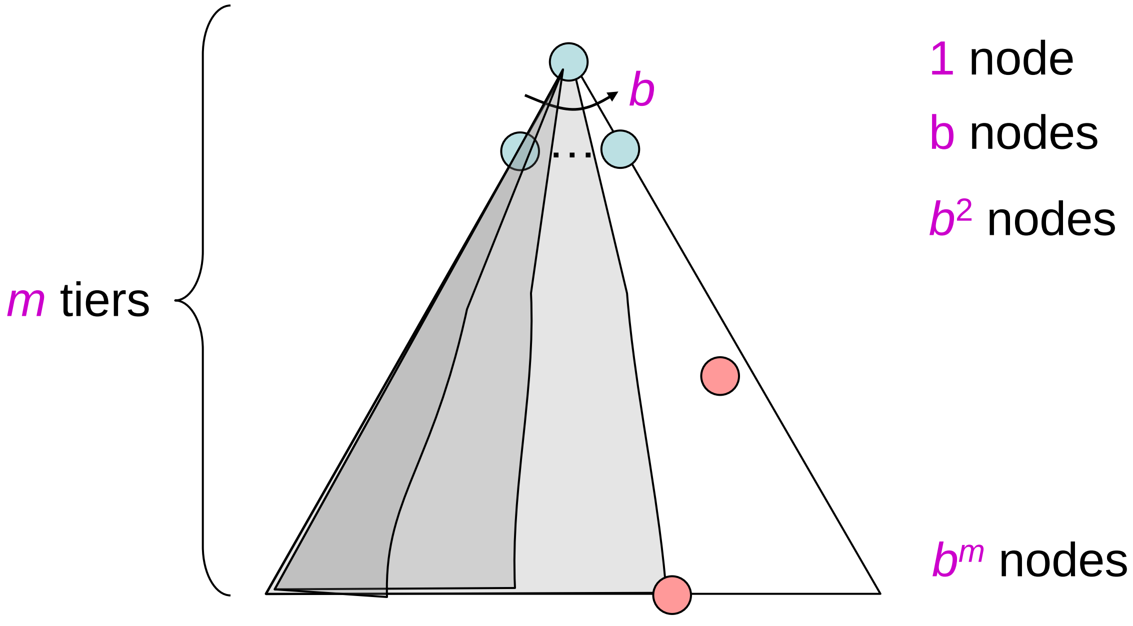


- Sketch of search tree:
  - b is the branching factor
  - m is the maximum depth
  - solutions at various depths

- Number of nodes in entire tree?
  - $1 + b + b^2 + \ldots.\ b^m = O(b^m)$

Depth-First Search (DFS) Properties

- What nodes does DFS expand?
  - Some left prefix of the tree down to depth m.
  - Could process the whole tree!
  - If m is finite, takes 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
  - preventing cycles may help (more later)
- Is it optimal?
  - No, it finds the "leftmost" solution, regardless of depth or cost



Depth-Limited Search

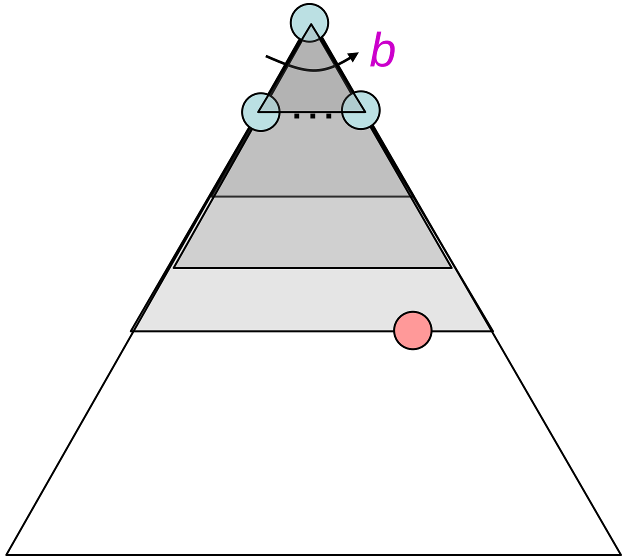
- Depth-first with depth cutoff k (depth at which nodes are not expanded)

- Three possible outcomes:
  - Solution
  - Failure (no solution)
  - Cutoff (no solution within cutoff)

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. …..

- Isn't that wastefully redundant?
  - Generally, most work happens in the lowest level searched, so not so bad!



Iterative Deepening Search

Provides the best of both breadth-first and depth-first search

Main idea:

Totally horrifying !

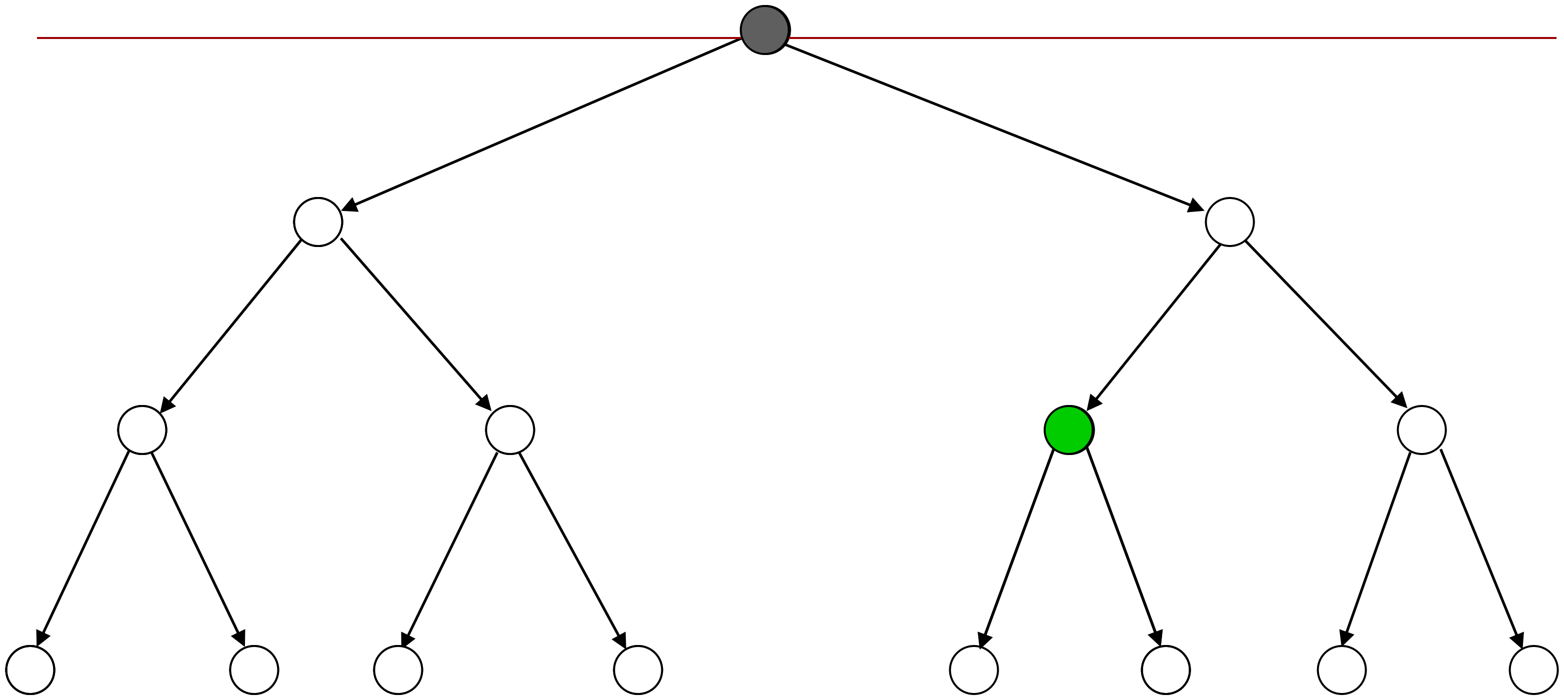
IDS

For k = 0, 1, 2, … do:

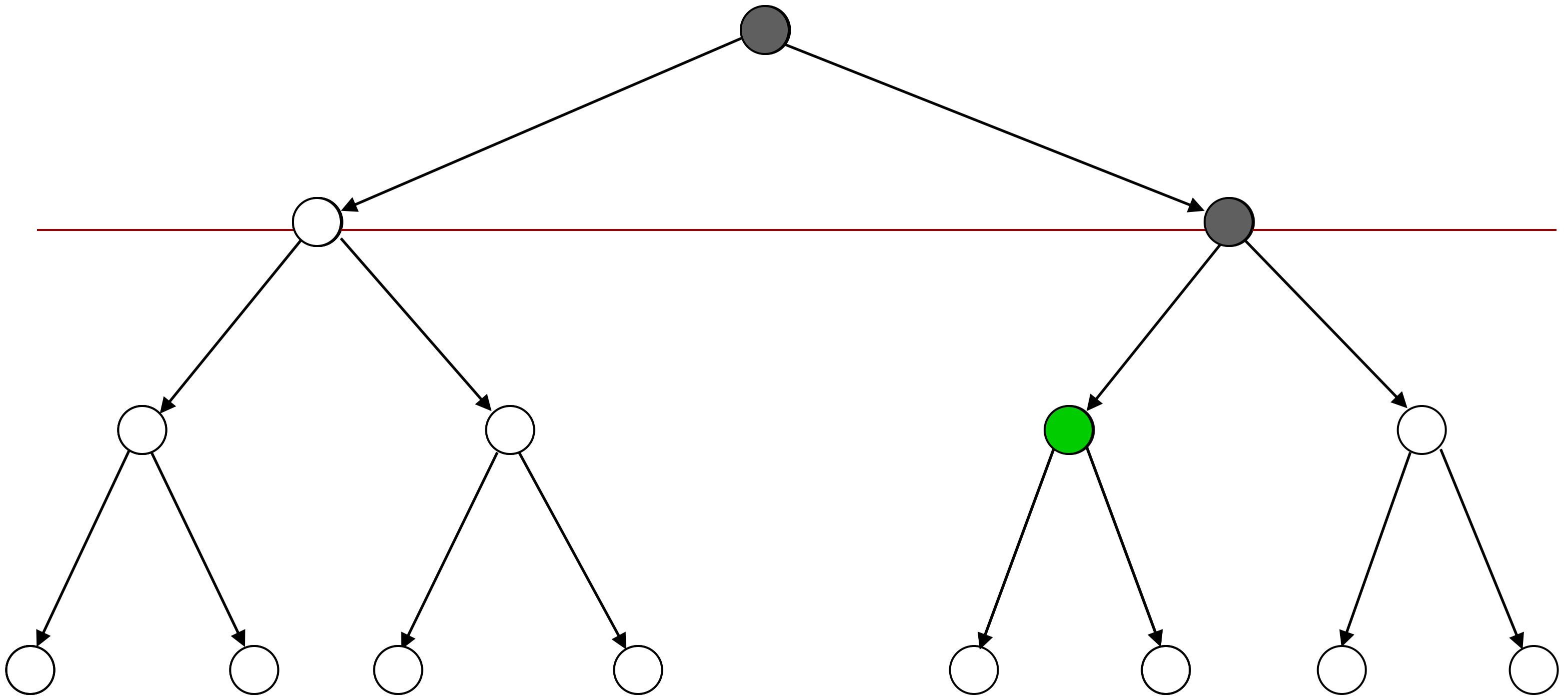
Perform depth-first search with depth cutoff k

(i.e., only generate nodes with depth $\leq$ k)

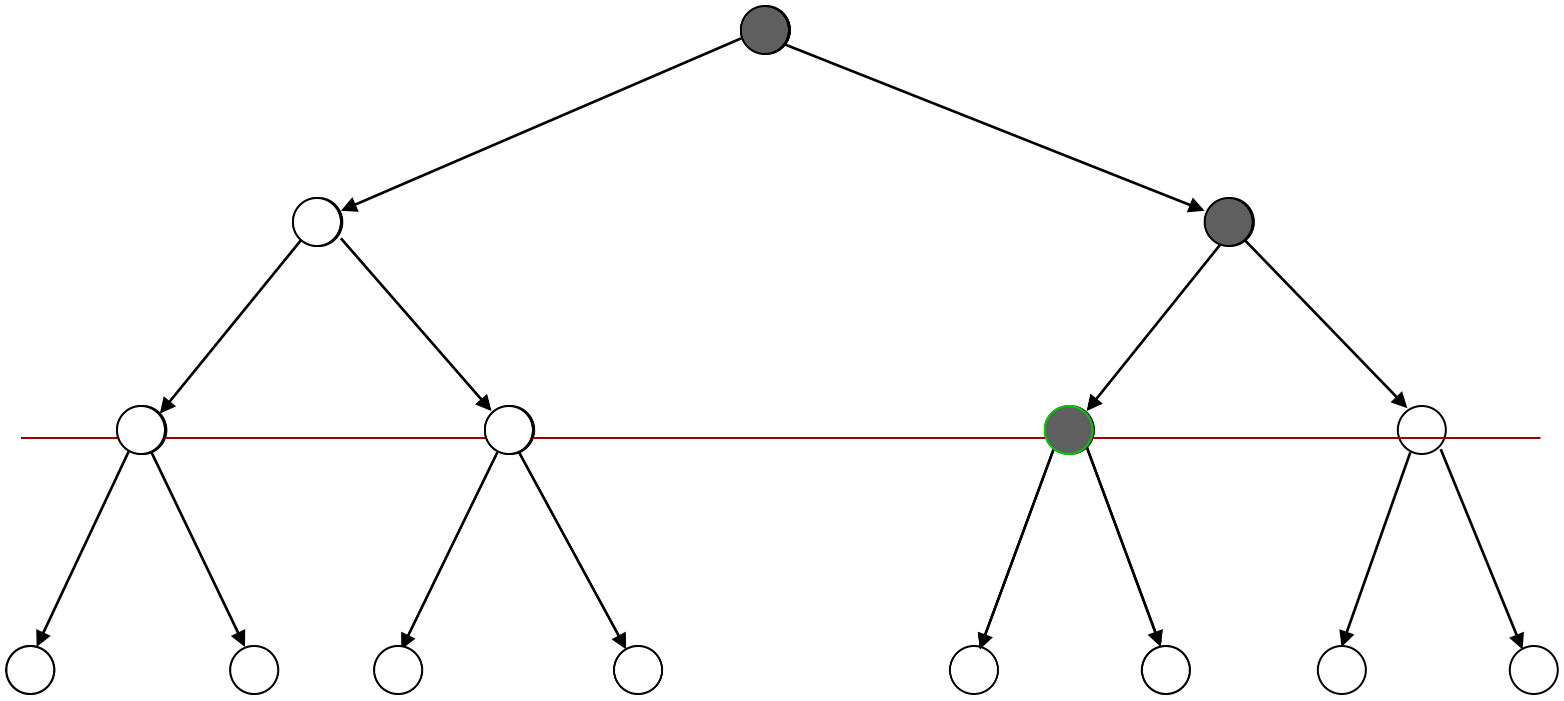
Iterative Deepening



Iterative Deepening



Iterative Deepening



Performance

- Iterative deepening search is:
  - Complete
  - Optimal if step cost =1
- Time complexity:

  $b^d + 2b^{d-1} + 3b^{d-2} + \dots + (d - 1)\, b^2 + db + (d + 1) = O(b^d)$

- Space complexity: $O(bd)$ or $O(d)$

Number of Generated Nodes (Breadth-First & Iterative Deepening)

d = 5 and b = 2

d	BF	ID
0	1	1 x 6 = 6
1	2	2 x 5 = 10
2	4	4 x 4 = 16
3	8	8 x 3 = 24
4	16	16 x 2 = 32
5	32	32 x 1 = 32
SUM	63	120

120/63 ~ 2

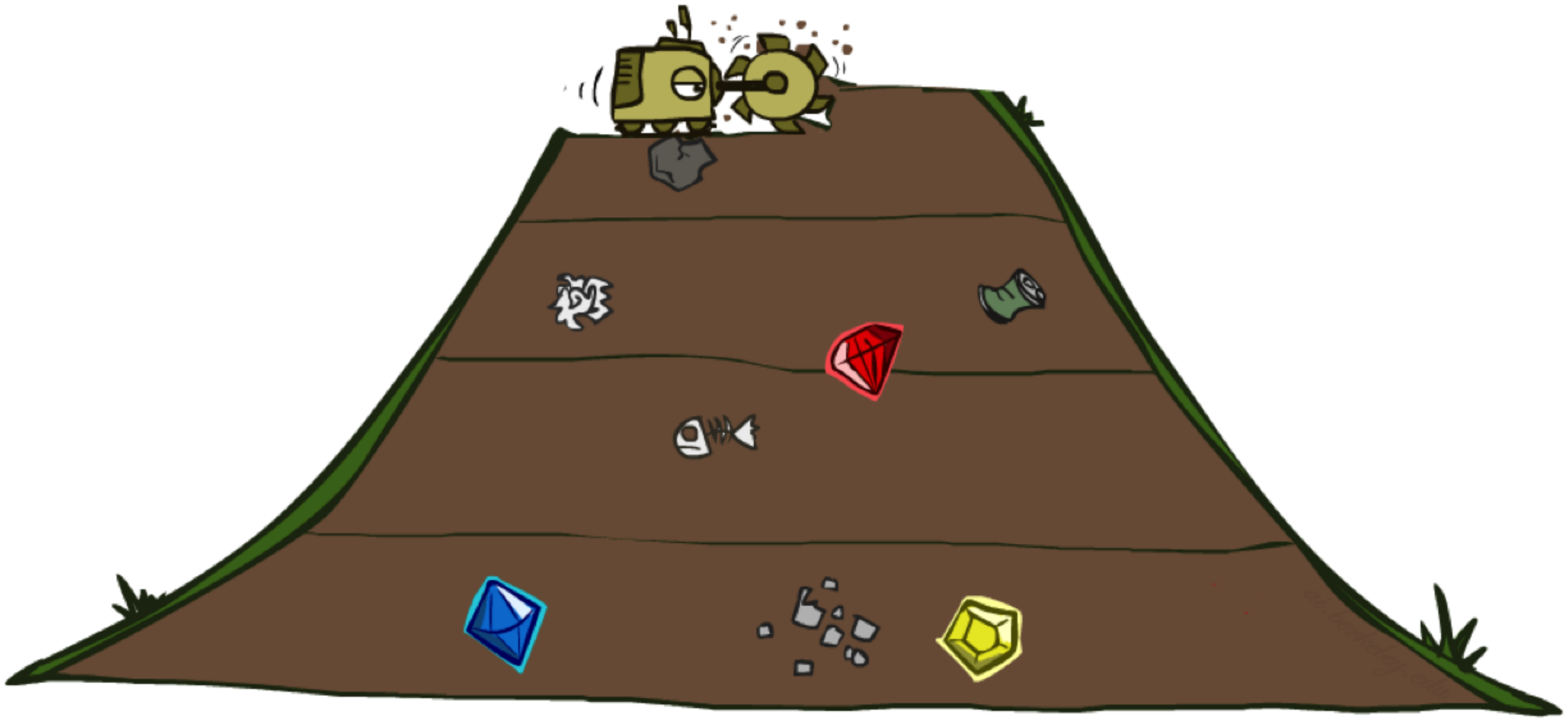
Number of Generated Nodes (Breadth-First & Iterative Deepening)

d = 5 and b = 10

d	BF	ID
0	1	6
1	10	50
2	100	400
3	1,000	3,000
4	10,000	20,000
5	100,000	100,000
SUM	111,111	123,456

123,456/111,111 ~ 1.111

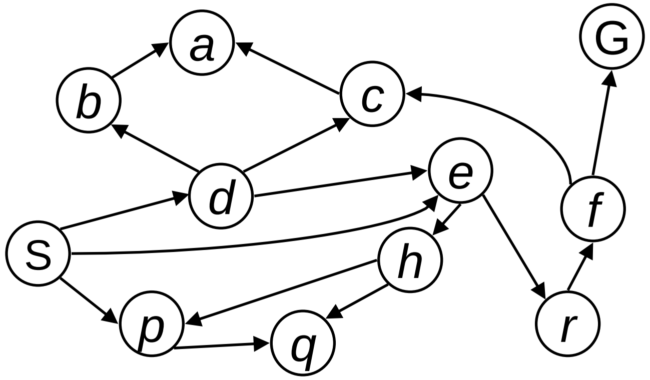
Breadth-First Search

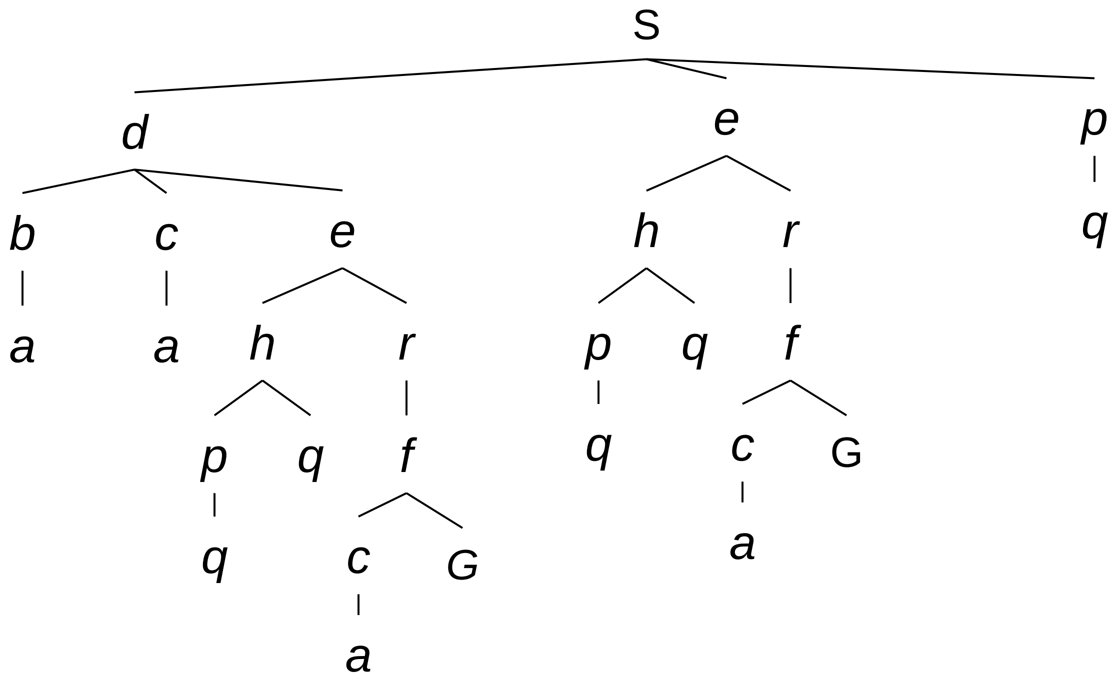


Depth-First Search

Strategy: expand a shallowest node first

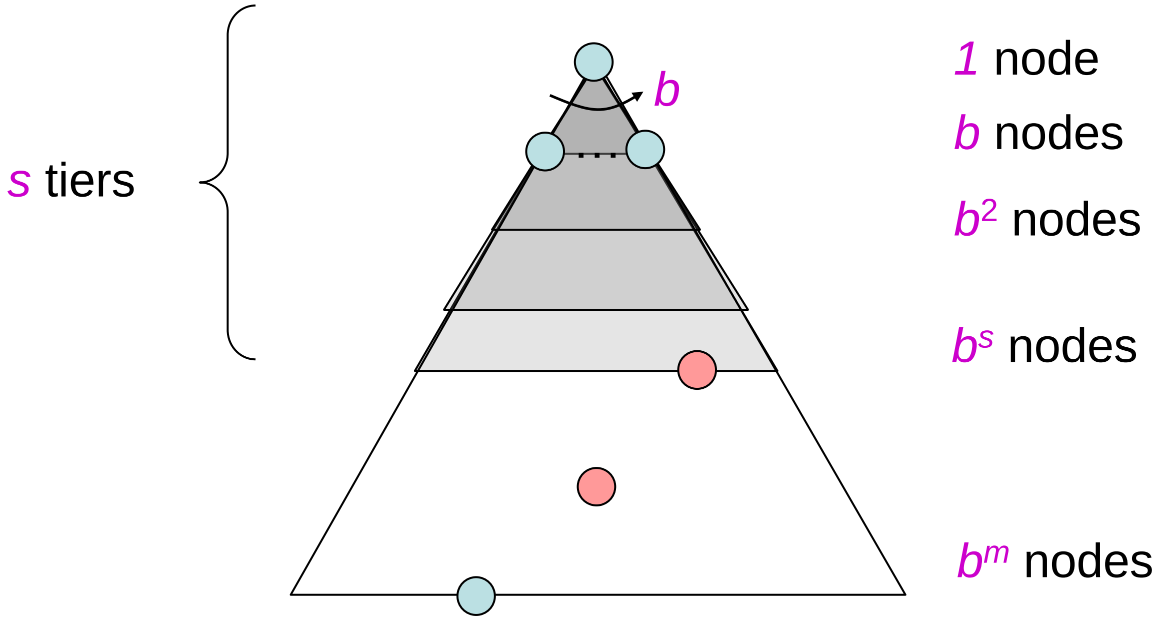
Implementation: Fringe is a FIFO queue





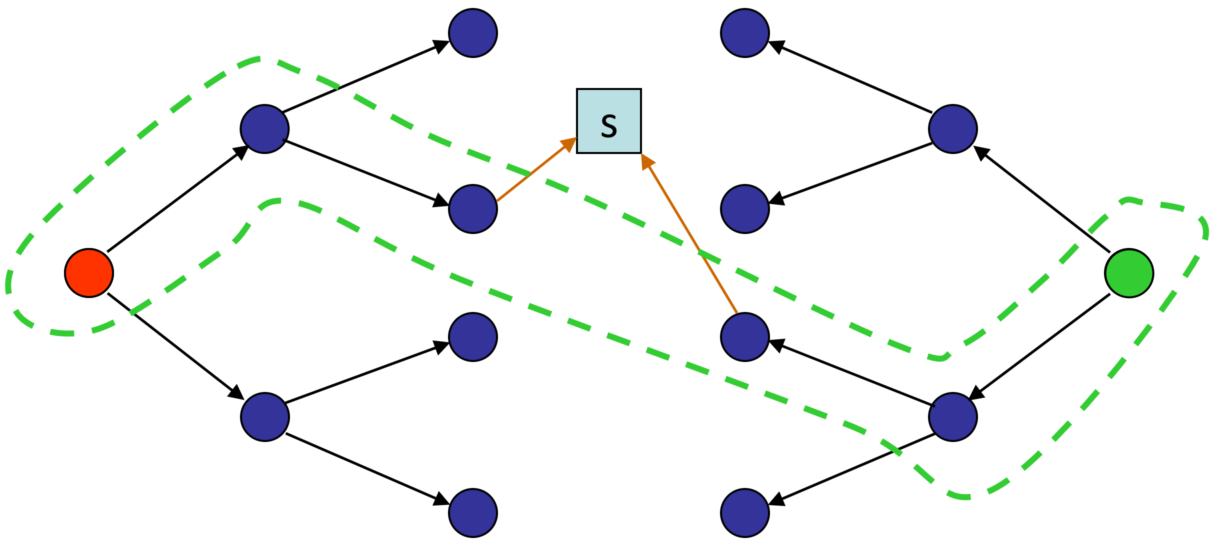
Breadth-First Search (BFS) Properties

- What nodes does BFS expand?
  - Processes all nodes above shallowest solution
  - Let depth of shallowest solution be s
  - Search takes time $O(b^s)$
- How much space does the frontier take?
  - Has roughly the last tier, so $O(b^s)$
- Is it complete?
  - s must be finite if a solution exists, so yes!
- Is it optimal?
  - If costs are equal (e.g., 1)



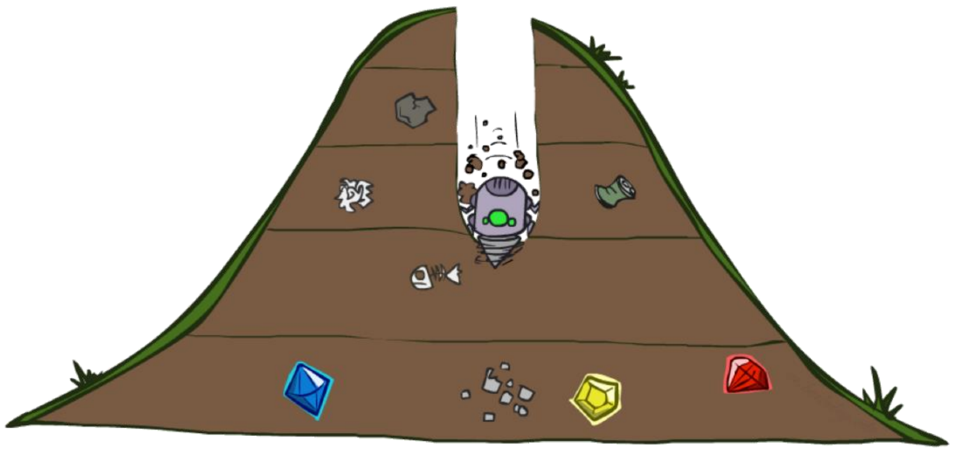
Bidirectional Strategy

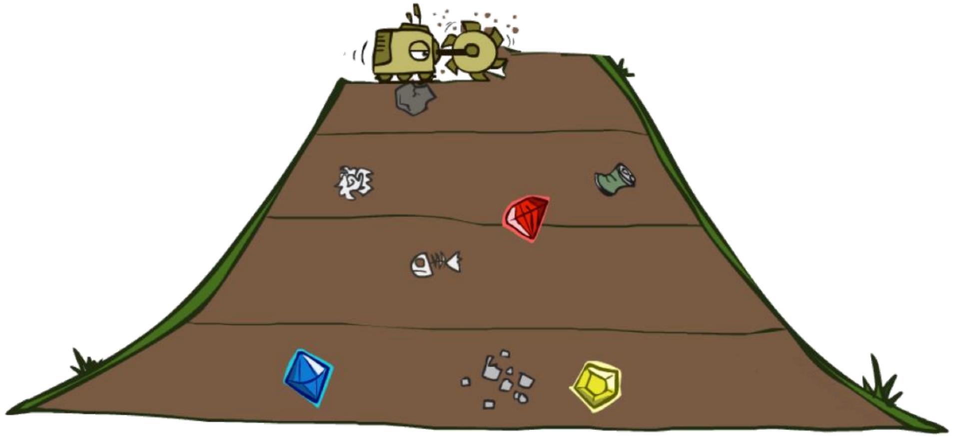
2 fringe queues: FRINGE1 and FRINGE2



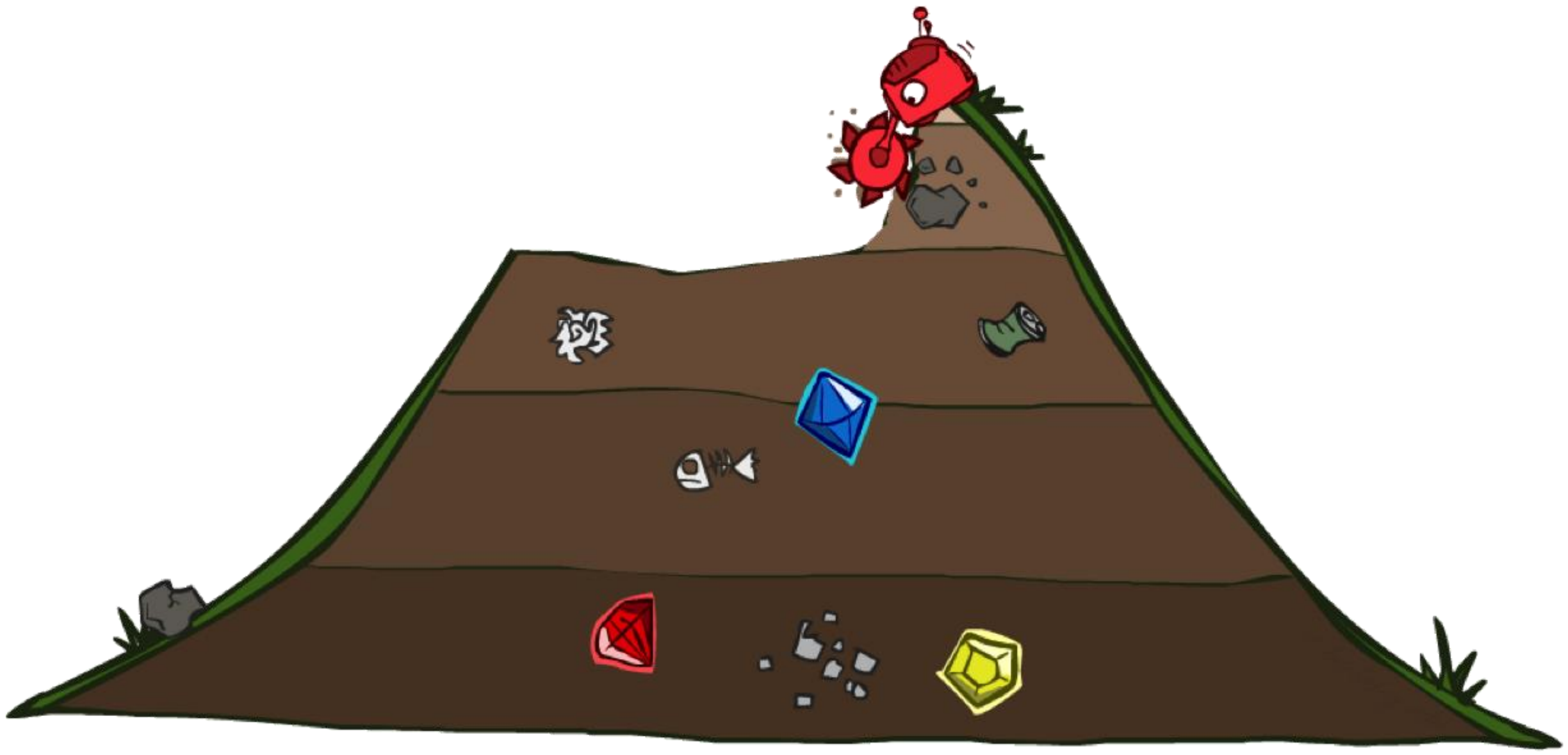
Time and space complexity is $O(b^{d/2}) << O(b^d)$
if both trees have the same branching factor b

Quiz: DFS vs BFS





Uniform Cost Search

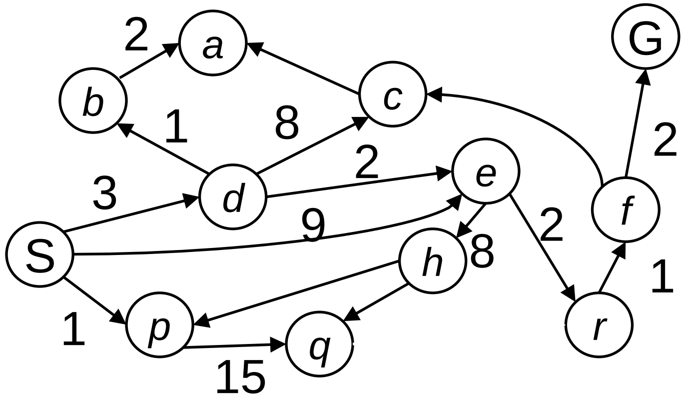


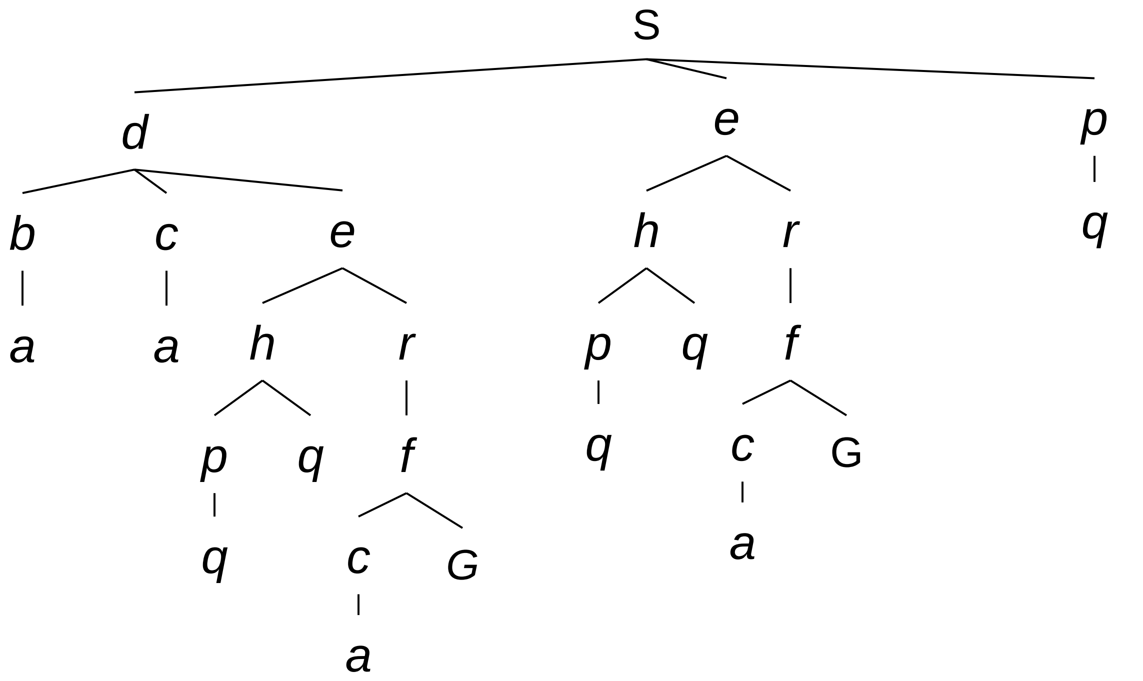
Uniform Cost Search

g(n) = cost from root to n

Strategy: expand lowest g(n)

Fringe is a priority queue sorted by g(n)



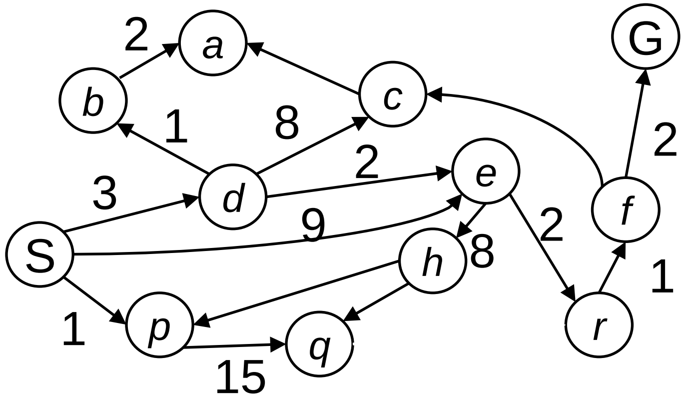


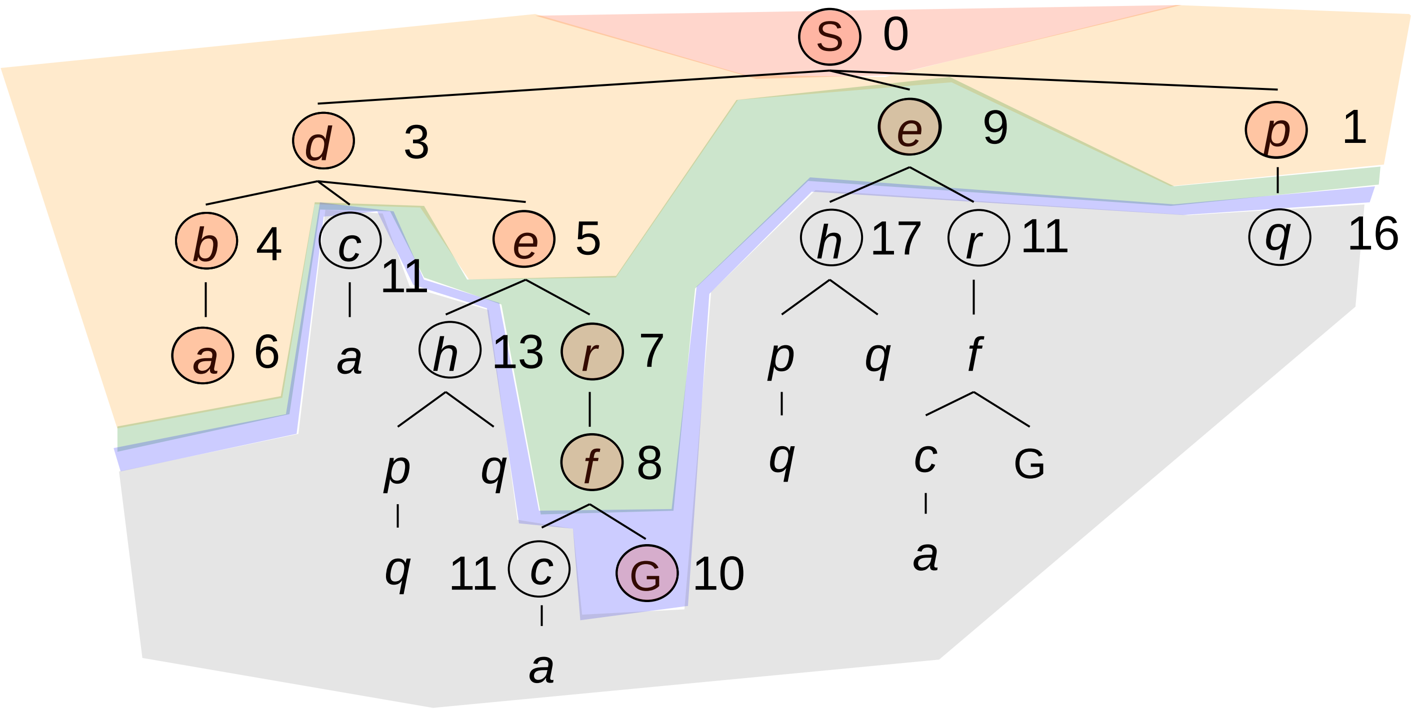
Uniform Cost Search

g(n) = cost from root to n

Strategy: expand lowest g(n)

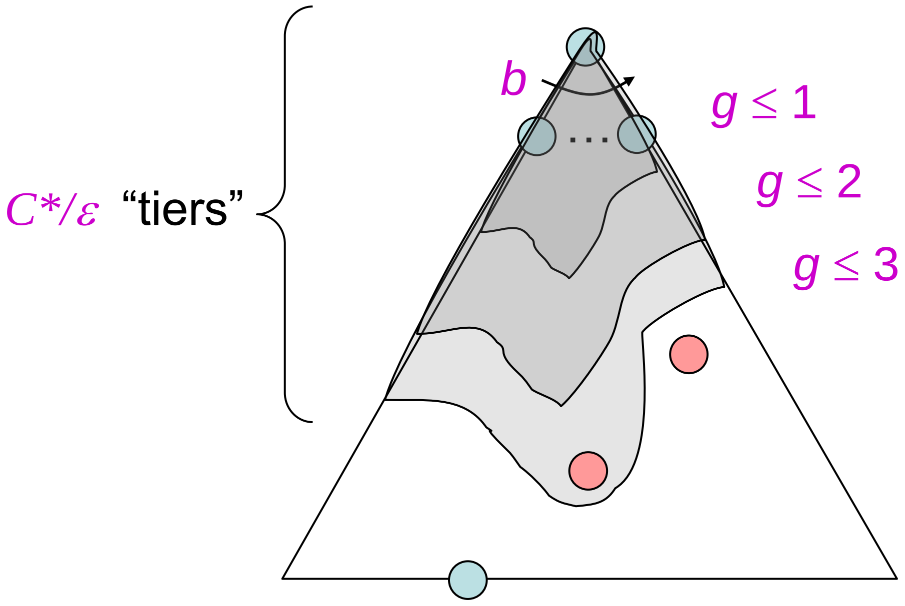
Fringe is a priority queue sorted by g(n)





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 C^* is finite and $\varepsilon > 0$, yes!
- Is it optimal?
  - Yes! (Proof next lecture via A*)



Comparison of Strategies

Measures	DFS	DLS	IDS	BFS	BDS	UCS
Time	b^d	b^l	b^d	b^d	$b^{\frac{d}{2}}$	b^d
Space	bm	b^l	bd	b^d	$b^{\frac{d}{2}}$	b^d
Optimal	No	No	Yes	Yes	Yes	Yes
Complete	No	$if\ l \geq d$	Yes	Yes	Yes	Yes