Solving Problems by Searching

Instructor: Tsung-Che Chiang
tcchiang@ieee.org
Department of Computer Science and Information Engineering
National Taiwan Normal University

Outline

- Problem-solving Agents
- Example Problems
- Searching for Solutions
- Uninformed Search Strategies
- Avoiding Repeated States
- Searching with Partial Information
- Summary

Problem-solving Agents

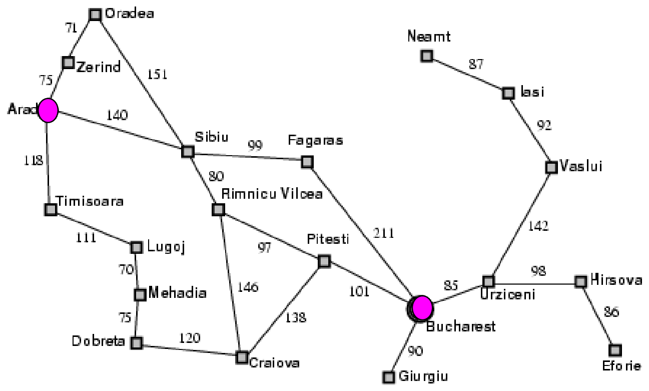
- Reflex agents vs. goal-based agents
  - Reflex agents cannot operate well in environments for which the state-action mapping is hard to store and learn.
  - Goal-based agents can succeed by considering future actions and the desirability of their outcomes.
- Problem-solving agents
  - They are a kind of goal-based agent.
  - They decide what to do by finding sequences of actions that lead to desirable states.

Problem-solving Agents

- Intelligent agents are supposed to maximize their performance measure.
- Achieving this is sometimes simplified by adopting a goal and aim at satisfying it.
- The agent's task is to find out an action sequence to a set of world states in which the goal is satisfied.
  - The process of looking for such a sequence is called search.

Problem-solving Agents

- Example: The agent is driving to Bucharest from Arad.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.2

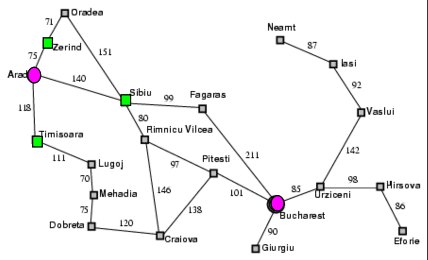
Problem-solving Agents

- Goal formulation,
  based on the current situation and the performance measure, is the first step in problem solving.

- Problem formulation
  is to decide what actions and states to consider, given a goal.

Problem-solving Agents

- If the agent has no knowledge, it can just choose a random road.
- If a map is given, it has
  - the information about the states it might get into and
  - the actions it can take



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.2

Problem-solving Agents

"An agent with several immediate options of unknown value can decide what to do by

first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence."

Problem-solving Agents

- Agent design

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation
    state ← UPDATE-STATE(state, percept)
    if seq is empty then do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Formulate
|
Search
|
Execute

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.1

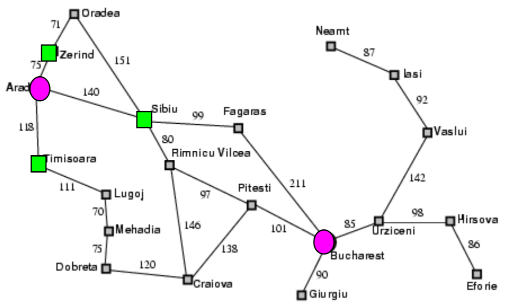
Problem-solving Agents

- Assumptions on the environment in the previous design
  - Static (formulation, searching, & execution)
  - Observable (initial state)
  - Discrete (enumeration of actions)
  - Deterministic

- We are dealing with a very easy environment.

Problem Formulation

- A problem can be defined formally by four components:
  - Initial state
    - e.g. In(Arad)
  - Possible actions
  - Goal test
  - Path cost



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.2

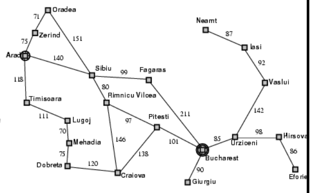
Problem Formulation

- Possible actions
  - The most common formulation uses a successor function.
  - e.g. From the state In(Arad), the successor function returns

    {<Go(Sibui), In(SIbui)>,
     <Go(Timisoara), In(Timisoara)>,
     <Go(Zerind), In(Zerind)>}

Problem Formulation

- Possible actions
  - The initial state and successor function implicitly define the state space of the problem – the set of all states reachable from the initial state.
  - The state space forms a graph in which the nodes are states and arcs are actions.
  - A path in the state space is a sequence of states connected by a sequence of actions.



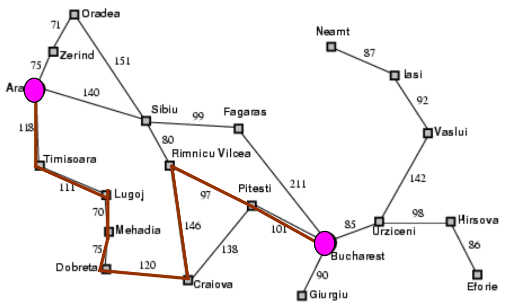
Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.2

Problem Formulation

- Goal test
  - Sometimes there is an explicit set of possible goal states.
  - Sometimes the goal is specified by an abstract property.

Problem Formulation

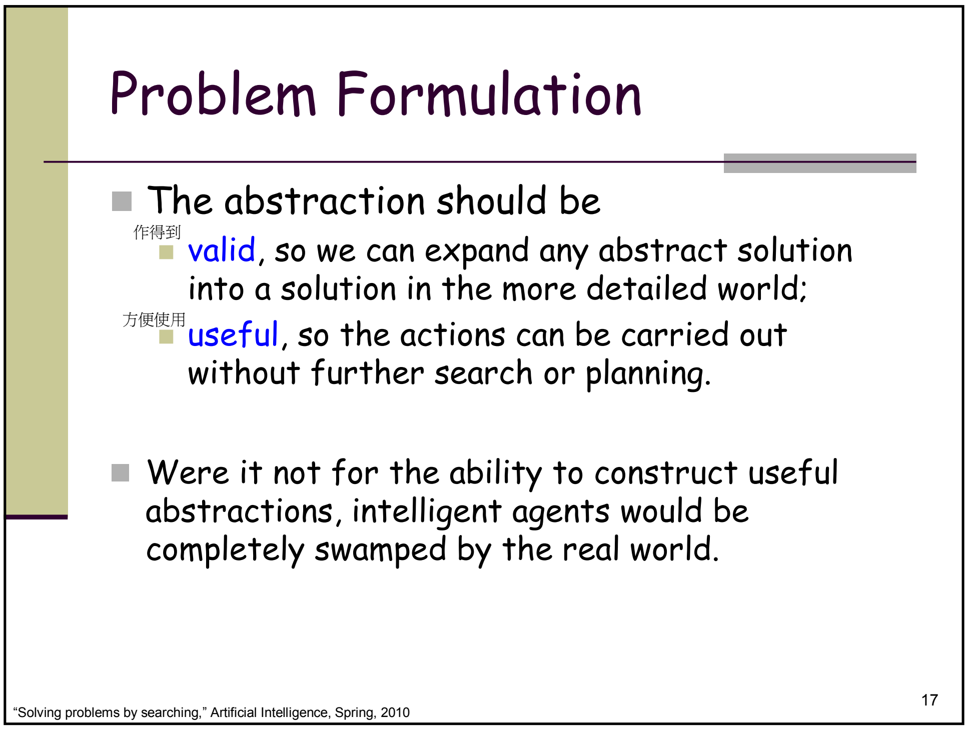
- A solution to a problem is a path from the initial state to a goal state.
- An optimal solution has the lowest path cost among all solutions.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.2

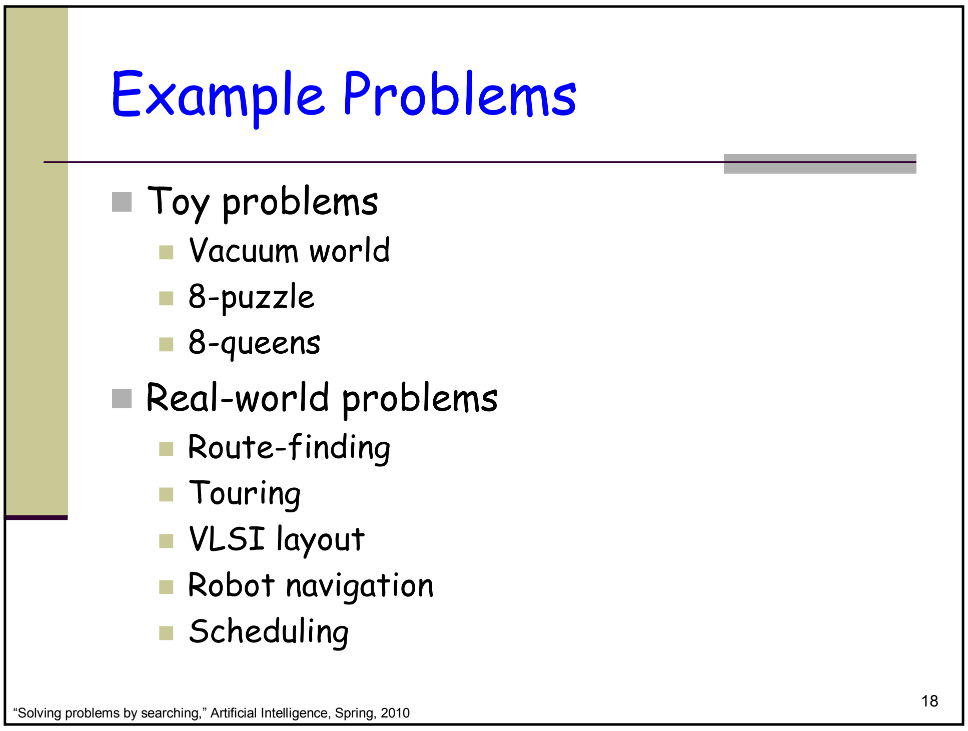
Problem Formulation

- During formulating a problem, considerations that are irrelevant to the problem (including both states and actions) can be filtered out – abstraction.



Problem Formulation

- The abstraction should be
  - 作得到 valid, so we can expand any abstract solution into a solution in the more detailed world;
  - 方便使用 useful, so the actions can be carried out without further search or planning.
- Were it not for the ability to construct useful abstractions, intelligent agents would be completely swamped by the real world.

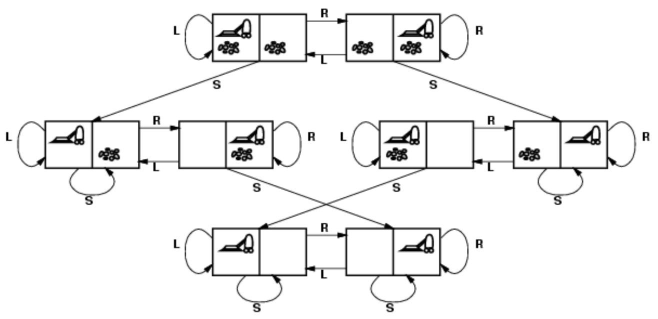


Example Problems

- Toy problems
  - Vacuum world
  - 8-puzzle
  - 8-queens
- Real-world problems
  - Route-finding
  - Touring
  - VLSI layout
  - Robot navigation
  - Scheduling

Example Problems

- Vacuum world
  - States: 2 location x {clean, dirty}^{2} = 8 states
  - Initial state: any state can be.
  - Successor function:



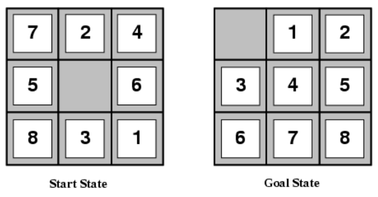
Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.3

Example Problems

- Vacuum world
  - Goal test: This checks whether all squares are clean.
  - Path cost: Each step costs 1.

Example Problems

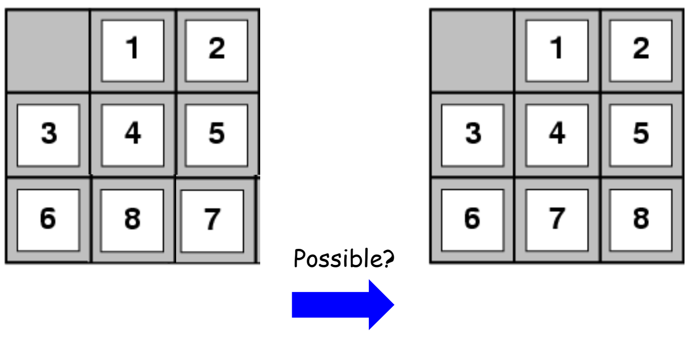
- 8-puzzle
  - States: locations of eight tiles and the blank
  - Initial state: any state can be.
  - Successor function: left, right, up, down
  - Goal test: any state can be.
  - Path cost: Each step costs 1



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.4

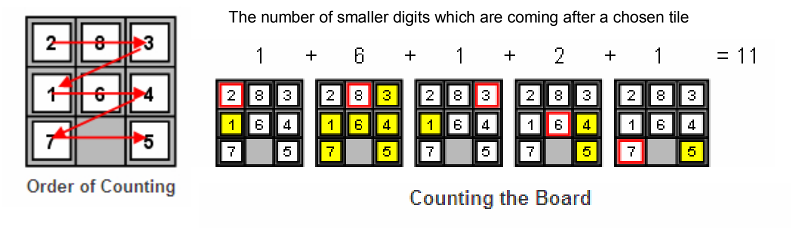
Example Problems

- 8-puzzle



Example Problems

- 8-puzzle
  - The states are divided into two disjoint sets.
  - Why?



The number of smaller digits which are coming after a chosen tile
1 + 6 + 1 + 2 + 1 = 11
Order of Counting
Counting the Board

    - Sliding the blank along a row doesn't change the sum.
    - Sliding the blank between rows doesn't change the sum%2.

http://www.8puzzle.com/8_puzzle_algorithm.html

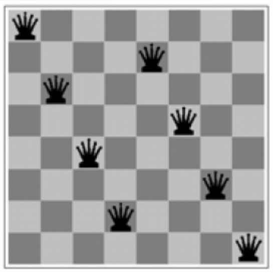
Example Problems

- 8-puzzle
  - It belongs to the family of sliding-block puzzles, known to be NP-complete.
  - Number of states
    - 8-puzzle: 9!/2 = 181,400 … very easy
    - 15-puzzle: 1.3 trillion states … still easy
    - 24-puzzle: 10^{25} states … quite difficult

Example Problems

- 8-queens (Basic formulation)
  - States: any arrangement of 0 to 8 queens
  - Initial state: no queens on the board
  - Successor function: add a queen to any empty square
  - Goal test: 8 queens on the board, none attacked
  - Path cost: 0

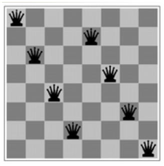
Note. The formulation here is incremental, we will see the complete formation later.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.5

Example Problems

- 8-queens (Smart formulation)
  - States:
    arrangement of n queens,
    one per column in the leftmost n columns,
    with no queen attacking another
  - Successor function:
    Add a queen to any square in the leftmost empty column such that it is not attacked by any other queen.
  - This formulation reduces the state space from 3×10^{14} to just 2,057.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.5

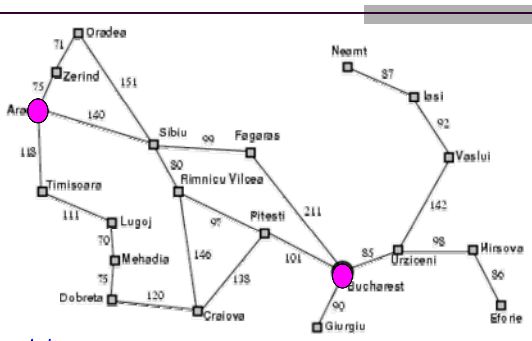
Searching for Solutions

- Searching through the state space generates a search tree (maybe a search graph).
  - It is important to distinguish between the state space and the search tree.

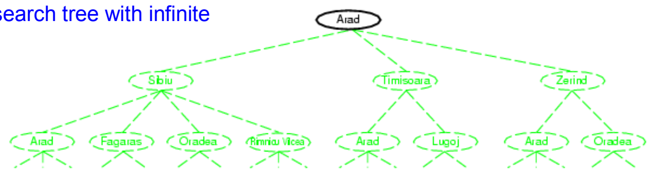
    state space: states + actions

    search tree: nodes + actions

Searching for Solutions

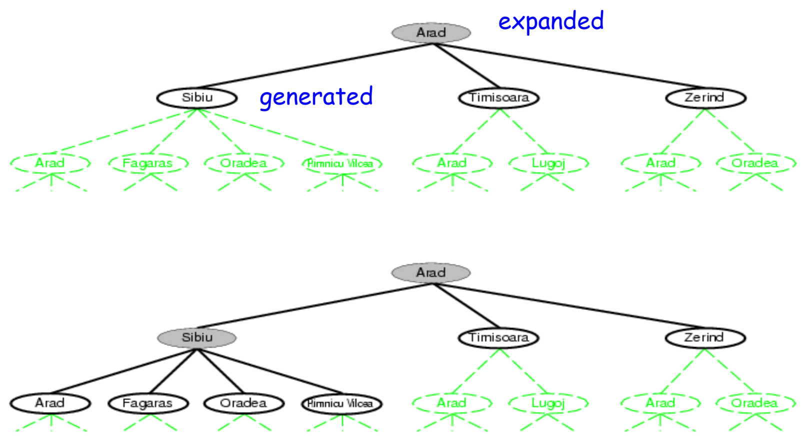


There are only 20 states in the state space.
But there could be a search tree with infinite number of nodes.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.2 & 3.6(a)

Searching for Solutions

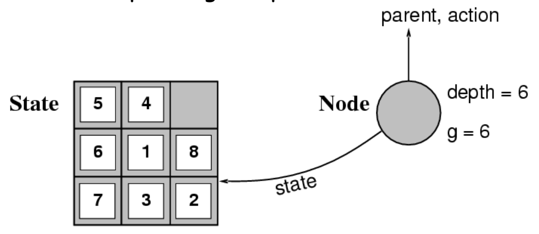


Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.6(b) & 3.6(c)

Searching for Solutions

- A node is assumed to have five components:
  - State
  - Parent-node
  - Action: action applied to the parent to generate the node
  - Path-cost: the cost from the initial state to the node
  - Depth: the number of steps along the path

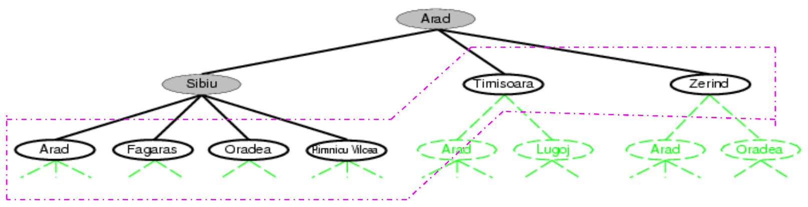
Note. States and node are different. Different nodes can contain the same world state.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.8

Searching for Solutions

- The fringe is the collection of nodes that have been generated but not yet expanded.
  - Every element of the fringe is a leaf node.
- The search strategy selects the next node to be expanded from the fringe.



Searching for Solutions

- General tree-search algorithm

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

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN[problem](STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.9

Searching for Solutions

- Measuring performance
  - Completeness
  - Optimality
  - Time complexity
    - usu. in terms of the number of nodes generated
  - Space complexity
    - maximum number of nodes stored in the memory

The complexity is usually expressed in terms of three quantities:

b, branching factor
d, depth of the shallowest goal node
m, maximum length of any path

Uninformed Search Strategies

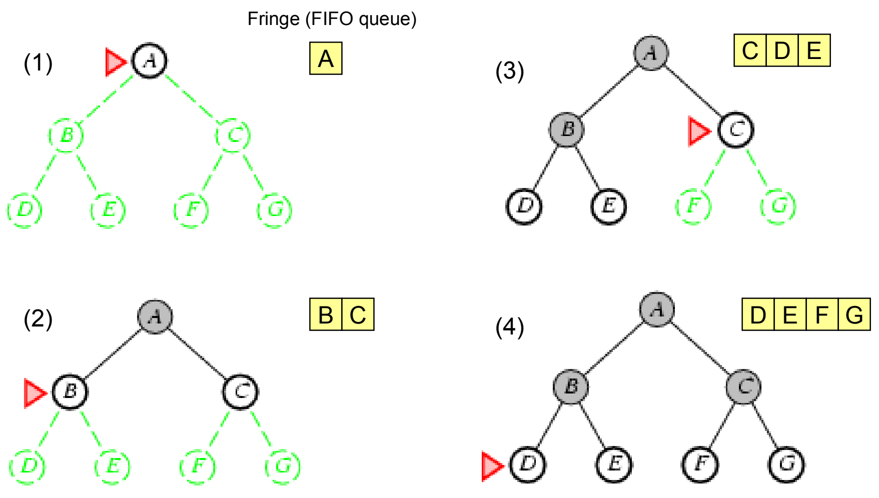
- Uninformed search (blind search) can only generate successors and distinguish a goal state from a non-goal state.
- Informed search (heuristic search) knows whether one non-goal state is "more promising" than another.
- All search strategies are distinguished by the order in which nodes are expanded.

Breadth-First Search

- BFS can be implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out (FIFO) queue.

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

Breadth-First Search



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.10

Breadth-First Search

- It is complete, provided the branching factor b is finite.
- The shallowest goal node is not necessarily optimal.
  - It is optimal when all step costs are equal.

Breadth-First Search

- Complexity analysis
  - Assume branching factor is b, the solution is at depth d.
  - In the worst case, we would expand all but the last node at level d.

Total number of nodes generated is
$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$. $\Leftarrow$ Time complexity

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. $\Leftarrow$ Space complexity

Breadth-First Search

Depth	Nodes	Time	Memory
2	1100	.11 seconds	1 megabyte
4	111,100	11 seconds	106 M
6	10^7	19 minutes	10 G
8	10^9	31 hours	1 T
10	10^{11}	129 days	101 T
12	10^{13}	35 years	10 P
14	10^{15}	3,523 years	1000 P

Assume that (1) b = 10; (2) 10,000 nodes can be generated per second; (3) a node requires 1000 bytes of storage.

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.11

Breadth-First Search

- Lessons we learned
  - The time requirements are a major factor.
  - The memory requirements are a bigger program for BFS than is the execution time.

    (In general, exponential-complexity search problems cannot be solved by uninformed methods for any but the smallest instances.)

Tea Time (Roomba)

- Wiimba
  http://www.youtube.com/watch?v=NqbcfSqPnLA
- Pacmba
  http://www.youtube.com/watch?v=2wsP_nmk_iw
- Surfin'ba
  http://www.youtube.com/watch?v=tLbprdjTX0w

Uniform-Cost Search

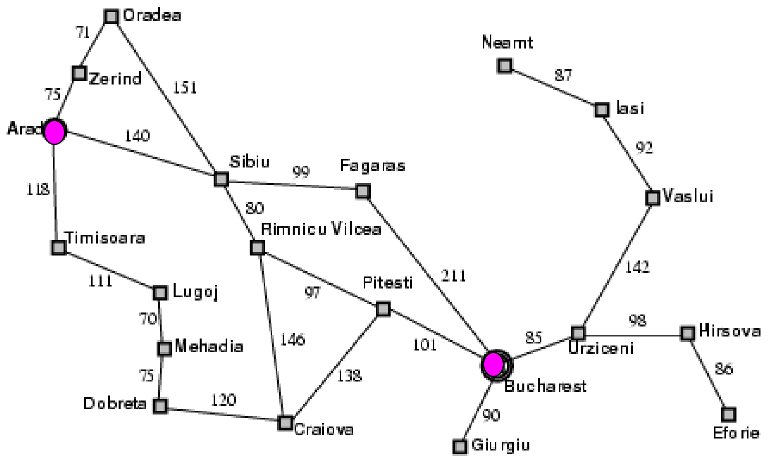
- It expands the node with the lowest path cost (not step cost!), instead of the shallowest node.
  - It is identical to BFS if all step costs are equal.
- It is complete and optimal with any step cost function

  provided the cost of every step is at least a small positive constant ε.

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.2

Uniform-Cost Search

- Exercise: BFS & UCS



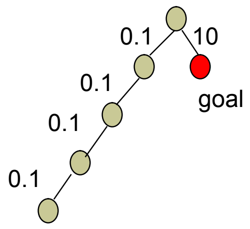
Uniform-Cost Search

- Worst-case time and space complexity:

  $O(b^{1+\lfloor C^*/\varepsilon \rfloor})$, where

  - C^* is the cost of the optimal solution, and
  - every action costs at least ε.

It often explores large trees of small steps before exploring paths involving large and perhaps useful steps.

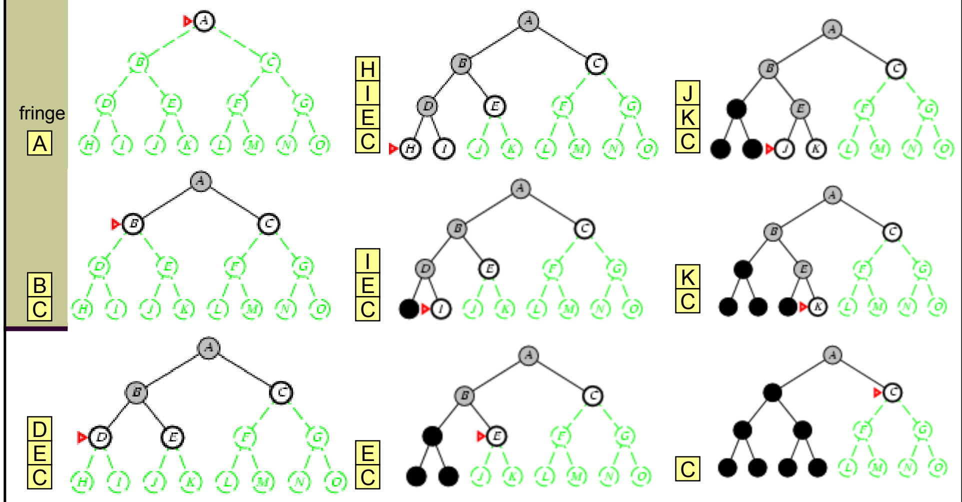


Depth-First Search

- It always expands the deepest node in the fringe.
- After reaching the deepest level, it backs up the next deepest node that still has unexplored successors.
- It can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue.

Depth-First Search

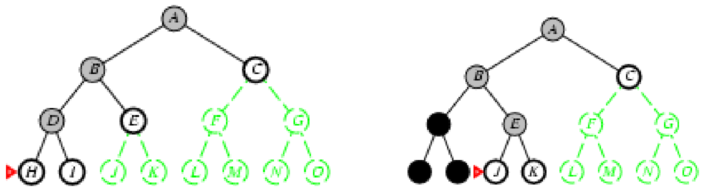
White: in the fringe
Gray: not in the fringe but still required (parents of nodes in the fringe)
Black: can be removed from memory



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.12

Depth-First Search

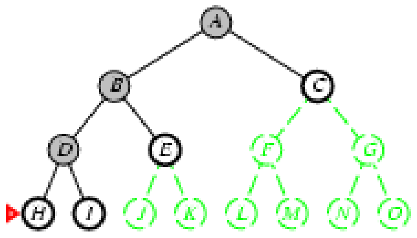
- It has very modest memory requirements since it only stores
  - a single path from the root to the leaf node, and
  - the remaining unexpanded sibling nodes.
- Once a node is expanded, it can be removed after all its descendants are fully explored.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.12

Depth-First Search

- Given branching factor b and maximum depth m, the memory requirement is $bm+1$ nodes. (cf. BFS: $O(b^d)$)



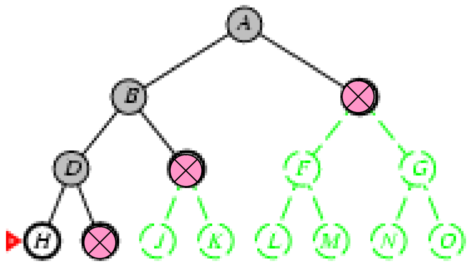
b = 2
m = 3

$b \cdot m$ + 1 = 7

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.12

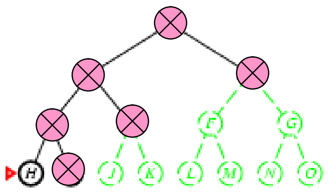
Depth-First Search

- A variant: Backtracking search
  - If each partially expanded node can remember which successor to generate next, only one successor is generated at a time.
    $\Rightarrow$ Memory requirement $O(bm) \to O(m)$



Depth-First Search

- A variant: Backtracking search
  - If we can undo the action when we go back, we can keep only one state rather than $O(m)$ states.



  - These techniques are critical to success for solving problems with large state descriptions such as robotic assembly.

Depth-First Search

- It is neither complete nor optimal.
  - It can make a wrong choice and get stuck going down a very long (even infinite) path.
- In the worst case, it will generate $O(b^m)$ nodes in the search tree.
  - Note that m can be much larger than d or even infinite if the tree is unbounded.

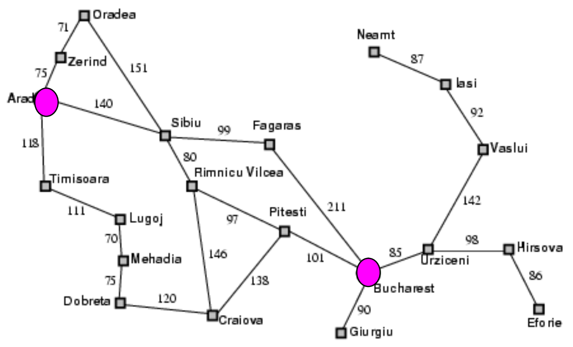
Depth-Limited Search

- The problem of unbounded tree can be alleviated supplying DFS with a depth limit ℓ.
- Unfortunately, it introduces an additional source of incompleteness if $\ell < d$.
- It is nonoptimal if $\ell > d$.
- Time complexity: $O(b^\ell)$
- Space complexity: $O(b\ell)$

Depth-Limited Search

- Sometimes depth limits can be set based on knowledge of the problem.

What value will you set for ℓ ?



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.2

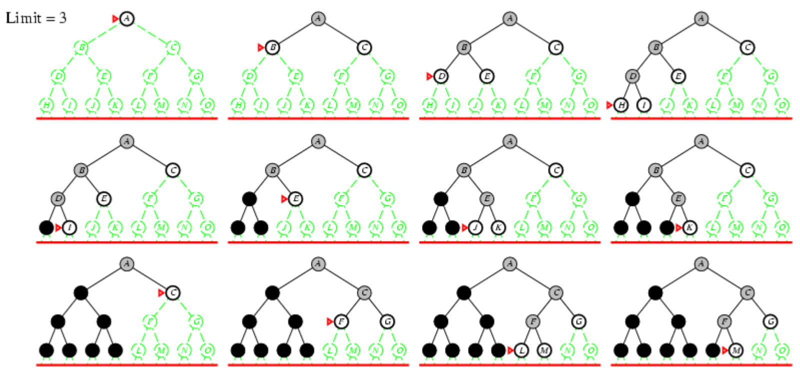
Depth-Limited Search

```
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
```

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.13

Depth-Limited Search



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.15

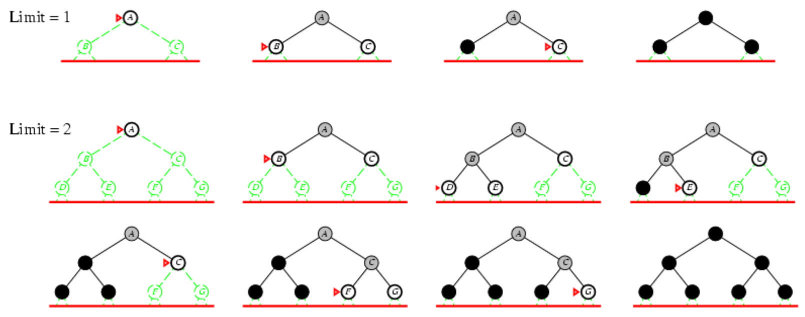
Iterative-Deepening DFS

- It combines the benefits of DFS and BFS.
  - Its memory requirement is modest. (DFS)
  - It is complete when the branching factor is finite. (BFS)
  - It is optimal when the path cost is a non-decreasing function of the depth. (BFS)

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

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.14

Iterative-Deepening DFS

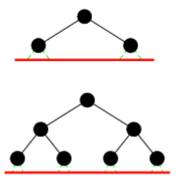


Iterative-Deepening DFS

- It is not as costly as you may imagine.

  N(IDS) = $(d)b + (d-1)b^2 + \ldots + (1)b^d$
  N(BFS) = $b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b)$

  If b = 10 and d = 5,
  N(IDS) = 123,450 and N(BFS) = 1,111,100.



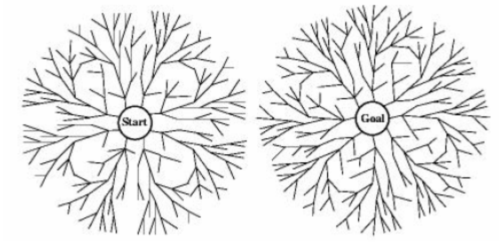
- In general, IDS is the preferred uninformed search method when there is a large search space and the depth of the solution is unknown.

Iterative-Deepening DFS

- It would seem worthwhile to develop an iterative analog to uniform-cost search.
  - The idea is to use increasing path-cost limits instead of increasing depth limits.
- The resulting algorithm, called iterative lengthening search, turns out to incur substantial overhead compared to uniform-cost search.

Bidirectional Search

- Motivation: $b^{d/2} + b^{d/2}$ is much less than b^d.
- It is implemented by having one or both of the searches check each node before expansion to see if it is in the fringe of the other search tree.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.16

Bidirectional Search

- Whether the algorithm is complete/optimal depends on the search strategies in both searches.
- Time complexity: $O(b^{d/2})$ (Assume BFS is used)
  - Checking node for membership in the other search tree can be done in constant time.
- Space complexity: $O(b^{d/2})$ (Assume BFS is used)
  - At least one of the search tree must be kept in memory for membership checking.

Bidirectional Search

- In fact, searching backward is not easy.
  - How can we find the predecessors of a state? Are actions reversible?
  - Is there only one goal state?
    - e.g. "clean world" in vacuum world,
      "checkmate" in chess.

Comparison of Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes[a]	Yes[a, b]	No	No	Yes[a]
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes*	Yes	No	No	Yes*

a: complete if b is finite
b: complete if step costs $\geq \varepsilon$ for positive ε.
*: optimal if step costs are all identical

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.17

Avoiding Repeated States

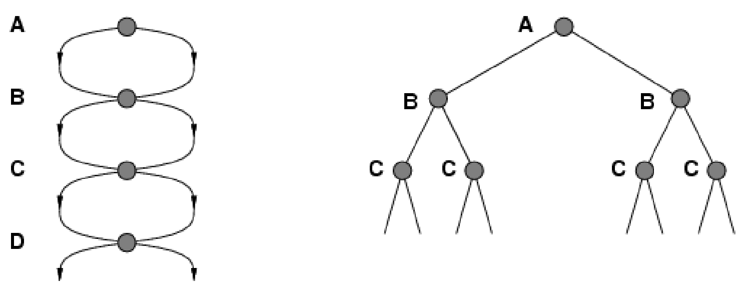
- For some problems, the state space is a tree, and there is only one path to each state.
  - e.g. the smart formulation of the 8-queens problem
- However, for some problems, we need to be careful about the possibility of expanding states that have been expanded.
  - e.g. the basic formulation of the 8-queens problem

Avoiding Repeated States

- For the problems with reversible actions, repeated states are unavoidable.
  - e.g. route-finding problems and sliding-blocks puzzles
- Repeated states can cause a solvable problem to become unsolvable.

Avoiding Repeated States

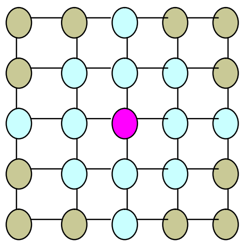
- An extreme example:
  a state space of size $d+1$ becomes a tree with 2^d leaves.



Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.18

Avoiding Repeated States

- A more realistic example
  - about $2d^2$ distinct states but 4^d leaves within d steps
  - for d = 20, this means about a trillion nodes but only about 800 distinct states

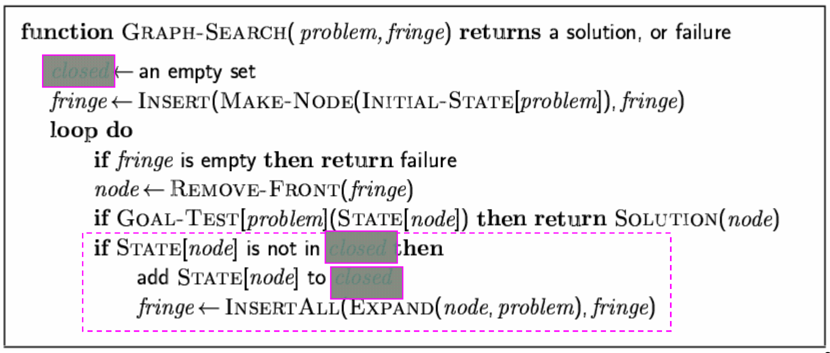


Avoiding Repeated States

- For DFS, the only nodes in memory are those on the path from the root to the current node.
  - Checking those nodes to the current node can detect the looping paths.
  - However, it cannot avoid the exponential proliferation of nonlooping paths.
  - There is a fundamental tradeoff between space and time.
    - "Algorithms that forget their history are doomed to repeat it."

Avoiding Repeated States

- If an algorithm remembers every state that it visited, then it can be viewed as exploring the state-space graph directly.



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

Artificial Intelligence: A Modern Approach, 2nd ed., Figure 3.19

Avoiding Repeated States

- On problems with many repeated states, GRAPH-SEARCH is much more efficient than TREE-SEARCH.
  - Its worst-case time and space complexity are proportional to the size of the state space. [This may be much smaller than $O(b^d)$.]
  - Because GRAPH-SEARCH keeps every node in memory, some searches are infeasible due to memory limitations.

Avoiding Repeated States

- The optimality of GRAPH-SEARCH
  - It needs to check whether a newly discovered path to a node is better than the original one.
    - BFS & UCS are already optimal graph-search strategies with (constant) step costs.
    - For DFS and IDS, we need to do the check.
  - If so, it might need to revise the depths and path costs of that node's descendants.

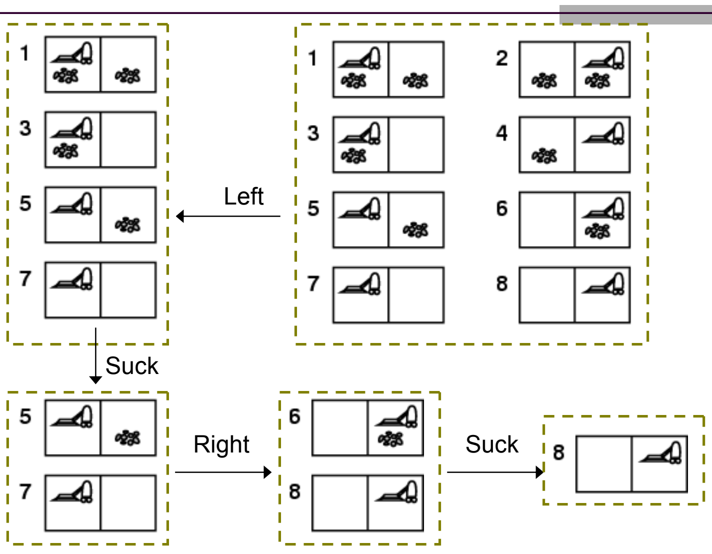
Searching with Partial Information

- What if the knowledge of the states or actions is incomplete? (Not fully observable or not deterministic)
- Different types of incompleteness lead to three distinct problem types:
  - Sensorless problems
  - Contingency problems
  - Exploration problems (Sec. 4.5)

Searching with Partial Information

- Sensorless problems (Example: the vacuum world)
  - The agent has no sensors at all. It could be in one of several possible initial states.
  - It knows all the effects of its actions. Each action might lead to one of several possible successor states.

Searching with Partial Information



Searching with Partial Information

- The agent can reach state 8 with the action sequence [Left, Suck, Right, Suck] without knowing the initial state.

  "When the world is not fully observable, the agent must reason about sets of states (belief state), rather than a single state."

Searching with Partial Information

- Searching in the space of belief states
  - An action is applied to a belief state by unioning the results of applying the action to each physical state in the belief state.
  - A path now connects belief states.
  - A solution now is a path leading to a belief state, all of whose members are goal states.
  - In general, there are 2^S belief states given S physical states. (But some of them may not be reachable, see the example on page 74.)

Searching with Partial Information

- The analysis is essentially the same if the environment is not deterministic.
- We just need to add the various outcomes of the action to the successor belief state.

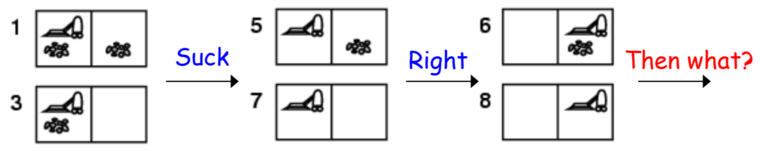
Searching with Partial Information

- Contingency problems
  - The environment is only partially observable and/or nondeterministic, but the agent can obtain new information from sensors after acting.
  - For example, in the vacuum world, the Suck action sometimes deposits dirt when there is no dirt already.

Searching with Partial Information

- Example

Percept [L, Dirty]



The agent has a location sensor and a "local" dirt sensor.
No fixed action sequence guarantees a solution to this problem.

Searching with Partial Information

- Many problems in the real world are contingency problems, because exact prediction is impossible.
- They lead to a different agent design, in which the agent can act before finding a guaranteed plan.
  - This type of interleaving of search and execution is also useful for exploration and game playing.

Summary

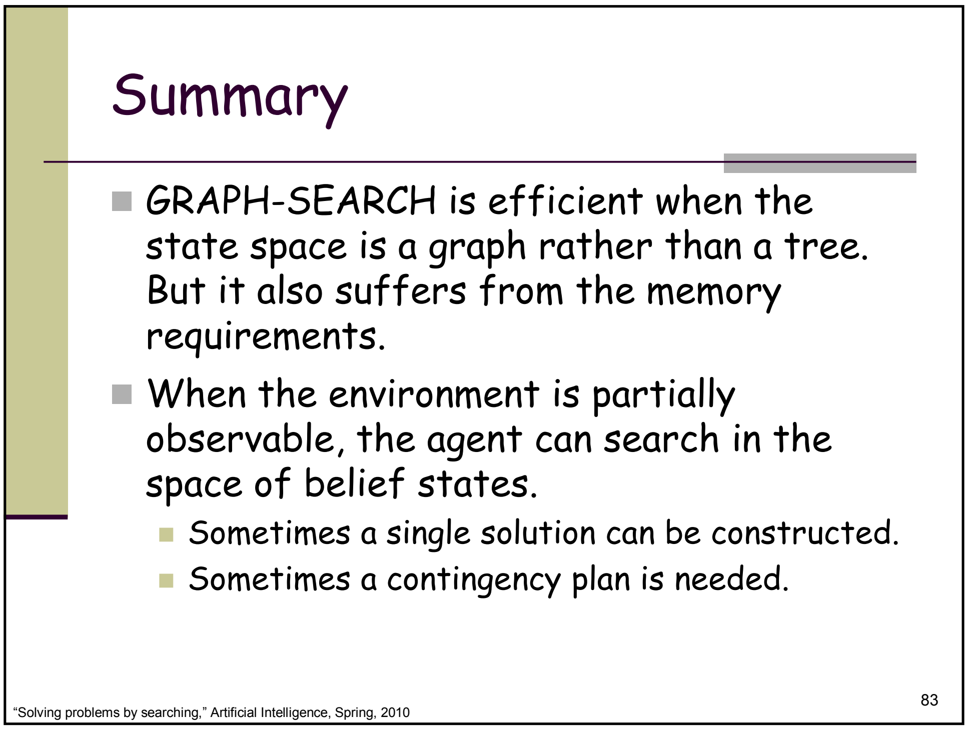
- In the deterministic, observable, and static environments, the agent can construct sequences of actions to achieve its goal – search.
- A problem consists of
  - the initial state,
  - a set of actions,
  - a goal test function, and
  - a path cost function

Summary

- Common blind search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

- Bidirectional search can be efficient, but not always applicable and may require too much space.



Summary

- GRAPH-SEARCH is efficient when the state space is a graph rather than a tree. But it also suffers from the memory requirements.
- When the environment is partially observable, the agent can search in the space of belief states.
  - Sometimes a single solution can be constructed.
  - Sometimes a contingency plan is needed.