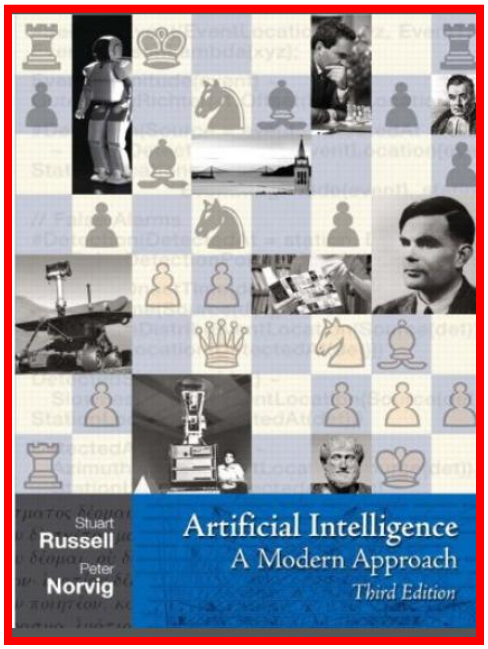


(Chapter-3)

Problem Solving and Search

Yanmei Zheng

WHY SEARCH?

- **Search** : Finding a good/best solution to a problem amongst many possible solutions.

- Many AI problems can be posed as search
- If goal found=>success; else, failure

- Not just city route search
  - Many AI problems can be posed as search
- **Game-playing**:
  Sequence of moves to win a game.
- **Speech Recognition**
  Sequence of moves to recognize the speech

O **Shortest path** on a map.

CHAPTER OUTLINE

- **Problem-solving agents**
- **Problem types, formulation & Examples**
- **Basic search algorithms**

1. **Uninformed search** algorithms (**blind search**)
   - (these algorithms are given no information about the problem other than its definition)
2. **Informed search** algorithms (**heuristic search**)
   - (these algorithms have some idea of where to look for solutions and whether one non goal state is more promising than another in reaching goal)

Problem-solving agents

- The simplest agent (reflex agent) which base their actions on direct mapping from states to actions
- Disadv: such agent cannot operate well in environments for which this mapping would be too large
- But Goal based agents can achieve successes by considering future actions desirability of their outcomes
  - On kind of goal based agent called problem solving agent
- Problem solving agents: decide what to do by finding sequences of actions that lead to desirable states

Problem types, formulation & Examples

How problem is solved?

Step 1	**Goal formulation**
Step 2.	**Problem formulation** – a process of deciding what actions and states to consider
Step 3	**Search** – systematic exploration of the sequence of alternative states that appear in a problem solving process
Step 4	**Solution** – reach the right action
Step 5	**Execution** – recommended actions can be accomplished

1. Define the problem and its solution

Formulate	Search	Executes

1. Define the problem and its solution

Formulate

- Agent task is to find out which sequence of actions will get to a goal state
- Hence, before it can do this , it needs to decide what sorts of **actions** & **states** to consider

1. Define the problem and its solution

Formulate

- Ex , **if** agent will consider details **"move left foot forward an inch "** or **" turn the steering wheel one degree left", then** the agent will probably never find a way out……why?

- Because at this level of details there are too many steps to find solution

Formulate =The process of deciding actions and states to consider

Note: The type of problem formulation can have a serious influence on the difficulty of finding a solution.

1. Define the problem and its solution

Search

- Ex , **if** agent at a specific city "**Riyad**" and " **want to go Madenah**", **and there are three paths to achieve the goal then** which to select ? May be random?

- If agent has a map (additional knowledge) , finding the best choice= Search

Search Algorithm =takes problem as input and returns a solution in the form of an action sequence

1. Define the problem and its solution

Search

Requirements of a good search strategy:

1. **It causes motion**

   Otherwise, it will never lead to a solution.

2. **It is systematic**

   Otherwise, it may use more steps than necessary.

3. **It is efficient**

   Find a good, but not necessarily the best, answer.

1. Define the problem and its solution

Executes

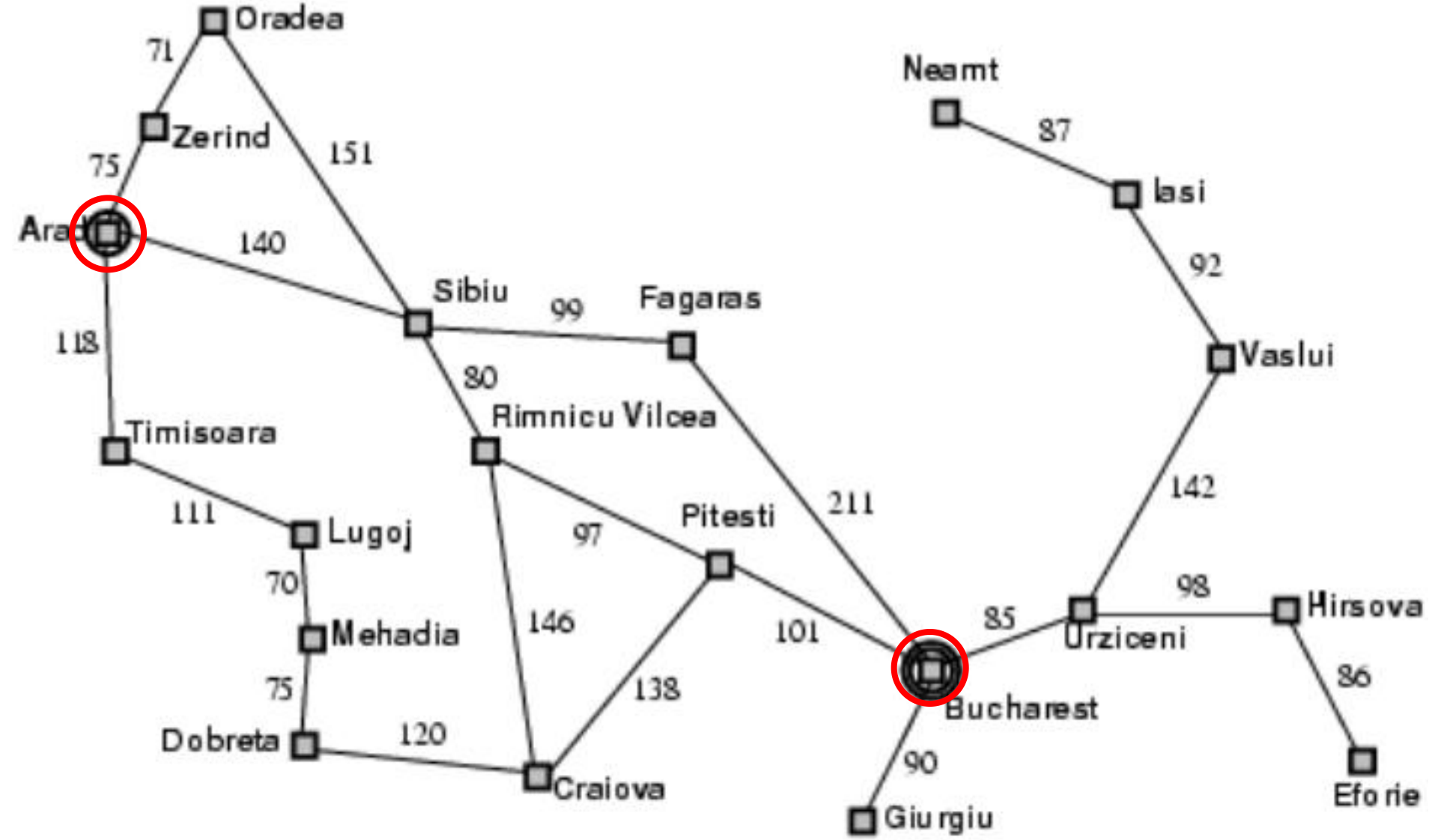
- Once a solution is found the action it recommends can be carried out

1. Define the problem and its solution

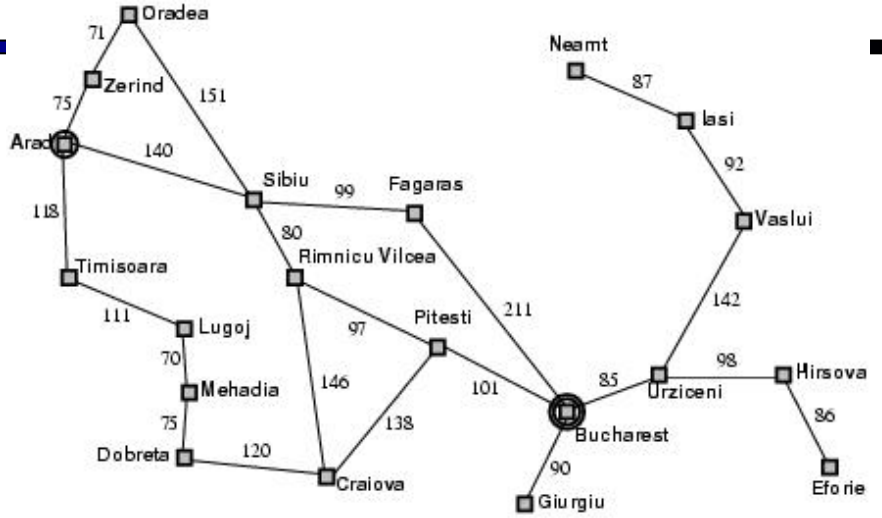
Initial state	Operator	Neighbour hood (Successor Function)	State Space	Goal test	Path cost
The **initial state** of the problem, defined in some suitable manner	**A set of actions** that moves the problem from one state to another	The set of all possible states reachable from a given state	The set of all states reachable from the initial state	A test applied to a state which returns if we have reached a state that solves the problem	How much it costs to take a particular path

Examples

Example: Traveling in Romania



State-space Problem Formulation



A problem is defined by four items:

1. initial state e.g., "at Arad“

2. actions or successor function

   S(x) = set of action–state pairs

   e.g., *S(Arad)* = {<*Arad* → *Zerind, Zerind*>, … }

3. goal test (or set of goal states)

   e.g., *x* = "at Bucharest”, *Checkmate(x)*

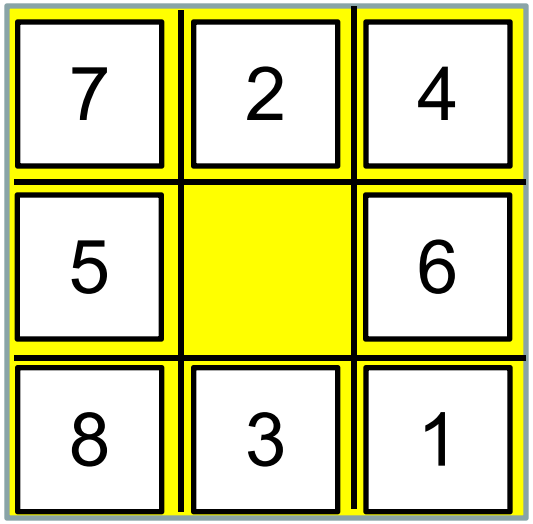
4. path cost (additive)

   e.g., sum of distances, number of actions executed, etc.

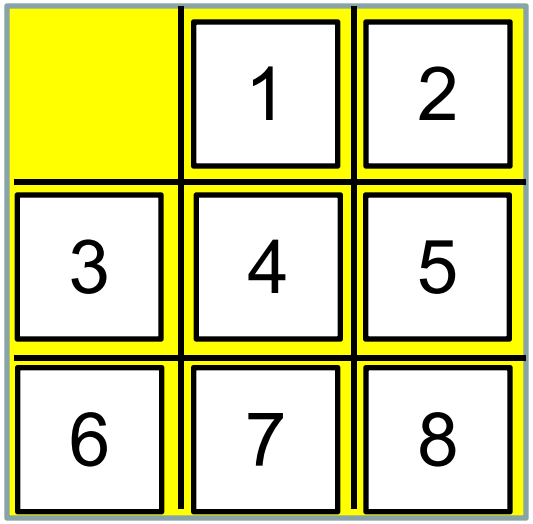
   c(x,a,y) is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state

Problem Ex: The 8-puzzle



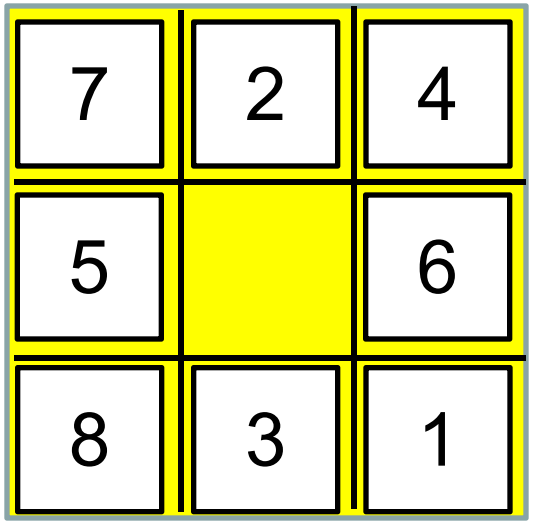
Initial state



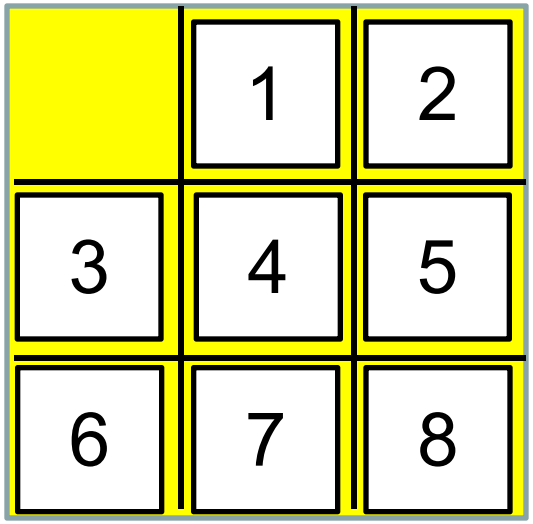
Goal state

- states?
- operators?
- goal test?
- path cost?

Real world task : the 8-puzzle



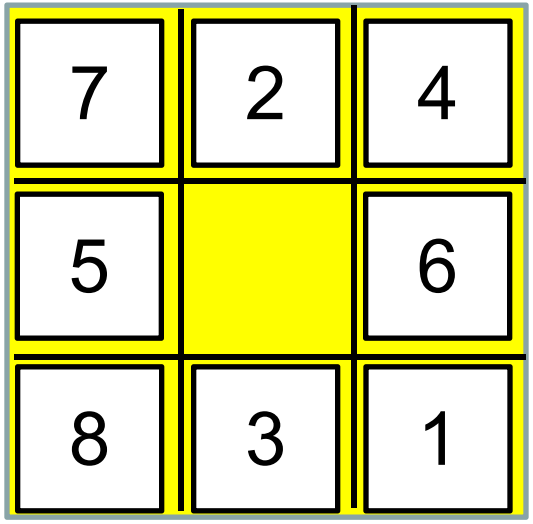
Initial state



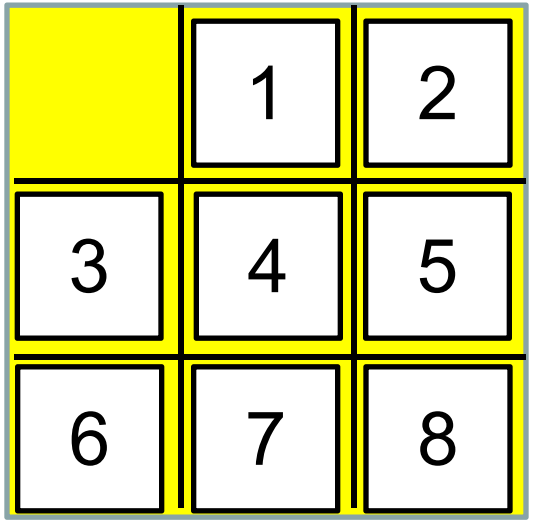
Goal state

- **State space**
  - integer locations of 8 tiles.
- **Successor function:**
  - Move blank (up, down, right, left)

8-puzzle problem



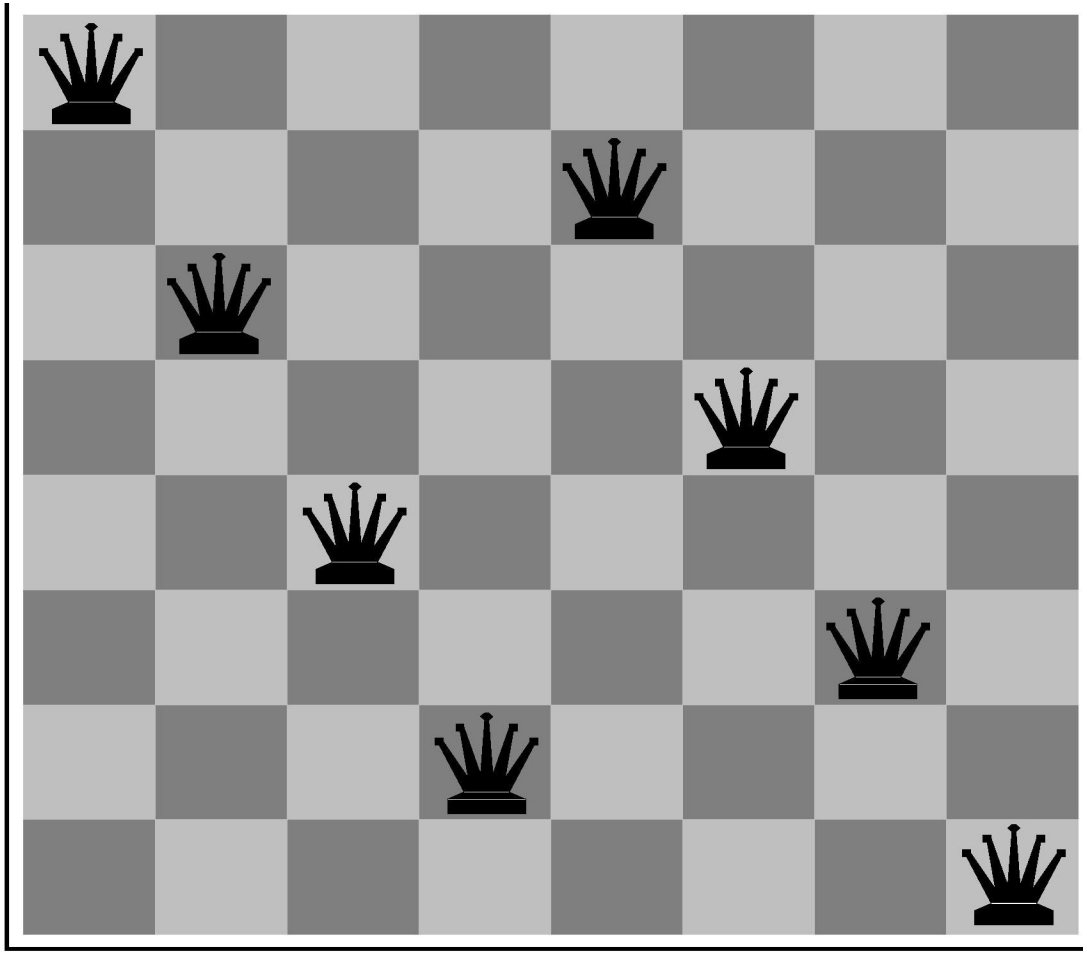
Initial state



Goal state

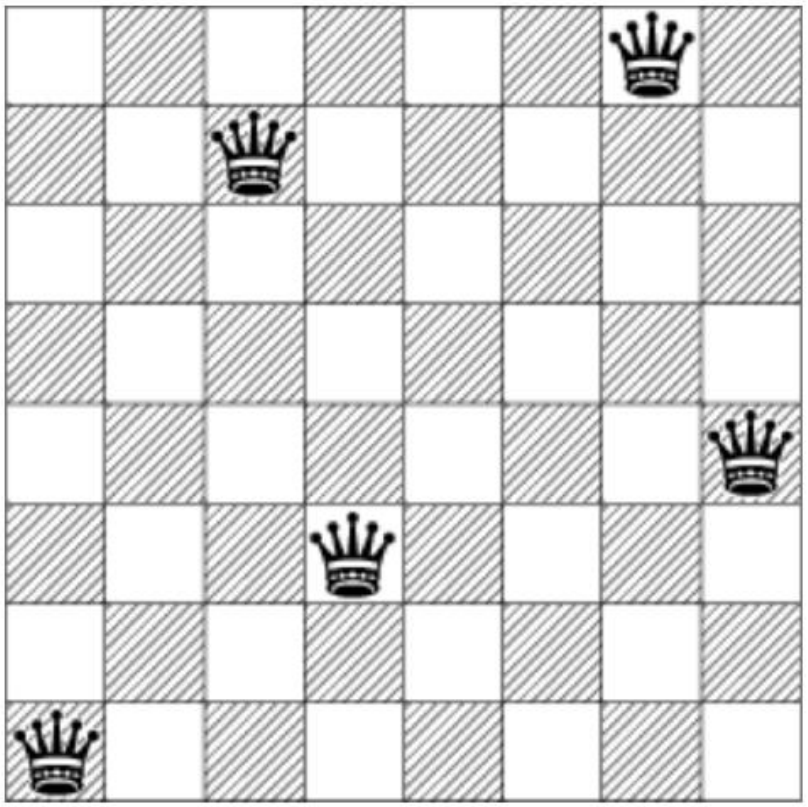
- state description
  - 3-by-3 array: each cell contains one of 1-8 or blank symbol
- two state transition descriptions
  - 8×4 moves: one of 1-8 numbers moves up, down, right, or left
  - 4 moves: one black symbol moves up, down, right, or left
- The number of nodes in the state-space graph:
  - 9! (= 362,880)

Problem Ex: The 8-queens problem



Real world task : the 8-puzzle

- **World state:**
  - Board blanks: 64
  - Queen number: 8
- **How many**
  - World states?
  - 64^8



Problem Ex: The 8-queens problem

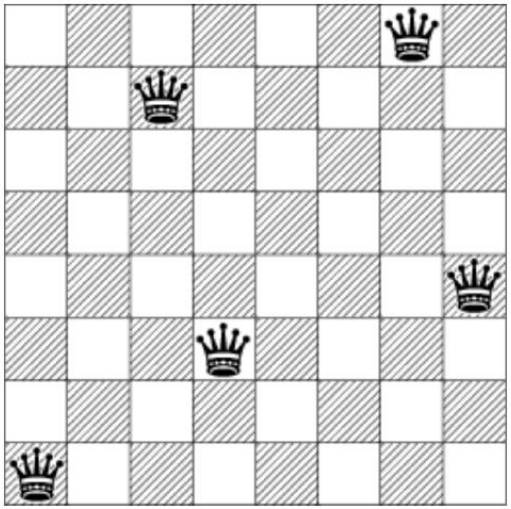
- states? -any arrangement of n<=8 queens

  -such that no queen attacks any other.[not on same row or same column or diagonal]

- initial state? no queens on the board
- actions? -add queen to any empty square

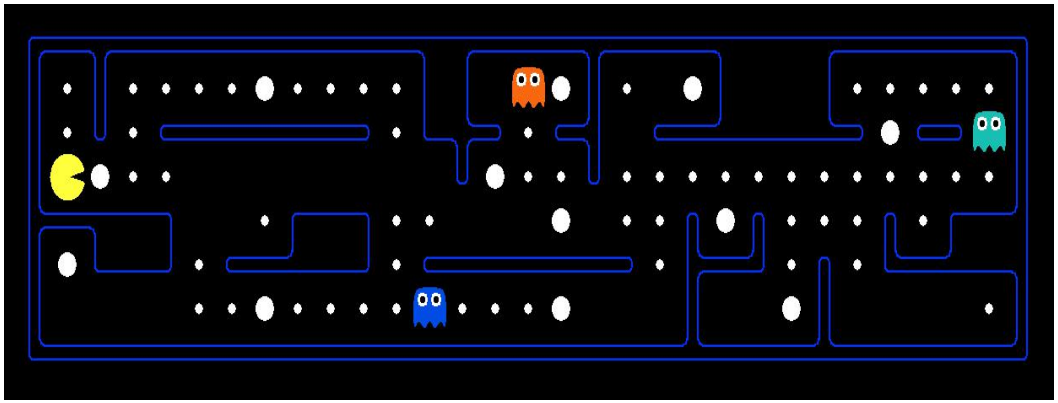
  -*or* add queen to leftmost empty square such that it is not attacked by other queens.

- goal test? 8 queens on the board, none attacked.
- path cost? 1 per move



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?

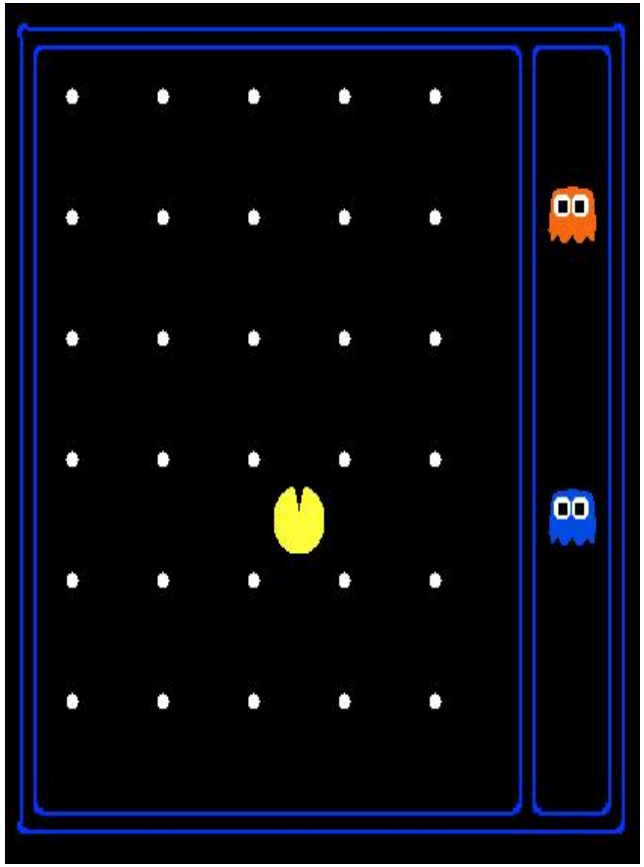
- **World state:**
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- **How many**
  - World states?

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

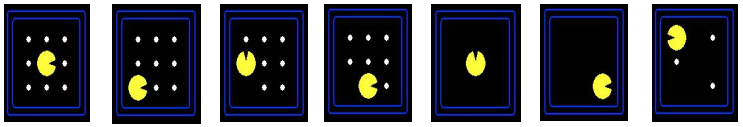
    120
  - States for eat-all-dots?

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

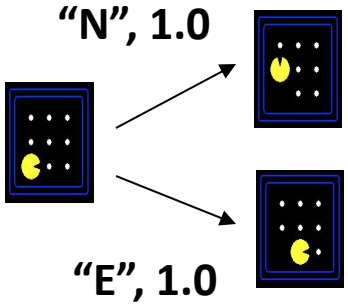


Search Problems

- 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

State Space Search: Water Jug Problem

- "You are given two jugs, a 4-litre one and a 3-litre one.
- Neither has any measuring markers on it.
- There is a pump that can be used to fill the jugs with water.
- How can you get exactly 2 litres of water into 4-litre jug."

A

B

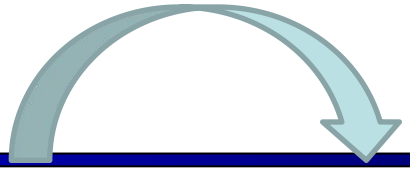
4-litre	3-litre
A	B
empty	empty

4-litre	3-litre

A
4
3
2
1

B
empty

4-litre



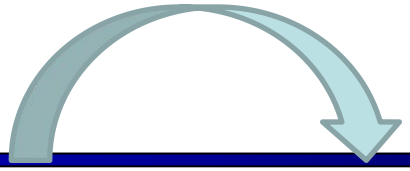
3-litre

A
4
3
2
1

Fill B from A

B
empty

4-litre

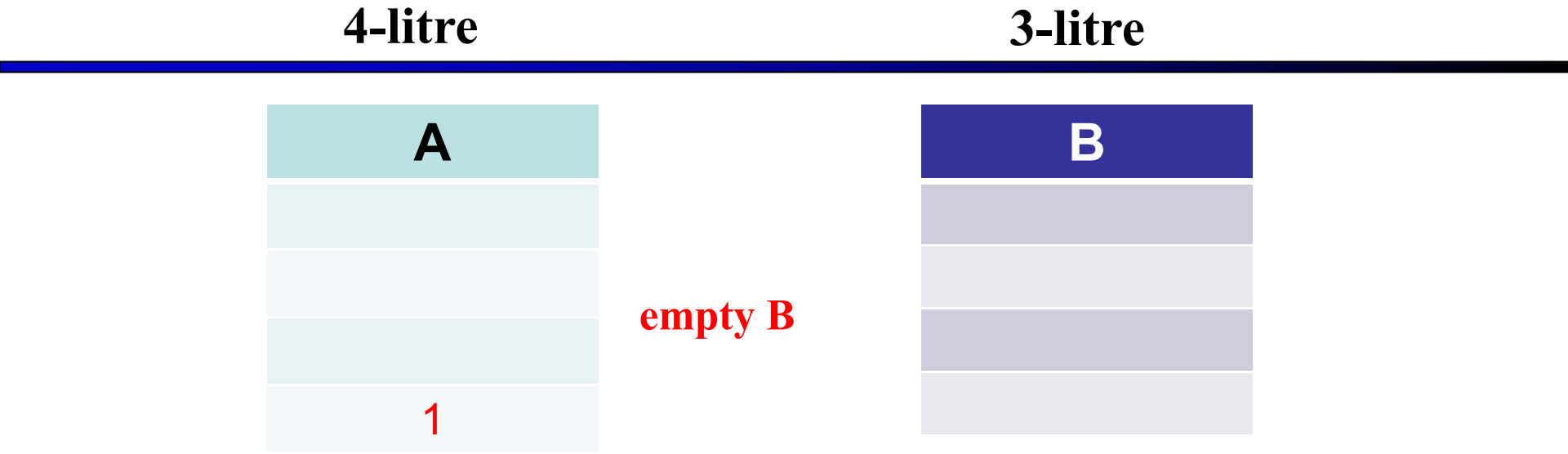


3-litre

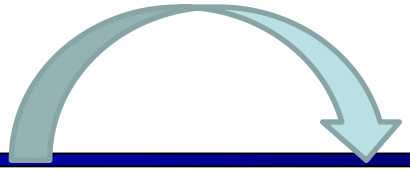
A
1

Fill B from A

B
3
2
1


4-litre
3-litre
A
B
empty B
1

4-litre



3-litre

A
1

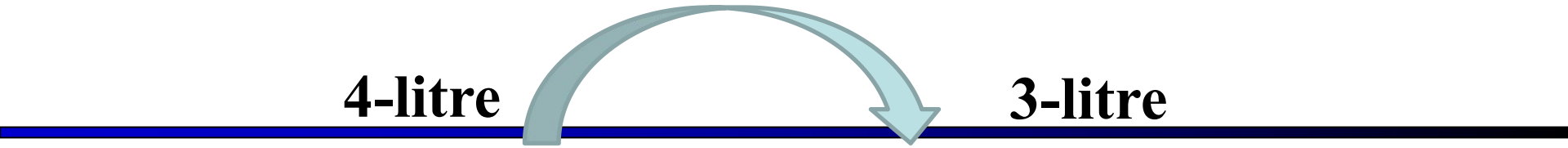
Transmit
from A to B

B

4-litre

3-litre

4-litre		3-litre
A		B
4		
3	Fill A again	
2		
1		1

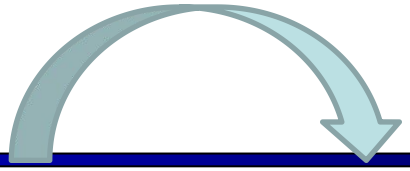


A
2
1

Transmit from A to B

B
3
2
1

4-litre



3-litre

A
2
1

empty B

B

goal

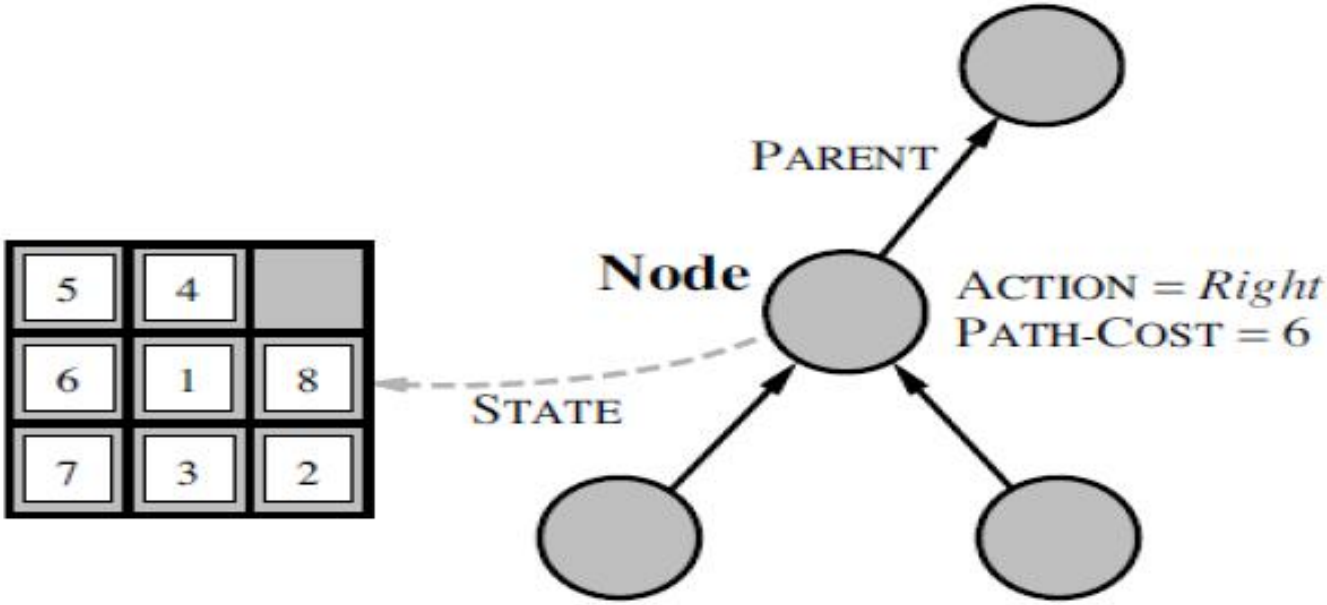
Water Jug Problem: A State Space Search

- State space:
  - set of ordered pairs of integers (x, y) such as
  - x = 0,1,2,3, or 4 for amount of water in 4-gallon jug,
  - y = 0, 1, 2, or 3 for amount of water in the 3-gallon jug.
  -
- The start state : (0,0).
- The goal state : is (2,n) for any value of n.

After formulating the problem ,
a **search** through the states is needed to find a solution

Data Structure

- 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).



Data Structure

- **Open list:**
  - **All nodes that have been generated but not yet extended(open nodes)**
  - **Fringe list**
- **Closed list:**
  - **Remember where one has been, remember every expanded node**
  - **Not necessary for tree search**

- **Many ways to represent node ,**
- **ex : data structure with 5 components**

Implementing a Search-What we need to store

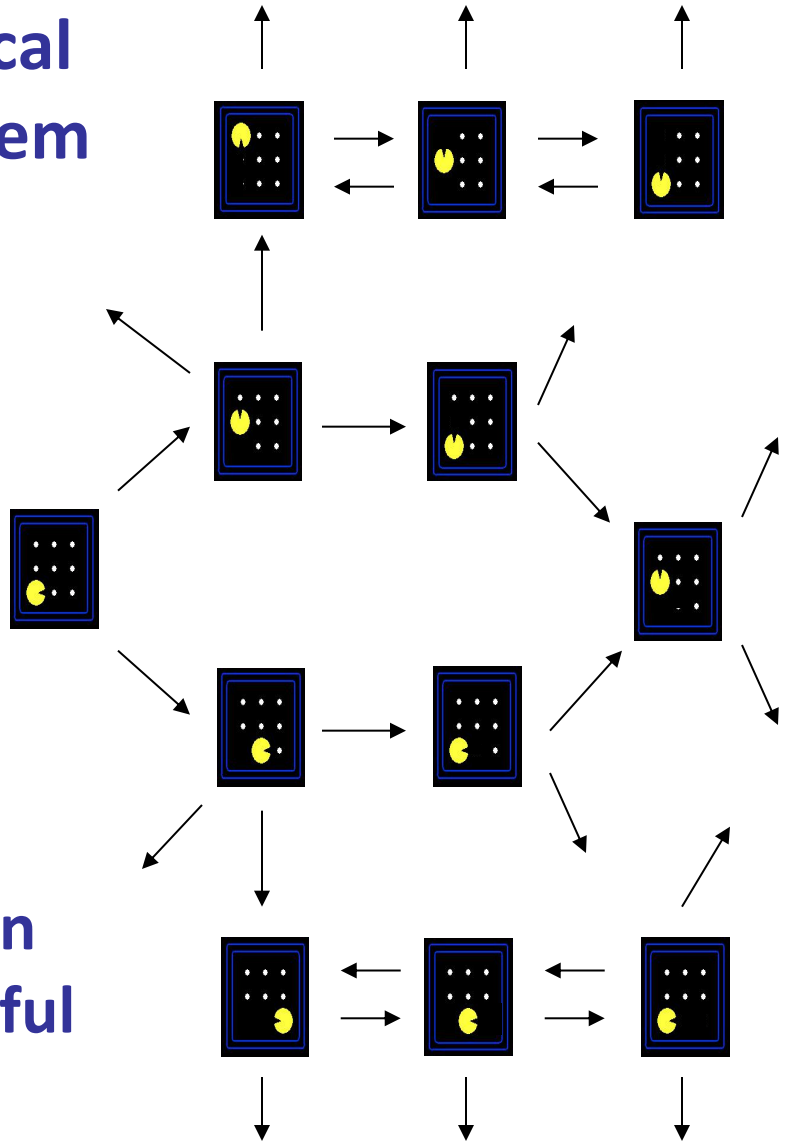
state	Parent node	action	Path cost	depth
The state in state space which the node corresponds	The node in search tree that generated this node	The action that was applied to parent to generate the node	Cost from initial state to the node	The number of steps along the path from the initial states

State-Space Search Algorithm

- **Search process constructs a "Search tree"**
- **Root is the start node (initial state).**
- **Leaf nodes are: unexpanded nodes (in the nodes list).**
- **"dead ends" (nodes that aren't goals and have no successors).**
- **Solution desired may be:**
  - **just the goal state.**
  - **a path from start to goal state .**
- **The search tree is the explicit *tree* generated during the search by the search strategy.**
- **The search space is the implicit *tree* (OR *graph*) defined by initial state and the operators.**

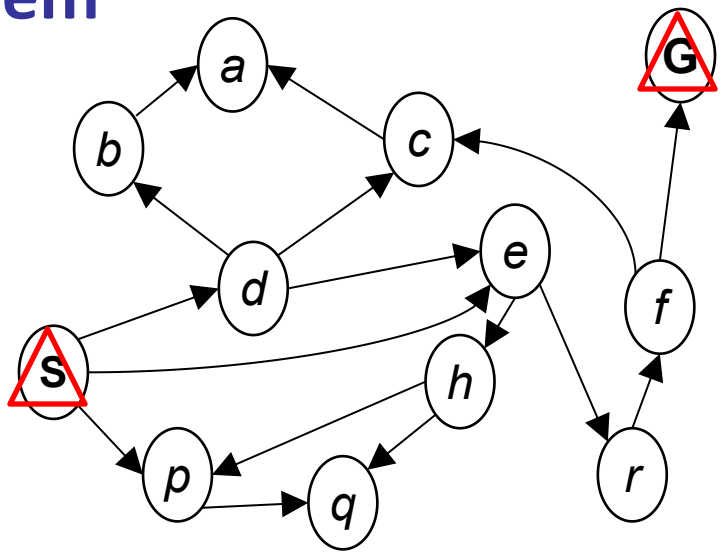
State Space Graphs

- 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



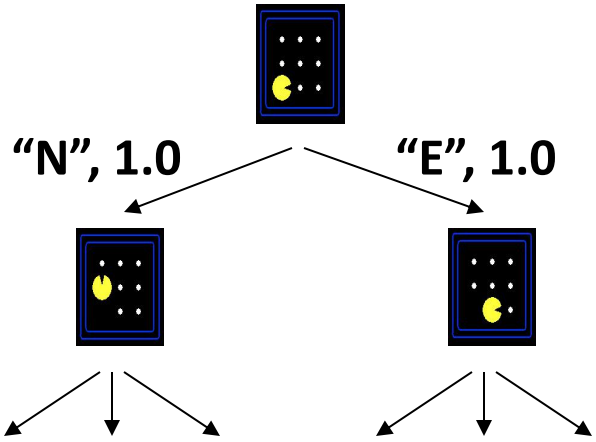
State Space Graphs

- **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 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**



Tiny search graph for a tiny search problem

Search Trees

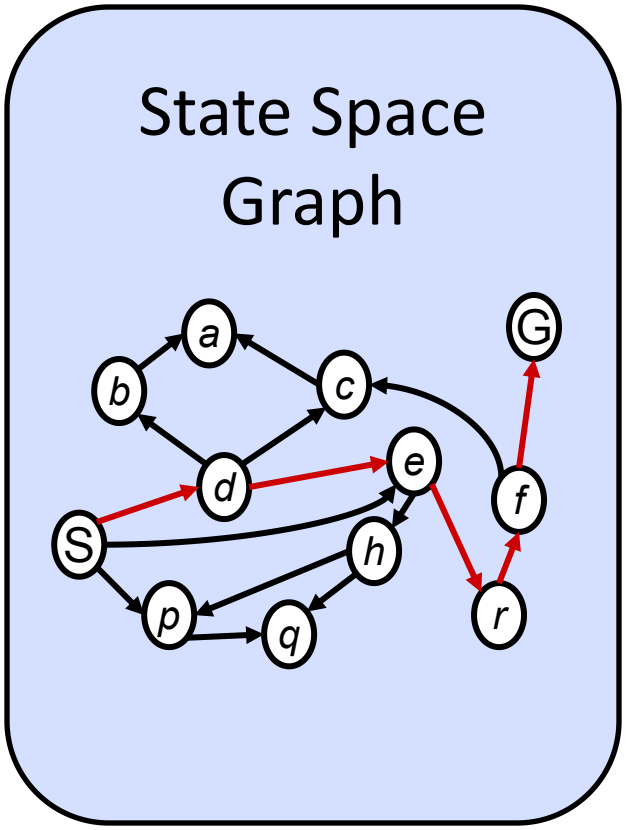


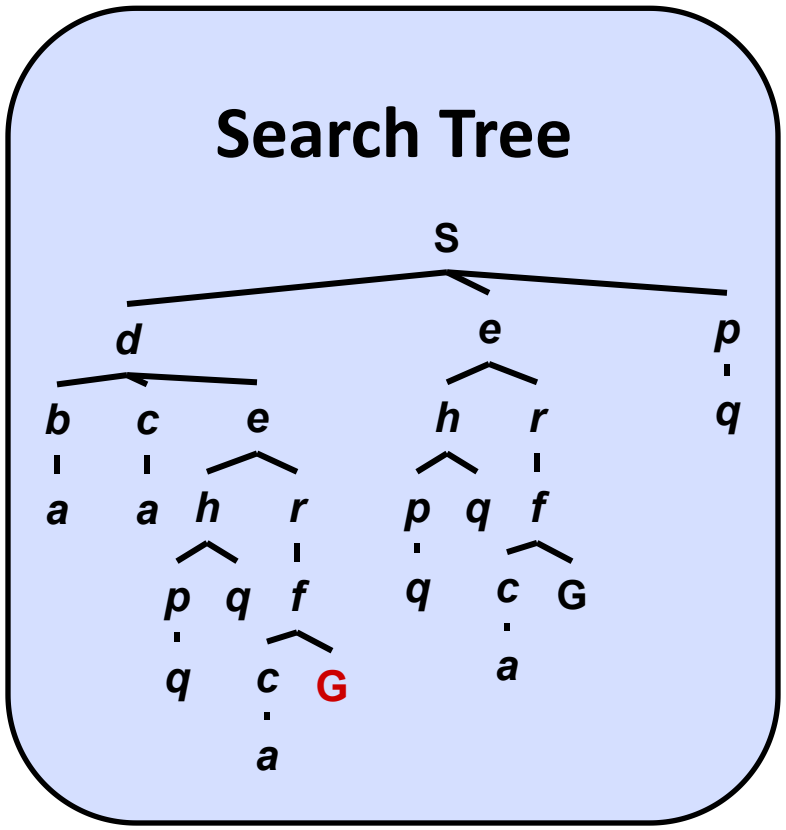
This is now / start

Possible futures

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

State Space Graphs vs. Search Trees

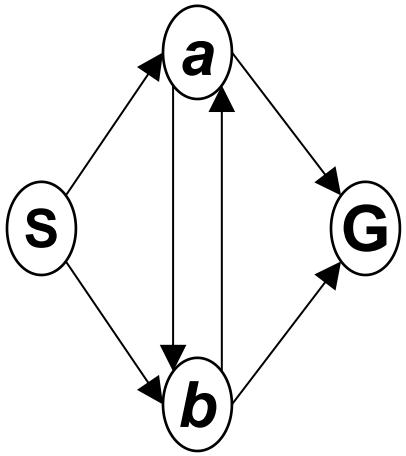




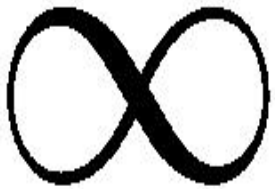
Each NODE in in the search tree is an entire PATH in the state space graph.

Quiz: State Space Graphs vs. Search Trees

Consider this 4-state graph:

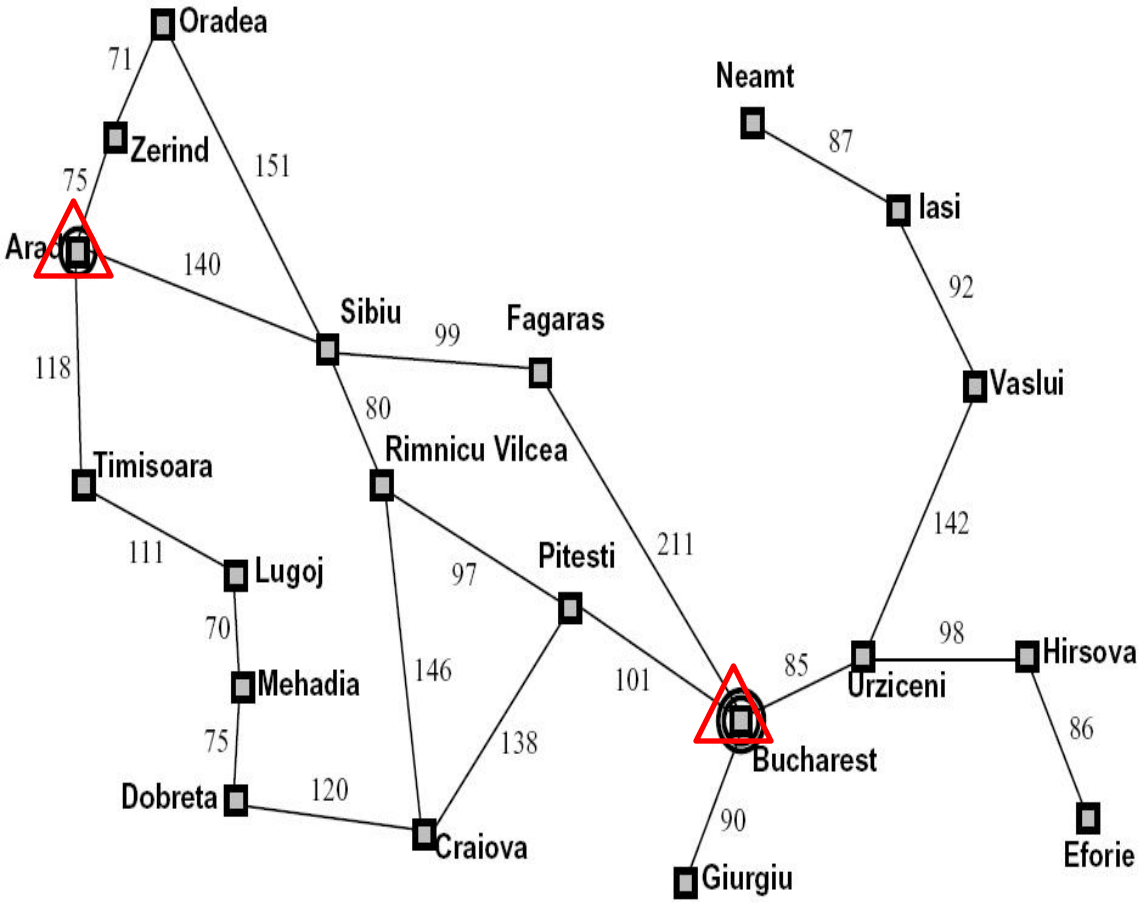


How big is its search tree (from S)?

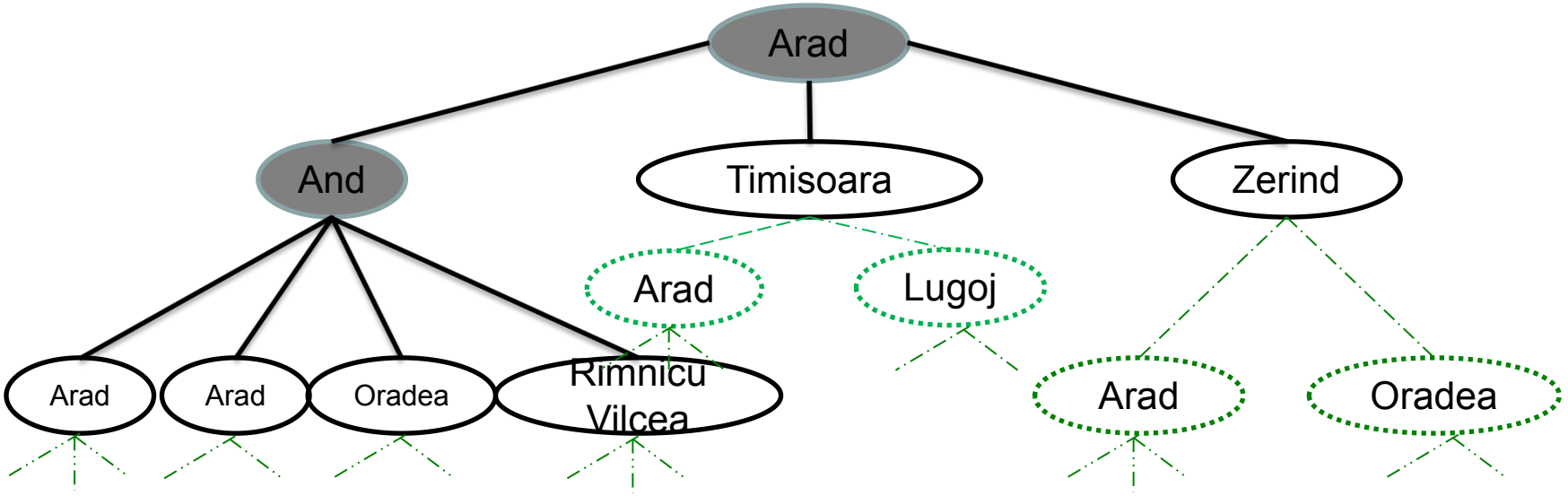


Important: Lots of repeated structure in the search tree!

Search Example: Romania



Searching with a Search Tree



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

Tree Search Algorithms

Basic idea:

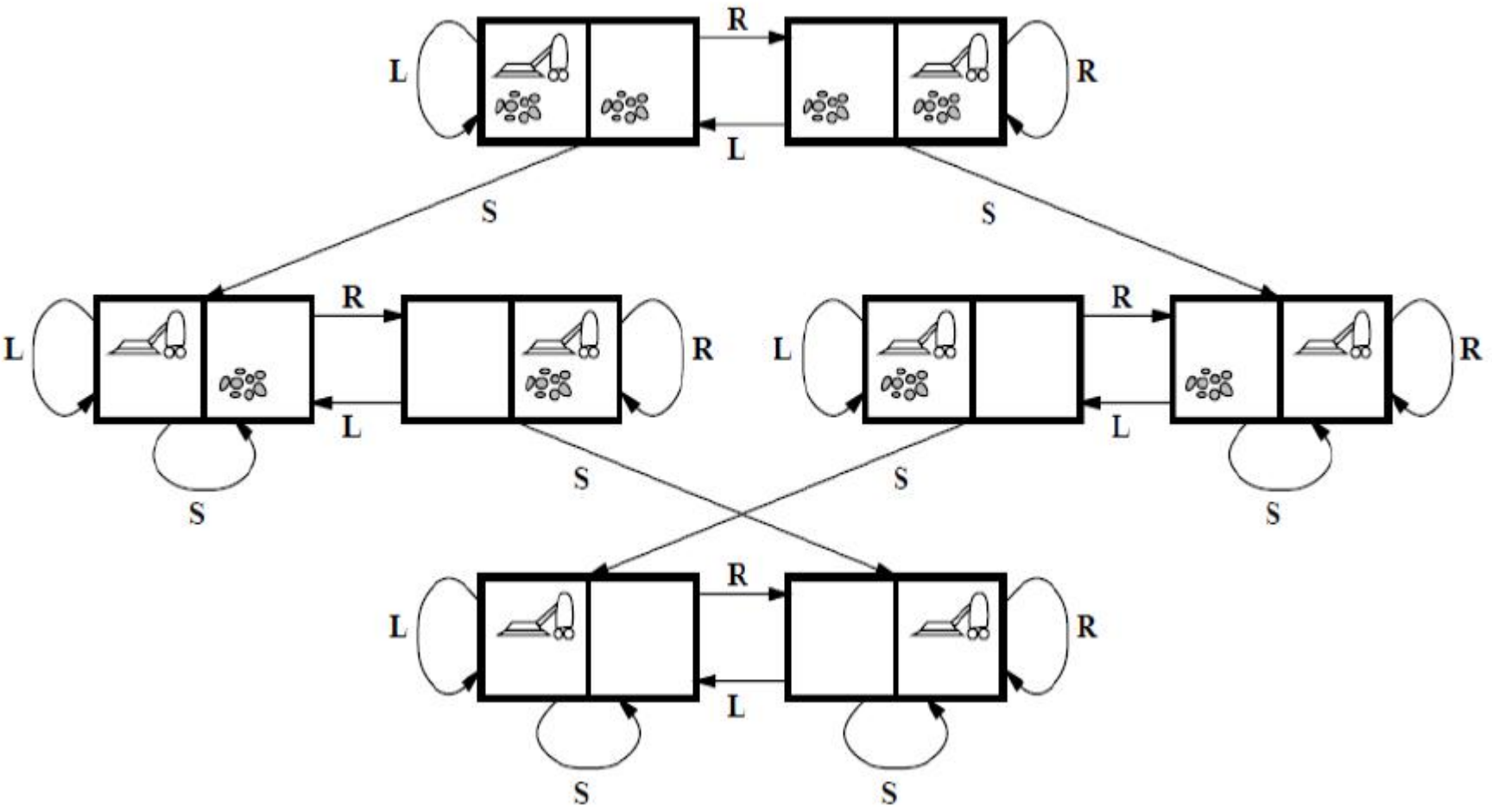
- **offline, simulated exploration of state space by generating successors of already-explored states (expanding states).**

General Tree Search

function TREE-SEARCH(problem， strategy)**returns** a solution,or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion **then return** failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state **then return** the corresponding solution
        else expand the node and add the resulting nodes to the search tree
end

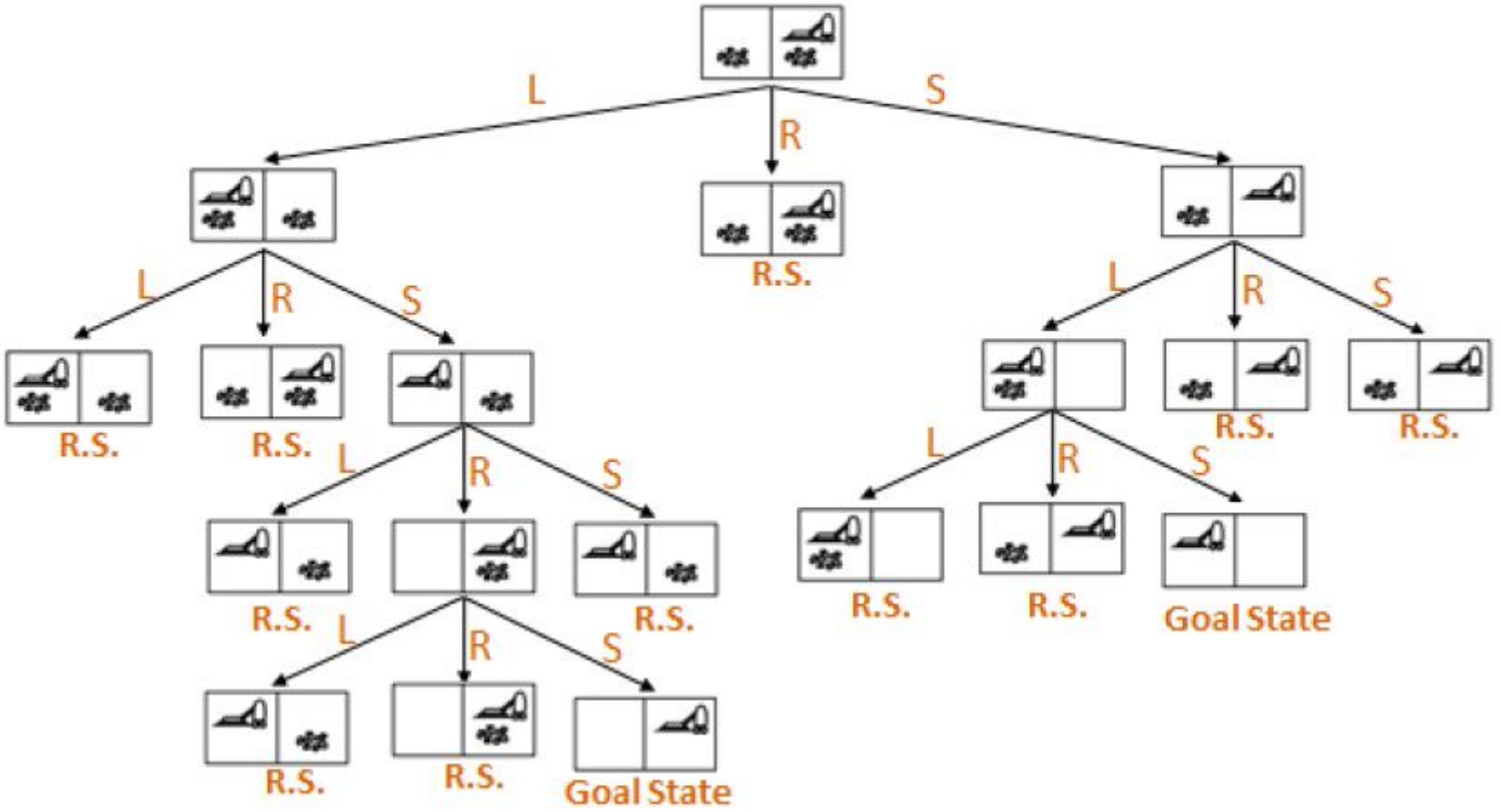
- **Important ideas:**
  - **Fringe**
  - **Expansion**
  - **Exploration strategy**
- **Main question: which fringe nodes to explore?**

Search Space of Vacuum World Problem: **Graph**

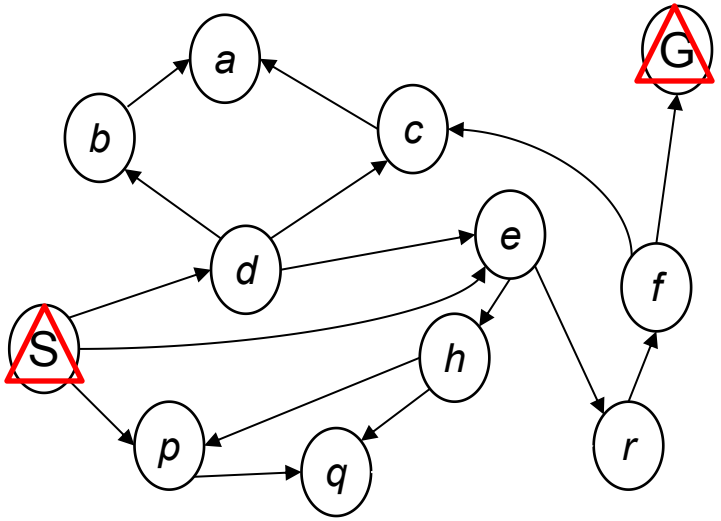


Search Space of Vacuum World Problem: **Tree**

- trace every path from the root until you reach a leaf node (goal) or a node already in that path (Repeated State or R.S.)



Example: Tree Search



How Good is the found Solution?

- **Completeness**
  - Is the strategy guaranteed to find a solution if there is a one
- **Time Complexity**
  - How long does it take to find a solution?
- **Space Complexity**
  - How much memory does it take to perform the search?
- **Optimality**
  - Does the strategy find the optimal solution where there are several solutions?

Search Algorithm Properties

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

Variables:

n	Number of states in the problem (huge)
b	The average branching factor B (the average number of successors)
C^*	Cost of least cost solution
s	Depth of the shallowest solution
m	Max depth of the search tree

Actions in Searching a Tree

Fundamental actions (operators) that you can take:

1. **"Expand": Ask a node for its children**
2. **"Test": Test a node for whether it is a goal**

Undiscovered Nodes

- The set of nodes that have not yet been discovered as being reachable from the root

Actions in Searching a Tree (cont..)

Fringe Nodes

This is the set of nodes that (open nodes)

- **have been discovered**
- **have not yet been "processed":**
  1. **have not yet expanded for the children**
  2. **(have not yet tested if they are a goal)**

Actions in Searching a Tree (cont..)

Visited Nodes

- This is the set of nodes that
  - have been discovered
  - have been processed:
    1. have discovered all their children
    2. (have tested whether are a goal)
- Also called
  - closed nodes

Action on finding a Goal

- "First match": Usually we just want one goal, or just to know whether or not one exists
  - on discovering a goal, then "return true"
- "All Matches": Sometimes want all goals
  - on discovering a goal, then record the fact that have found it, but continue with the search

Searching algorithm

Uninformed Search Algorithms(Blind Search)

3.1 Breadth first Search
3.2 Depth First Search
3.3 Depth limited Search
3.4 Iterative Deeping Search
3. 5 Bidirectional Search

Informed Search (Heuristic Search)

Best First Search
Greedy Search
Perfect Information Search
A* Search
Iterative Deepening A* Search
A* with PathMax

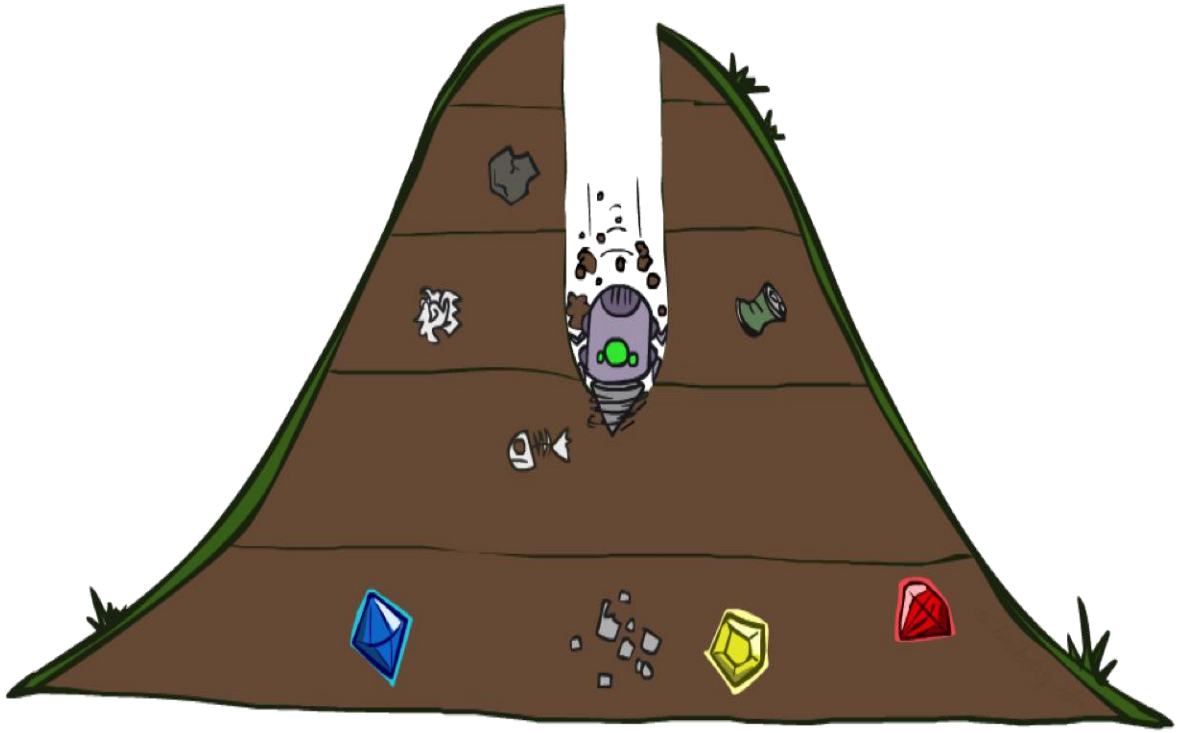
Uninformed Search Algorithms(Blind Search)

1. Depth First Search
2. Breadth first Search
3. Depth limited Search
4. Iterative Deeping Search
5. Uniform Cost Search (UCS)
6. Bidirectional Search

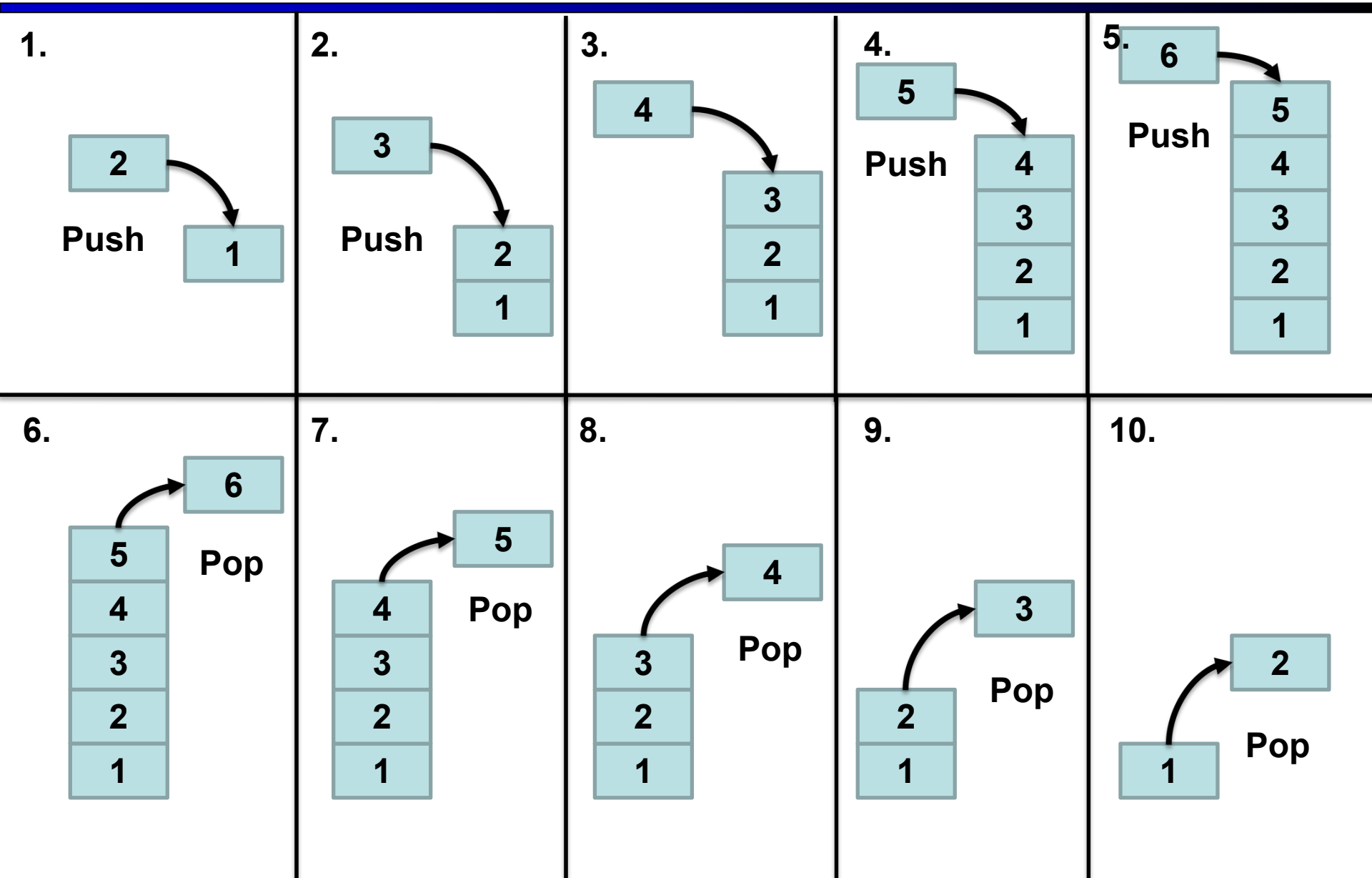
Blind Searches - Characteristics

- **Simply searches the State Space**
- **Can only distinguish between a goal state and a non-goal state**
- Blind Searches have no **preference** as to which state (node) that is expanded next.
- The different types of blind searches are characterised by the **order** in which they **expand the nodes**.

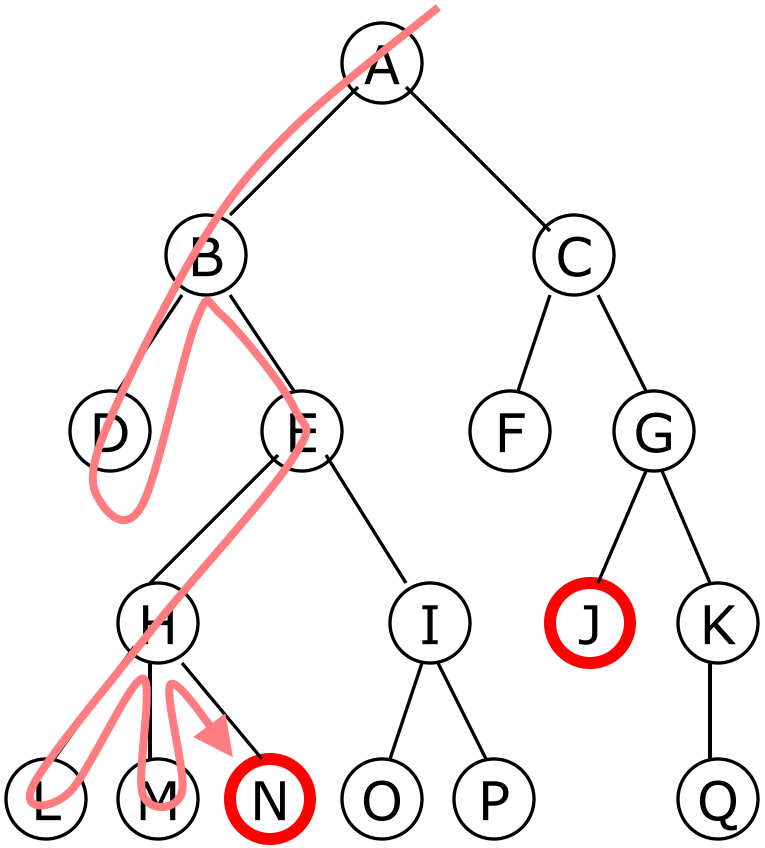
Depth-First Search



Last In First Out



Depth-first searching



- A **depth-first** search (**DFS**) explores a path all the way to a leaf before **backtracking** and exploring another path
- For example, after searching **A**, then **B**, then **D**, the search backtracks and tries another path from **B**
- Node are explored in the order **A B D E H L M N I O P C F G J K Q**
- **N** will be found before **J**

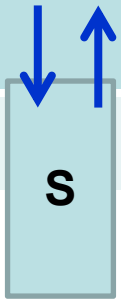
- Always expands the deepest node in the current fringe
- The search proceeds immediately to deepest level of search tree , where nodes have no successors
- As those nodes are expanded they are dropped from fringe
- Can be implemented by LIFO queue (stack)
- For memory , this algorithm need only to store one path

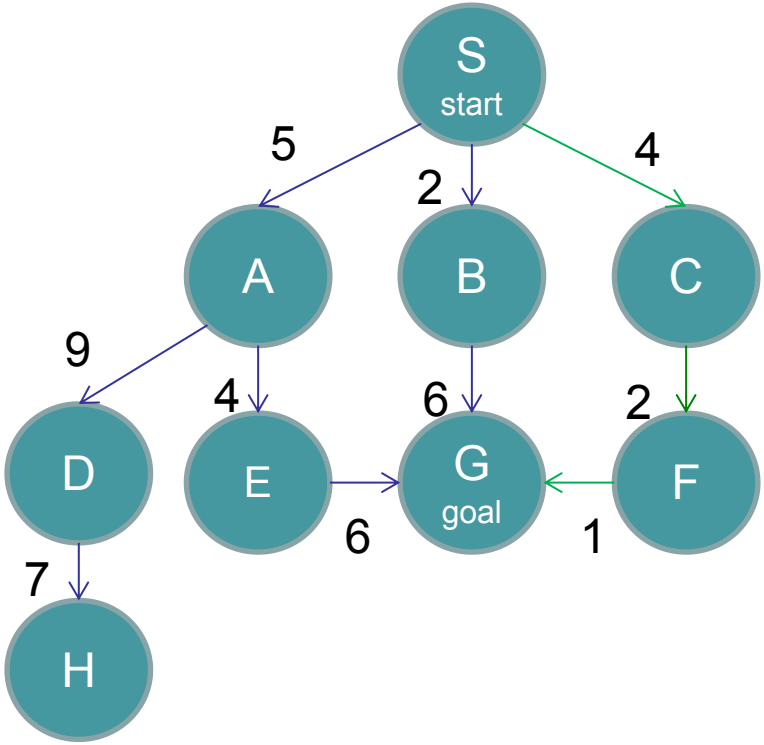
Example: DFS

generalSearch(problem, Stack)

# of nodes tested: 0, expanded: 0

Expnd.node	Open list
	{S}



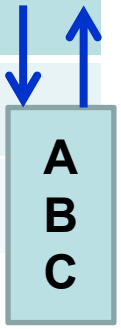


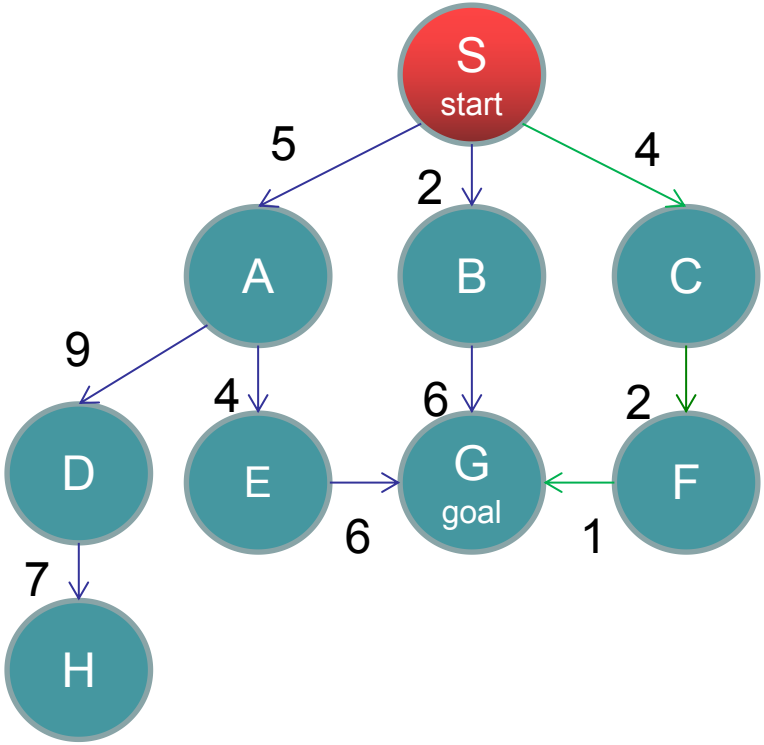
Example: DFS

generalSearch(problem, Stack)

# of nodes tested: 1, expanded: 1

Expnd.node	Open list
	{S}
S not goal	{A,B,C}



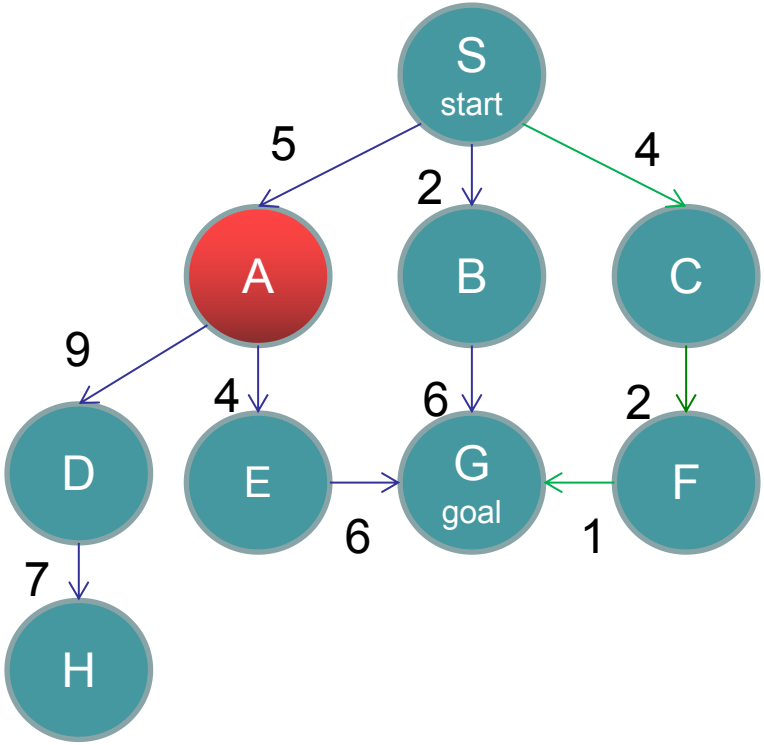


Example: DFS

generalSearch(problem, Stack)

of nodes tested: 2, expanded: 2

Expnd.node	Open list
	{S}
S	{A,B,C}
A not goal	{D,E,B,C}

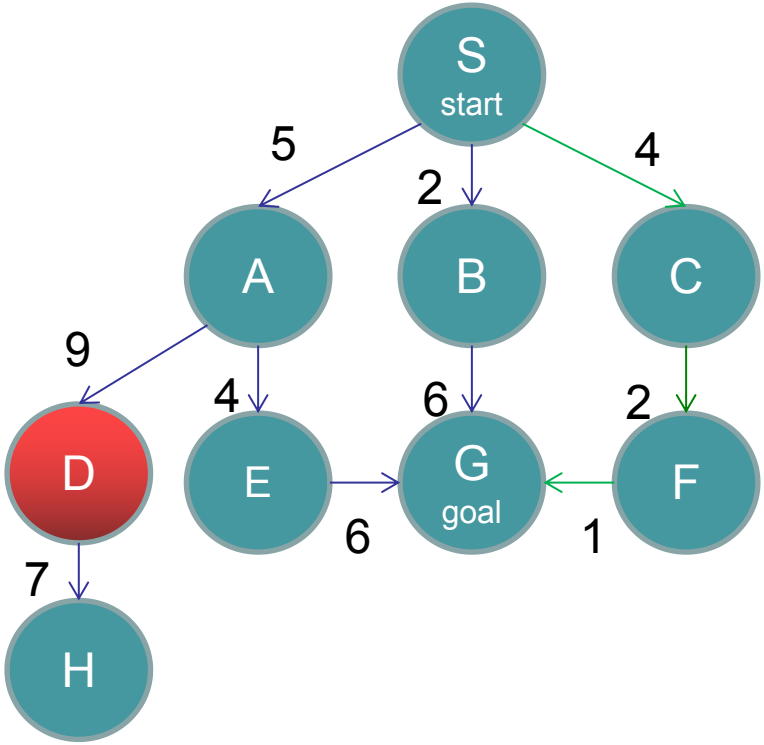


Example: DFS

generalSearch(problem, Stack)

of nodes tested: 3, expanded: 3

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D not goal	{H,E,B,C}

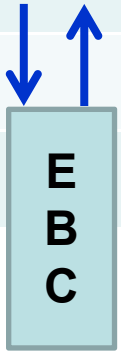


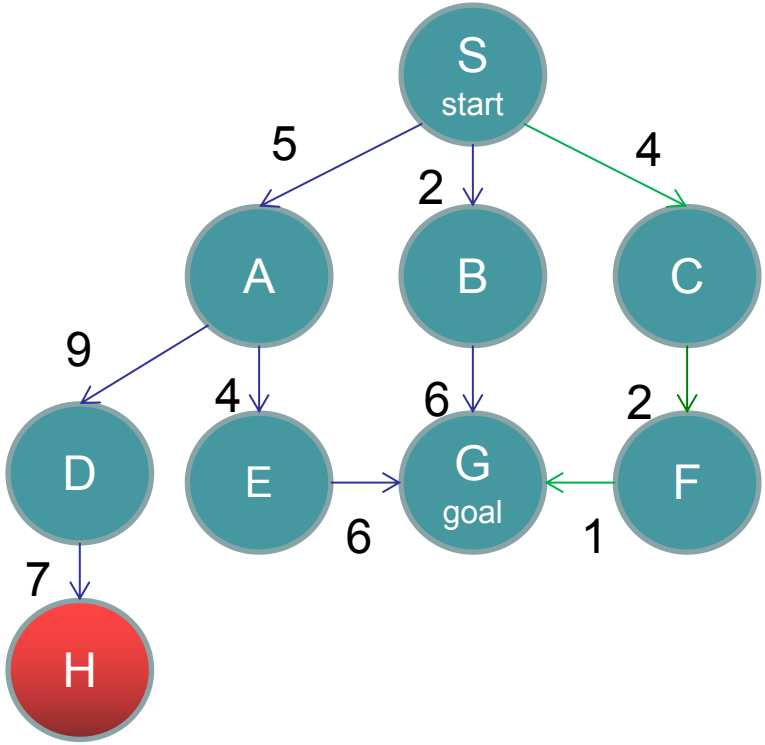
Example: DFS

generalSearch(problem, Stack)

of nodes tested: 4, expanded: 4

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H not goal	{E,B,C}



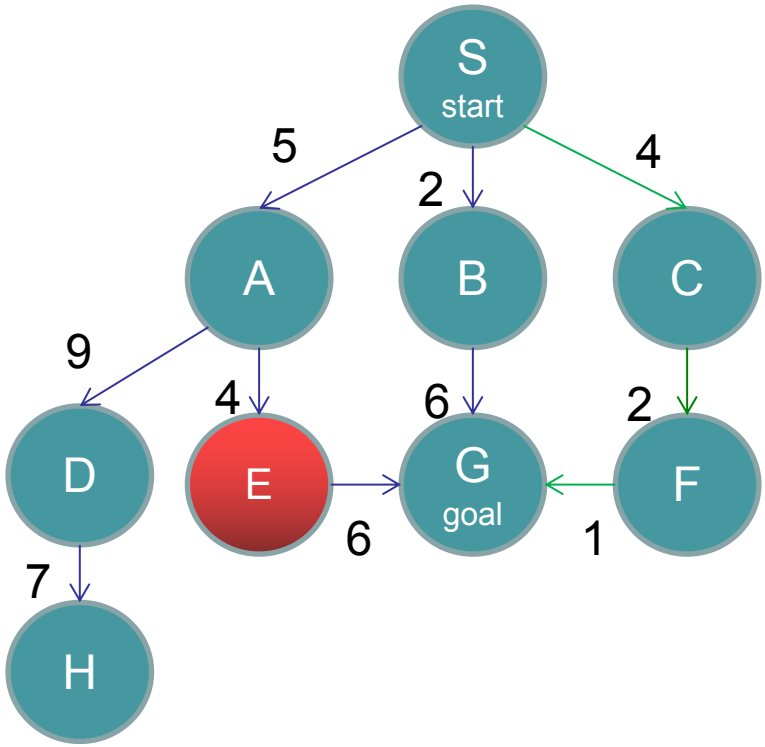


Example: DFS

generalSearch(problem, Stack)

of nodes tested: 5, expanded: 5

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E not goal	{G,B,C}

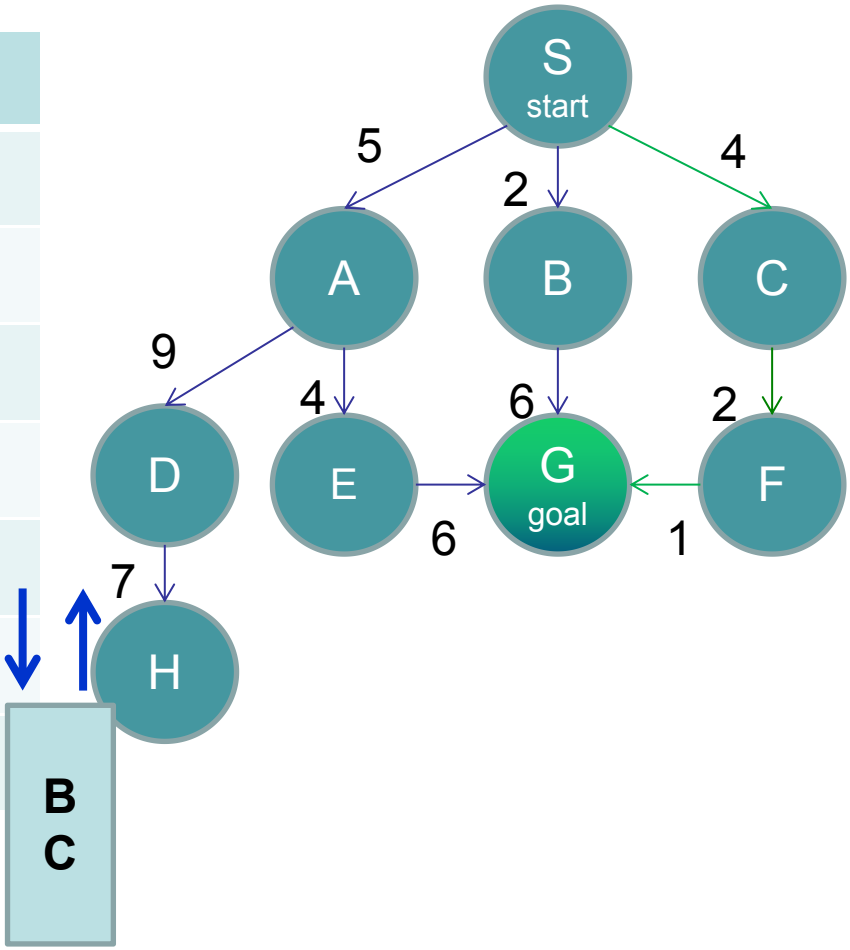


Example: DFS

generalSearch(problem, Stack)

of nodes tested: 6, expanded: 5

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G goal	{B,C}no expand

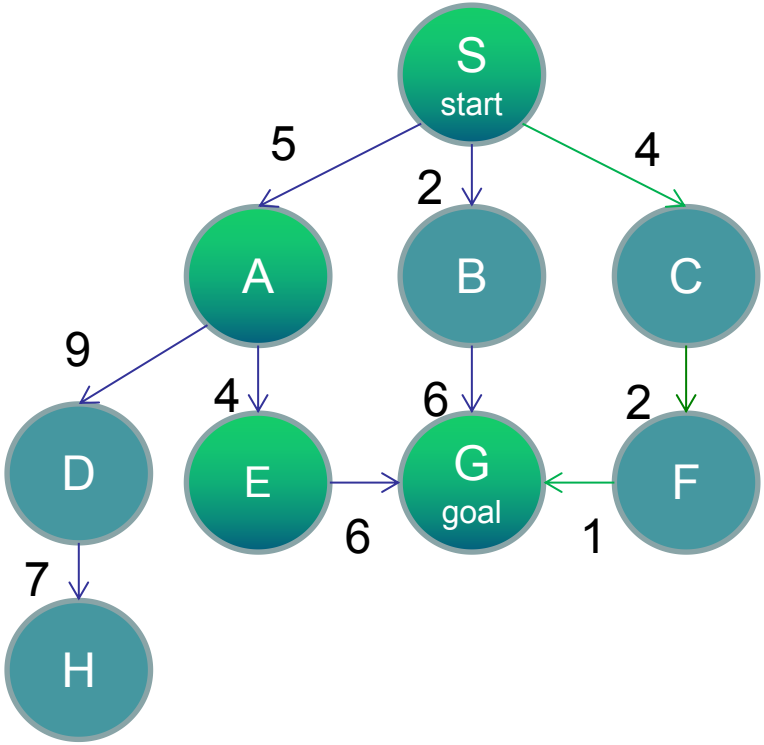


Example: DFS

generalSearch(problem, Stack)

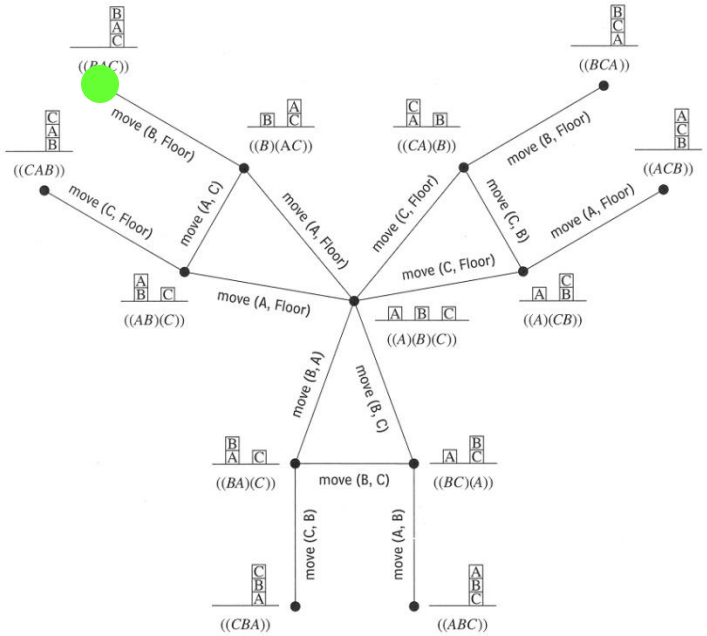
of nodes tested: 6, expanded: 5

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{D,E,B,C}
D	{H,E,B,C}
H	{E,B,C}
E	{G,B,C}
G	{B,C}

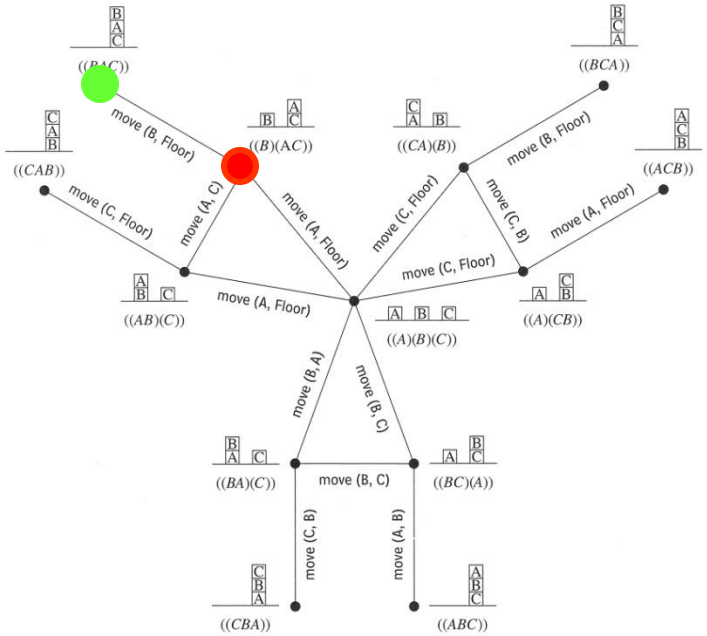


Path: S,A,E,G
Cost:15

Uninformed Search: Depth-First



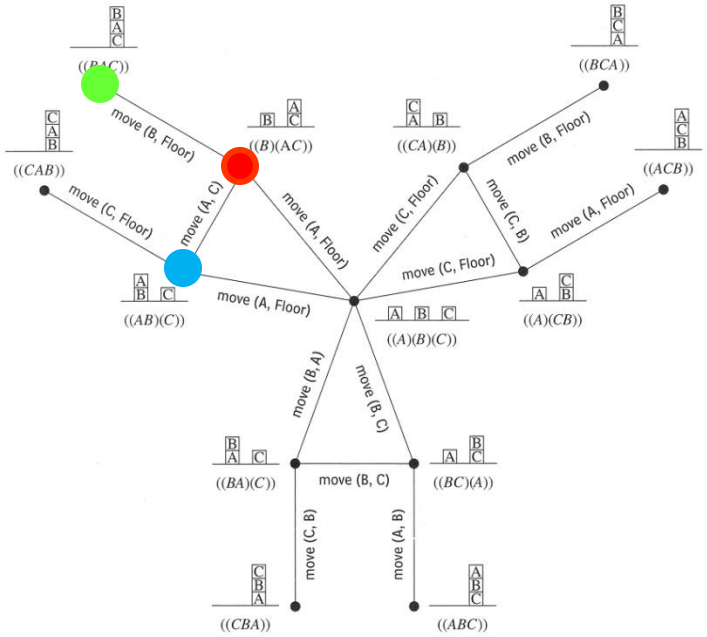
Uninformed Search: Depth-First

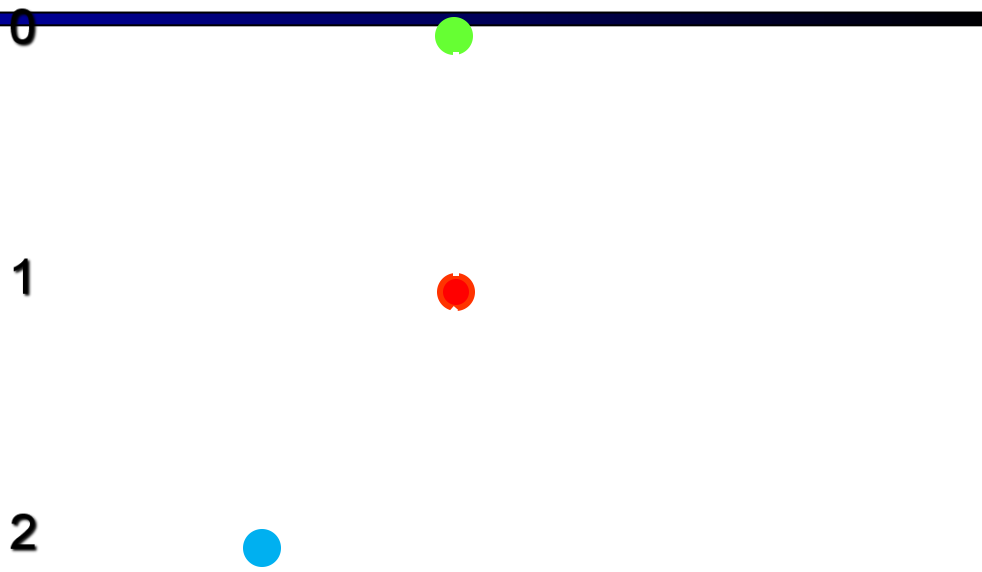


0

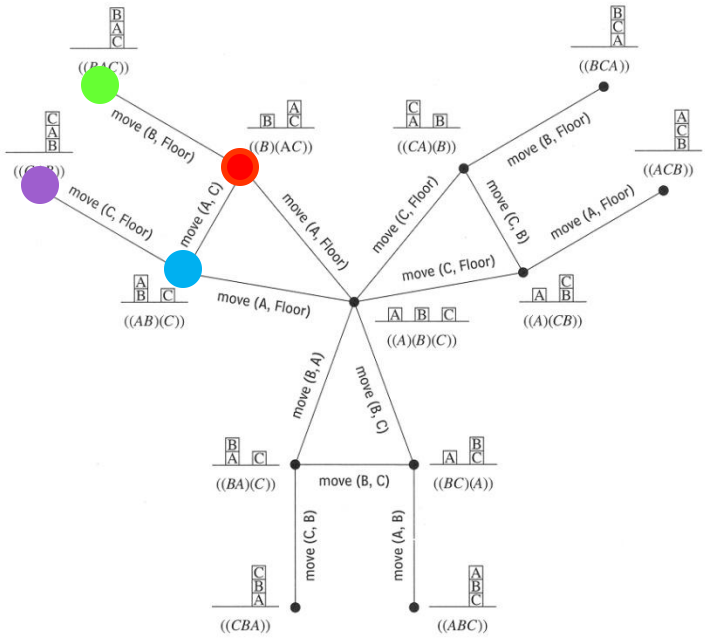
1

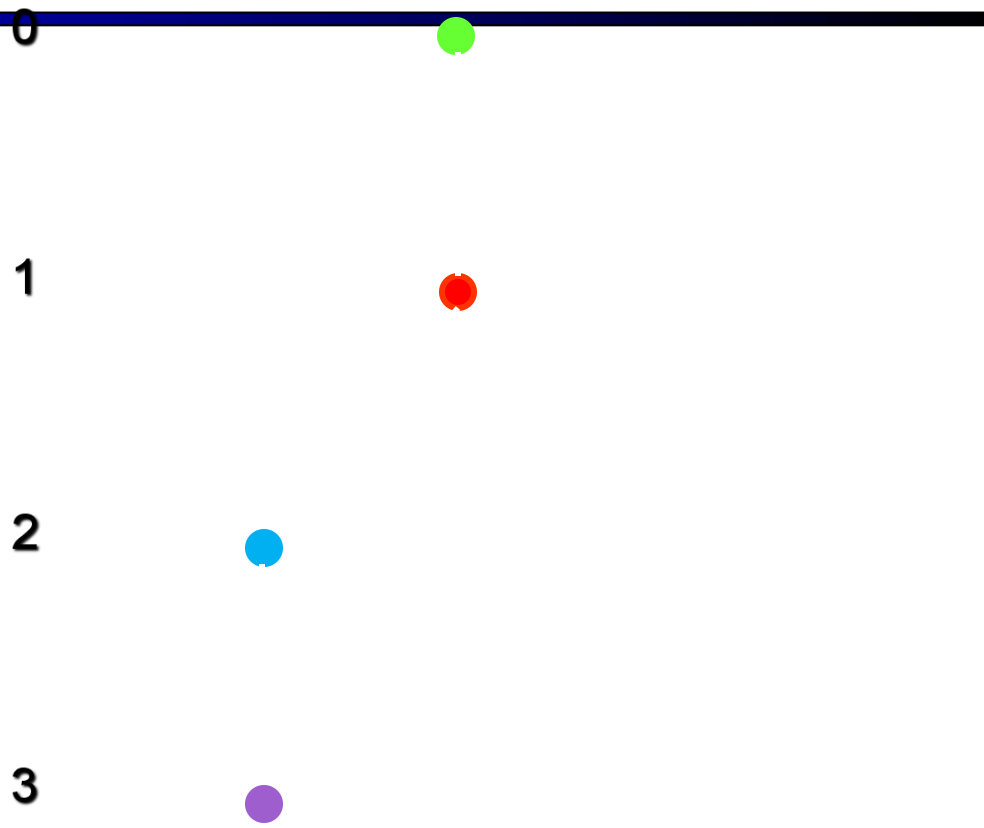
Uninformed Search: Depth-First



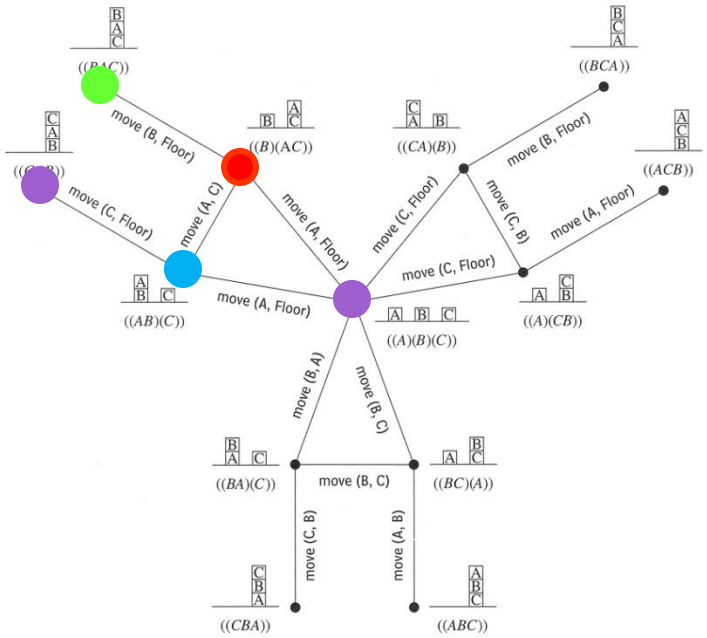


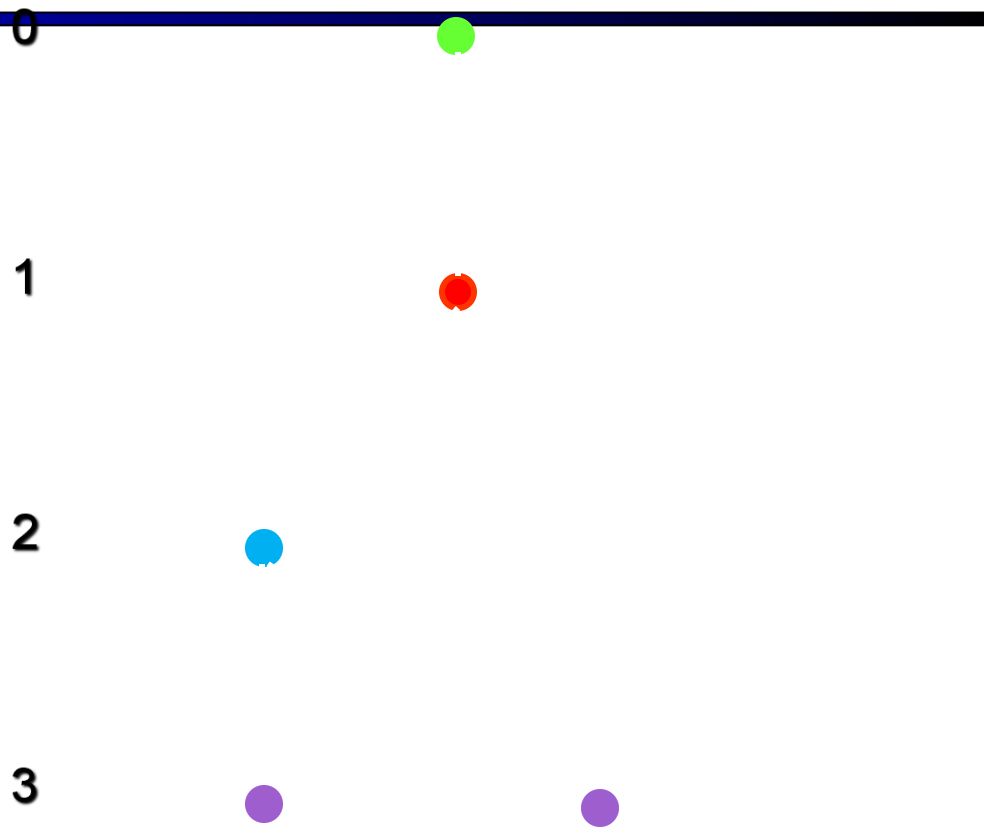
Uninformed Search: Depth-First



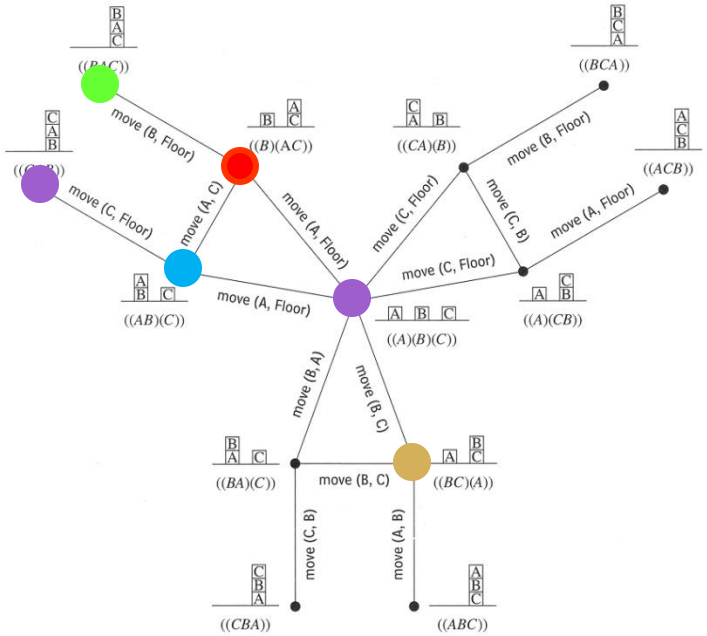


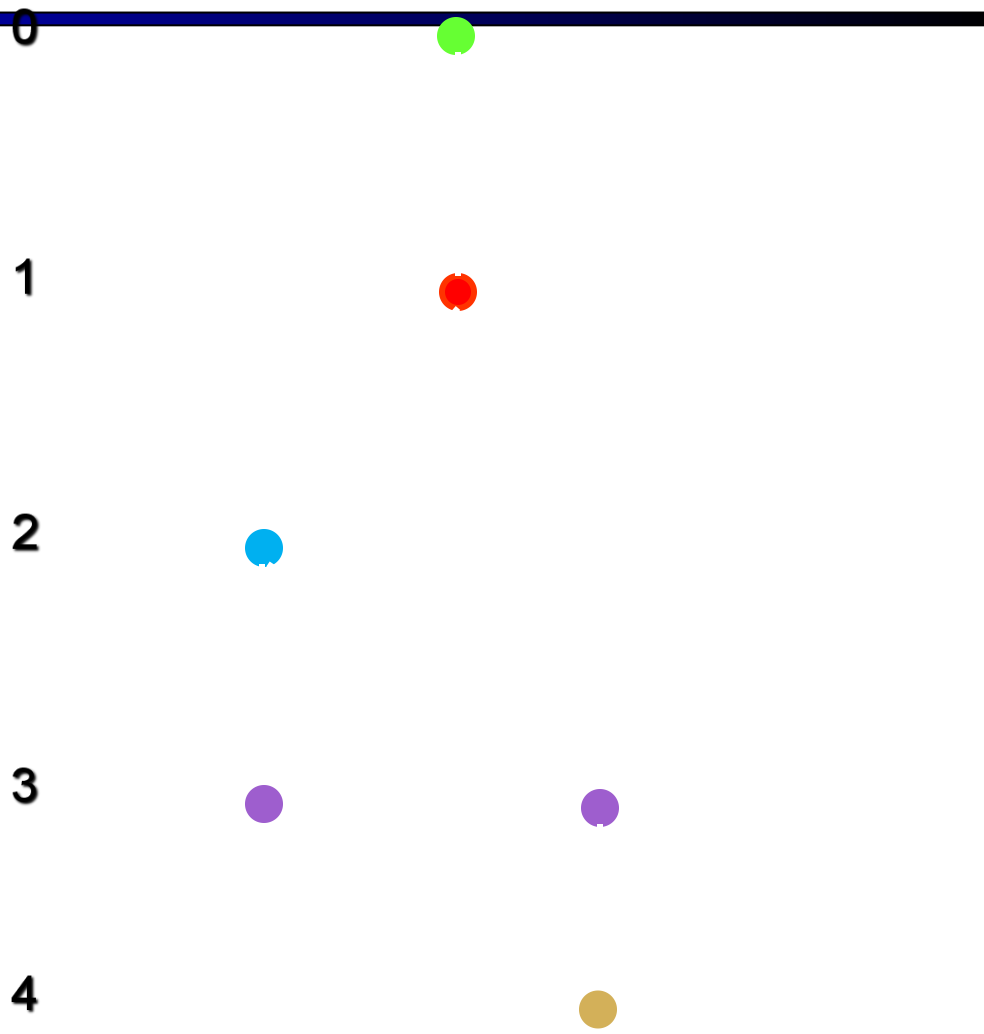
Uninformed Search: Depth-First



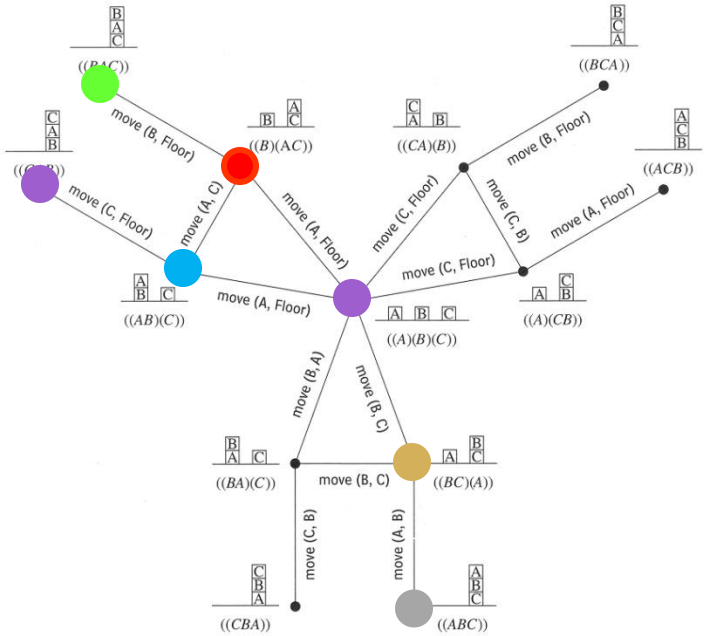


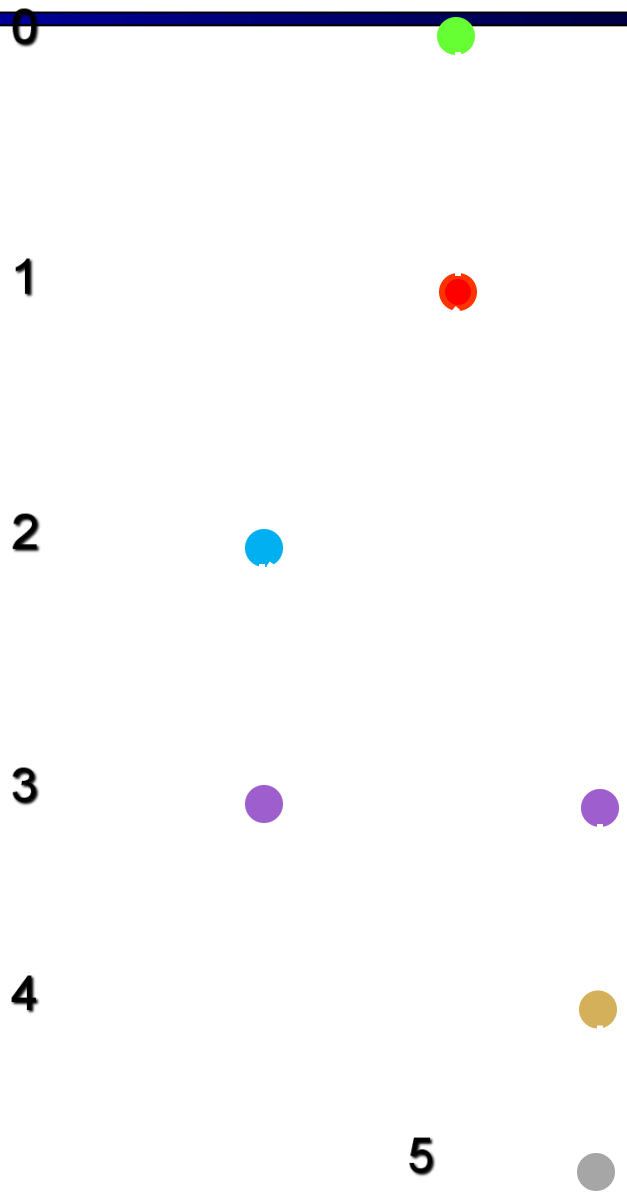
Uninformed Search: Depth-First



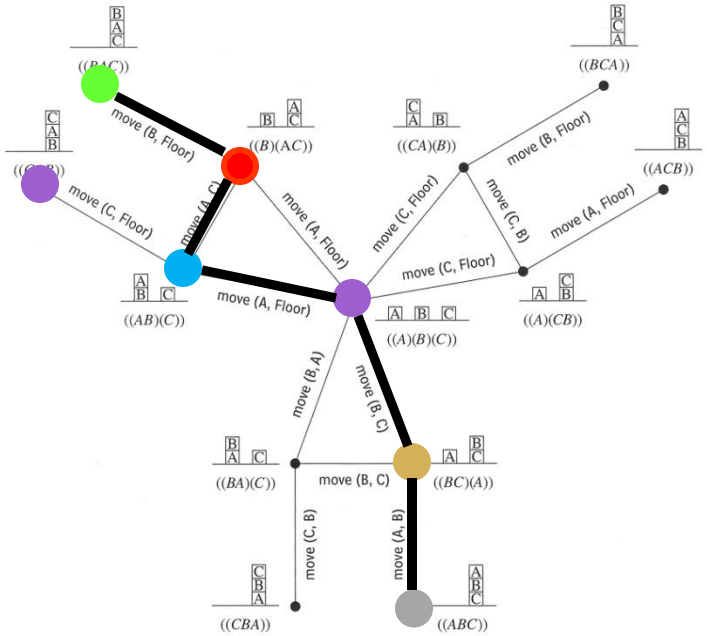


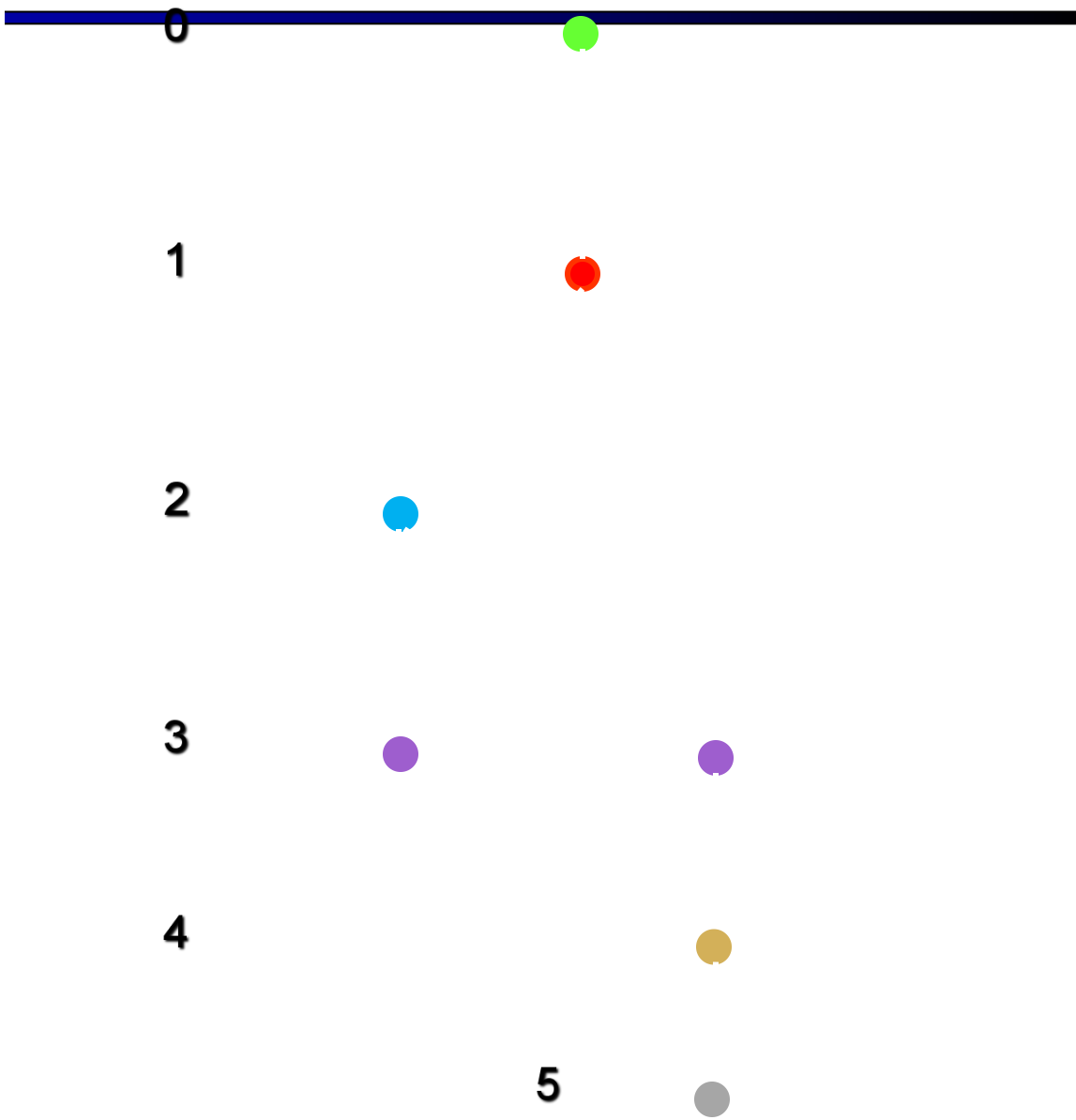
Uninformed Search: Depth-First





Uninformed Search: Depth-First

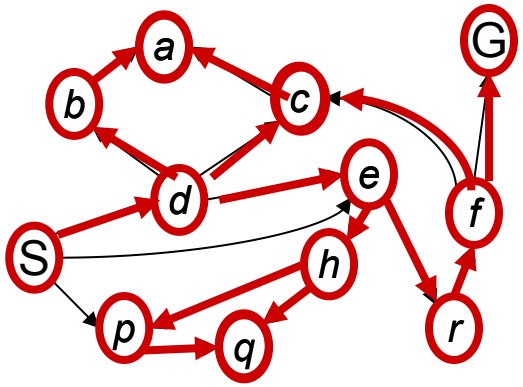




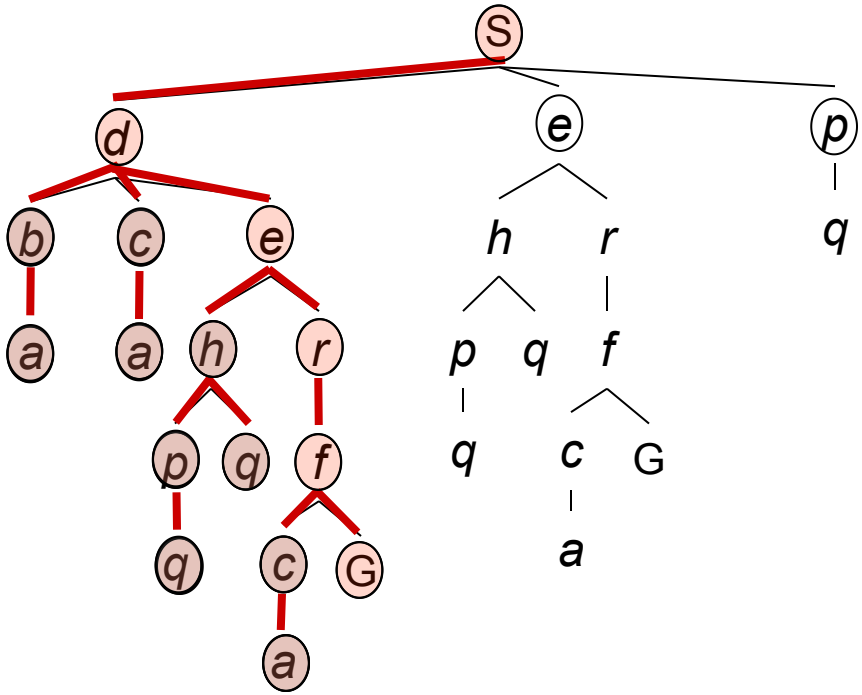
Depth-First Search

Strategy: expand a deepest node first

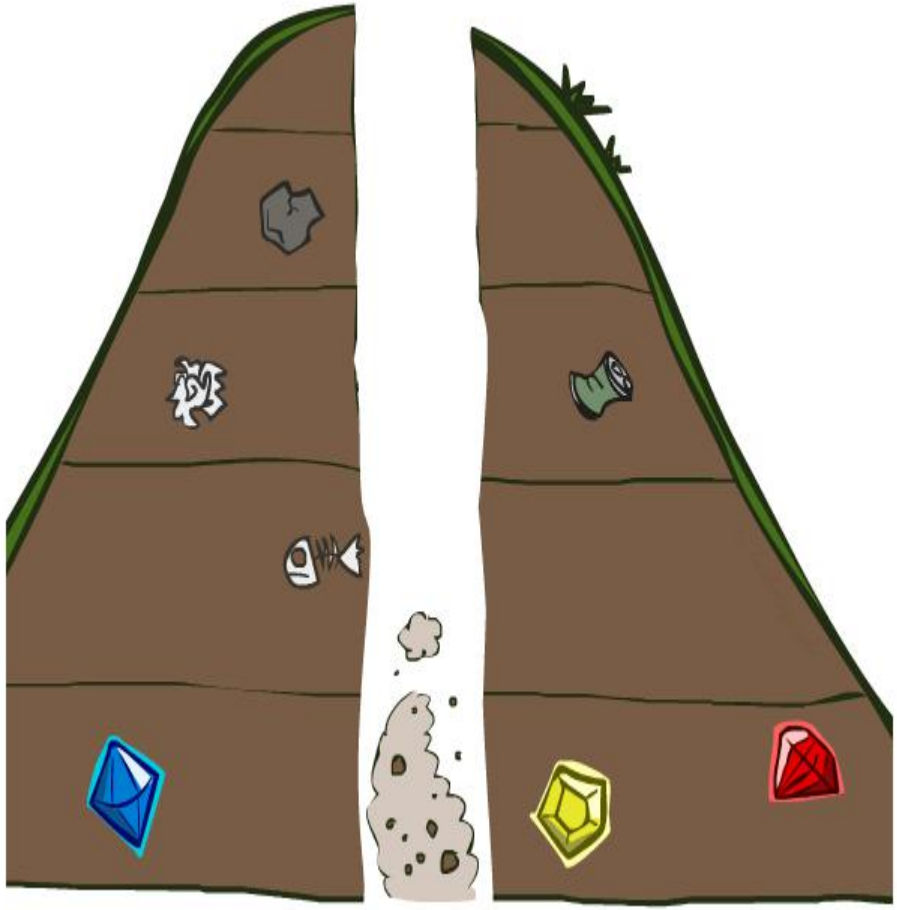
Implementation: Fringe is a LIFO stack



Last In First Out

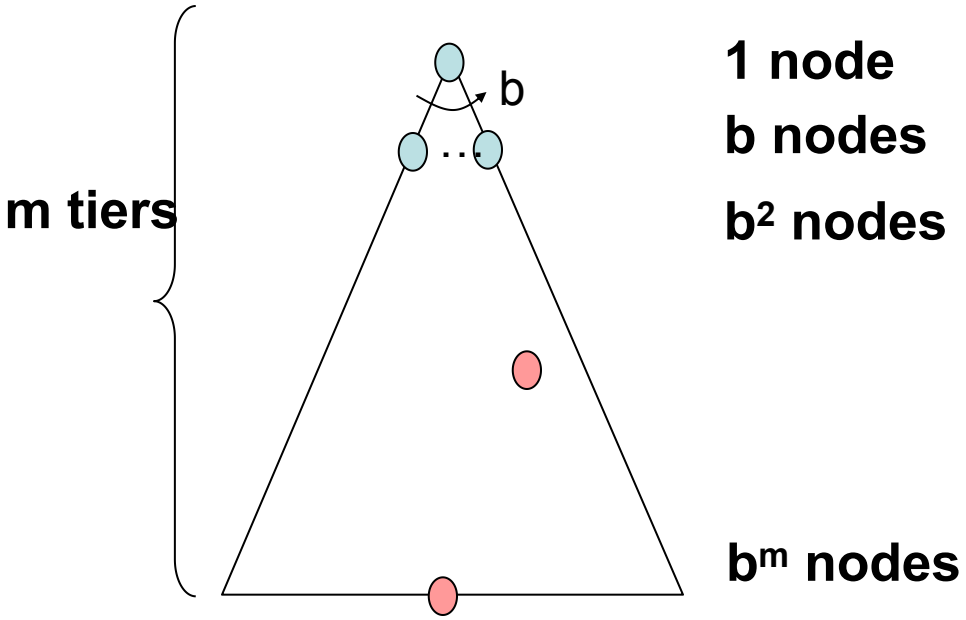


Search Algorithm Properties



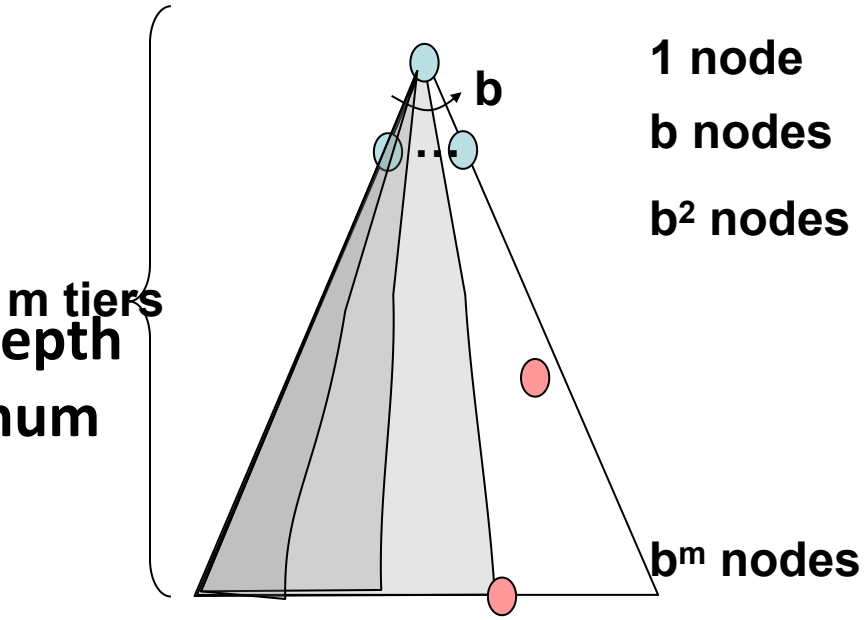
Search Algorithm Properties

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



Depth-First Search (DFS) Properties

- **What nodes DFS expand?**
  - Some left prefix of the tree.
  - Could process the whole tree!
  - Each state has b successors, the depth of the target node is d, the maximum depth of the search tree is m
  - If m is finite, takes time $O(b^m)$

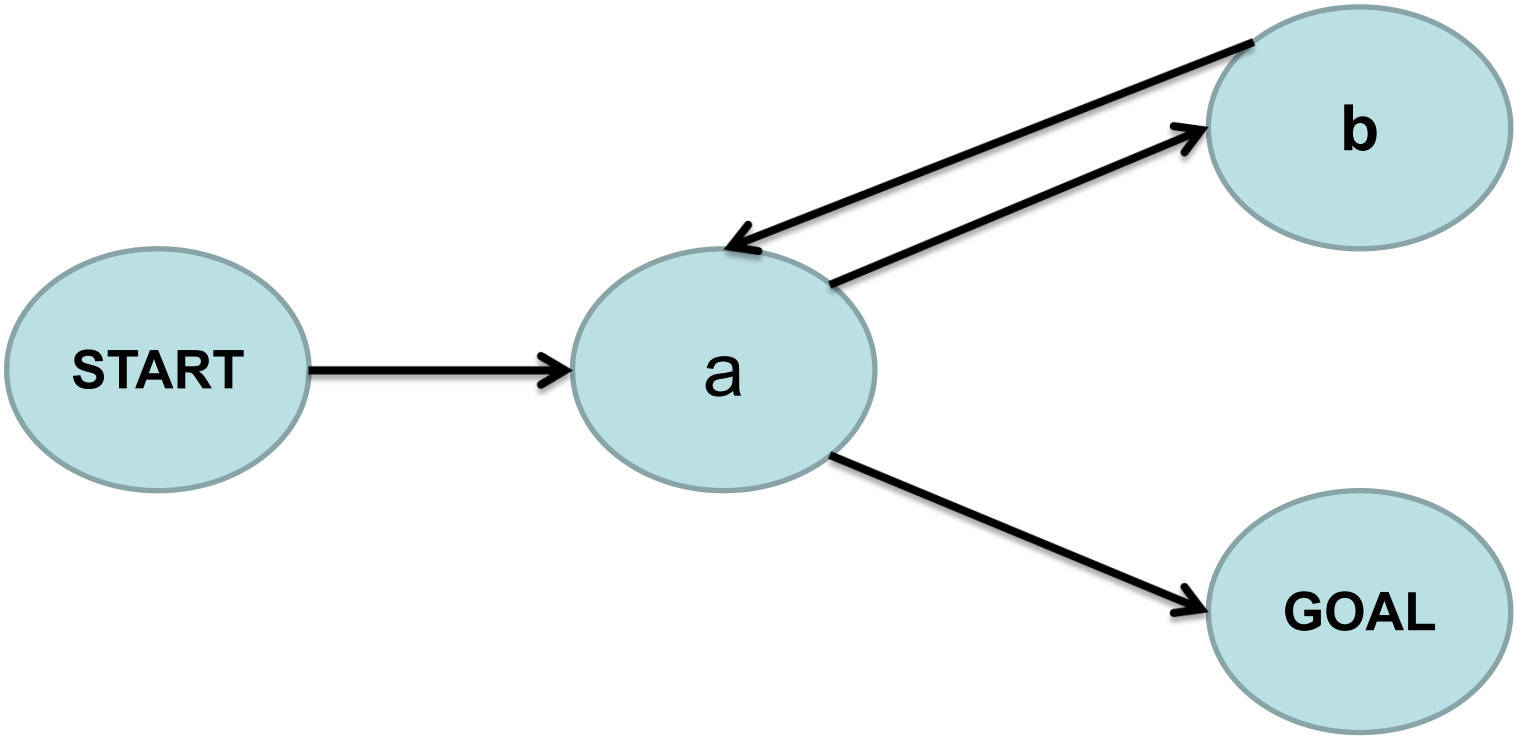


- **Is it complete?**
  - m could be infinite, so only if we prevent cycles (more later)
- **Is it optimal?**
  - No, it finds the "leftmost" solution, regardless of depth or cost

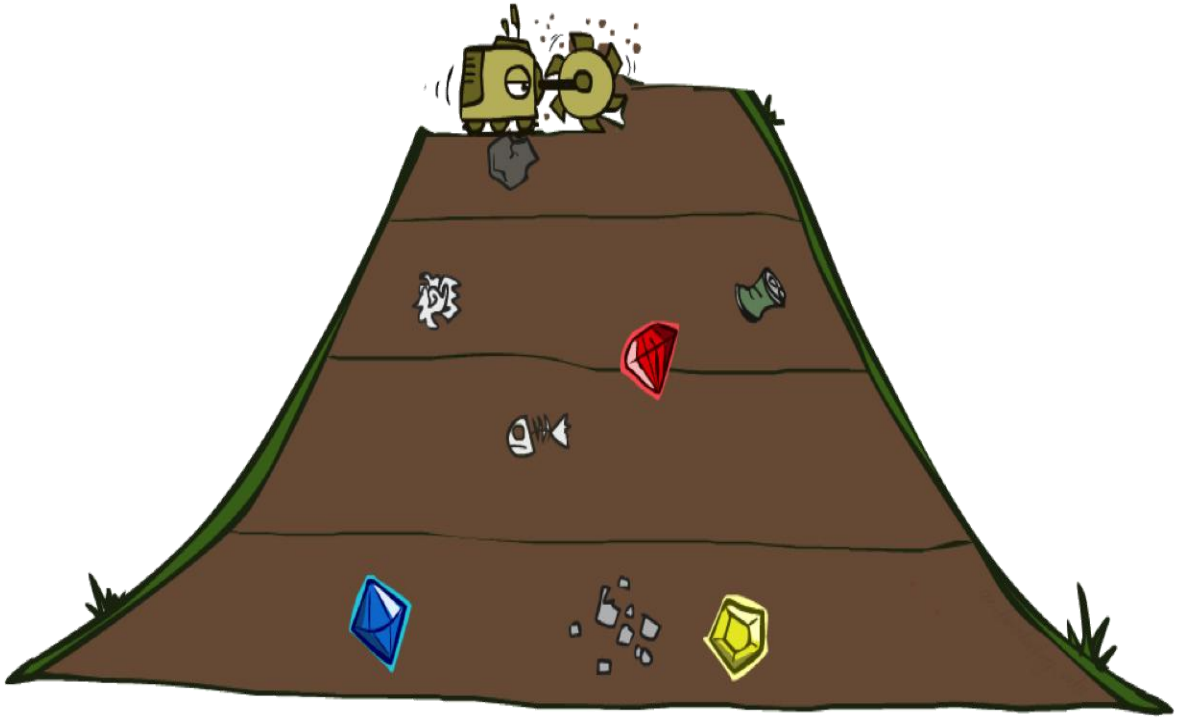
Terrible if m is much larger than d (depth of optimal solution)

DFS

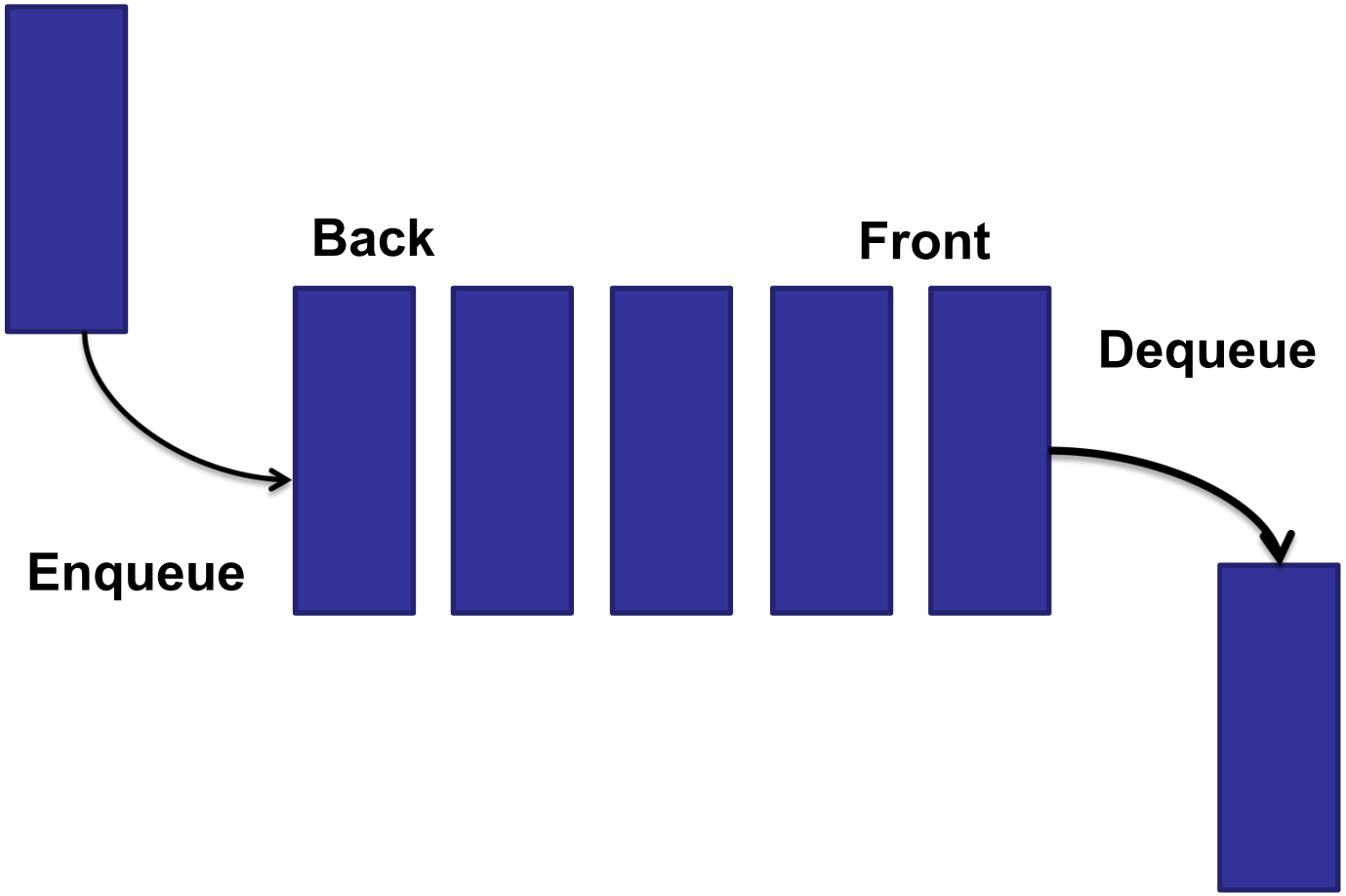
- Infinite paths make DFS incomplete…
- How can we fix this?
- When is DFS optimal?



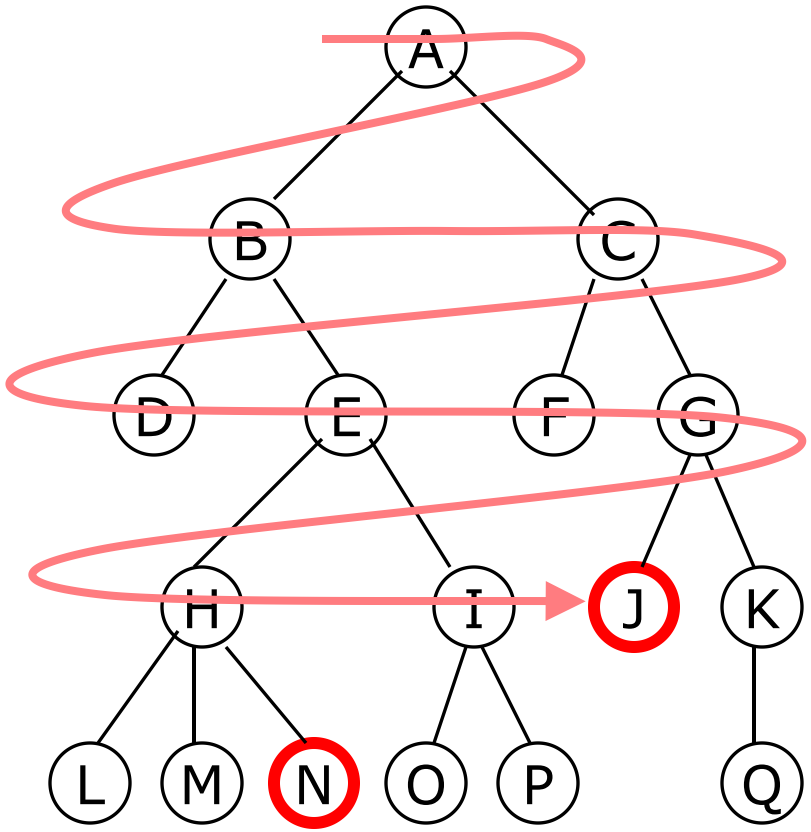
Breadth-First Search



First in, First out



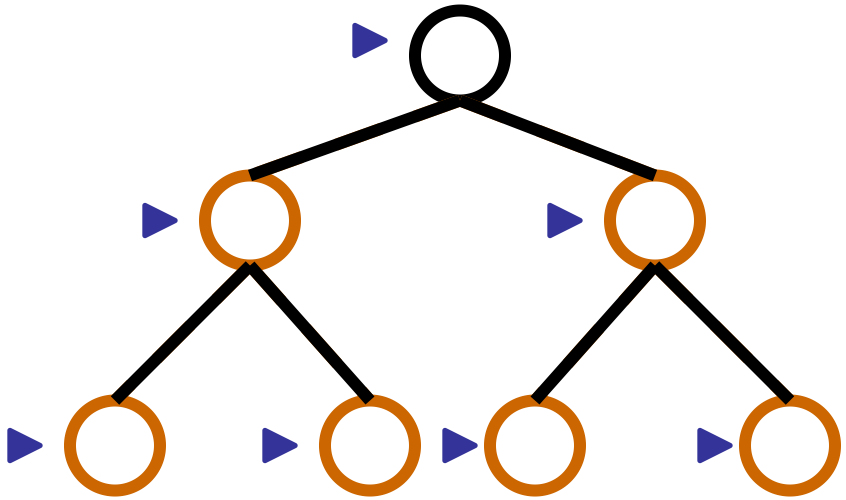
Breadth-first searching



- A **breadth-first** search (**BFS**) explores nodes nearest the root before exploring nodes further away
- For example, after searching **A**, then **B**, then **C**, the search proceeds with **D**, **E**, **F**, **G**
- Node are explored in the order **A B C D E F G H I J K L M N O P Q**
- **J** will be found before **N**

BFS, Tree search

- Expand shallowest unexpanded node
- Implementation: open is a FIFO queue

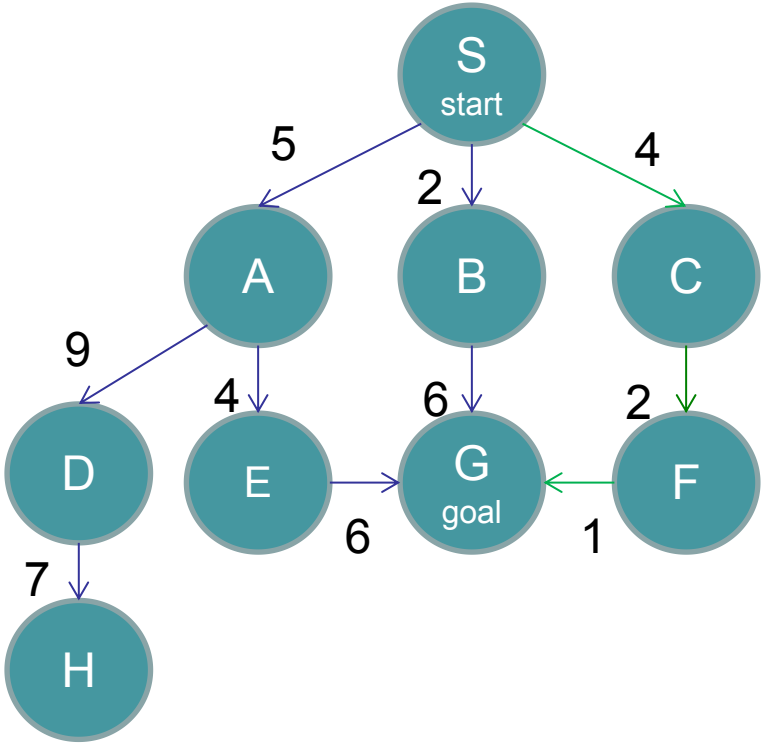


Example: BFS

generalSearch(problem, Queue)

# of nodes tested: 0, expanded: 0

Expnd.node	Open list
	{S}

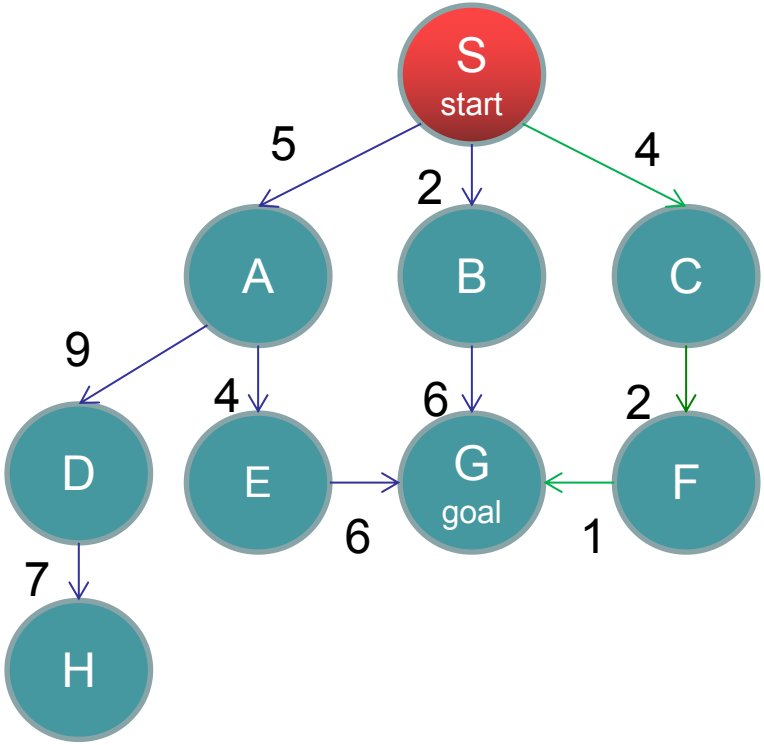


Example: BFS

generalSearch(problem, Queue)

of nodes tested: 1, expanded: 1

Expnd.node	Open list
	{S}
S not goal	{A,B,C}

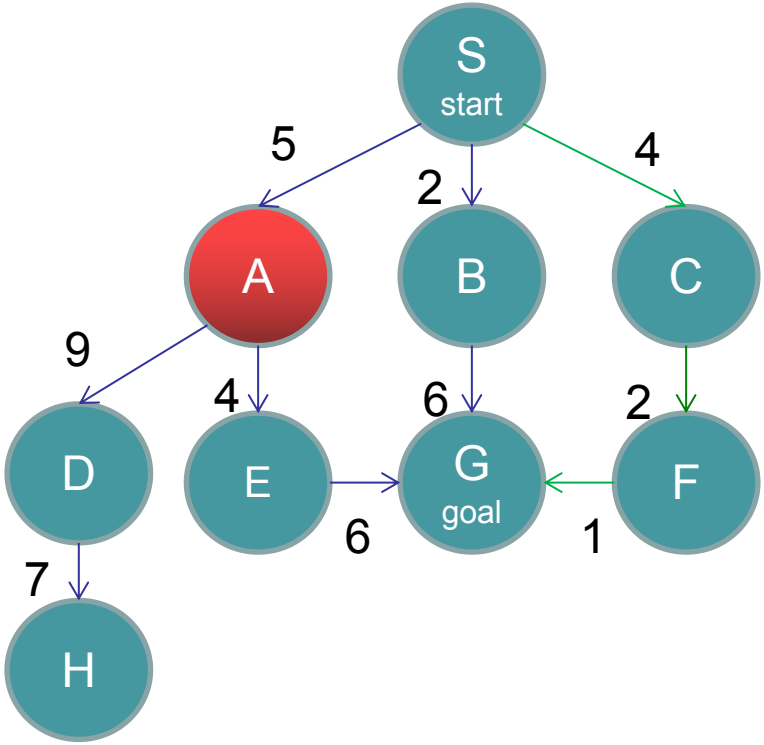


Example: BFS

generalSearch(problem, Queue)

of nodes tested: 2, expanded: 2

Expnd.node	Open list
	{S}
S	{A,B,C}
A not goal	{B,C,D,E}

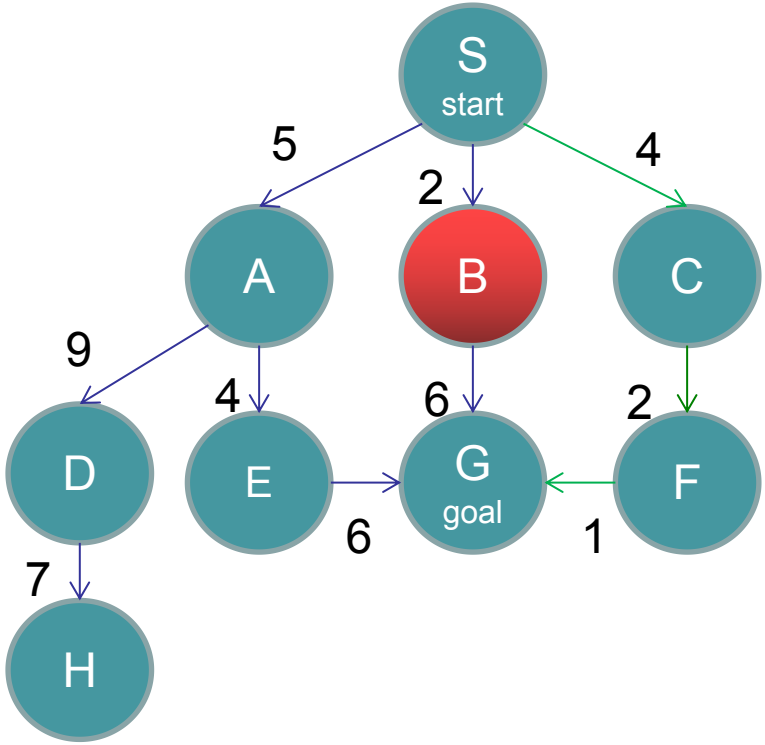


Example: BFS

generalSearch(problem, Queue)

of nodes tested: 3, expanded: 3

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B not goal	{C,D,E,G}

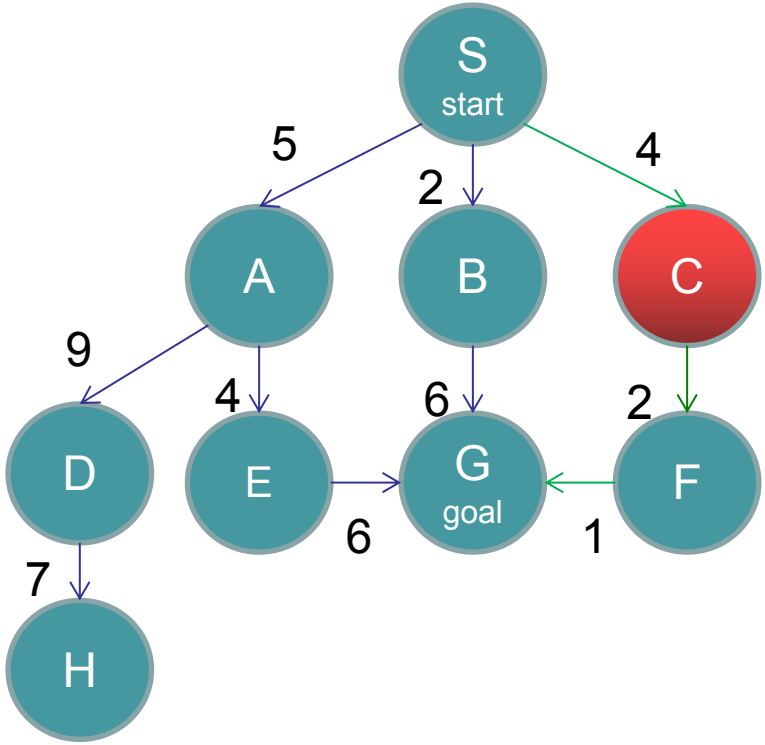


Example: BFS

generalSearch(problem, Queue)

of nodes tested: 4, expanded: 4

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C not goal	{D,E,G,F}

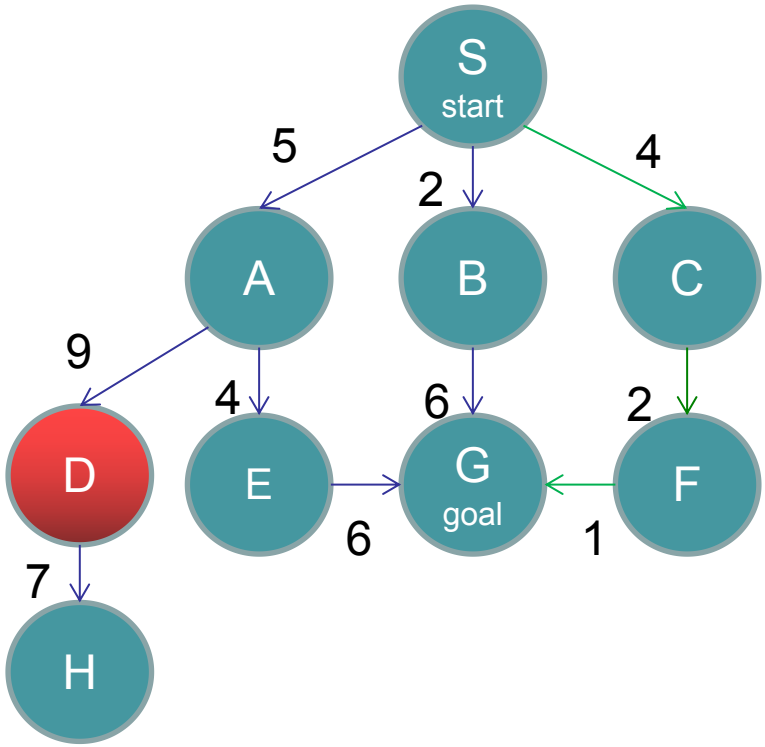


Example: BFS

generalSearch(problem, Queue)

of nodes tested: 5, expanded: 5

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D not goal	{E,G,F,H}

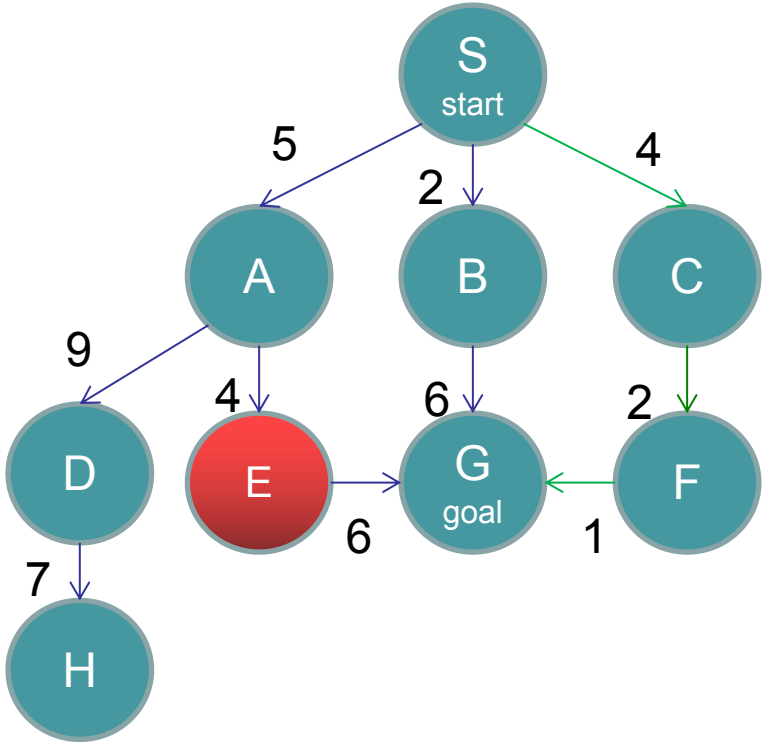


Example: BFS

generalSearch(problem, Queue)

of nodes tested: 6, expanded: 6

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E not goal	{G,F,H,G}

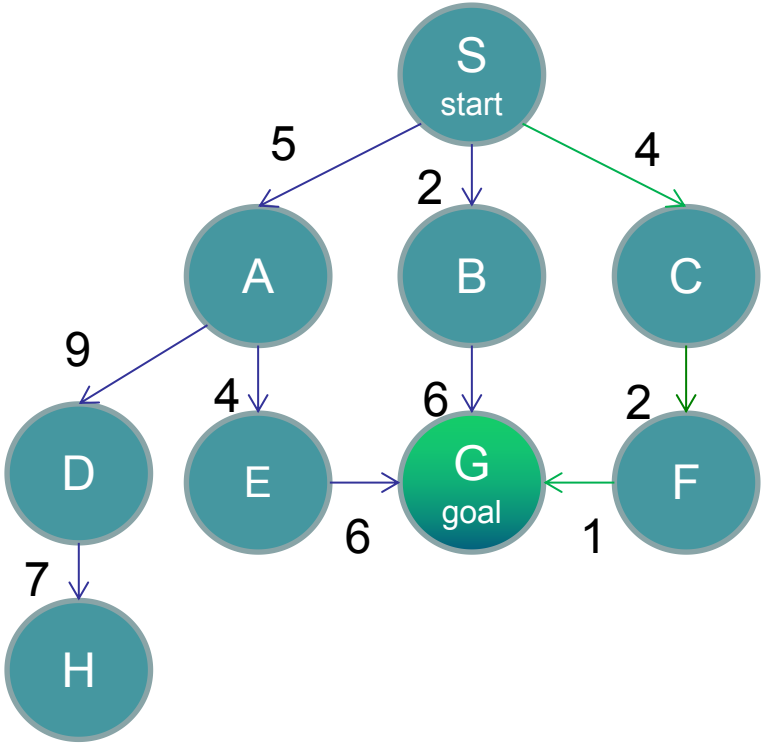


Example: BFS

generalSearch(problem, Queue)

of nodes tested: 7, expanded: 6

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G goal	{F,H,G} no expand

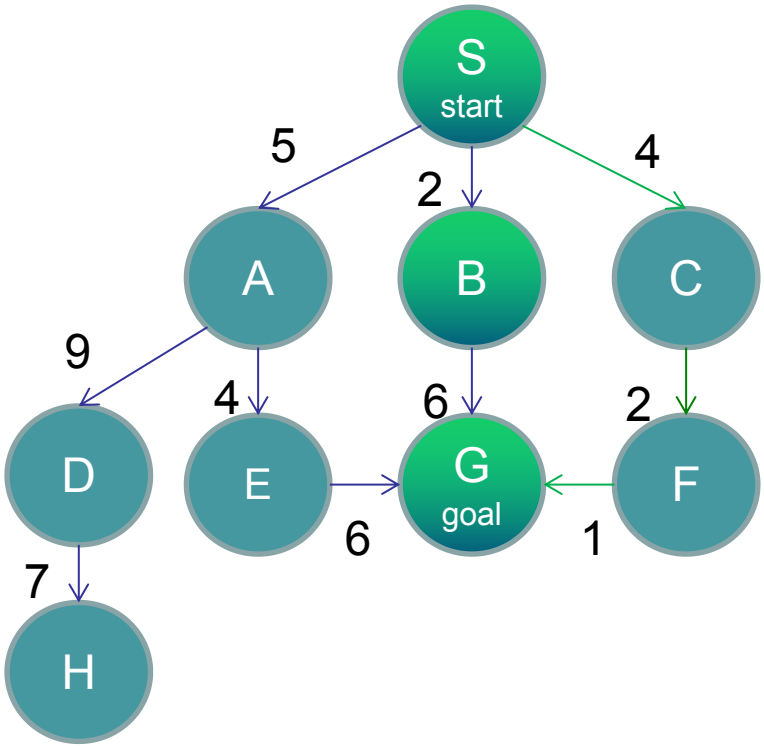


Example: BFS

generalSearch(problem, Queue)

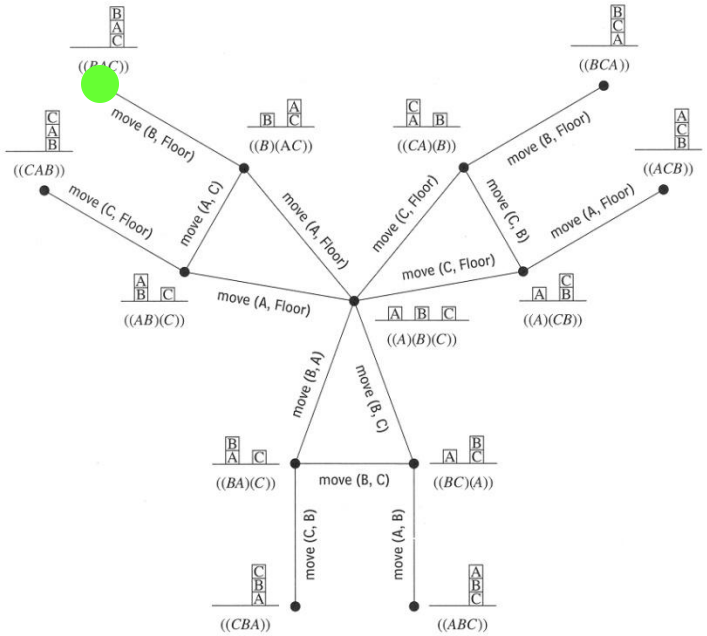
of nodes tested: 7, expanded: 6

Expnd.node	Open list
	{S}
S	{A,B,C}
A	{B,C,D,E}
B	{C,D,E,G}
C	{D,E,G,F}
D	{E,G,F,H}
E	{G,F,H,G}
G	{F,H,G}

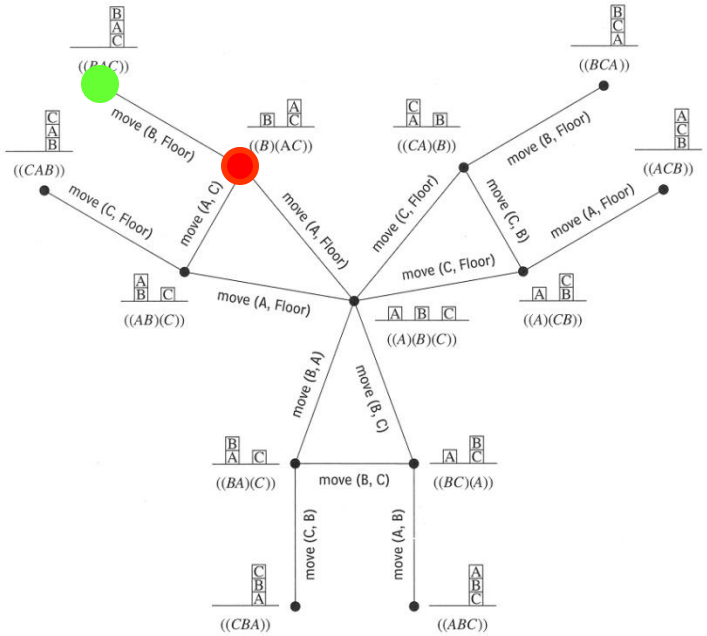


Path: S,B,G
Cost:8

Uninformed Search: Breadth-First



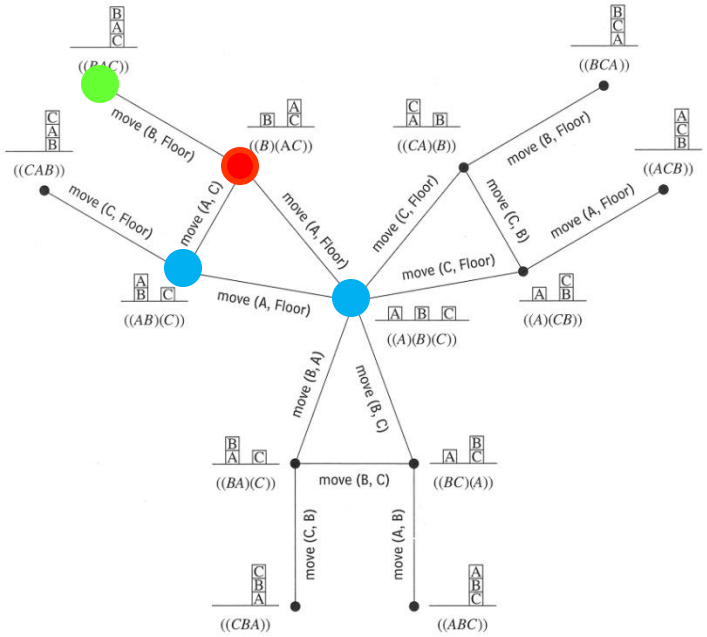
Uninformed Search: Breadth-First

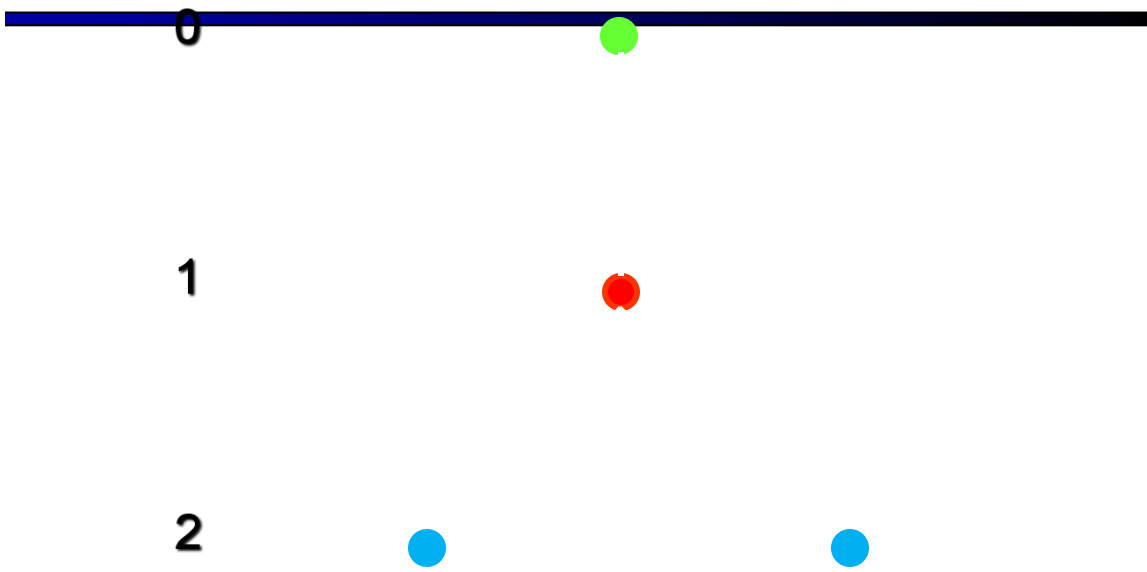


0

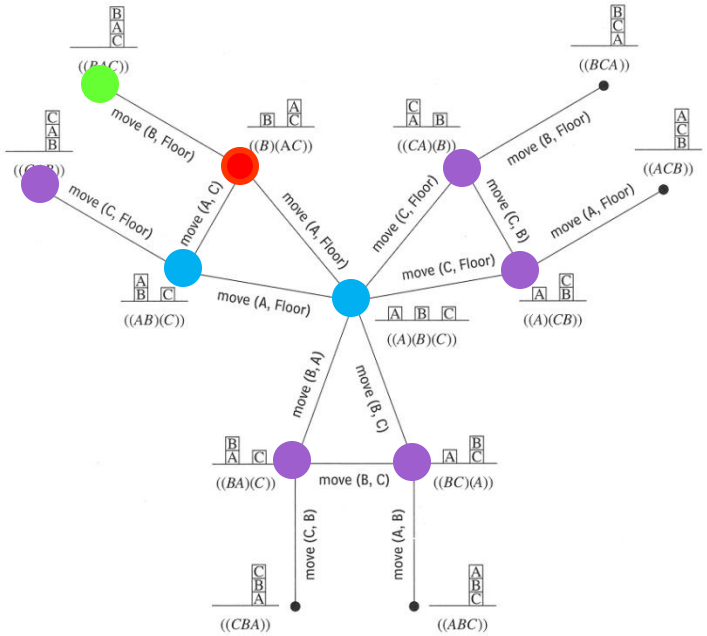
1

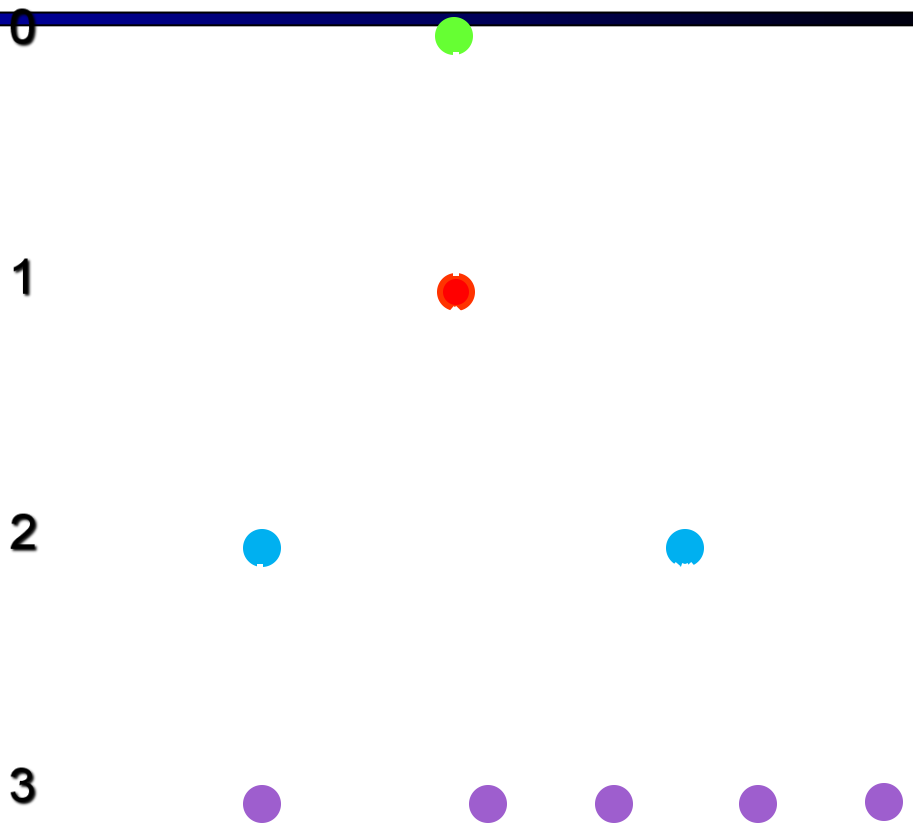
Uninformed Search: Breadth-First



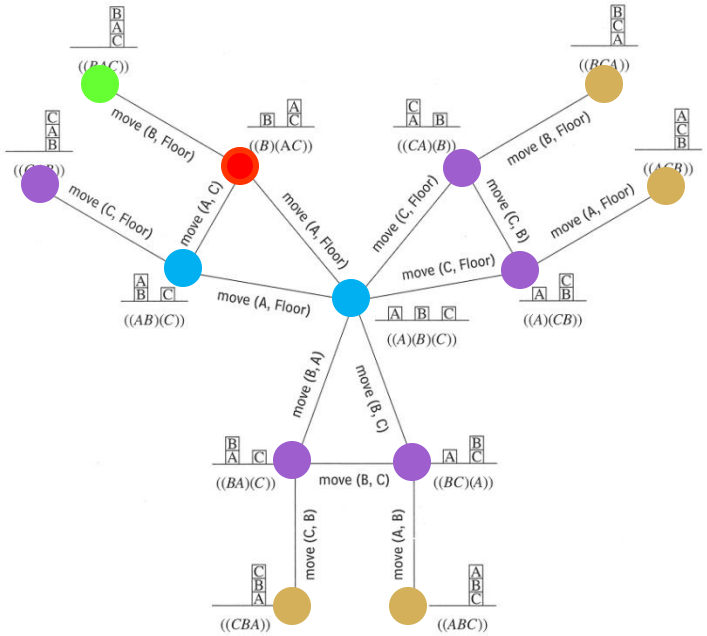


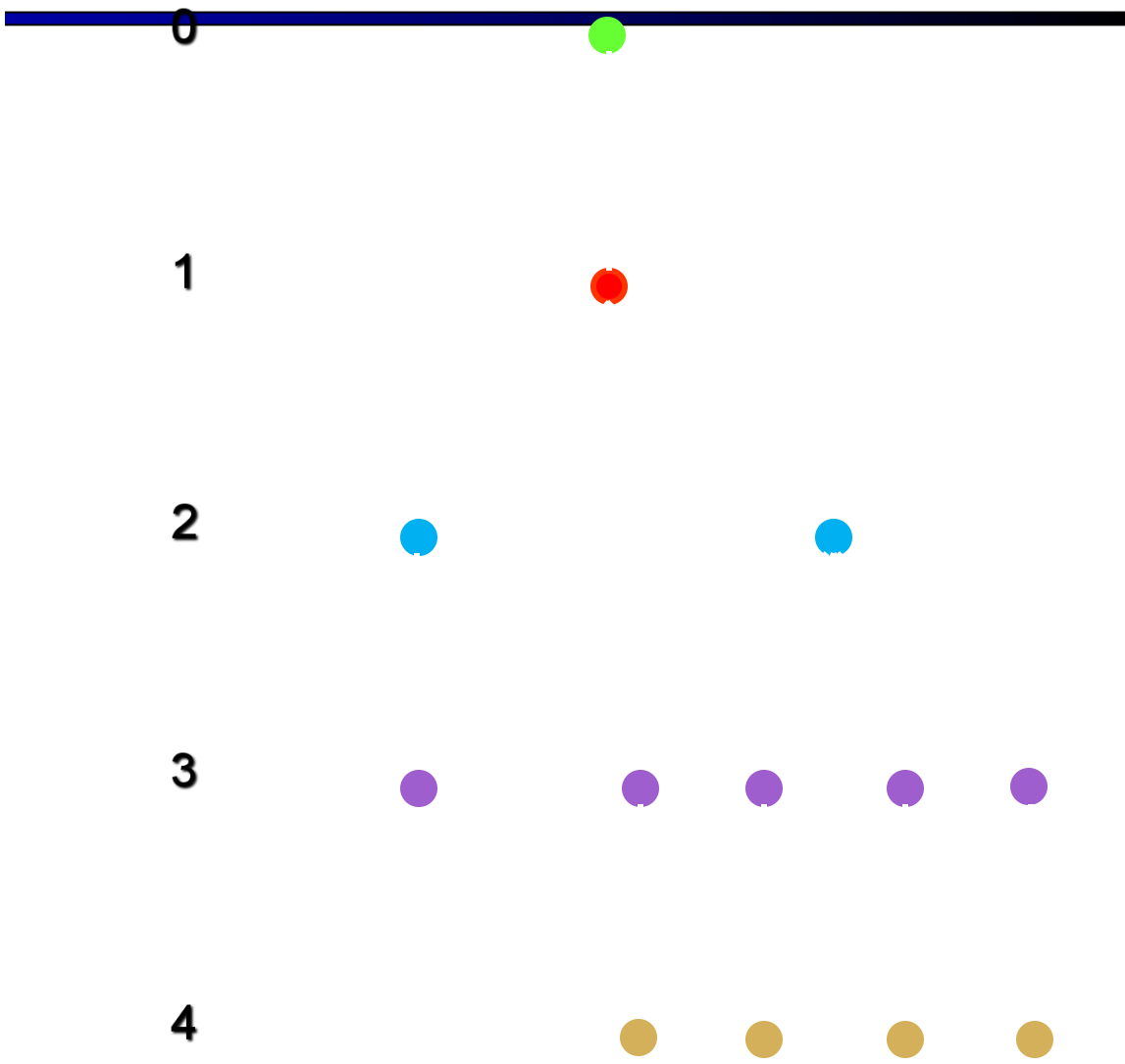
Uninformed Search: Breadth-First



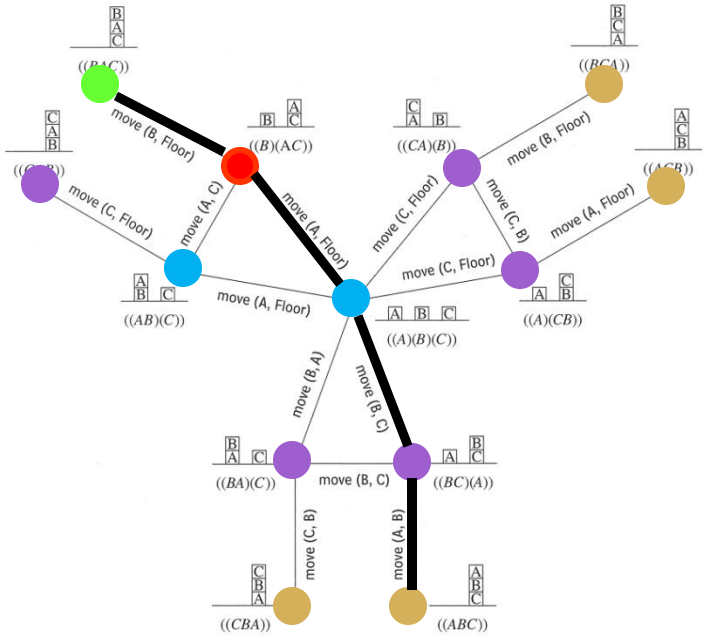


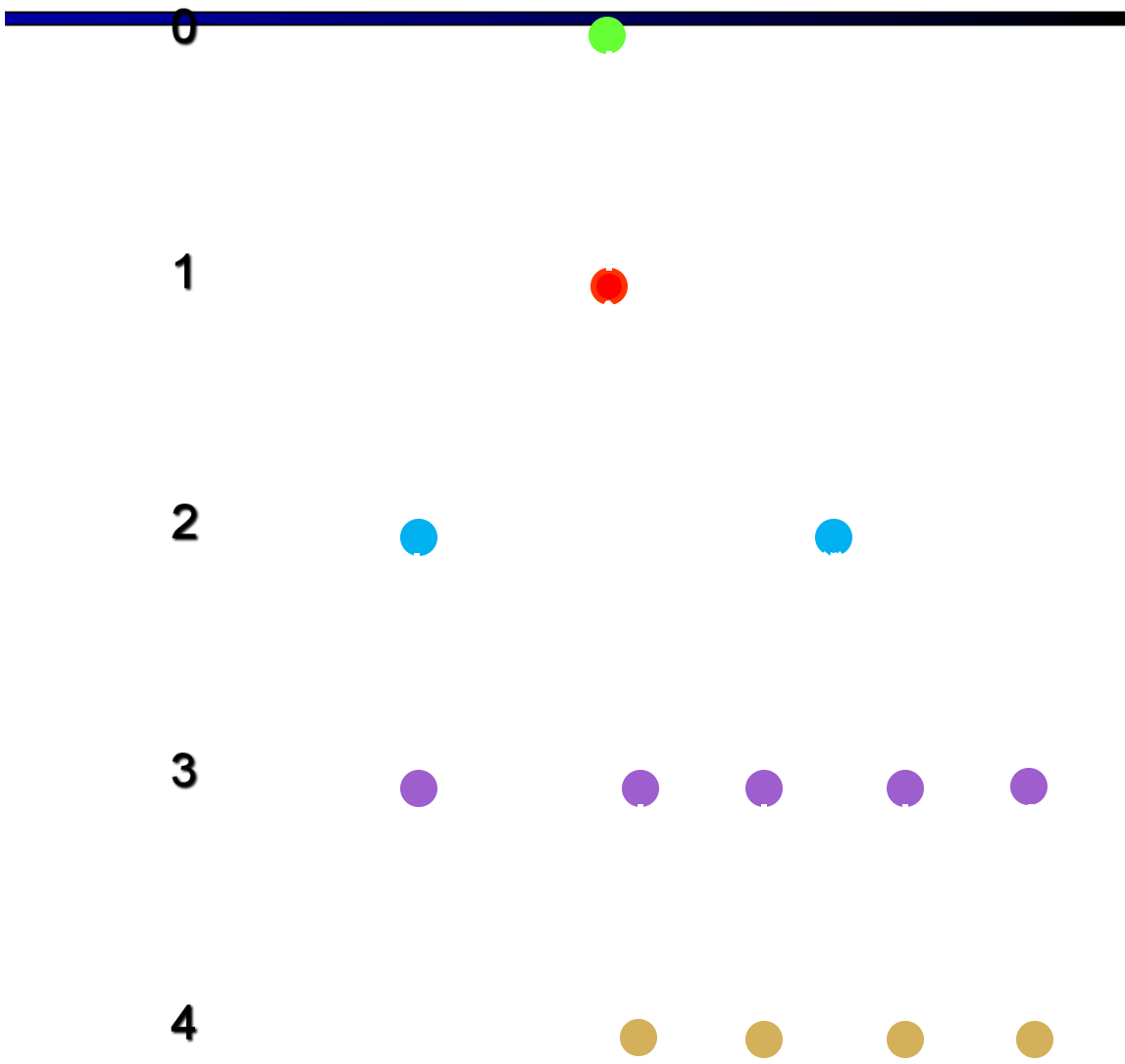
Uninformed Search: Breadth-First

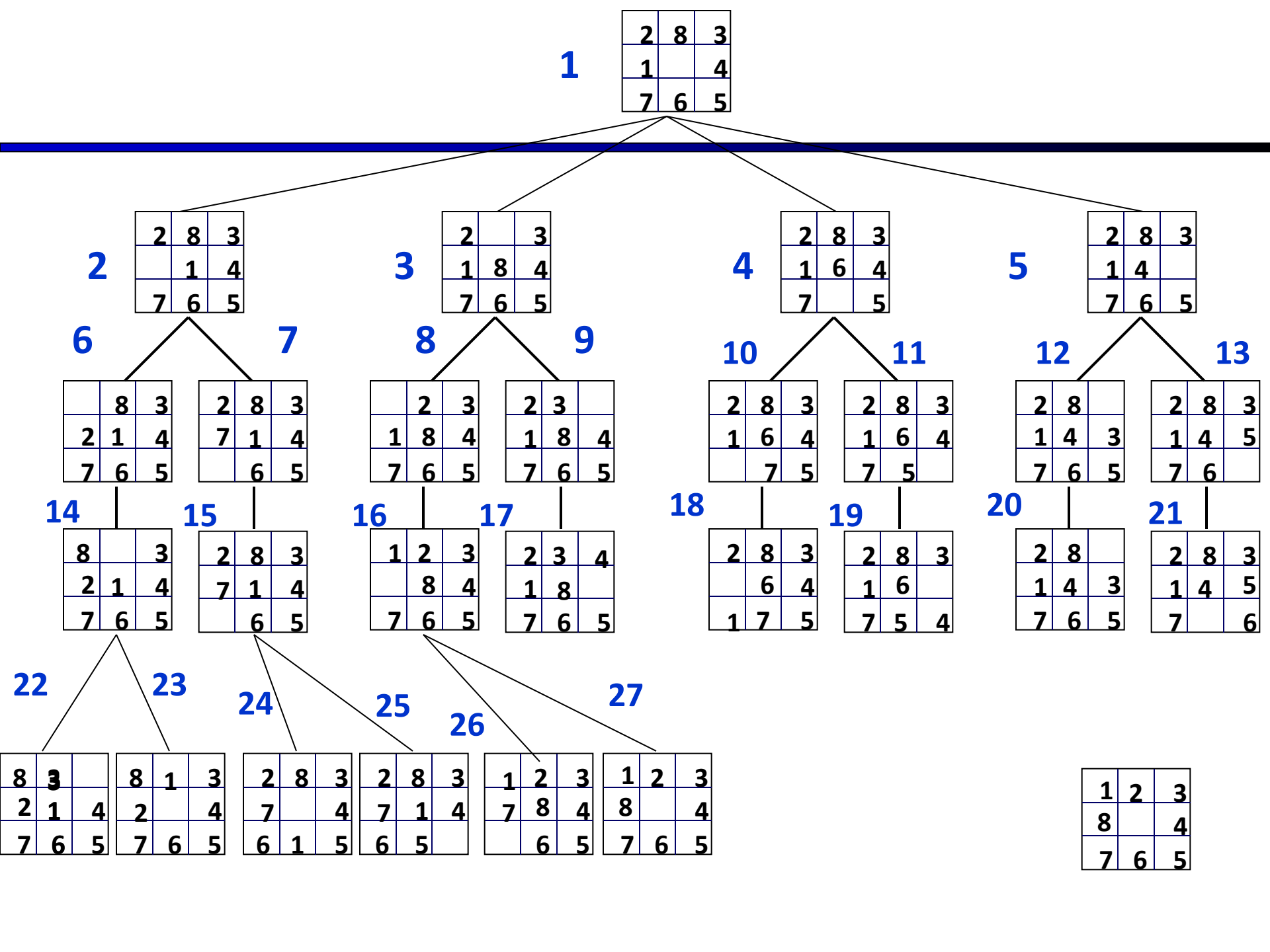




Uninformed Search: Breadth-First






1
2 8 3
1 4
7 6 5
2
2 8 3
1 4
7 6 5
3
2 3
1 8 4
7 6 5
4
2 8 3
1 6 4
7 5
5
2 8 3
1 4
7 6 5
6
8 3
2 1 4
7 6 5
7
2 8 3
7 1 4
6 5
8
2 3
1 8 4
7 6 5
9
2 3
1 8 4
7 6 5
10
2 8 3
1 6 4
7 5
11
2 8 3
1 6 4
7 5
12
2 8
1 4 3
7 6 5
13
2 8 3
1 4 5
7 6
14
8 3
2 1 4
7 6 5
15
2 8 3
7 1 4
6 5
16
1 2 3
8 4
7 6 5
17
2 3 4
1 8
7 6 5
18
2 8 3
6 4
1 7 5
19
2 8 3
1 6
7 5 4
20
2 8
1 4 3
7 6 5
21
2 8 3
1 4 5
7 6
22
8 3
2 1 4
7 6 5
23
8 1 3
2 4
7 6 5
24
2 8 3
7 4
6 1 5
25
2 8 3
7 1 4
6 5
26
1 2 3
7 8 4
6 5
27
1 2 3
8 4
7 6 5
1 2 3
8 4
7 6 5

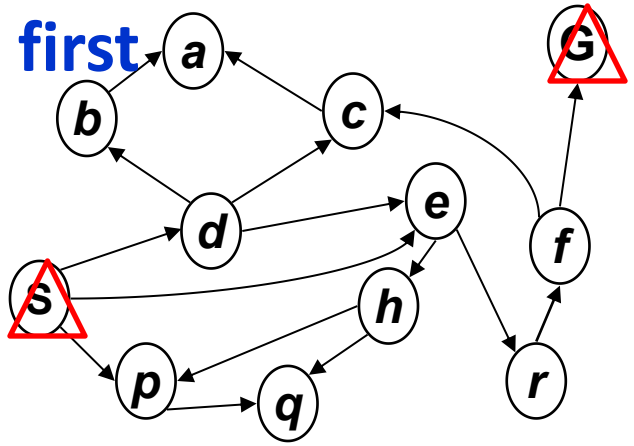
Breadth-First Search

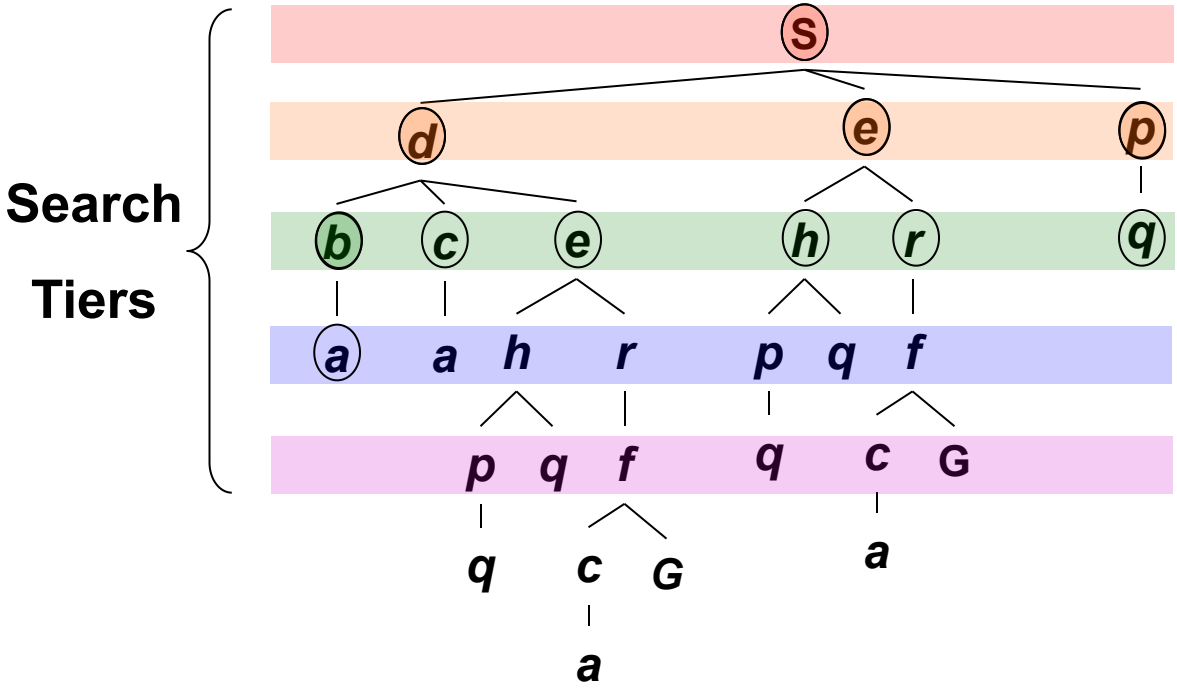
Strategy: expand a shallowest node first

Implementation:

Fringe is a FIFO queue

First in, First out

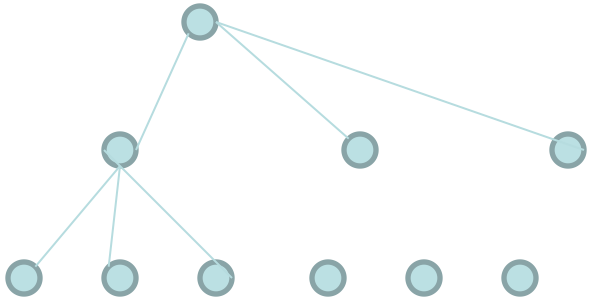




Evaluating Breadth First Search

- **Observations**
  - **Very systematic**
  - **If there is a solution breadth first search is guaranteed to find it**
  - **If there are several solutions then breadth first search will always find the shallowest (having little depth) goal state first and**
  - **if the cost of a solution is a non-decreasing function of the depth then it will always find the cheapest solution**

Evaluating Breadth First Search

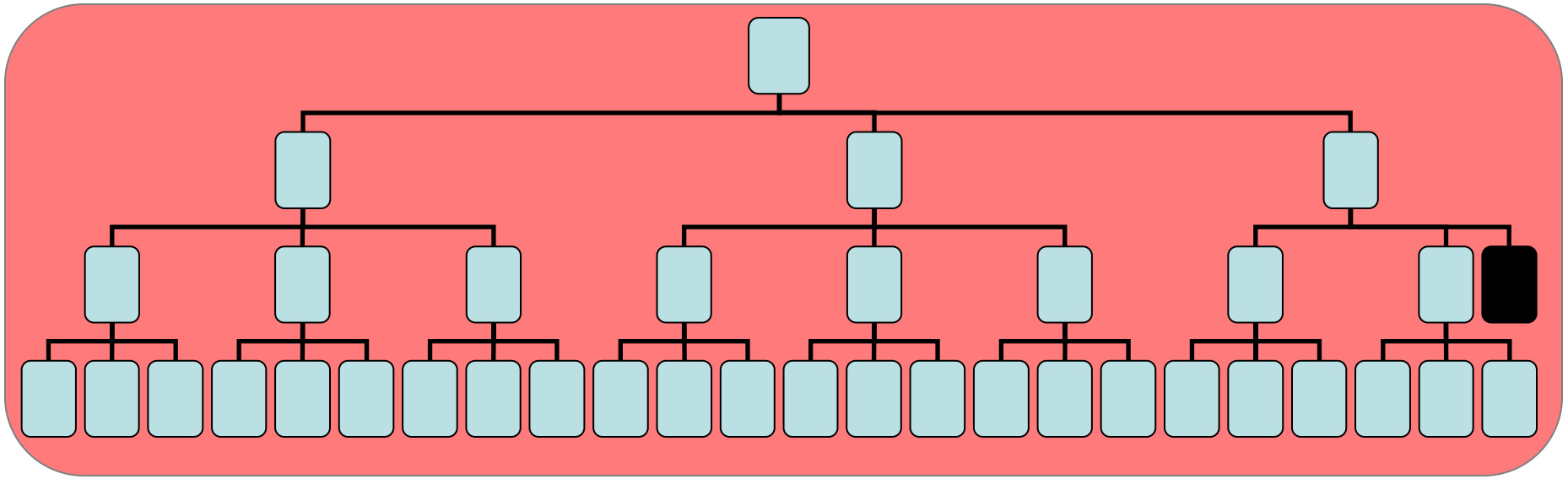


- **Space Complexity : $1 + b + b^2 + b^3 + \ldots + b^d$ i.e $O(b^d)$**
- **Time Complexity : $1 + b + b^2 + b^3 + \ldots + b^d$ i.e. $O(b^d)$**

- **Where b is the branching factor and d is the depth of the search tree**

- **Note : The space/time complexity could be less as the solution could be found anywhere on the d^{th} level.**

Evaluating Breadth First Search

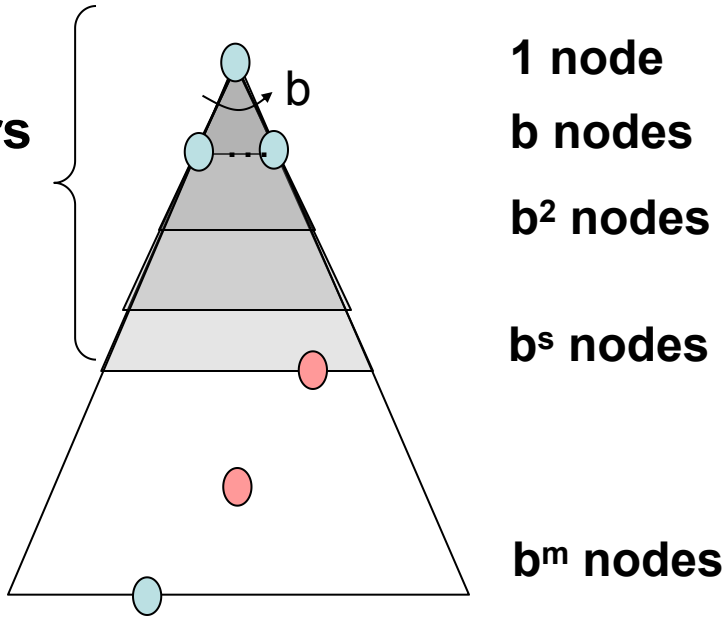


- Every node that is generated must remain in memory so space complexity is therefore as time complexity

- Memory requirements are a bigger problem for breadth first search than is the execution

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 fringe 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?
  - Only if costs are all 1 (more on costs later)



BFS complexity: concretely

- Assume branching factor b = 10

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.

Breadth-First vs. Depth-First

- **Uninformed breadth-first search:**

- Requires the construction and storage of almost the complete search tree.

  → Space complexity for search depth n is $O(e^n)$.

- Is guaranteed to find the shortest path to a solution.

- **Uninformed depth-first search:**

- Requires the storage of only the current path and the branches from this path that were already visited.

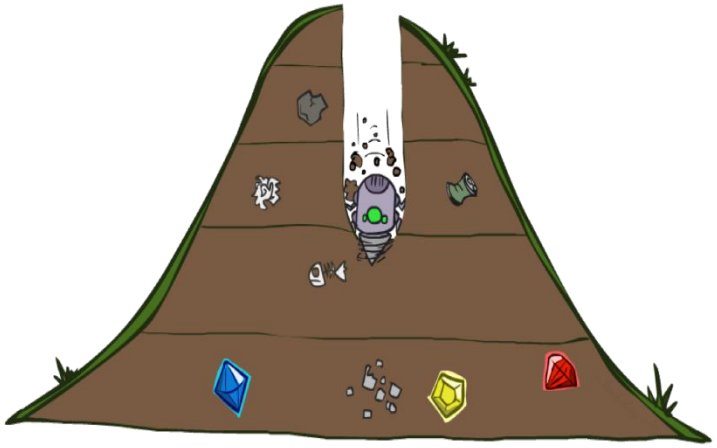
  → Space complexity for search depth n is O(n).

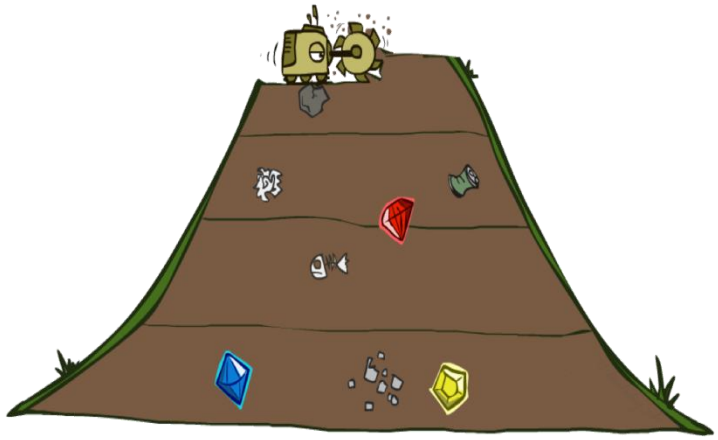
- May search unnecessarily deep for a shallow goal.

Quiz: DFS vs BFS

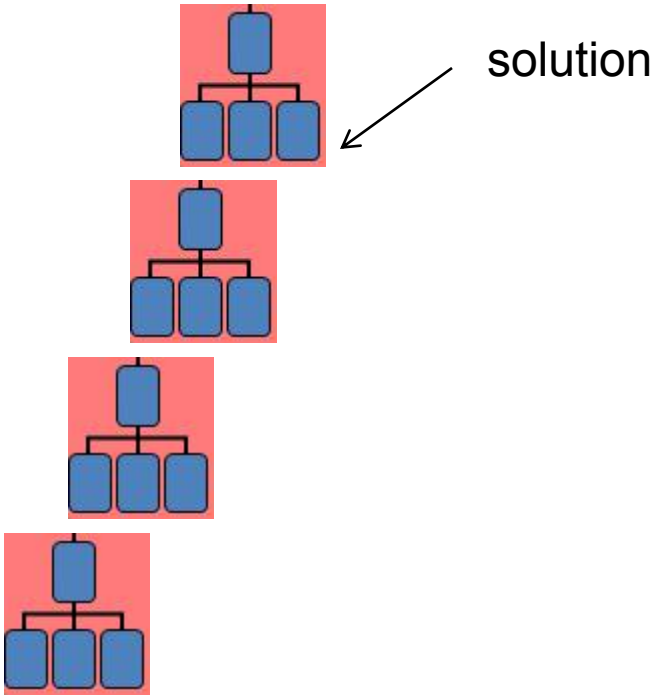
- **When will BFS outperform DFS?**
  - The branch factor is relatively small.
  - The depth of the optimal solution is relatively shallow.

- **When will DFS outperform BFS?**
  - The tree is deep and the answer is frequent.





Depth-limited search



- The depth first search disadvantage is that the algorithm go deep and deep while solution may be near root
- It is depth-first search with an imposed limit on the depth of exploration, to guarantee that the algorithm ends.

Depth-limited search

= depth-first search with depth limit *l*,

i.e., nodes at depth *l* have no successors

- **Recursive implementation:**

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Iterative Deepening Search,Tree search

- **What?**
  - **A general strategy to find best depth limit *l*.**
    - **Goals is found at depth *d*, the depth of the shallowest goal-node.**
  - **Combines benefits of DF- and BF-search**

```
function ITERATIVE_DEEPENING_SEARCH(problem)
return a solution or failure
        inputs: problem
        for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED_SEARCH(problem, depth)
                if result ≠ cuttoff then return result
```

Iterative Deepening

- **Iterative deepening** is an interesting combination of breadth-first and depth-first strategies:
  - Space complexity for search depth n is O(n).
  - Is guaranteed to find the shortest path to a solution without searching unnecessarily deep.
- **How does it work?**
- The idea is to successively apply depth-first searches with **increasing depth bounds** (maximum search depth).

Iterative Deeping Search

- The algorithm consists of iterative, depth-first searches, with a maximum depth that increases at each iteration. Maximum depth at the beginning is 1.
- Only the actual path is kept in memory; nodes are regenerated at each iteration.
- DFS problems related to infinite branches are avoided.
- To guarantee that the algorithm ends if there is no solution, a general maximum depth of exploration can be defined.

Iterative Deeping Search

- **Depth-limited search, with**
  - **depth = 0**
  - **then again with depth = 1**
  - **then again with depth = 2**
  - **... until you find a solution**

5-Iterative Deeping Search

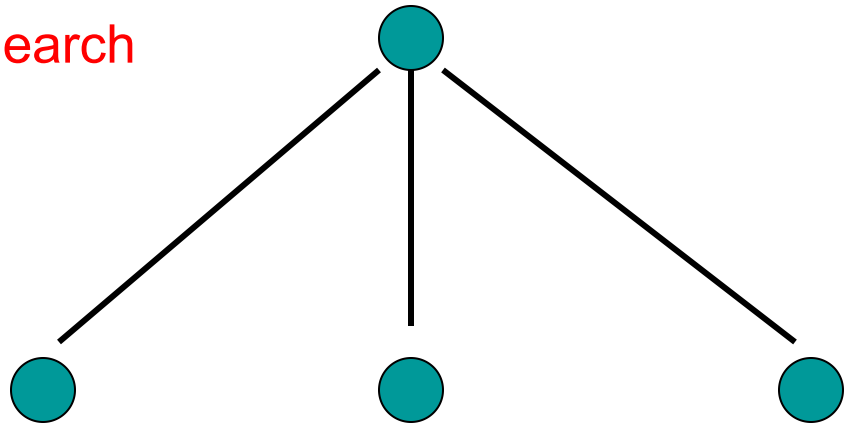
Depth = 0

Depth Limit = 0

5-Iterative Deeping Search

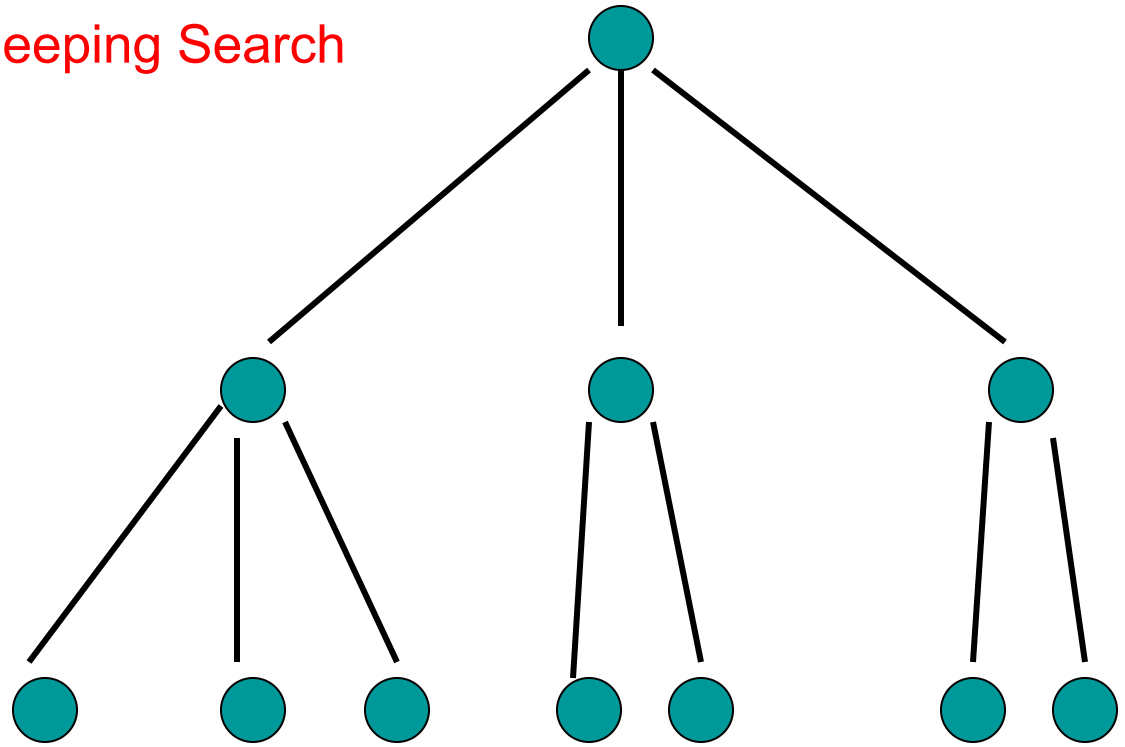
Depth = 0

Depth Limit = 1

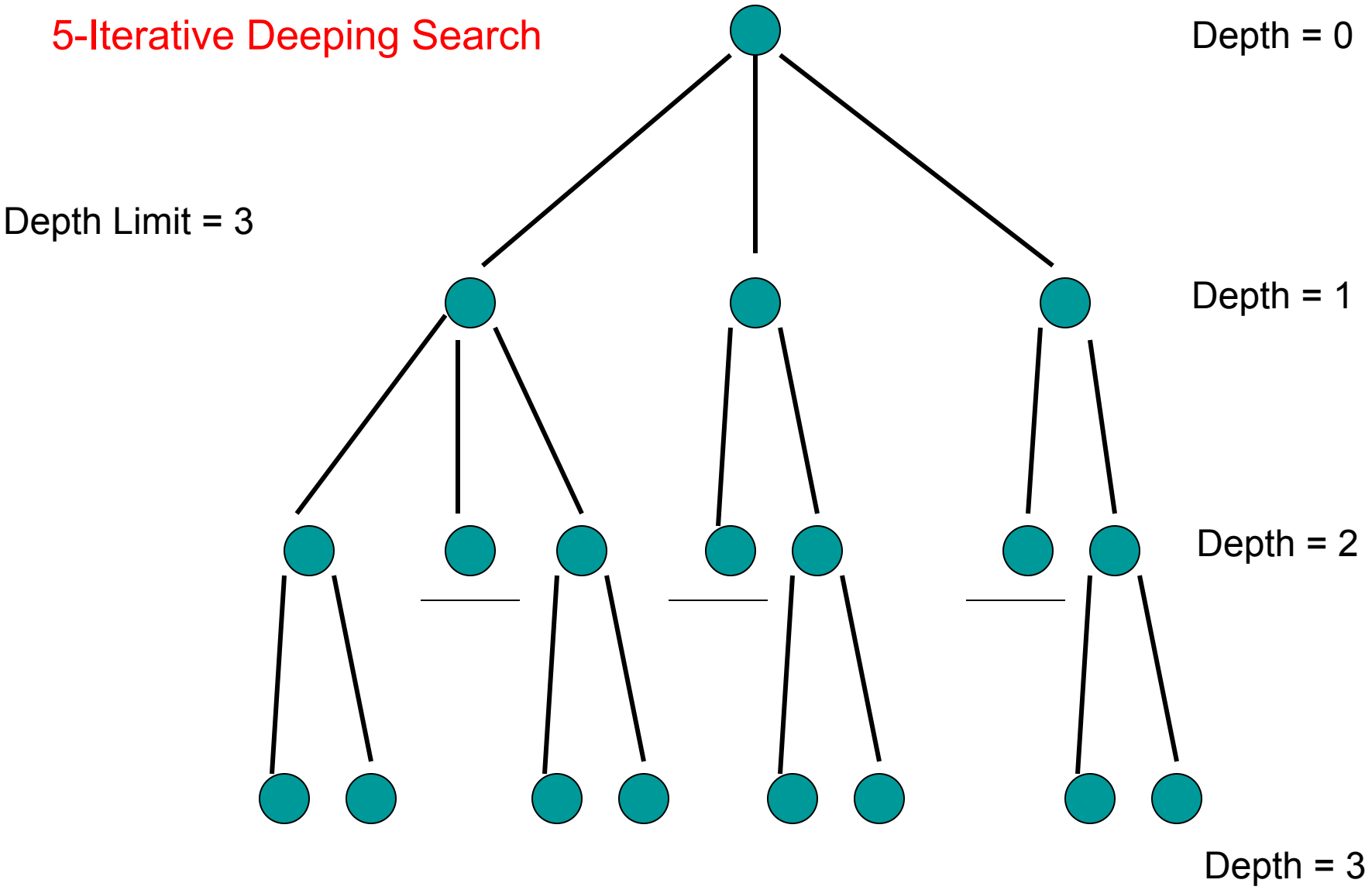


5-Iterative Deeping Search

Depth Limit = 2



5-Iterative Deeping Search



ID-search, example

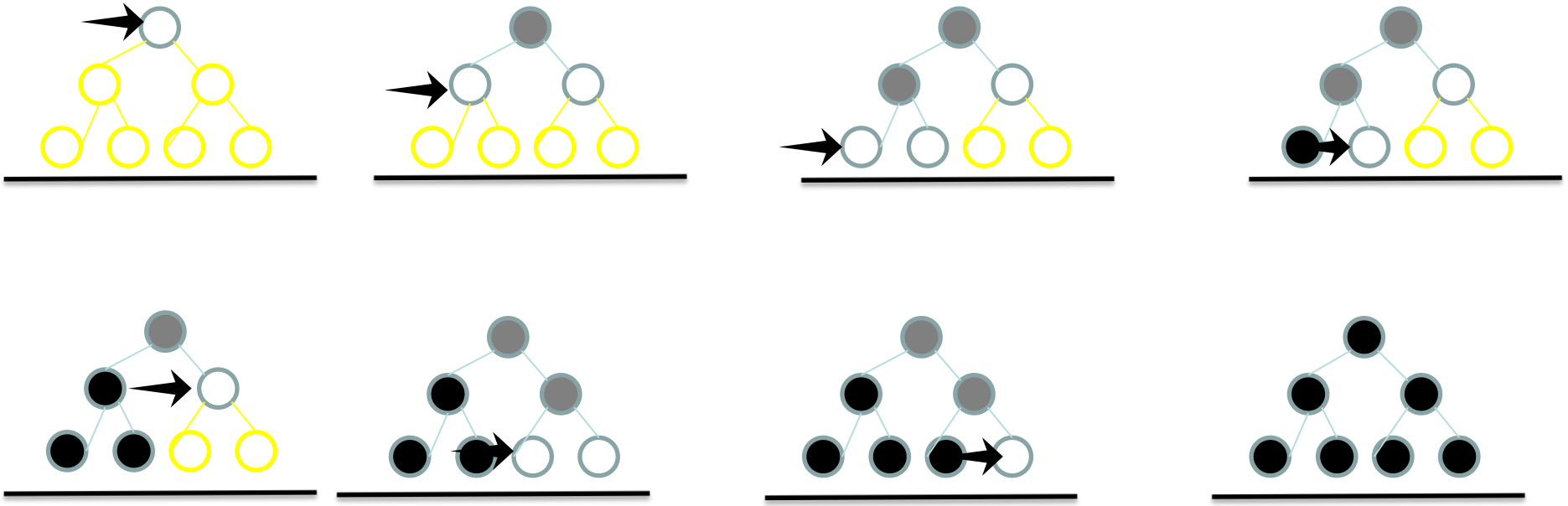
- Limit=0

ID-search, example

- Limit=1

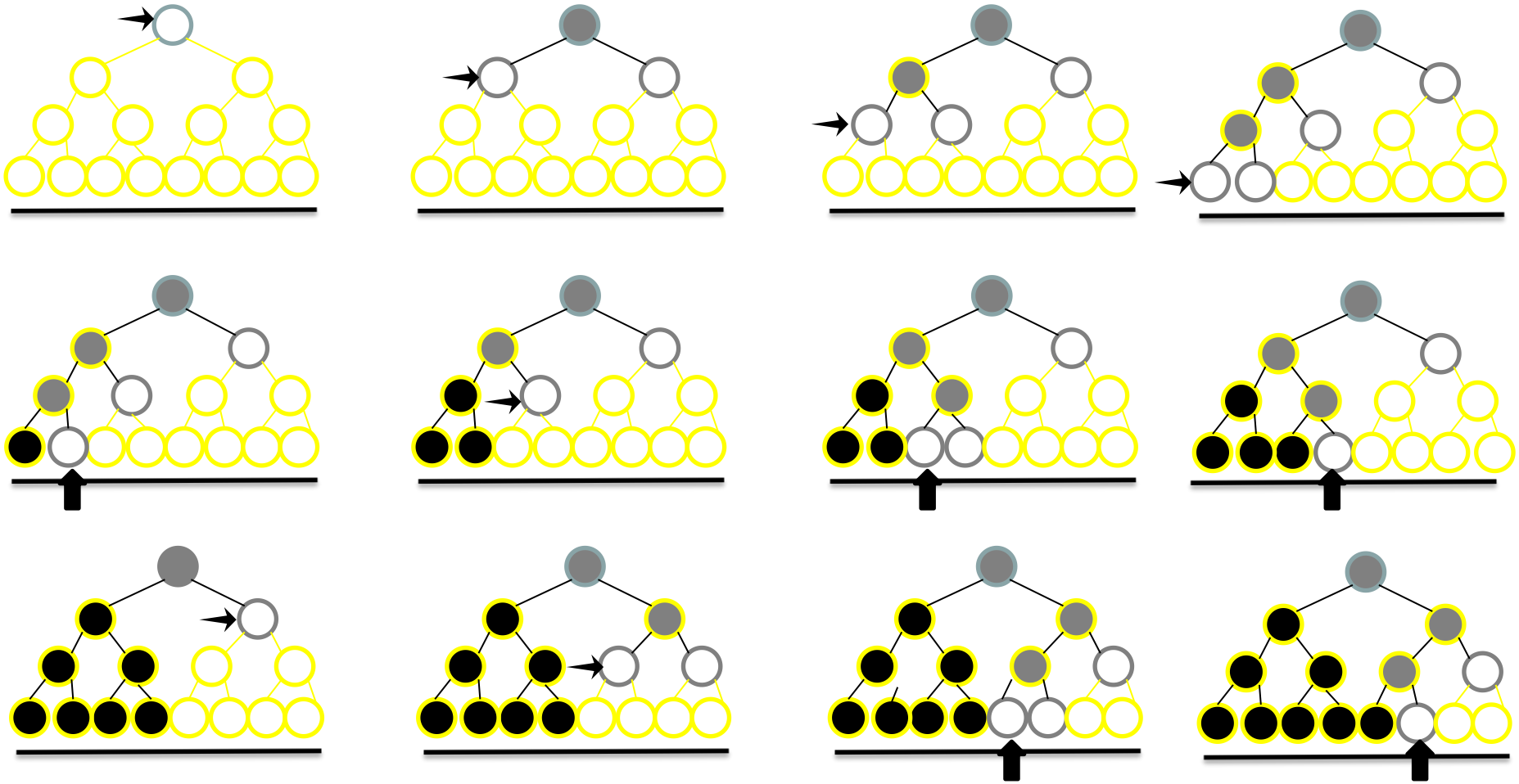
ID-search, example

- Limit=2

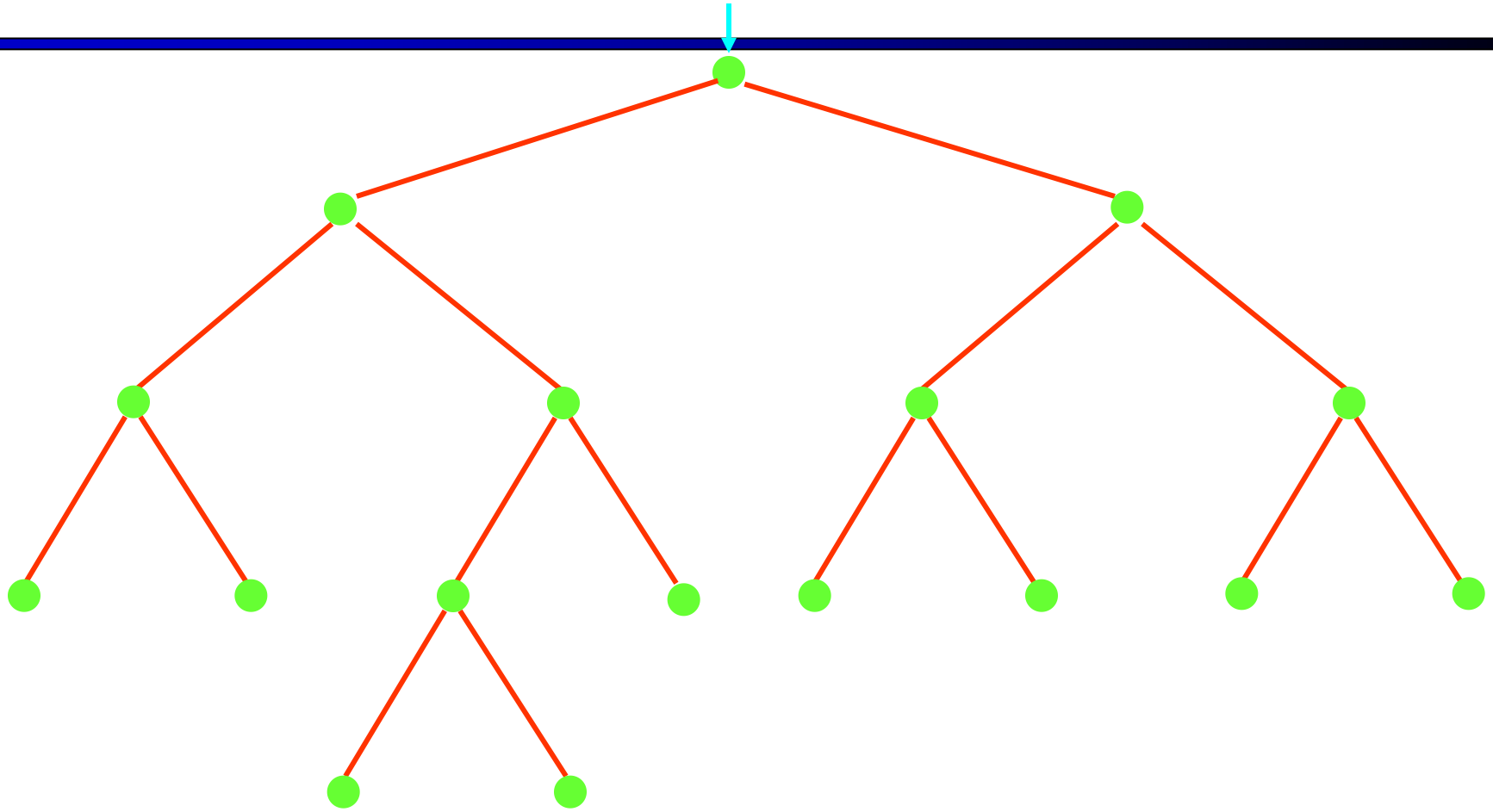


ID-search, example

- Limit=3

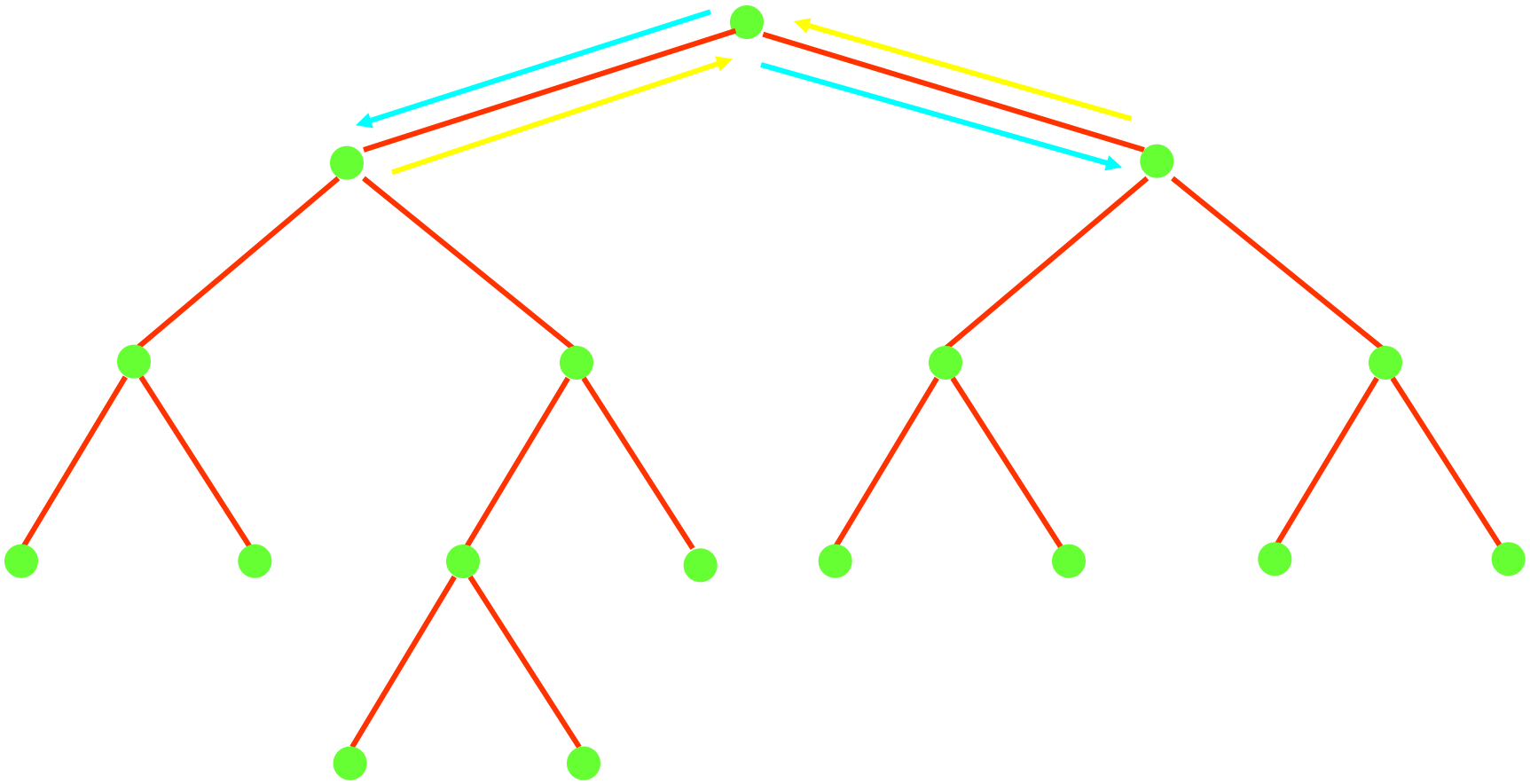


Iterative Deepening



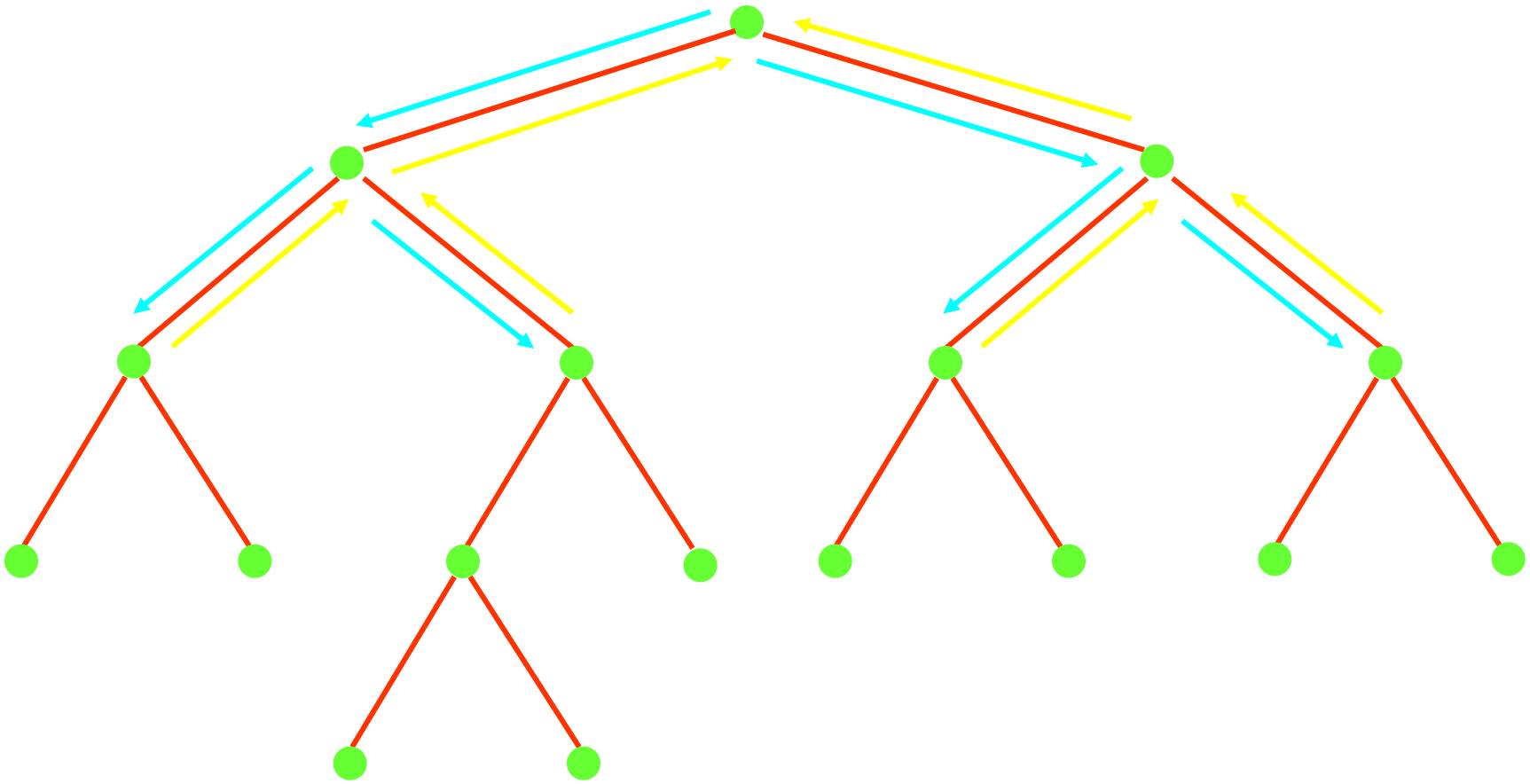
- maximum search depth = 0 (only root is tested)

Iterative Deepening



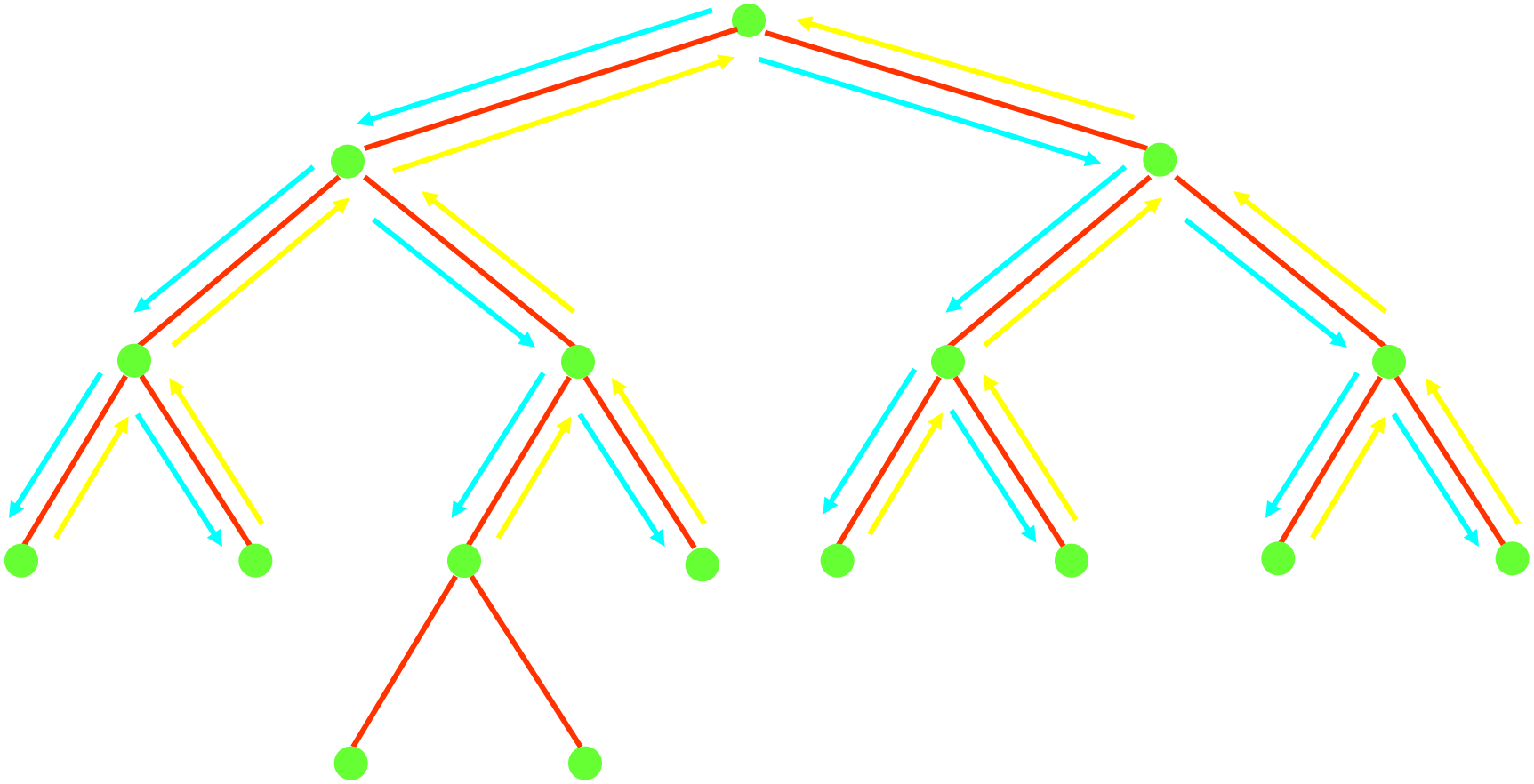
- maximum search depth = 1

Iterative Deepening



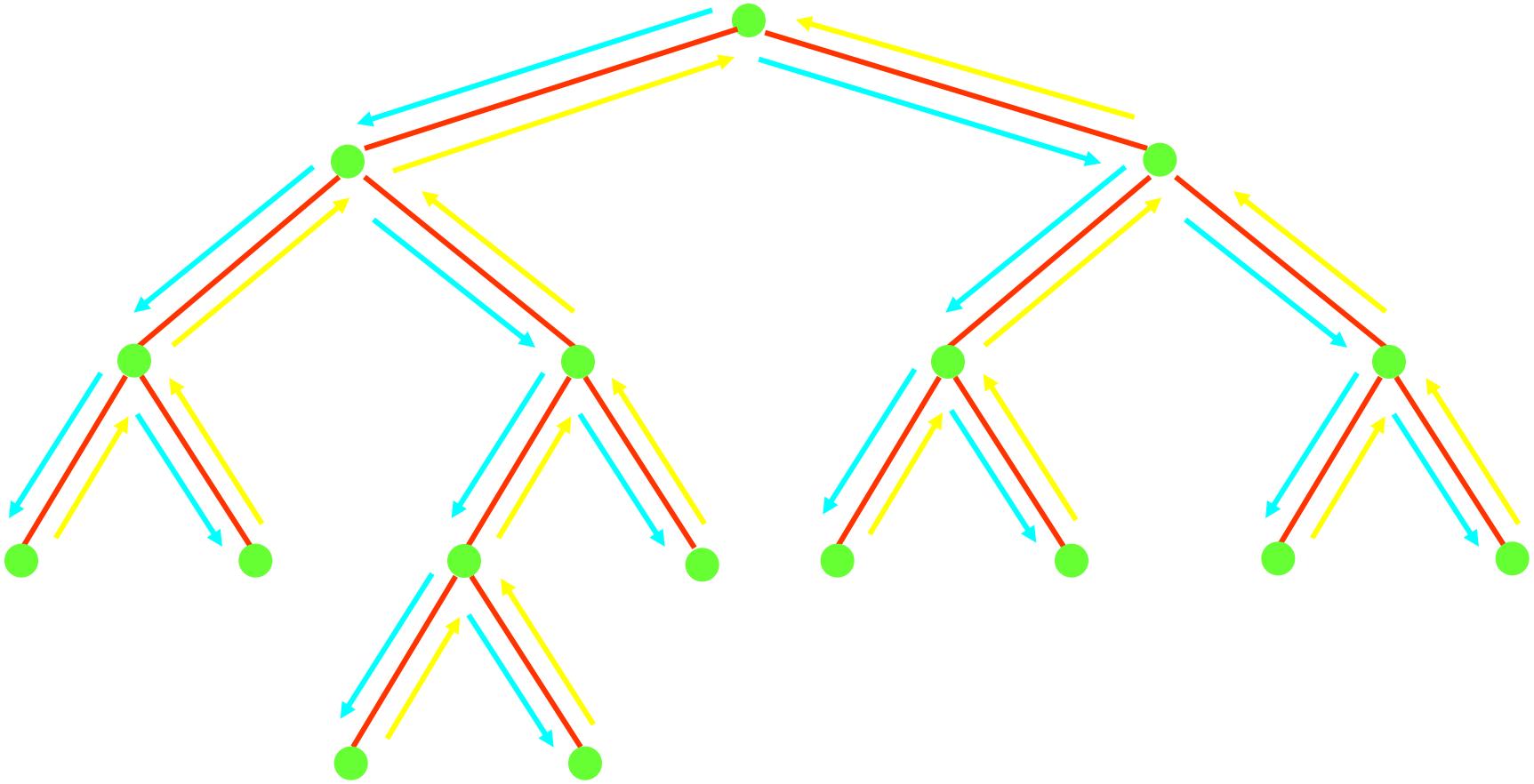
- maximum search depth = 2

Iterative Deepening



- maximum search depth = 3

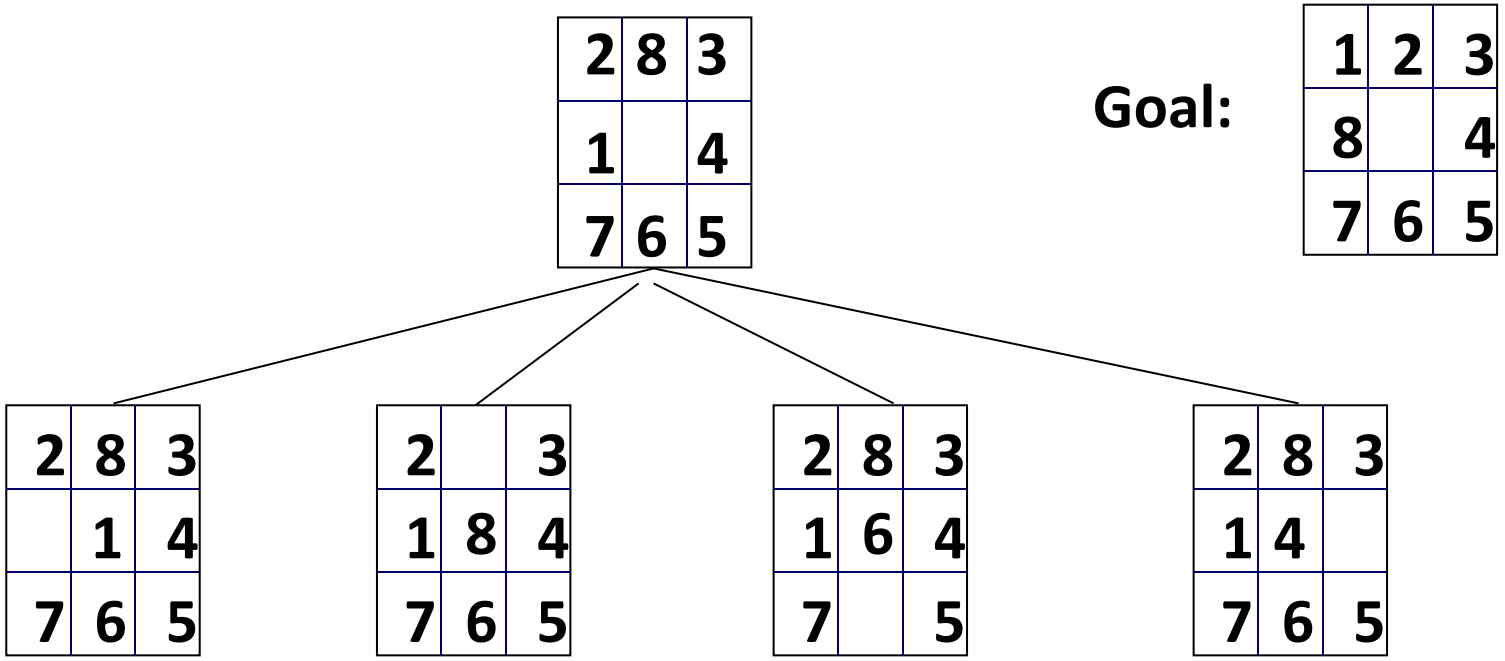
Iterative Deepening



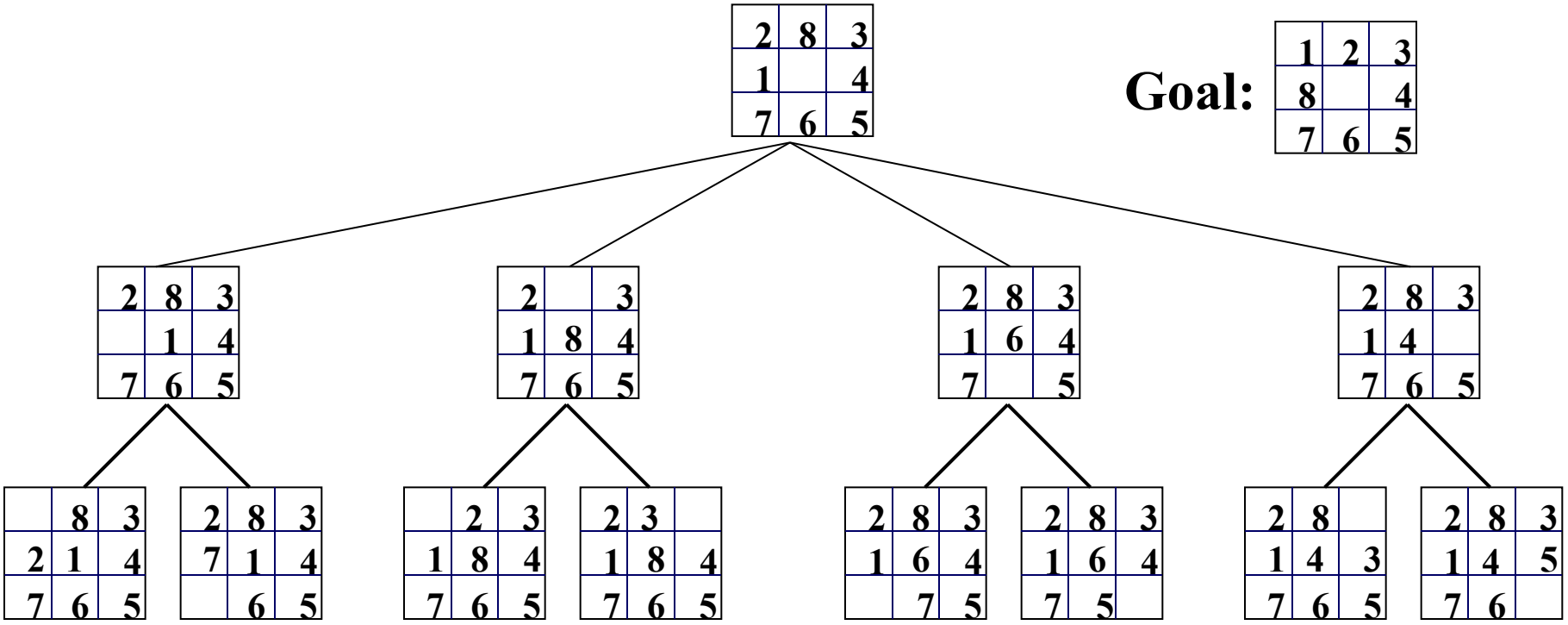
- maximum search depth = 4

ID-search, example

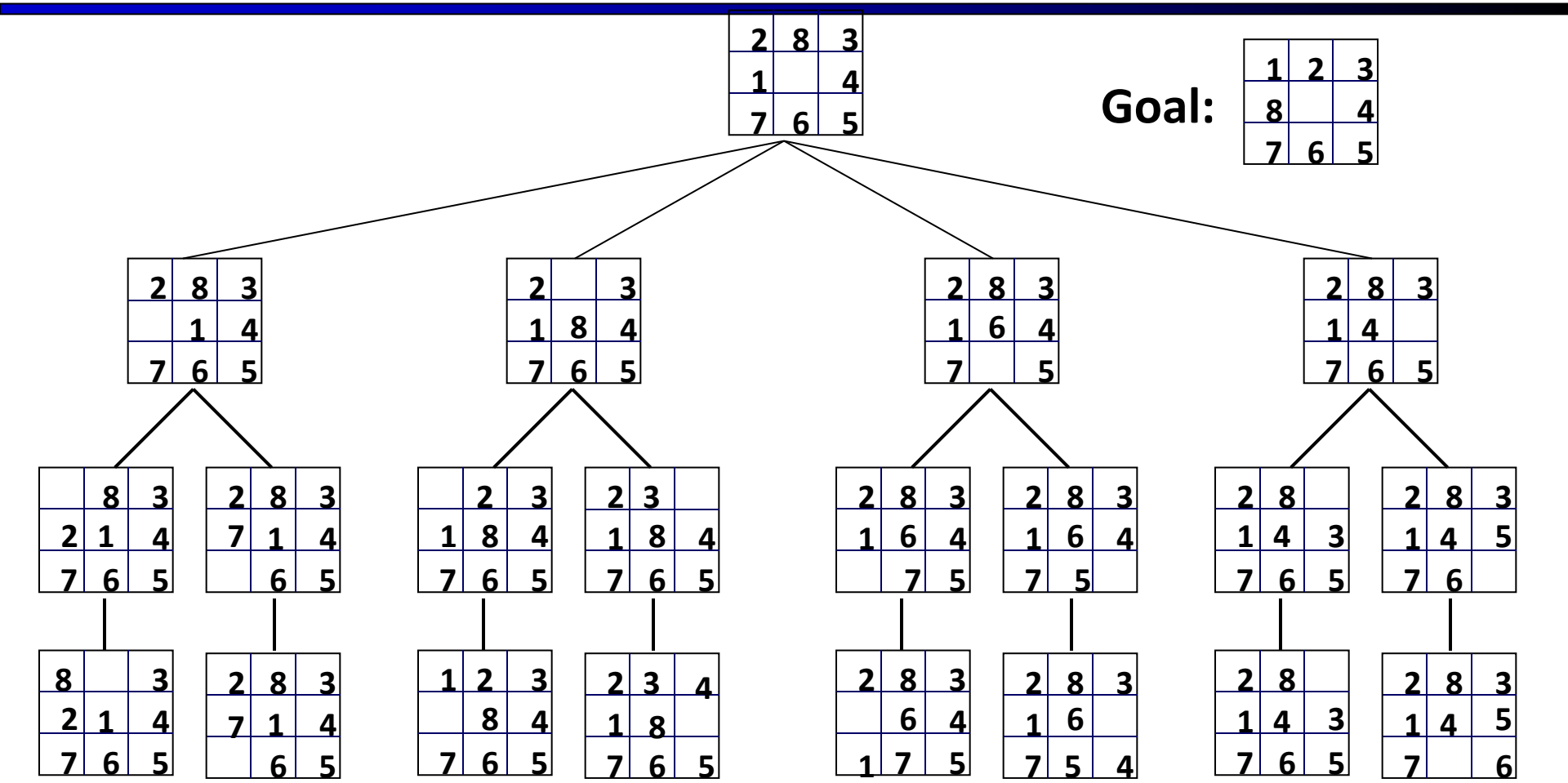
Depth-limited=1

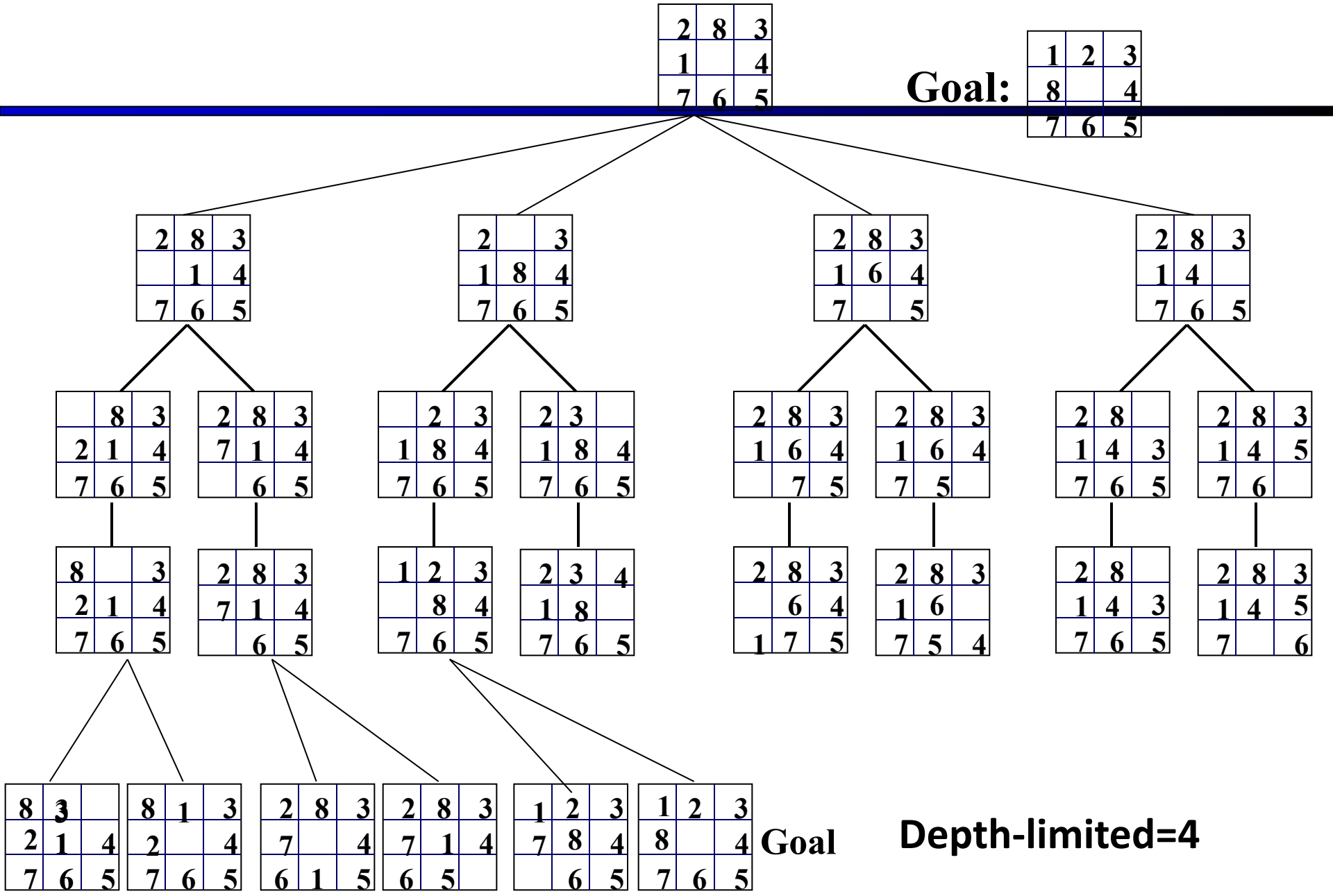


Depth-limited=2



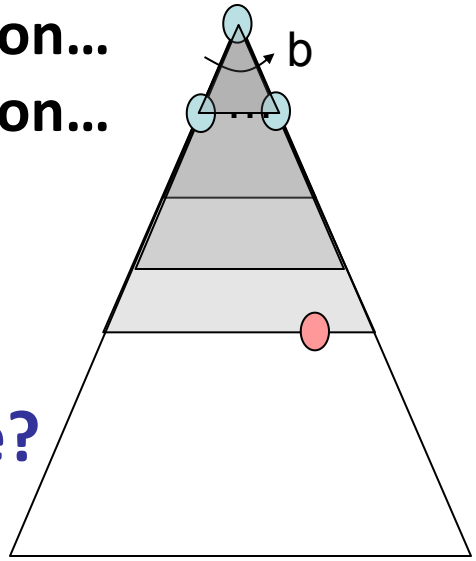
Depth-limited=3



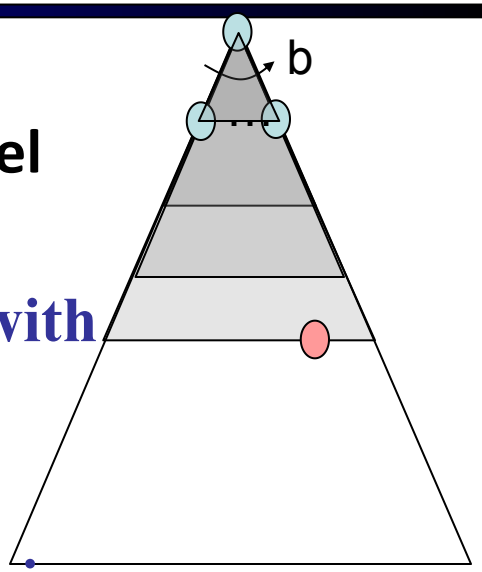

Goal:
Goal
Depth-limited=4

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 expand?
  - $O(b^d)$
- How much space does the fringe take?
  - O(bd)
- Is it complete?
  - yes!
- Is it optimal?
  - Yes! (if the cost is equal per step)



Iterative Deepening



- **Isn't that wastefully redundant?**
  - **Generally most work happens in the lowest level searched, so not so bad!**
- **Number of nodes generated in a *BFS* to depth *d* with branching factor *b*:**

$$N_{BFS} = b^1 + b^2 + \ldots + b^{d-1} + b^d$$

- **Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:**

$$N_{IDS} = (d+1)b^0 + d\,b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

  - **For $b = 10$, $d = 5$,**
    $N_{BFS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - **$N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$**
  - **Overhead = (123,456 - 111,111)/111,111 = 11% Compare to BFS**

IDS的性能

- **Number of nodes generated in a *BFS* to depth *d* with branching factor *b*:**

$$N_{BFS} = b^1 + b^2 + \ldots + b^{d-1} + b^d$$

- **Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:**

$$N_{IDS} = (d+1)b^0 + d\,b + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

  - **For $b = 10$, $d = 5$,**
    $N_{BFS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - **$N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$**
  - **Overhead = (123,456 - 111,111)/111,111 = 11% Compare to BFS**

Iterative Deepening

- But it seems that the time complexity of iterative deepening is much higher than that of breadth-first search!
- Well, if we have a branching factor b and the shallowest goal at depth d, then the worst-case number of nodes to be expanded by breadth-first is:

$$N_{bf} = 1 + b + b^2 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

Iterative Deepening

- In order to determine the number of nodes expanded by iterative deepening, we have to look at depth-first search.
- What is the worst-case number of nodes expanded by depth-first search for a branching factor b and a maximum search level j ?

$$N_{df} = 1 + b + b^2 + \ldots + b^j = \frac{b^{j+1} - 1}{b - 1}$$

Iterative Deepening

- **Therefore, the worst-case number of nodes expanded by iterative deepening from depth 0 to depth d is:**

$$N_{id} = \sum_{j=0}^{d} \frac{b^{j+1} - 1}{b - 1}$$

$$= \frac{1}{b-1}\left[b\left(\sum_{j=0}^{d} b^{j} \right) - \sum_{j=0}^{d} 1 \right]$$

$$= \frac{1}{b-1}\left[b\left(\frac{b^{d+1} - 1}{b - 1} \right) - (d+1) \right]$$

$$N_{id} = \frac{b^{d+2} - 2b - bd + d + 1}{(b-1)^2}$$

Iterative Deepening

- Let us now compare the numbers for breadth-first search and iterative deepening:

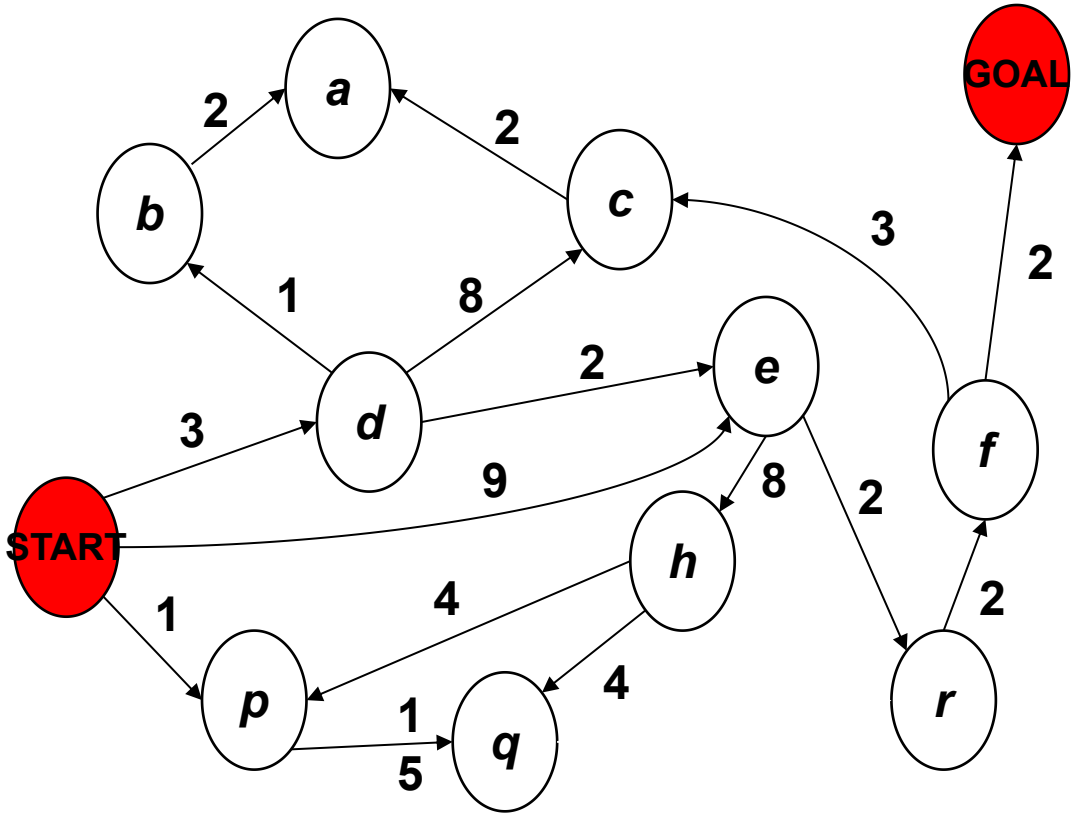
$$N_{bf} = 1 + b + b^2 + \ldots + b^d = \frac{b^{d+1} - 1}{b - 1}$$

$$N_{id} = \frac{b^{d+2} - 2b - bd + d + 1}{(b-1)^2}$$

For large d, you see that N_{id}/N_{bf} approaches b/(b – 1), which in turn approaches 1 for large b.

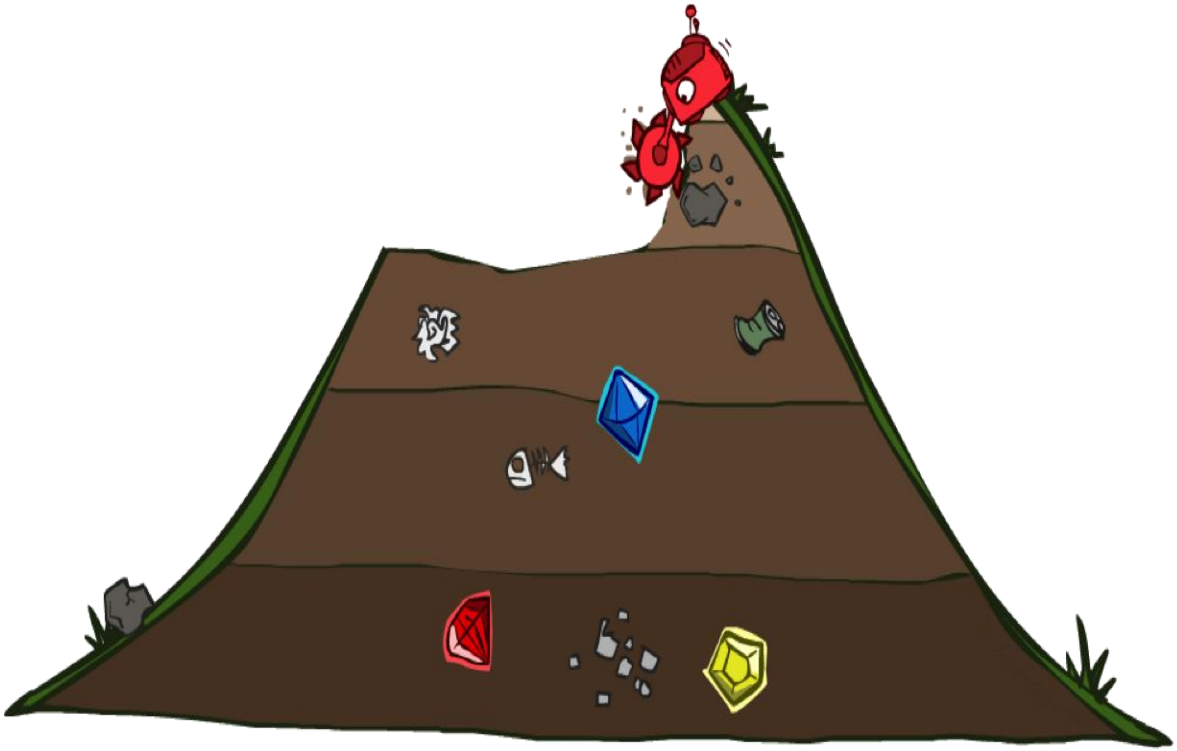
So for big trees (large b and d), iterative deepening does not expand many more nodes than does breadth-first search (about 11% for b = 10 and large d).

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



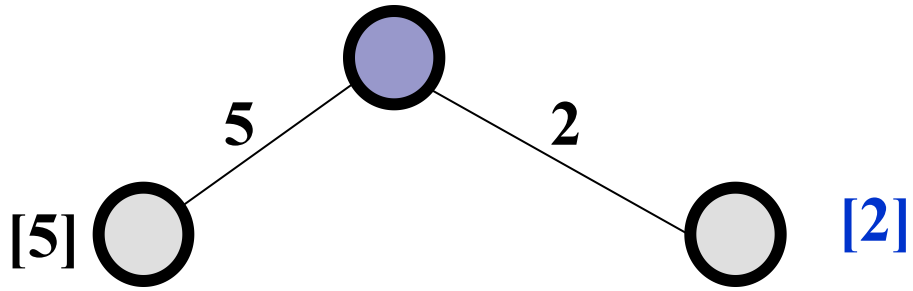
Uniform-cost search, UCS

- Expanded version of BFS
  - Expand node with lowest path cost
- Proposed in 1959 by Dijkstra, also known as Dijkstra's algorithm
- Let g(n) = cost of path from start node s to current node n
- Implementation: open =Priority Queue(queue ordered by g).
- UCS is the same as BFS when all step-costs are equal.

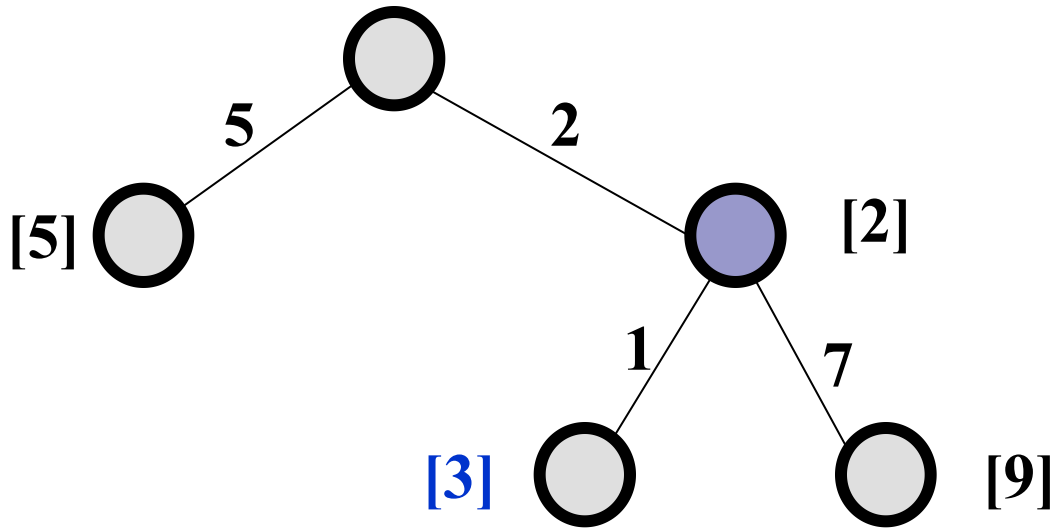
Uniform Cost Search (UCS)

- Expand the **cheapest** node.
- **Main idea**: Expand the cheapest node. Where the cost is the path cost g(n).
- **Implementation**: Enqueue nodes in order of cost g(n) in the queue **frontier**(insert in order of increasing path cost).
    - If a node n already exists in Frontier and a new path to n is found with a smaller cost, remove the node n from Frontier and insert the new node n with the new cost into Frontier

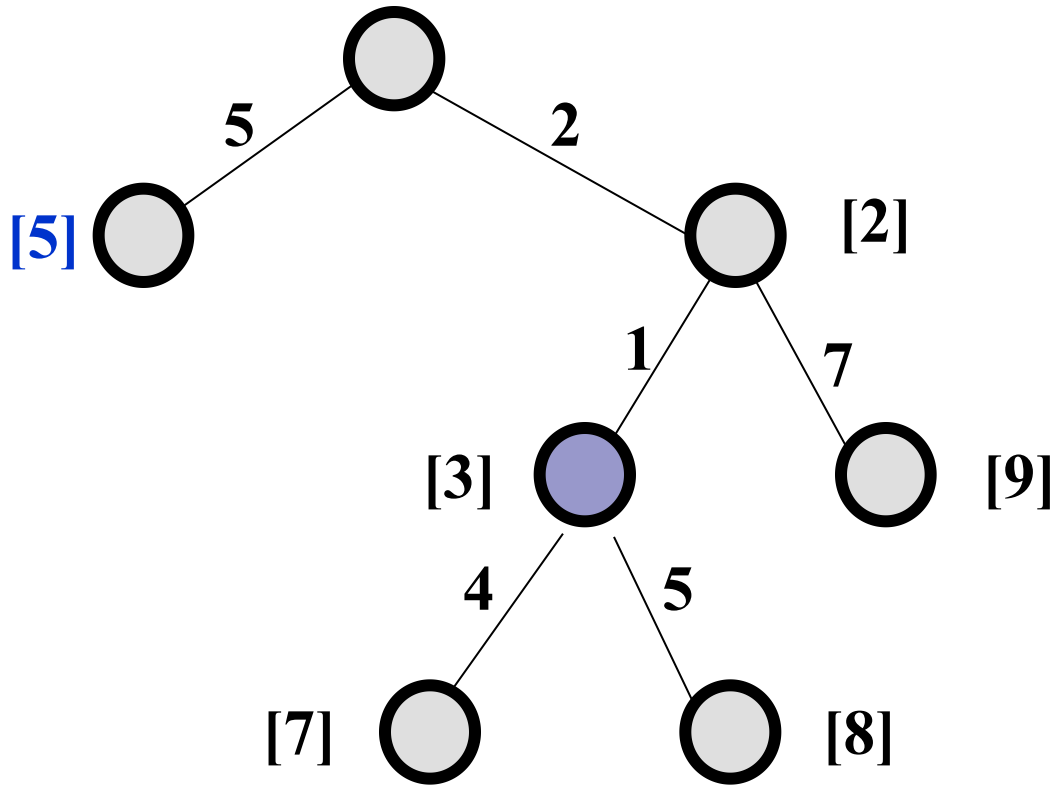
Uniform Cost Search (UCS)



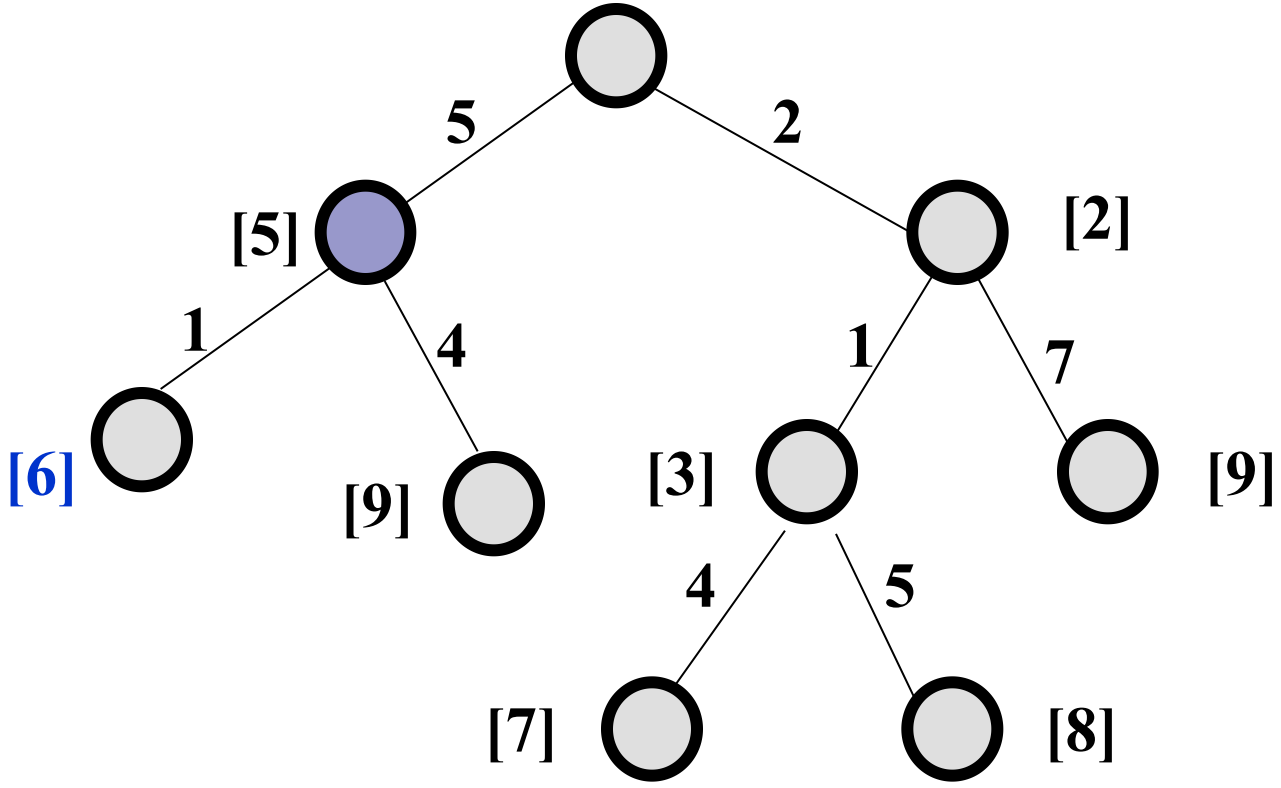
Uniform Cost Search (UCS)



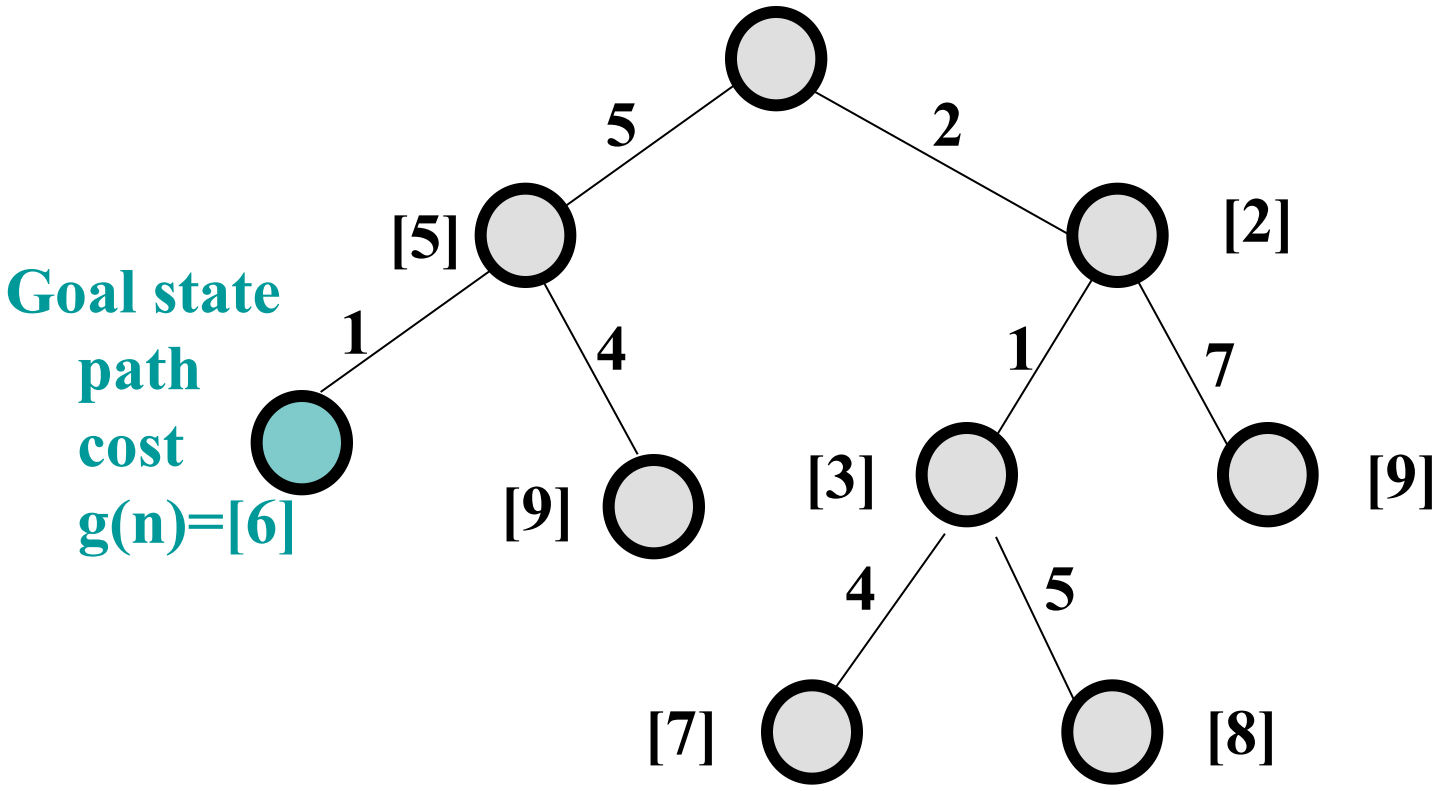
Uniform Cost Search (UCS)



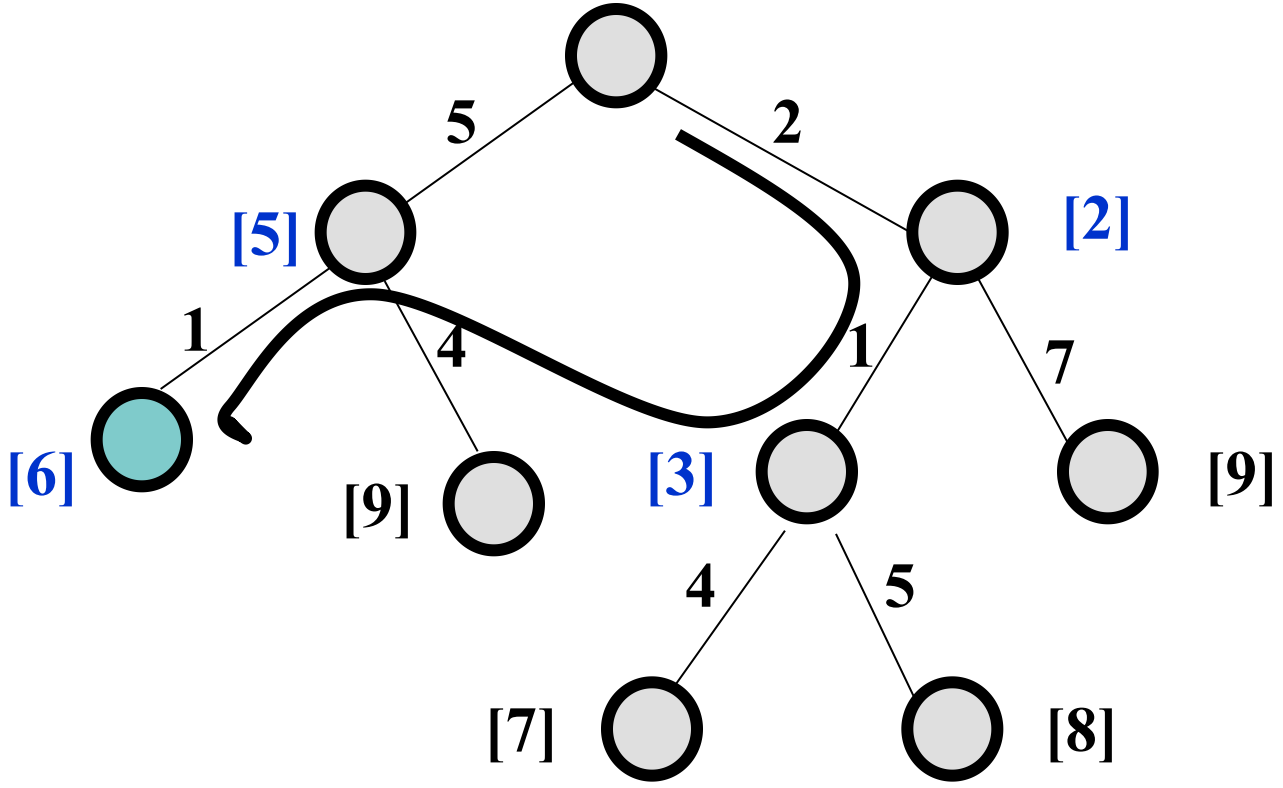
Uniform Cost Search (UCS)



Uniform Cost Search (UCS)



Uniform Cost Search (UCS)

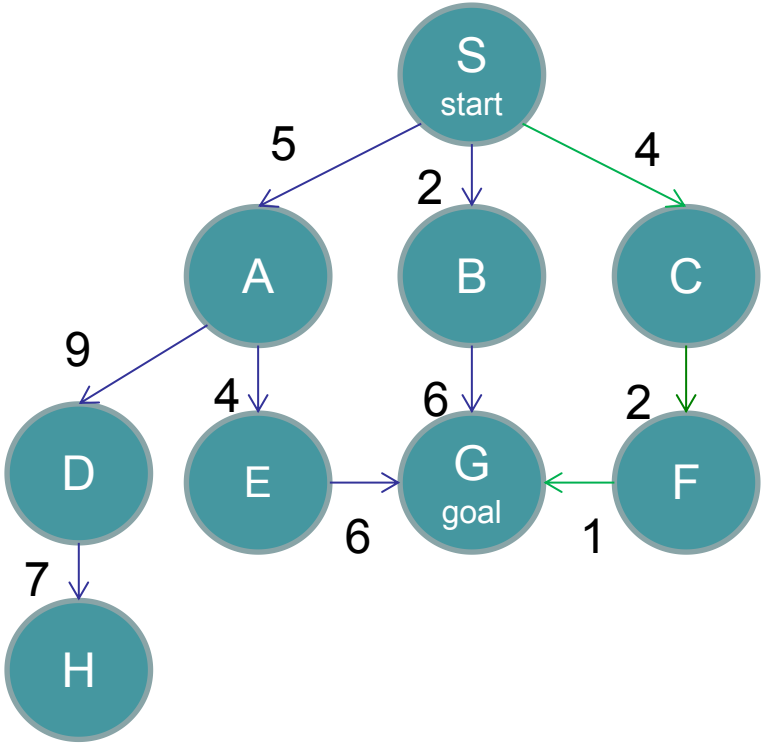


Example: UCS

generalSearch(problem, priorityQueue)

# of nodes tested: 0, expanded: 0

Expnd.node	Open list
	{S}

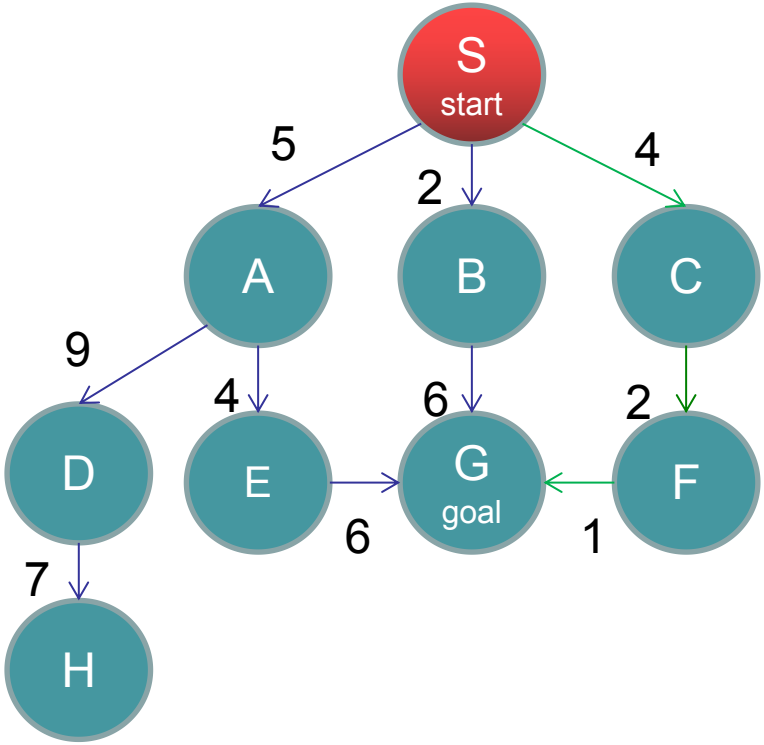


Example: UCS

generalSearch(problem, priorityQueue)

of nodes tested: 1, expanded: 1

Expnd.node	Open list
	{S:0}
S not goal	{B:2,C:4,A:5}

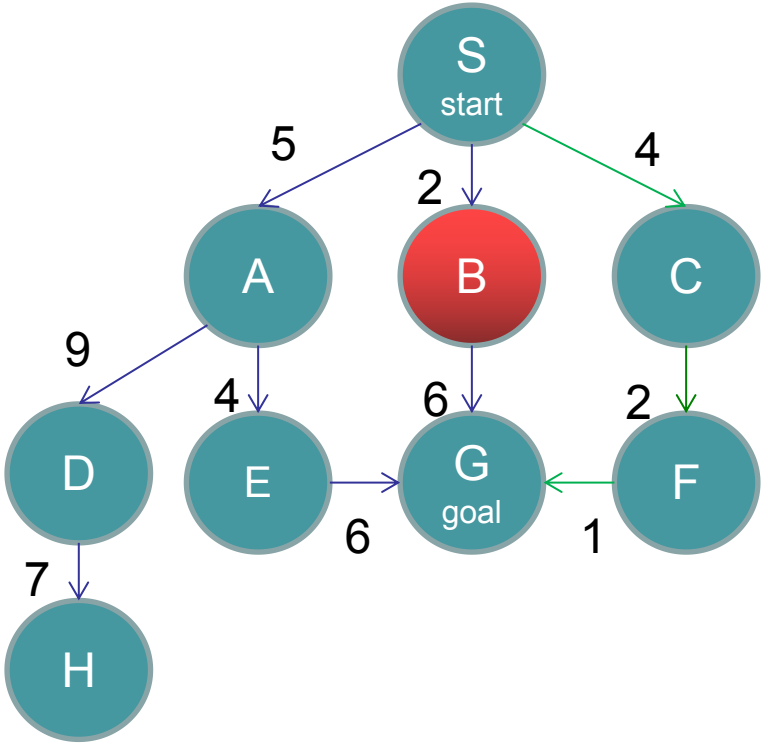


Example: UCS

generalSearch(problem, priorityQueue)

of nodes tested: 2, expanded: 2

Expnd.node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B not goal	{C:4,A:5,G:2+6}

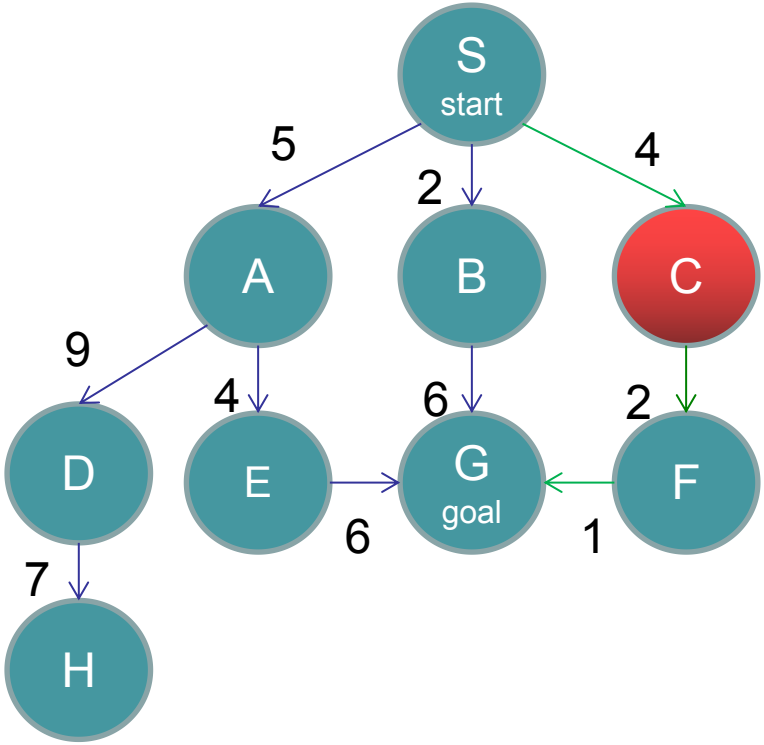


Example: UCS

generalSearch(problem, priorityQueue)

# of nodes tested: 3, expanded: 3

Expnd.node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C not goal	{A:5,F:4+2,G:8}

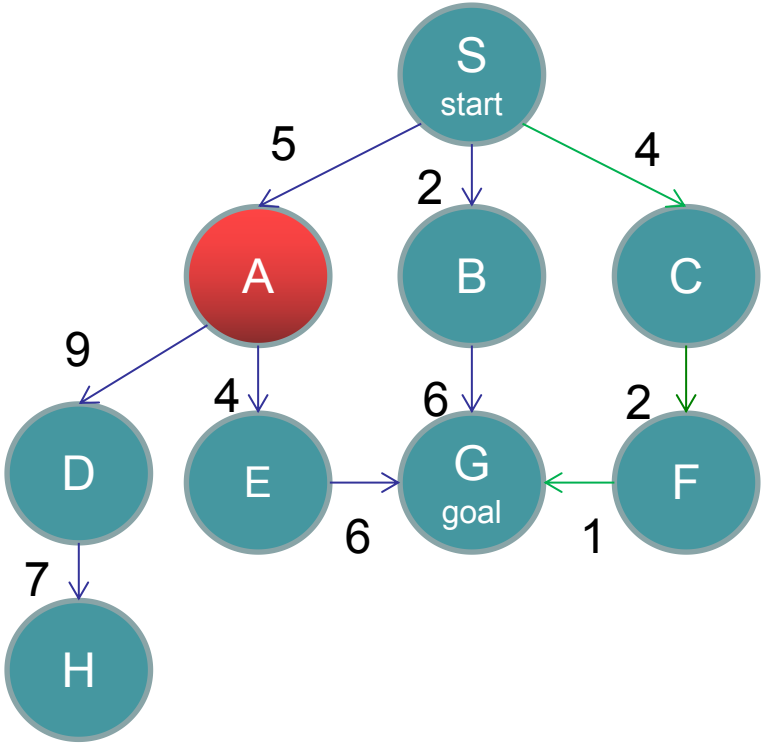


Example: UCS

generalSearch(problem, priorityQueue)

of nodes tested: 4, expanded: 4

Expnd.node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A not goal	{F:6,G:8,E:5+4,D:5+9}

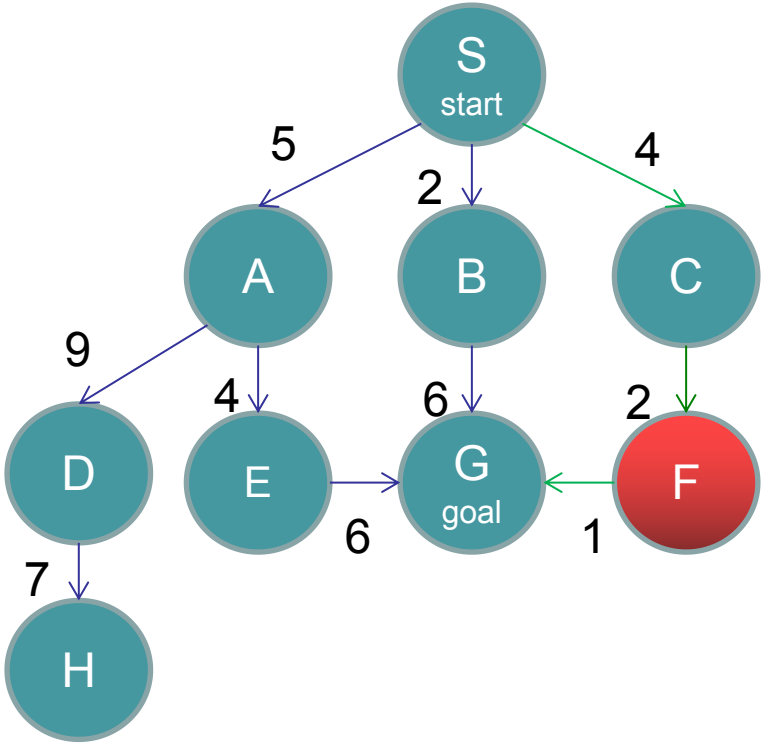


Example: UCS

generalSearch(problem, priorityQueue)

of nodes tested: 5, expanded: 5

Expnd.node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F not goal	{G:4+2+1,G:8,E:9, D:14}

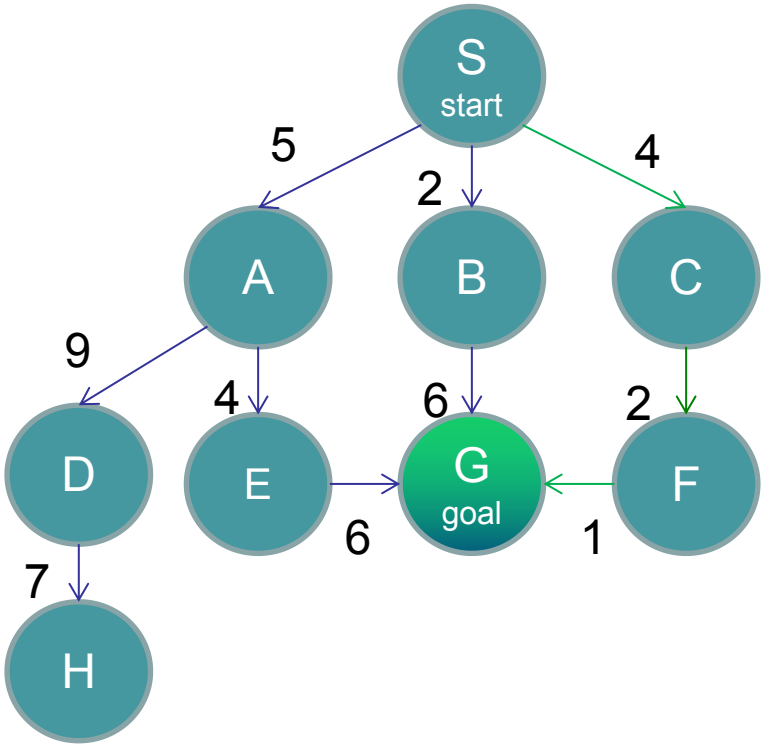


Example: UCS

generalSearch(problem, priorityQueue)

of nodes tested: 6, expanded: 5

Expnd.node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,E:9,D:14}
G goal	{E:9,D:14} no expand

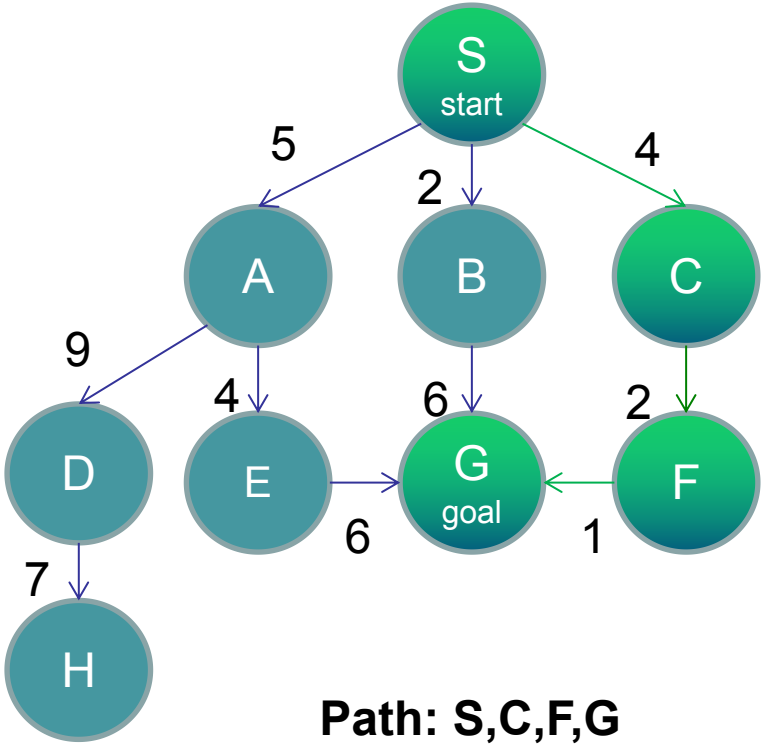


Example: UCS

generalSearch(problem, priorityQueue)

# of nodes tested: 6, expanded: 5

Expnd.node	Open list
	{S:0}
S	{B:2,C:4,A:5}
B	{C:4,A:5,G:8}
C	{A:5,F:6,G:8}
A	{F:6,G:8,E:9,D:14}
F	{G:7,G:8,E:9,D:14}
G	{E:9,D:14}

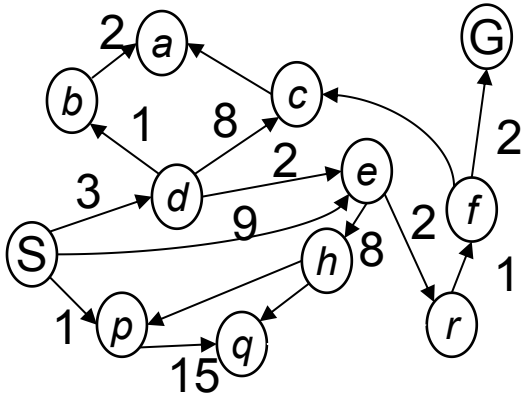


Path: S,C,F,G
Cost:7

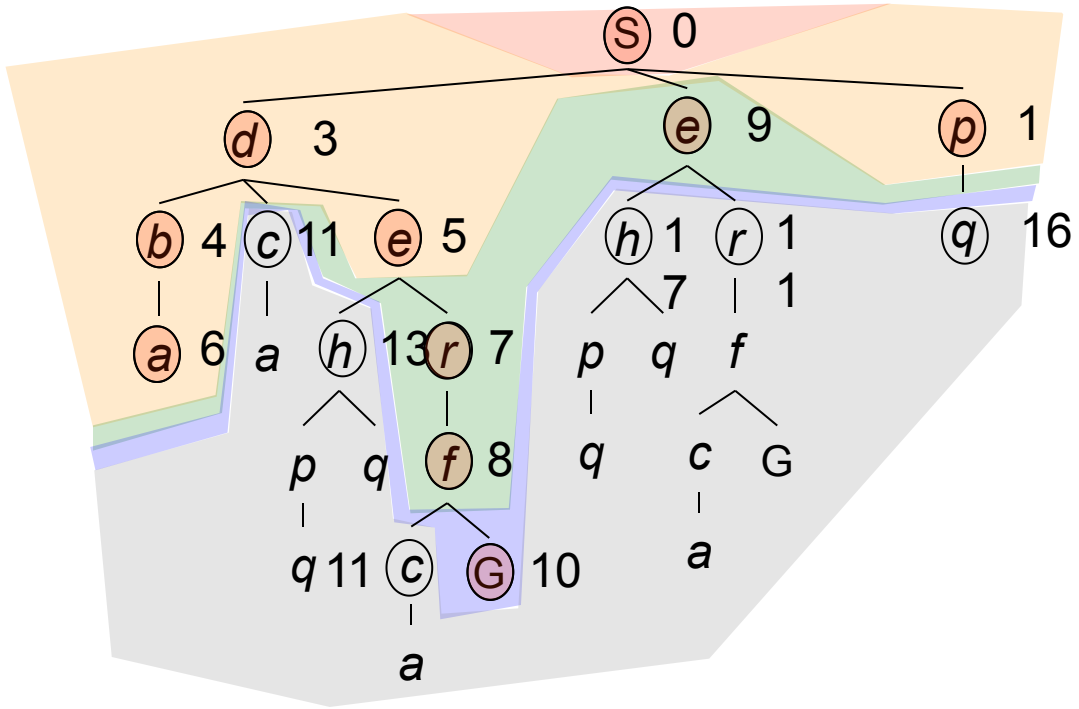
Uniform Cost Search

Strategy: expand a cheapest node first:

Fringe is a priority queue (priority: cumulative cost)

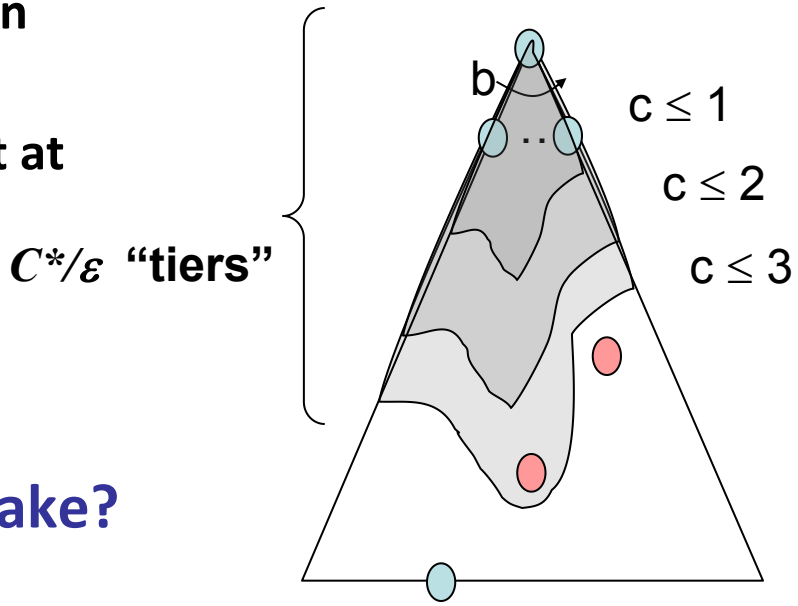


Cost contours
C* = 10



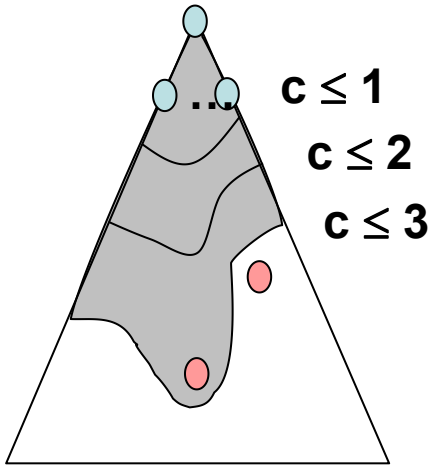
Uniform Cost Search (UCS) Properties

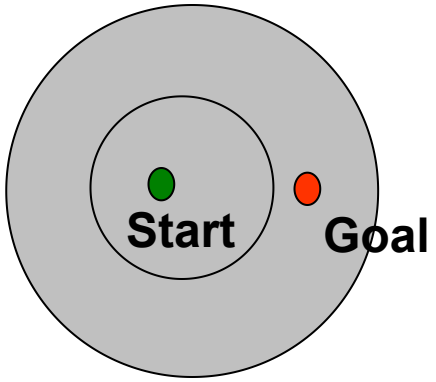
- What nodes does UCS expand?
  - Processes all nodes with cost less than cheapest solution!
  - If that solution costs C^* and arcs cost at least ε , then the "effective depth" is roughly C^*/ε
  - Takes time $O(b^{C^*/\varepsilon})$ (exponential in effective depth)
- How much space does the fringe 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! (Proof next lecture via A*)



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





Bidirectional Search

- Start searching forward from initial state and backwards from goal, and try to meet in the middle

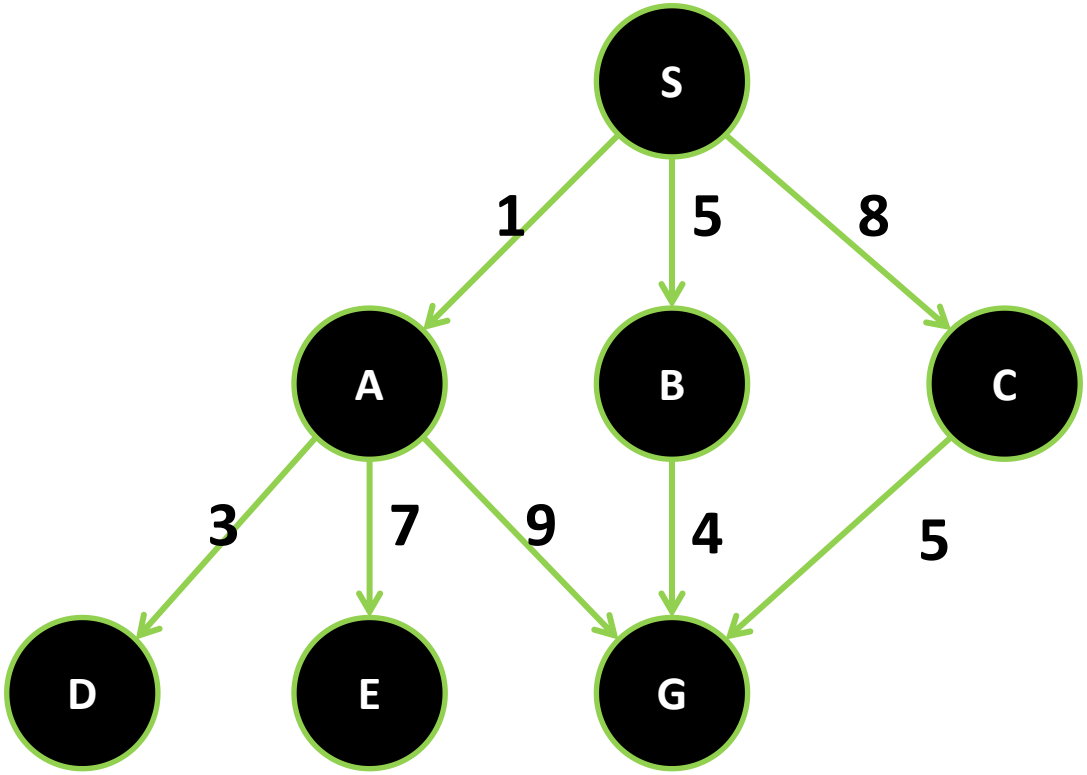
Summary of algorithms

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

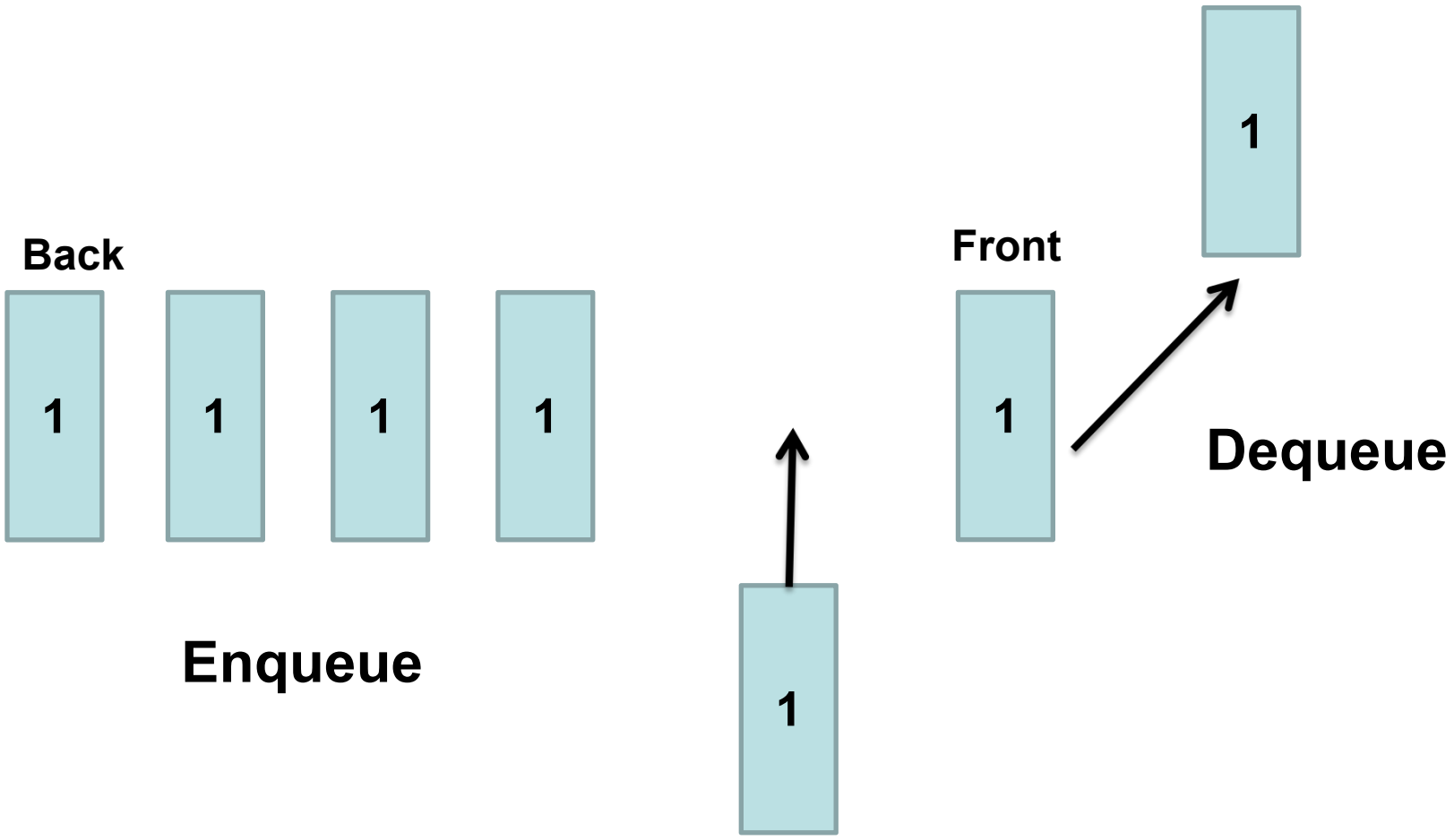
For BFS,Suppose the branching factor b is finite and step costs are identical;

Quiz

- Given the following state space, list the extension order and the final solution path with different search algorithms as below
  - DFS
  - BFS
  - UCS
  - IDS



Priority Queue

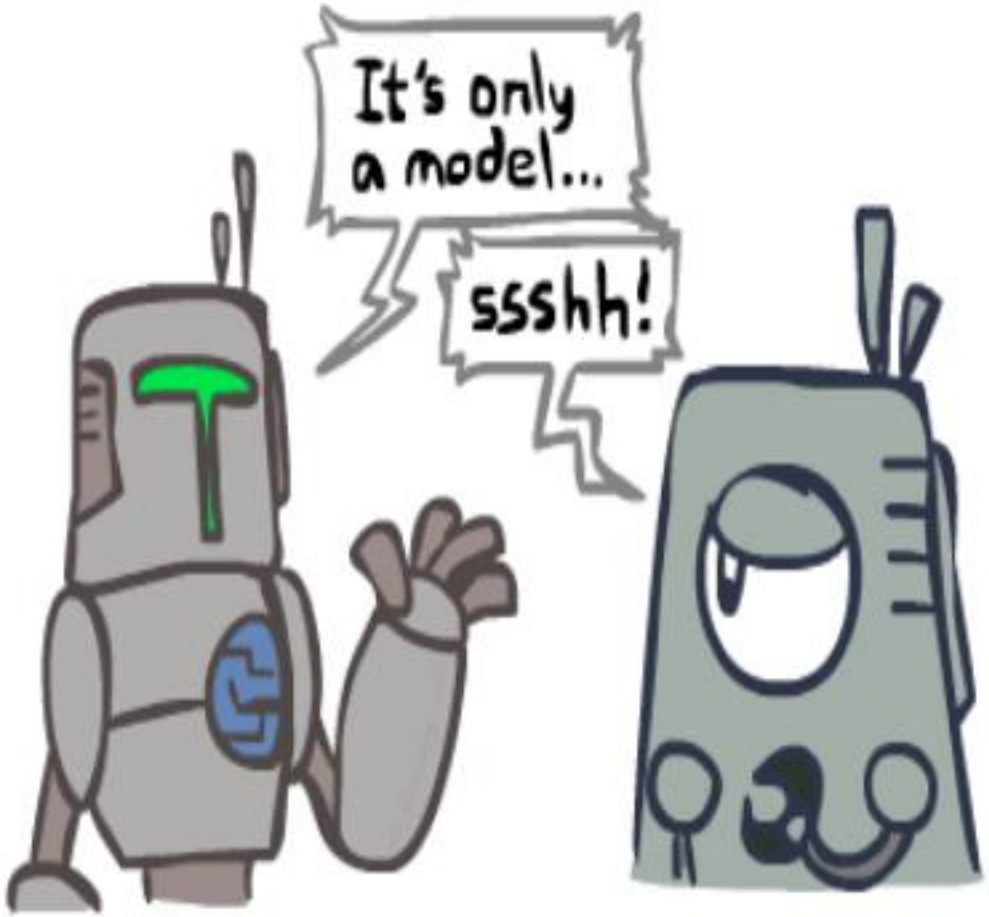


The One Queue

- **All these search algorithms are the same except for fringe strategies**
  - **Conceptually, all fringes 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**

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…



Some Hints for P1

- Graph search is almost always better than tree search (when not?)
- Implement your closed list as a dict or set!
- Nodes are conceptually paths, but better to represent with a state, cost, last action, and reference to the parent node

Summary

- Search Problems
- Uninformed Search Methods
  - Depth-First Search
  - Breadth-First Search
  - Uniform-Cost Search
- Next time: informed search, A*