Lecture 2a: Uninformed Search

CSCI 360

Introduction to Artificial Intelligence

USC

Exponential growth is scary

- *"Folding the piece of paper 3 times will get you about the thickness of a fingernail."*

- *10 folds → about the **width of a hand**.*
- *18 folds → the **height of an average person**.*
- *23 folds → taller than the **Empire State Building** in NYC.*
- *30 folds → reaching the **outer space** (100km high).*
- *42 folds → reaching the **Moon**.*

  …

- *And finally, at 103 folds, it will get outside of the **observable Universe**.*

Outline

[Chapters 3.3-3.4]

- **Recap: Problem-Solving Agents**
- **Implementation of Search Algorithms**
- **Measuring Performance**
- **Uninformed Search Strategies**

Outline

[Chapters 3.3-3.4]

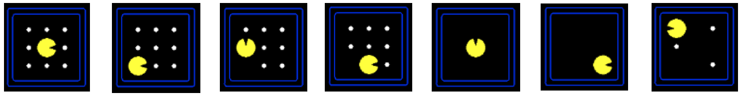
- **Recap: Problem-Solving Agents**
- **Implementation of Search Algorithms**
- **Measuring Performance**
- **Uninformed Search Strategies**
  - **Breadth-first search**
  - **Uniform-cost search**
  - **Depth-first search**
  - **Depth-limited search**
  - **Iterative deepening depth-first search**
  - **Bidirectional search**

Recap: Many AI applications can be reduced to “search”

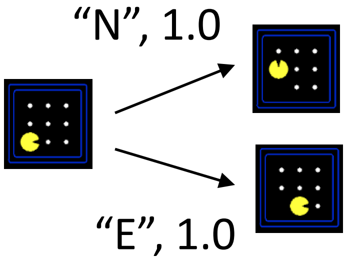
- States:
- Initial state:
- Actions:
- Transition model:
- Goal test:

Recap: An example search problem

- A **search problem** consists of:
  - A state space



  - A successor function (with actions, costs)



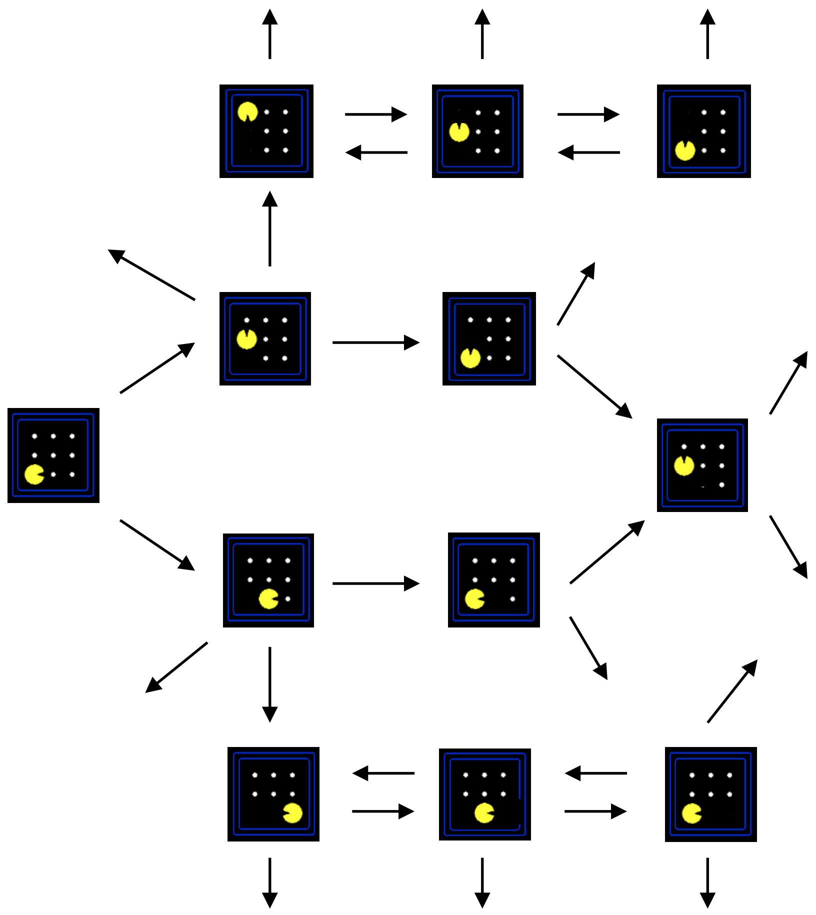
  - A start state and a goal test

- A **solution** is a sequence of actions that transforms the start state to a goal state

Recap: The idea of a “state space graph”

- A mathematical representation of a search problem
  - Nodes are world configurations
  - Edges lead to successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

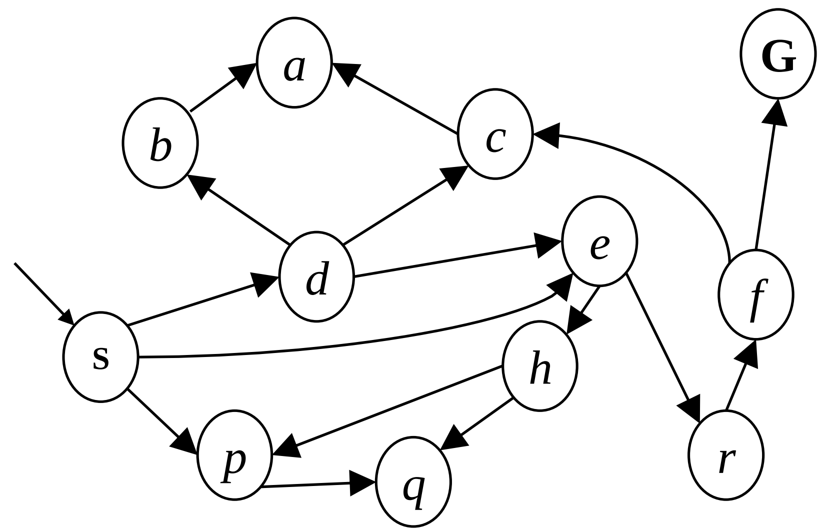
- In practice, we can rarely build this full graph (since it’s too big), but conceptually, it’s a useful idea



Recap: The idea of a "state space graph"

- A mathematical representation of a search problem
  - Nodes are world configurations
  - Edges lead to successors (action results)
  - The goal test is a set of goal nodes (maybe only one)

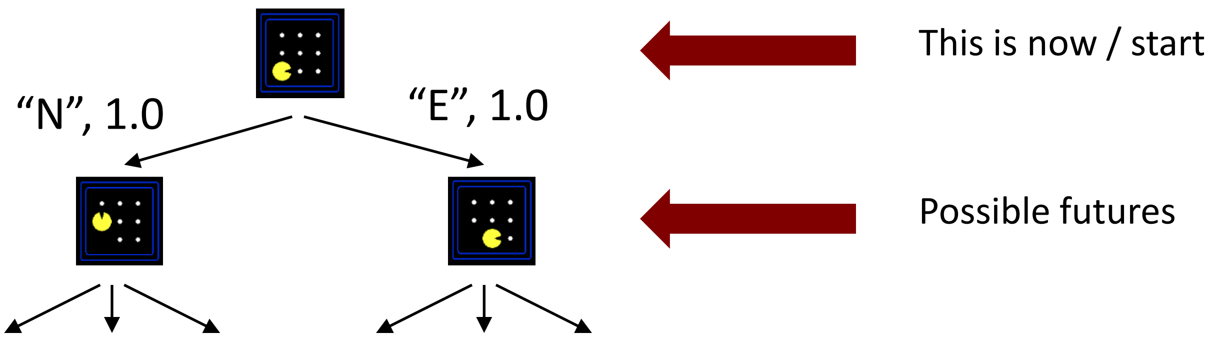
- In practice, we can rarely build this full graph (since it's too big), but conceptually, it's a useful idea



Tiny search graph for a tiny search problem

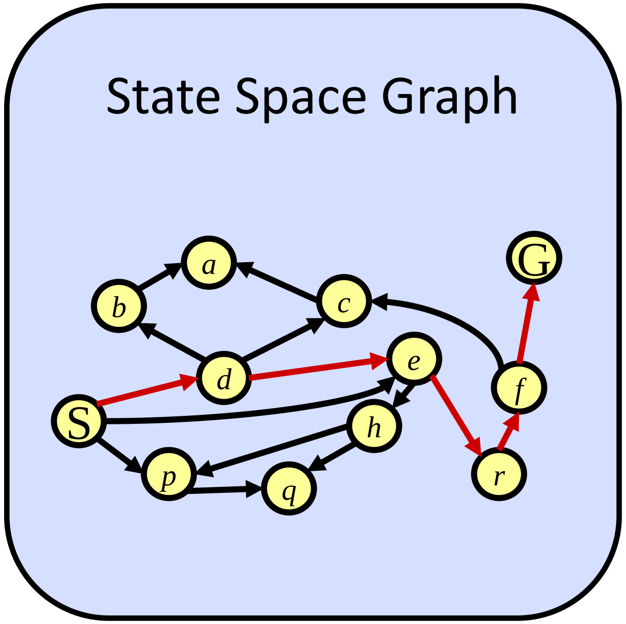
Recap: The notion of a “search tree”

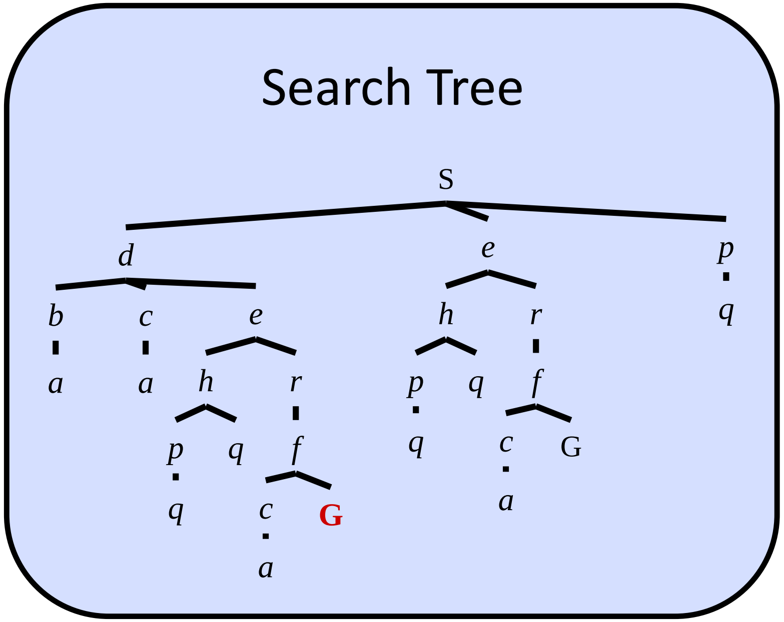
– A “what if” tree of plans and their outcomes
– The start state is the root node
– Children correspond to successors
– For most problems, however, we can never actually build the whole tree



Recap: State Space Graphs vs. Search Trees

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





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

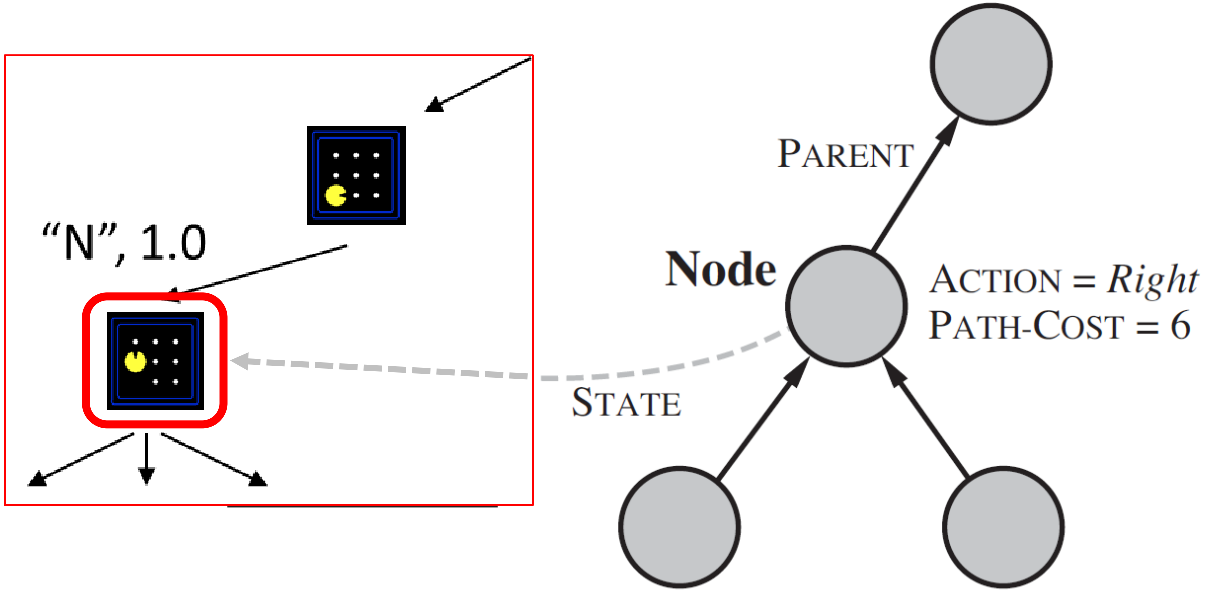
Outline

[Chapters 3.3-3.4]

- **Recap: Problem-Solving Agents**
- **Implementation of Search Algorithms**
- **Measuring Performance**
- **Uninformed Search Strategies**

Nodes versus States

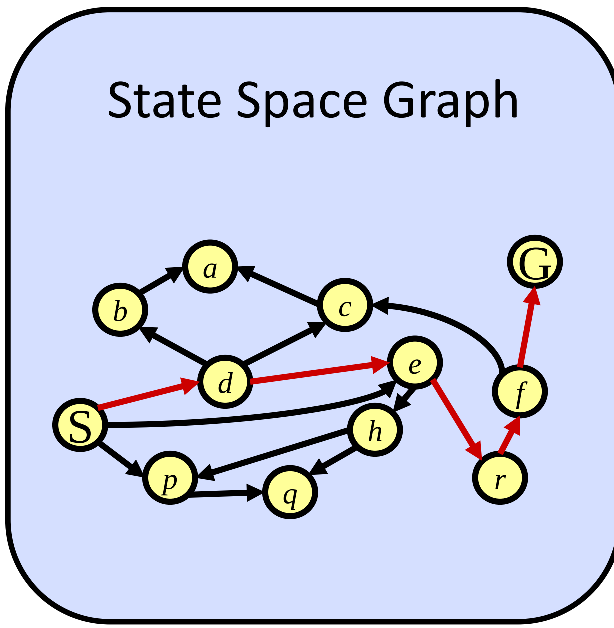
- **State** is a configuration of an environment/agent
- **Node** is a data structure constituting part of a search tree
  - may include fields such as (state, parent, action, path cost, …)
  - Two nodes in a tree may correspond to the same (game) state



Implementation details (Graph data structure)

- Recall that in classic algorithms, the given is often given as a set of “nodes” and “edges”
- But, AI algorithms don’t have such a graph as input

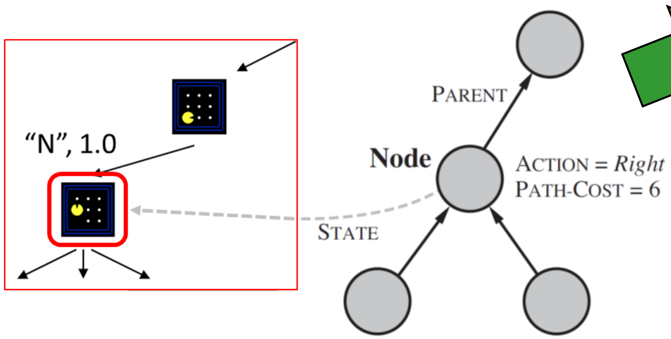
```
class Graph {
  vector<Node> nodes;
  vector<Edge> edges;
  …
}
```



Implementation details (related to states)

```
class problem {
    State  INITIAL_STATE;
    State  RESULT( State st, Action act ) { … }
    bool   GOAL_TEST( State st ) { … }
    …
}
```

Input given to an AI search algorithm



Node for constructing the search tree

```
class Node {
    State  STATE;      // the corresponding game state
    Node   PARENT;     // node that generated this node
    Action ACTION;     // action that generated this node
    int    PATH_COST;
    …
}
```

Implementation details (child-node)

- Given current node, how to get **child** and **parent** nodes?
  - Basic operations used by a search algorithm

function CHILD-NODE($problem, parent, action$) **returns** a node
  return a node with
    STATE = $problem$.RESULT($parent$.STATE, $action$),
    PARENT = $parent$, ACTION = $action$,
    PATH-COST = $parent$.PATH-COST + $problem$.STEP-COST($parent$.STATE, $action$)

```
Node problem = …
Node parent = …
Action action = …
 …
Node node = CHILD_NODE(problem, parent, action);
```

```
Node node = …
 …
Node parent = node.PARENT;
```

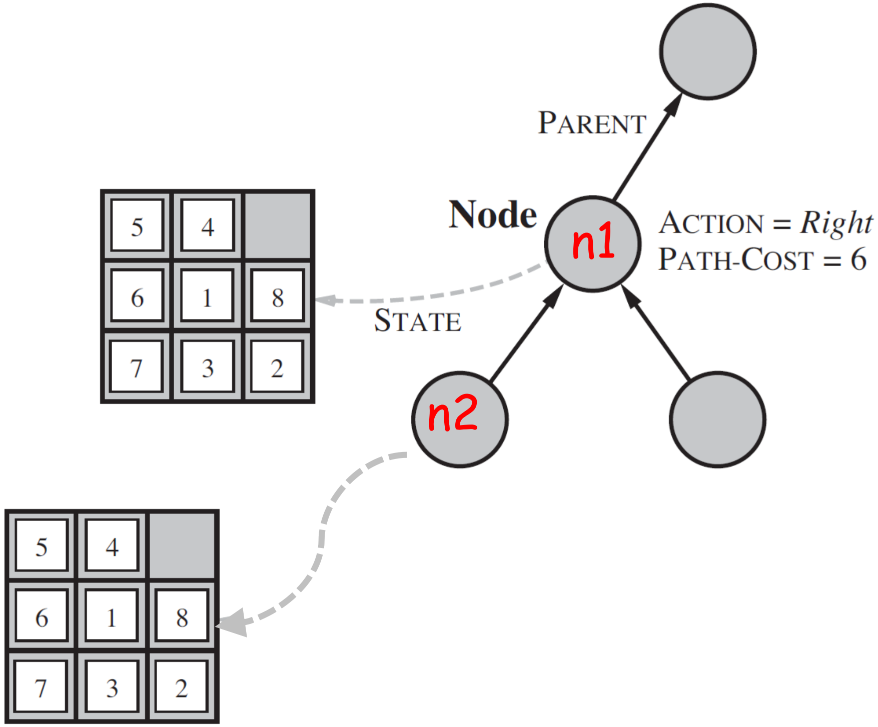
Recall the “graph search” algorithm

- The ***explored set*** helps avoiding the exploration of the same states again and again

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

Implementation details (the explored set)

- The explored set can be implemented with a **hash table**
- The notion of equality between states (instead of nodes)
  - Why?



The states of two nodes (n1 and n2) may be equal

(n1.STATE == n2.STATE)

Use them to compute hash key

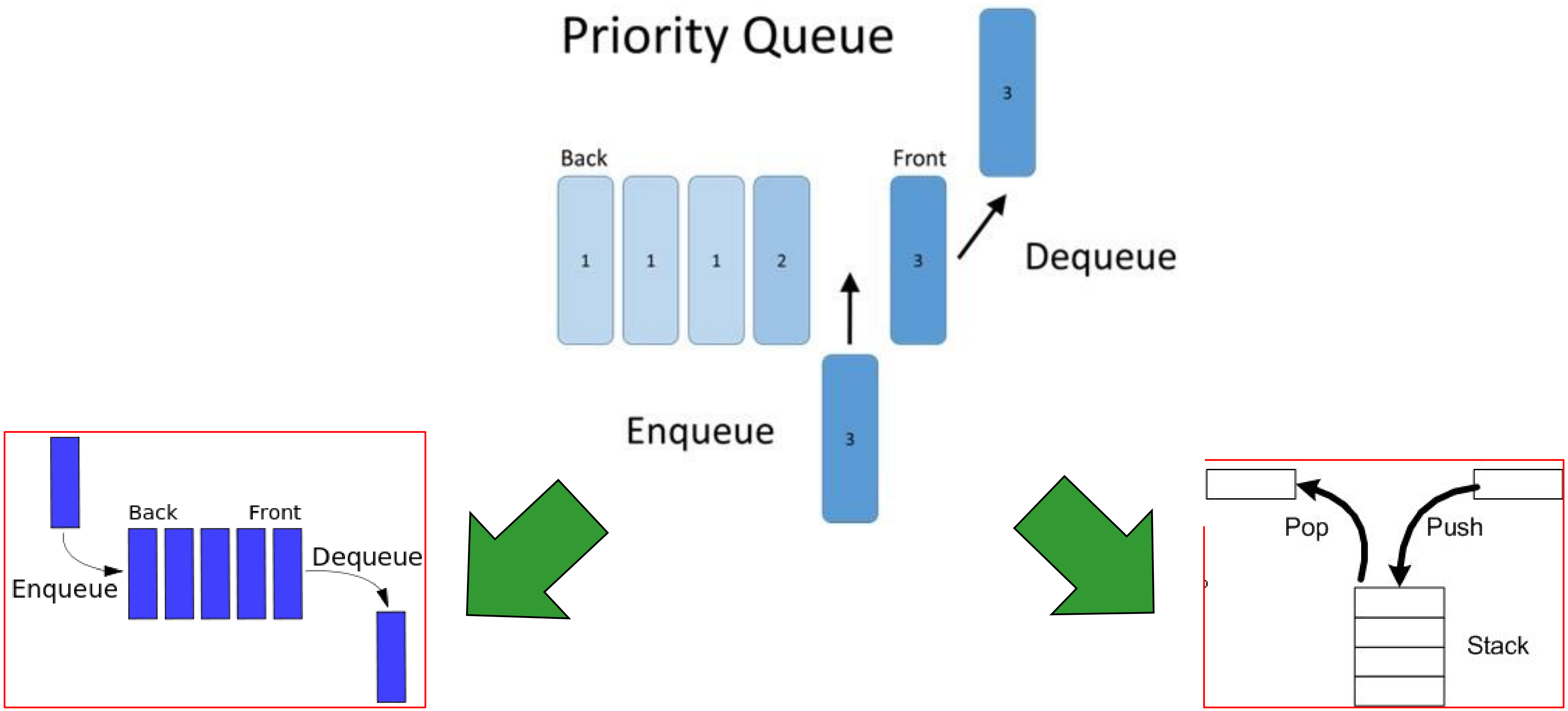
Recall the "graph search" algorithm

- The ***frontier*** stores the nodes to be expanded next

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

Implementation details (the frontier)

- How to implement the frontier set? It’s a (priority) queue!
  - FIFO (first-in, first out), LIFO (last-in, first out), Priority-Based



Outline

[Chapters 3.3-3.4]

- **Recap: Problem-Solving Agents**
- **Implementation of Search Algorithms**
- **Measuring Performance**
- **Uninformed Search Strategies**

Completeness, Optimality, and Complexity

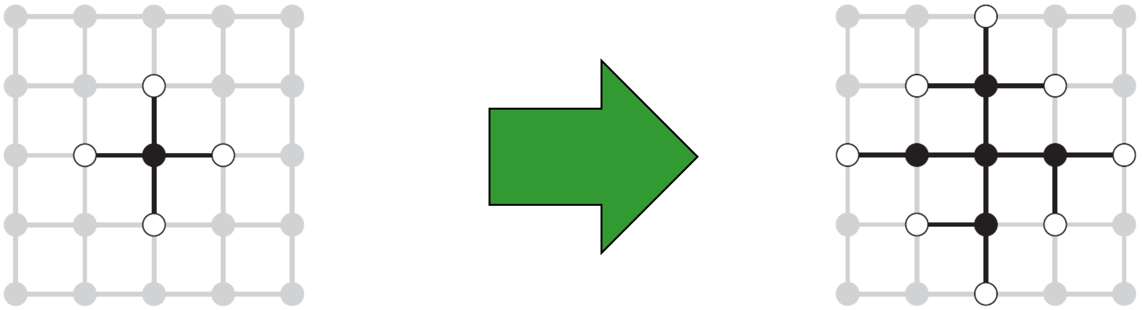
- Completeness:
  - Guaranteed to find a solution when there is one
- Optimality:
  - Guaranteed to find the optimal solution
- Time complexity:
  - How long does it take to complete the search?
- Space complexity:
  - How much memory does it need to perform the search

Two Ways of Quantifying Complexity

- Theoretical Computer Science
  - |V| -- number of nodes in a graph
  - |E| -- number of edges in a graph
  - **O (|V|+|E|)**

- Artificial Intelligence (AI) applications
  - b -- **branching factor** (maximum number of successors of any node)
  - d -- **depth** of shallowest goal state along any path in state space

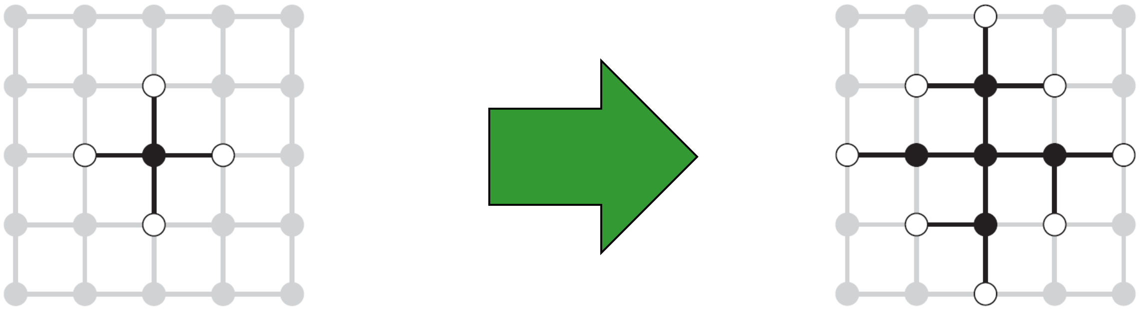
State Space Explosion (where b=4)



Tree Search **at depth d** $\boxed{4^d}$

1 trillion (d=20)

State Space Explosion (where b=4)



Tree Search **at depth d** $\boxed{4^d}$

1 trillion (d=20)

Graph Search **at depth d** $\boxed{4d}$

80 states (d=20)

What is Polynomial?

- ***4d*** -- this is called “linear” in terms of “d”
- ***4^d*** – this is called “exponential” in terms of “d”

- $O(1)$ – constant
- $O(d)$ – linear
- $O(d^2)$ – quadratic (or degree-2 polynomial)
- $O(d^3)$ – cubic (or degree-3 polynomial)
- …

- In the study of algorithms (e.g., CSCI 104), a **polynomial-time** solvable problem is considered to be a “**tractable**” problem, whereas a exponential-time solvable problem is considered to be “intractable”

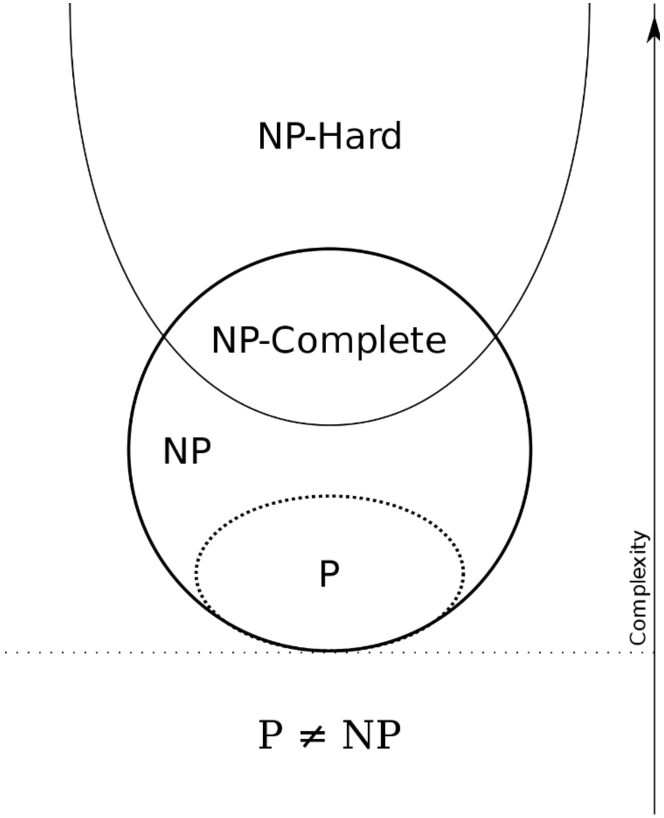
What is NP?

- NP means “Non-deterministic Polynomial”
  - If you can **“guess”** the solution, and then, in **polynomial time** **“check”** if the solution is indeed a valid solution

- The time taken to **come up with a solution** is often longer than the time taken to **check the validity of the solution**
  - If you have a “***non-deterministic Turing machine***”, you can solve the problem in polynomial time
  - If you only have a “***deterministic Turing machine***”, you may need exponential time

What is NP-complete?

- The hardest problems in NP
  - If you can solve one of them in polynomial time, you can solve all of them in polynomial time



What is NP-hard?

- At least as hard as the hardest problem in NP (and could be harder or even undecidable)
  - Every problem in NP can be reduced to this problem (and the reduction process itself takes polynomial time)

- Unfortunately, many AI search problems are NP-hard
  - So far, there are only "exponential-time" algorithms for solving them

Total Cost = search cost + solution cost

- Example: *finding a route from Arad to Bucharest*
  - Search cost: the time taken to find a solution (in milliseconds)
  - Solution cost: the path cost of the route found (in kilometers)

- **Question**: how do you add up the two kinds of cost?
  - Answer: you just add them
  - No “official exchange rate” between the two
    - Car’s average speed (kilometers per millisecond)

Outline

[Chapters 3.3-3.4]

- **Recap: Problem-Solving Agents**
- **Implementation of Search Algorithms**
- **Measuring Performance**
- **Uninformed Search Strategies**

Uninformed vs. Informed Search

- **Informed Search**: the agent knows whether one state is “more promising” than another in reaching the goal

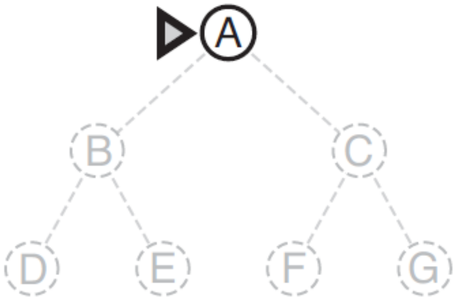
- **Uninformed search** (or blind search): the agent does not know such information
  - All the agent can do is to
  - (1) generate successor states and
  - (2) check if a state is a goal state

Breadth-first search (BFS)

- The “shallowest” node in frontier is chosen for expansion
  - The frontier is implemented using a FIFO queue

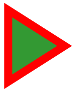
Breadth-first search (algorithm)

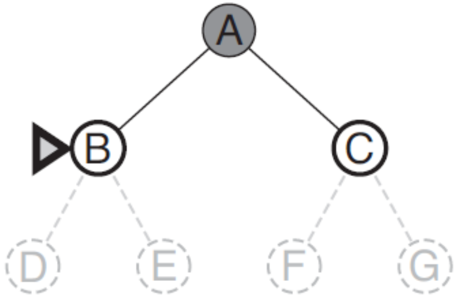
At each step, the node to be expanded next is indicated by the marker



Breadth-first search (algorithm)

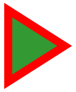
At each step, the node to be expanded next is indicated by the marker

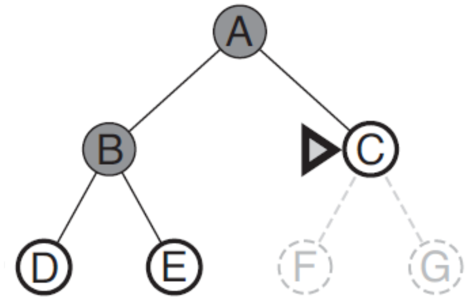




Breadth-first search (algorithm)

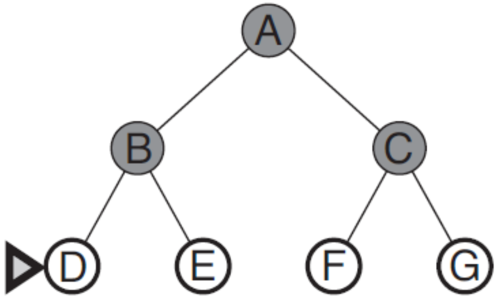
At each step, the node to be expanded next is indicated by the marker





Breadth-first search (algorithm)

At each step, the node to be expanded next is indicated by the marker



Breadth-first search (complete and optimal)

- BFS is complete
  - If the shallowest goal node is at some finite depth, ***d***, then BFS will eventually find the goal node
    - After generating all the shallower nodes
- BFS is optimal
  - If the path cost is a non-decreasing function

Breadth-first search (pseudo code)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)  /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Breadth-first search (time complexity)

- Parameters of the input problem
  - **Branching factor** (b): every node has b successors
  - **Depth** (d): the shallowest goal set is at some finite depth d

- Total nodes generated is
  - $b + b^2 + b^3 + \ldots + b^d = \boldsymbol{O(b^d)}$

Breadth-first search (time & memory blowups)

- Total nodes generated, and the time taken, are both
  - $b + b^2 + b^3 + \ldots + b^d = O(b^d)$

This is still possible

This is unrealistic

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.

Uniform-cost Search

- "**shallowest**" node ➔ node with the "**lowest path cost**"

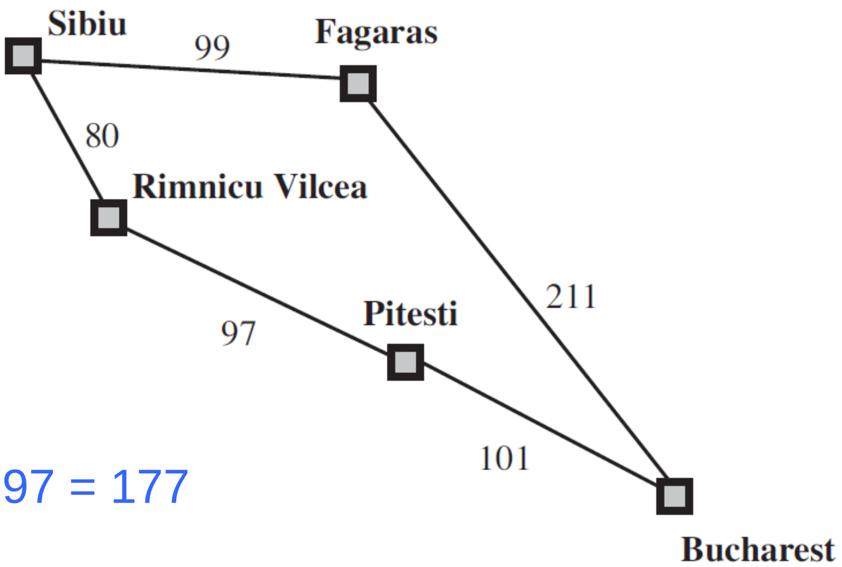
```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
      if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              frontier ← INSERT(child, frontier)
          else if child.STATE is in frontier with higher PATH-COST then
              replace that frontier node with child
```

Uniform-cost Search (optimality)

- Two subtle differences from BFS
  - **Goal test** is applied to a node when it's selected for expansion (not when it is first generated as in BFS) -- **Why?**
  - **Replace a node in the frontier** if a better path (to the same state associated with a node) is found – **Why?**

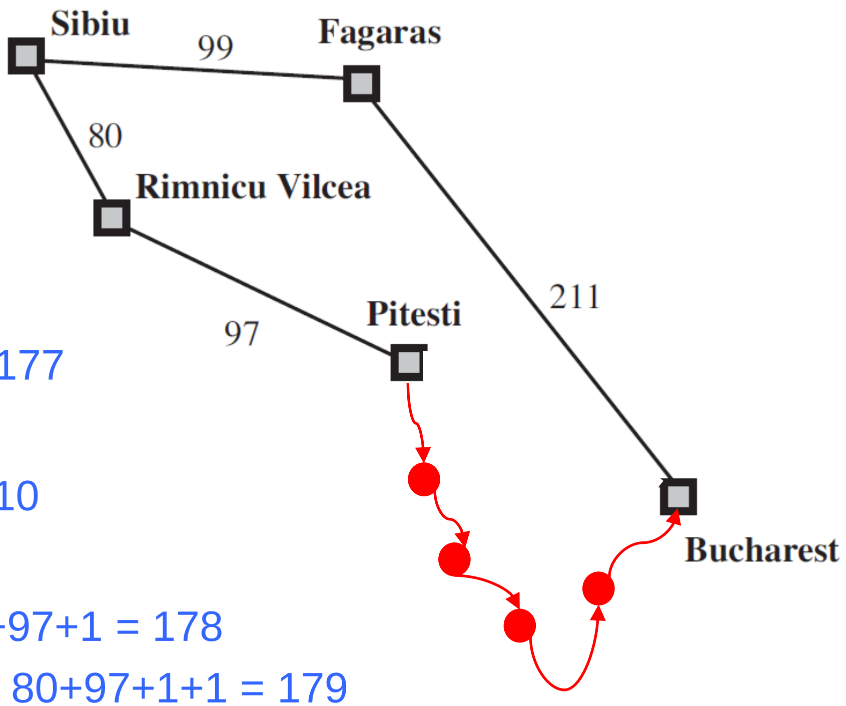
- **Theorem:** Uniform-cost search always expands nodes in order of their optimal path cost
  - The first goal node selected for expansion must be the optimal solution

Uniform-cost Search (example run)



- After Step 1:
  - Sibiu to Fagaras: 99
  - Sibiu to Rimnicu Vilcea: 80
- After Step 2:
  - Sibiu to Rimnicu Vilcea **to Pitesti**: 80+97 = 177
- After Step 3:
  - Sibiu to Fagaras **to Bucharest**: 99+211 = 310
- After Step 4:
  - Sibiu to Rimnicu Vilcea to Pitesti **to Bucharest**: 80+97+101 = 278
- After Step 5:
  - Sibiu to Rimnicu Vilcea to Pitesti to Bucharest **to somewhere else …**

Uniform-cost Search (example run 2)



- After Step 1:
  - Sibiu to Fagaras: 99
  - Sibiu to Rimnicu Vilcea: 80
- After Step 2:
  - Sibiu to Rimnicu Vilcea to Pitesti: 80+97 = 177
- After Step 3:
  - Sibiu to Fagaras to Bucharest: 99+211 = 310
- After Steps 4,5,6,7,8:
  - Sibiu to Rimnicu Vilcea to Pitesti **to A:**: 80+97+1 = 178
  - Sibiu to Rimnicu Vilcea to Pitesti to A **to B:** 80+97+1+1 = 179
  - Sibiu to Rimnicu Vilcea to Pitesti to A to B **to C:**: 80+97+1+1+1 = 180
  - Sibiu to Rimnicu Vilcea to Pitesti to A to B to C **to D:**: 80+97+1+1+1+1 = 181
  - Sibiu to Rimnicu Vilcea to Pitesti to A to B to C to D **to Bucharest: … =** 278
- After Step 9:
  - Sibiu to Rimnicu Vilcea to Pitesti to ABCD to Bucharest **to somewhere else …**

Uniform-cost Search (potential pitfall)

- Negative (or zero-cost) circles
  - It will get stuck in an infinite loop

- Completeness is guaranteed *"only if"* **the cost of every step exceeds some small positive constant *(ε)***

Uniform-cost Search (time complexity)

- Does not depend on "**d**" (the depth); instead, let's use **C*** (path cost of the optimal solution)

$$\boxed{O(b^{1+\lfloor C^*/\epsilon \rfloor})}$$

- When ***[C*/ε] = d***, the time complexity becomes similar to that of BFS

$$\boxed{b^{d+1}}$$

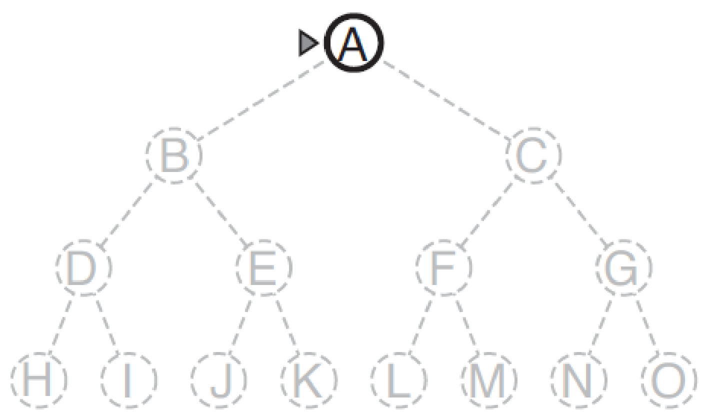
Outline

[Chapters 3.3-3.4]

- **Recap: Problem-Solving Agents**
- **Implementation of Search Algorithms**
- **Measuring Performance**
- **Uninformed Search Strategies**
  - **Breadth-first search**
  - **Uniform-cost search**
  - **Depth-first search**
  - **Depth-limited search**
  - **Iterative deepening depth-first search**
  - **Bidirectional search**

Depth-first search (DFS)

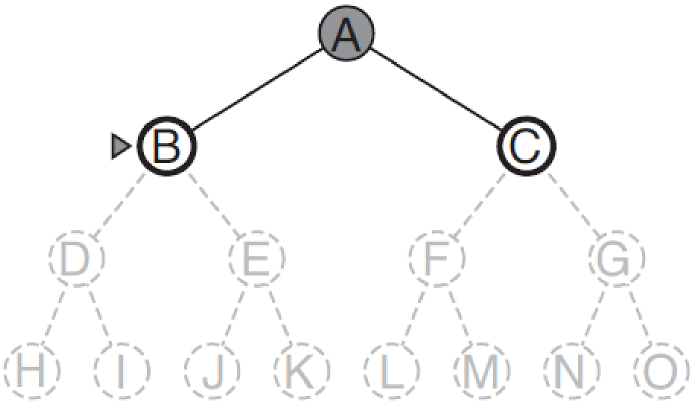
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

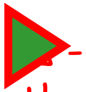


-- node to be expanded next is indicated by the marker

Depth-first search (DFS)

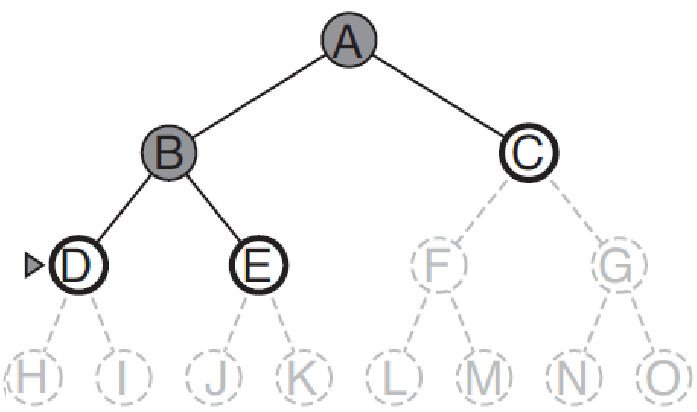
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)



 -- node to be expanded next is indicated by the marker

Depth-first search (DFS)

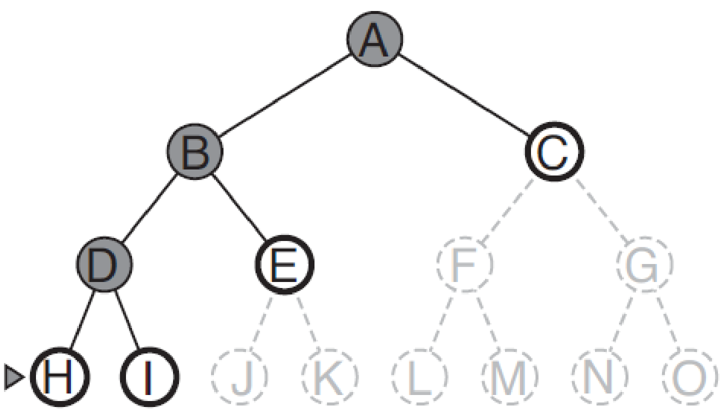
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

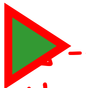


-- node to be expanded next is indicated by the marker

Depth-first search (DFS)

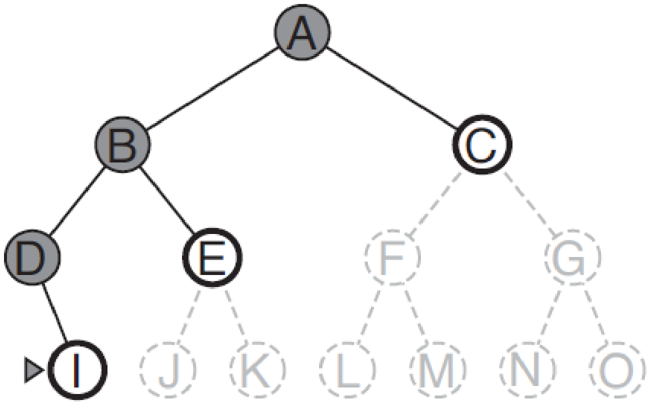
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

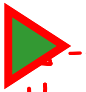



-- node to be expanded next is indicated by the marker

Depth-first search (DFS)

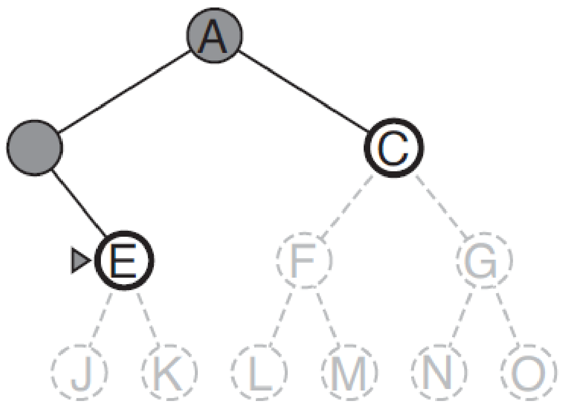
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)



 -- node to be expanded next is indicated by the marker

Depth-first search (DFS)

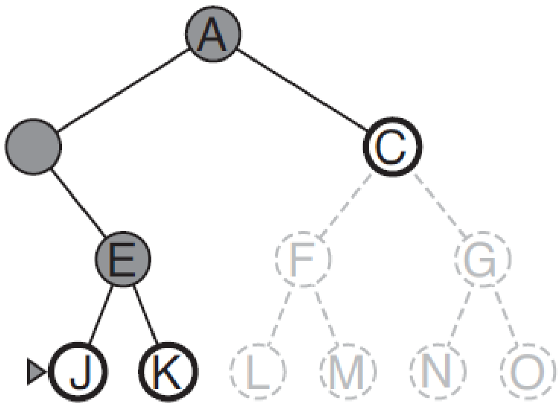
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

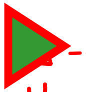


-- node to be expanded next is indicated by the marker

Depth-first search (DFS)

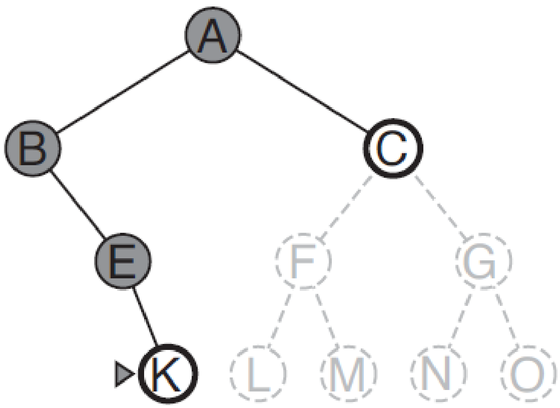
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)




-- node to be expanded next is indicated by the marker

Depth-first search (DFS)

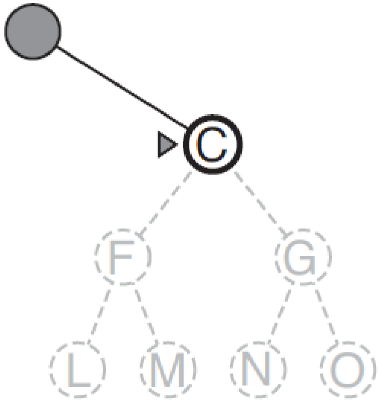
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

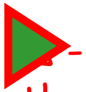


 -- node to be expanded next is indicated by the marker

Depth-first search (DFS)

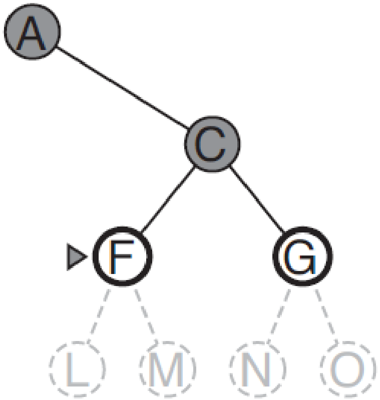
- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

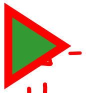


 -- node to be expanded next is indicated by the marker

Depth-first search (DFS)

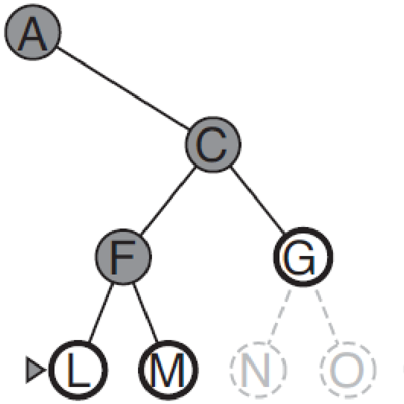
- The "**deepest**" nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

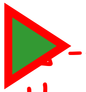


 -- node to be expanded next is indicated by the marker

Depth-first search (DFS)

- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)

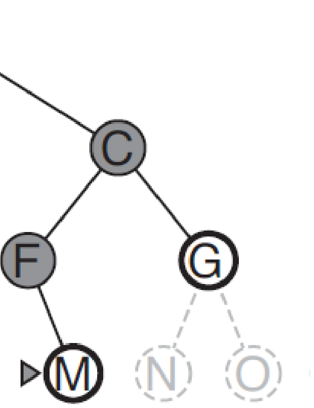




-- node to be expanded next is indicated by the marker

Depth-first search (DFS)

- The “**deepest**” nodes in frontier is chosen for expansion
  - The frontier is implemented using a LIFO queue (or stack)



-- node to be expanded next is indicated by the marker

DFS may be incomplete

- Unless the "**graph search**" is used (as opposed to the tree search) in a "**finite**" state space

- The tree search version, in general, is incomplete
  - Example: Arad—Sibiu—Arad—Sibiu loop forever

DFS's time complexity

- Parameter of input problem
  - **b** – branching factor
  - **m** – maximum depth of any state
    - Can be much larger than "d", the depth of the shallowest goal state

- Number of nodes explored
  - ***$O(b^m)$***

$O(b^d)$ for BFS looks much better;
so why use DFS, at all?

DFS's space complexity

- **Depth-first tree search** is actually the "**basic workhorse**" of many areas of AI because of its memory efficiency

- Require storage of only *O(bm)* nodes

Depth	Nodes	Time	Memory	DFS
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	156 kilobytes

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.

Variant of DFS: Backtracking search

- It uses still less memory, by generating “one successor” node at a time (as opposed to all successors)

- Comparison
  - BFS: **$O(b^d)$**
  - DFS: **$O(bm)$**
  - Backtracking search: **$O(m)$**

Variant of DFS: Backtracking search (cont'd)

- Here is yet another memory-saving trick: generating a successor node by "modifying current state description" rather than copying it
  - BFS: **$O(b^d)$**
  - DFS: **$O(bm)$**
  - Backtracking search: **$O(m)$**
  - Modified backtracking search: **$O(1)$ state + $O(m)$ actions**
- More improvement (no-cost backtracking): does not need to restore the state description back to the original form
  - In certain applications, this may be possible

Outline

[Chapters 3.3-3.4]

- **Recap: Problem-Solving Agents**
- **Implementation of Search Algorithms**
- **Measuring Performance**
- **Uninformed Search Strategies**
  - **Breadth-first search**
  - **Uniform-cost search**
  - **Depth-first search**
  - **Depth-limited search**
  - **Iterative deepening depth-first search**
  - **Bidirectional search**

Depth-limited Search

- Aim to make DFS terminating
  - Pros: Memory efficient O(bm)
  - Cons: **Incomplete**, **Not optimal**

May not terminate

May not find the best solution

Depth-limited Search (algorithm)

- Nodes at depth (l) are treated as if they have no successors

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)
```

```
function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Depth-limited Search (how to set the limit?)

- Number of States
  - Example: Map of Romania has 20 states → limit = 20

- Diameter
  - Example: Map of Romania has a diameter 9 → limit = 9

- Reachable Diameter
  - Any reachable city can be reached from the initial state in (k) steps ➔ limit = k

Depth-limited Search (difficulty in setting limit)

- In general, finding a tight depth “limit” is a hard problem
  - This can be a challenging research problem in itself

Outline

[Chapters 3.3-3.4]

- Recap: Problem-Solving Agents
- Implementation of Search Algorithms
- Measuring Performance
- **Uninformed Search Strategies**
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - **Iterative deepening depth-first search**
  - **Bidirectional search**

Best of Both Worlds(DFS + BFS)?

- Pseudo code

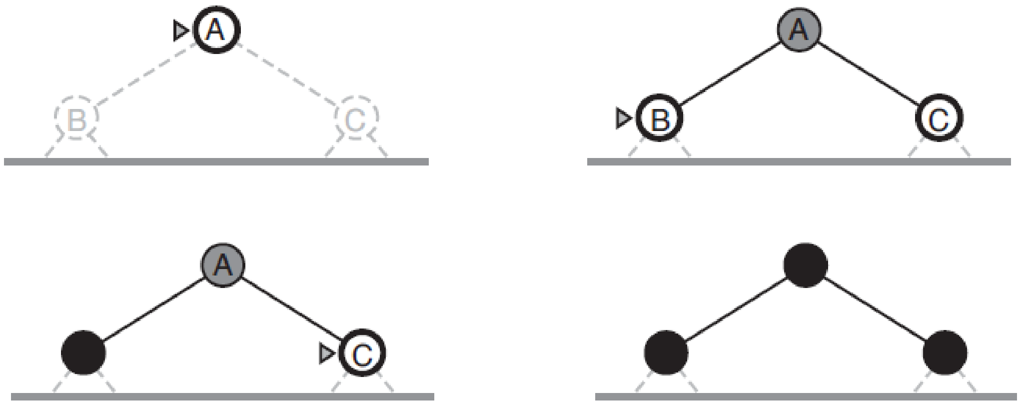
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

- Advantages
  - **Complete** and **Optimal** (just like BFS)
  - **O(bd)** storage requirement (just like DFS)

Iterative Deepening (example)

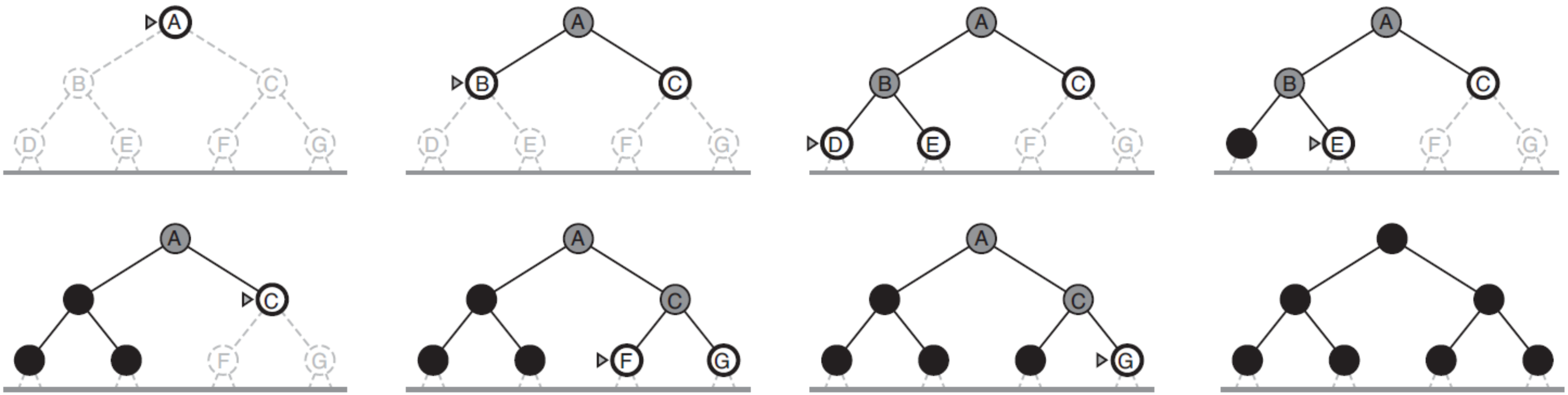
- Limit = 0

- Limit = 1



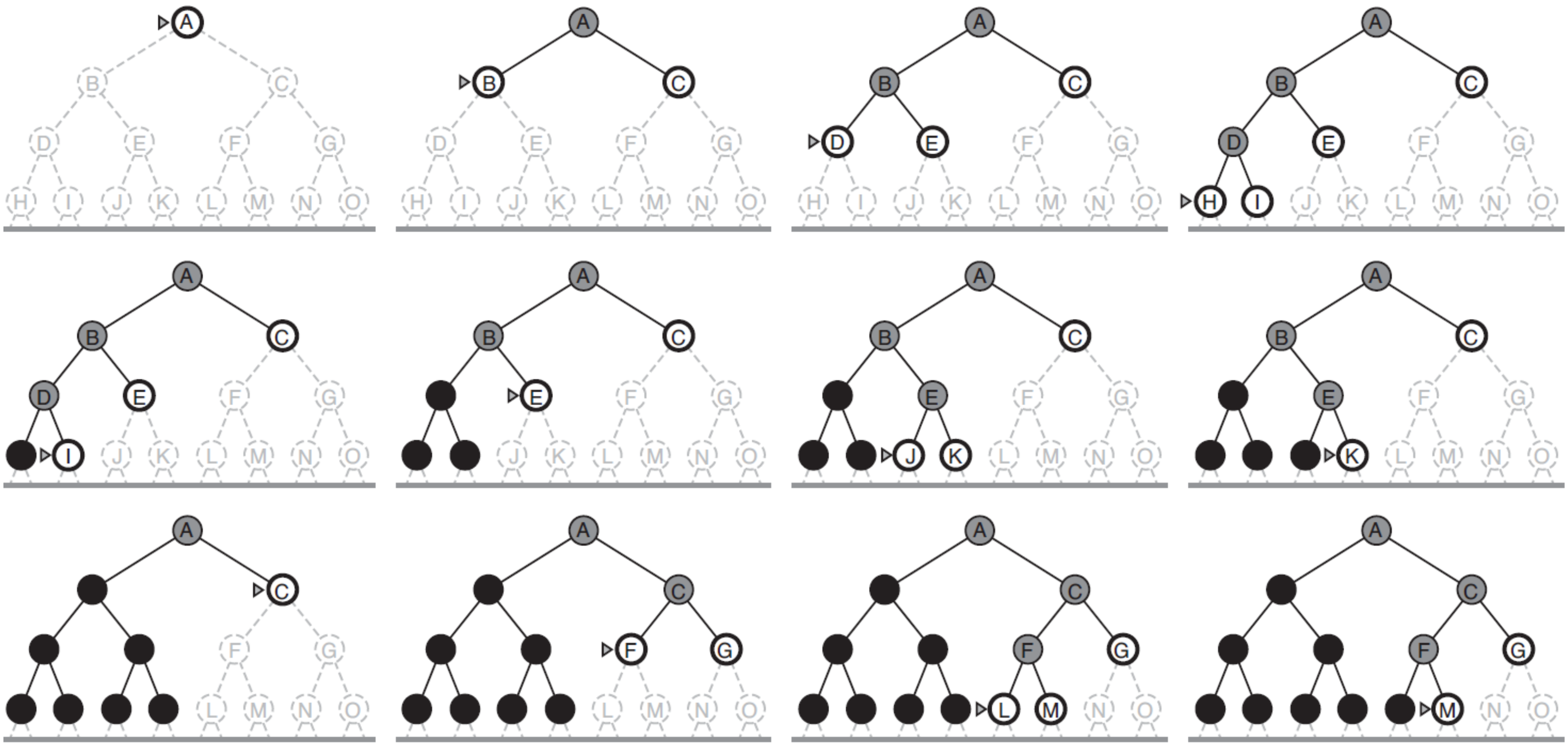
Iterative Deepening (example)

- Limit = 2



Iterative Deepening (example)

- Limit = 3



Iterative Deepening (Time Complexity)

- Except for (*limit = d*), each level is visited multiple times

$$N(\text{IDS}) = (d)b + (d-1)b^2 + \cdots + (1)b^d$$

- Compare to BFS with shallowest goal depth (d) → *$O(b^d)$*

- *Example: Let (b=10) and (d=5)*

$$\boxed{\begin{aligned} N(\text{IDS}) &= 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \\ N(\text{BFS}) &= 10 + 100 + 1,000 + 10,000 + 100,000 = 111,110 \end{aligned}}$$

  - *There is some extra cost for generating upper levels multiple times, but the extra cost is not large*

Iterative Deepening (conclusion)

- **Iterative deepening** is the preferred uninformed search method for many AI applications
  - E.g., when the search space is large and
  - the depth of the solution is not known

Iterative Deepening (uniform-cost search)

- Can you apply the idea of "iterative deepening" to uniform-cost search?
  - Using "path cost" instead of "depth" to set the limit
  - Can be done, but does not have the same time/space efficiency…

Bidirectional Search

- Simple observation regarding the “exponential” complexity

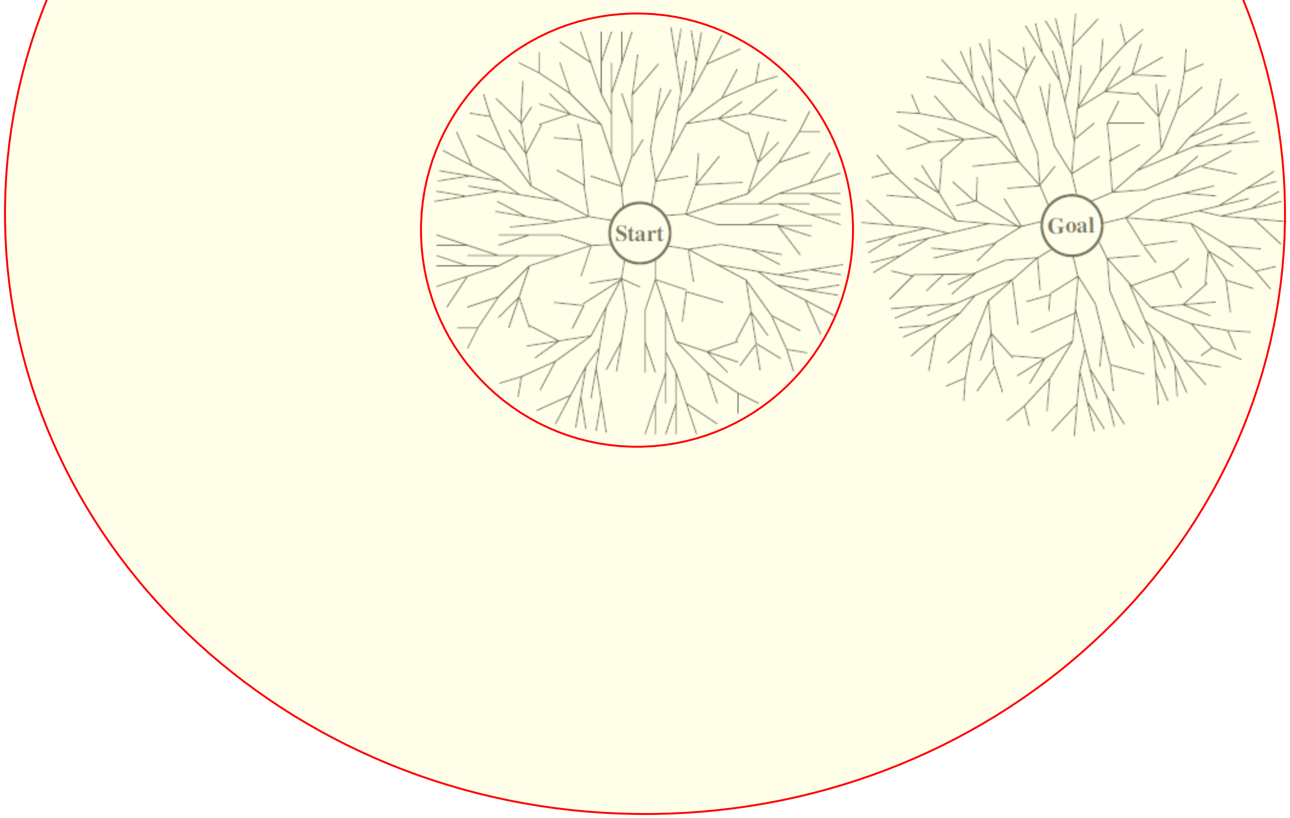
$$\boxed{b^{d/2} + b^{d/2} \text{ is much less than } b^d}$$

- Example: Let (b=10) and (d=6)
  - 2,220 states
  - 1,111,110 states

Bidirectional Search

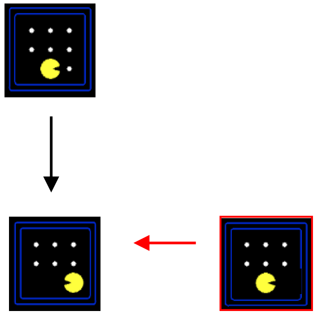
- Simple observation regarding the "exponential" complexity

$$b^{d/2} + b^{d/2} \text{ is much less than } b^d$$



Bidirectional Search (implementation issues)

- Computing "predecessors" may not be easy
  - The notion of "reversible actions" may not exist in the application
  - Even if actions are revisable, the predecessors may be many



Comparison of uninformed search algorithms

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

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Outline

[Chapters 3.3-3.4]

- **Recap: Problem-Solving Agents**
- **Implementation of Search Algorithms**
- **Measuring Performance**
- **Uninformed Search Strategies**
  - **Breadth-first search**
  - **Uniform-cost search**
  - **Depth-first search**
  - **Depth-limited search**
  - **Iterative deepening depth-first search**
  - **Bidirectional search**