Problem Solving and Search

Chapter 3
Outline

• Problem-solving agents
• Problem formulation
• Example problems
• Basic search algorithms
Problem-Solving Agents

In the *simplest* case, an agent will:

- formulate (or be given) a goal and a problem;
- search for a sequence of actions that solves the problem;
- then execute the actions.

When done it may formulate another goal and start over.

- In this case the performance measure is simply whether or not the goal is attained.
Problem-solving agents

Restricted form of general agent:

Function Simple-Problem-Solving-Agent(percept) returns an action
Problem-solving agents

Restricted form of general agent:

Function Simple-Problem-Solving-Agent(percept) returns an action

- persistent seq an action sequence, initially empty
- state some description of the current world state
- goal a goal, initially null
- problem a problem formulation
Problem-solving agents

Restricted form of general agent:

Function **Simple-Problem-Solving-Agent**(percept) **returns** an action

*persistent 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
  goal ← Formulate-Goal(state)
  problem ← Formulate-Problem(state,goal)
  seq ← Search(problem)
  if seq = fail then return null

action ← First(seq,state);    seq ← Rest(seq,state)

return action
Problem-solving agents

- This is *offline* problem solving, executed “eyes closed.”
  - Requires complete knowledge about the domain
- *Online* problem solving involves acting without necessarily having complete knowledge.
Example: Romania

- On holiday in Romania; currently in Arad.
  - Flight leaves tomorrow from Bucharest
- Formulate goal
  - Be in Bucharest
- Formulate problem
  - \textit{states}: various cities
  - \textit{actions}: drive between cities
- Find solution
  - Sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Example: Romania
Problem Formulation: State-Space Search

A problem is defined by five items:

1. The set of states, including the initial state, e.g. “at Arad”
2. Actions available to the agent, e.g. Vacuum: Suck, Left, …
3. Transition model: What actions do, defines a graph.
   • I.e. RESULT(s, a) = state resulting from doing a in s.
     e.g. RESULT(In(Arad), Go(Zerind)) = In(Zerind)
4. 1.–3. define the state space
5. Goal test. Can be explicit, e.g. x = “at Bucharest”
   implicit, e.g. NoDirt(x)
6. Path cost (additive) e.g. sum of distances, number of actions, etc.
   c(x, a, y) is the step cost, assumed to be \( \geq 0 \)

A solution is a sequence of actions from initial state to a goal state.
Problem Formulation: State-Space Search

A \textit{problem} is defined by five items:

1. The set of \textit{states}, including the \textit{initial state} e.g. “at Arad”
Problem Formulation: State-Space Search

A *problem* is defined by five items:

1. The set of *states*, including the *initial state* e.g. “at Arad”
2. *Actions* available to the agent E.g. Vacuum: Suck, Left, . . .
Problem Formulation: State-Space Search

A *problem* is defined by five items:

1. The set of *states*, including the *initial state* e.g. “at Arad”
2. *Actions* available to the agent E.g. Vacuum: Suck, Left, ...
   - I.e. $RESULT(s, a) =$ state resulting from doing $a$ in $s$.
     e.g. $RESULT(In(Arad), Go(Zerind)) = In(Zerind)$
4.–3. define the *state space*
Problem Formulation: State-Space Search

A problem is defined by five items:

1. The set of states, including the initial state e.g. “at Arad”
2. Actions available to the agent E.g. Vacuum: Suck, Left, . . .
3. Transition model: What actions do; defines a graph.
   - i.e. \( \text{RESULT}(s, a) = \text{state resulting from doing } a \text{ in } s. \)
   - e.g. \( \text{RESULT}(\text{In}(\text{Arad}), \text{Go}(\text{Zerind})) = \text{In}(\text{Zerind}) \)

1.–3. define the state space

4. Goal test. Can be explicit, e.g. \( x = \text{“at Bucharest”} \)
   implicit, e.g. \( \text{NoDirt}(x) \)
Problem Formulation: State-Space Search

A problem is defined by five items:

1. The set of **states**, including the **initial state** e.g. “at Arad”
2. **Actions** available to the agent E.g. Vacuum: Suck, Left, ...
3. **Transition model**: What actions do; defines a graph.
   - I.e. $RESULT(s, a) =$ state resulting from doing $a$ in $s$.
     e.g. $RESULT(\text{In}(\text{Arad}), \text{Go}(\text{Zerind})) = \text{In}(\text{Zerind})$

1.–3. define the **state space**

4. **Goal test**. Can be **explicit**, e.g. $x =$ “at Bucharest”
   **implicit**, e.g. $\text{NoDirt}(x)$

5. **Path cost** (additive)
   e.g. sum of distances, number of actions , etc.
   $c(x, a, y)$ is the **step cost**, assumed to be $\geq 0$
Problem Formulation: State-Space Search

A problem is defined by five items:

1. The set of states, including the initial state e.g. “at Arad”
2. Actions available to the agent E.g. Vacuum: Suck, Left, …
3. Transition model: What actions do; defines a graph.
   - I.e. $\text{RESULT}(s, a) =$ state resulting from doing $a$ in $s$.
     e.g. $\text{RESULT}(\text{In(Arad)}, \text{Go(Zerind)}) = \text{In(Zerind)}$
1.–3. define the state space

4. Goal test. Can be explicit, e.g. $x =$ “at Bucharest”
   implicit, e.g. $\text{NoDirt}(x)$

5. Path cost (additive)
   e.g. sum of distances, number of actions, etc.
   $c(x, a, y)$ is the step cost, assumed to be $\geq 0$

A solution is a sequence of actions from initial state to a goal state.
Selecting a State Space

- The real world is highly complex and contains lots of irrelevant information.
  ⇒ state space must be *abstracted* for problem solving
- (Abstract) state will have irrelevant detail removed.
- Similarly, actions must be at the right level of abstraction
  - e.g., “Go(Zerind)” omits things like starting the car, steering, etc.
- (Abstract) solution =
  set of paths that are solutions in the real world
Example: Vacuum World State Space Graph

states:
actions:
transition model:
goal test:
path cost:
Example: Vacuum World State Space Graph

states: dirt and robot locations (so $2 \times 2^2$ possible states)

actions:

transition model:

goal test:

path cost:
Example: Vacuum World State Space Graph

states: dirt and robot locations
actions: Left, Right, Suck, NoOp
transition model:
goal test:
path cost:
Example: Vacuum World State Space Graph

states: dirt and robot locations

actions: Left, Right, Suck, NoOp

transition model: actions as expected, except moving left (right) in the right (left) square is a NoOp

goal test:

path cost:
Example: Vacuum World State Space Graph

- **states:** dirt and robot locations
- **actions:** *Left, Right, Suck, NoOp*
- **transition model:** actions as expected, except moving left (right) in the right (left) square is a *NoOp*
- **goal test:** no dirt
- **path cost:**
Example: Vacuum World State Space Graph

- **states:** dirt and robot locations
- **actions:** *Left*, *Right*, *Suck*, *NoOp*
- **transition model:** actions as expected, except moving left (right) in the right (left) square is a *NoOp*
- **goal test:** no dirt
- **path cost:** 1 per action (0 for *NoOp*)
Example: The 8-puzzle

states:
actions:
transition model:
goal test:
path cost:
Example: The 8-puzzle

states:  (integer) locations of tiles.
\[\text{Ignore intermediate positions}\]

actions:

transition model:

goal test:

path cost:
Example: The 8-puzzle

- **states**: locations of tiles
- **actions**: move blank left, right, up, down
- **transition model**:
- **goal test**:
- **path cost**:

Start State

```
7 2 4
5  _ 6
8 3 1
```

Goal State

```
1 2 3
4 5 6
7 8  _
```
Example: The 8-puzzle

states: locations of tiles
actions: move blank left, right, up, down
transition model: given a state and action give the resulting state
goal test:
path cost:
Example: The 8-puzzle

states: locations of tiles

actions: move blank left, right, up, down

transition model: given a state and action give the resulting state

goal test: = goal state (given)

path cost:
Example: The 8-puzzle

states: locations of tiles
actions: move blank left, right, up, down
transition model: given a state and action give the resulting state
goal test: = goal state (given)
path cost: 1 per move

[Aside: optimal solution of $n$-Puzzle family is NP-hard]
Example: Airline Travel

states:
Example: Airline Travel

**states:** Include locations (airports), current time.

- Also perhaps fares, domestic/international, and other “historical aspects”.

**initial state:**
Example: Airline Travel

**states:** Include locations (airports), current time.

- Also perhaps fares, domestic/international, and other “historical aspects”.

**initial state:** Given by a user’s query

**actions:**
Example: Airline Travel

states: Include locations (airports), current time.
- Also perhaps fares, domestic/international, and other “historical aspects”.

initial state: Given by a user’s query

actions: Flight from current location with attributes such as seat class, departure time, etc.

transition model:
Example: Airline Travel

states: Include locations (airports), current time.

- Also perhaps fares, domestic/international, and other “historical aspects”.

initial state: Given by a user’s query

actions: Flight from current location with attributes such as seat class, departure time, etc.

transition model: The state resulting from taking a flight, including destination and arrival time.

goal test:
Example: Airline Travel

**states:** Include locations (airports), current time.

- Also perhaps fares, domestic/international, and other “historical aspects”.

**initial state:** Given by a user’s query

**actions:** Flight from current location with attributes such as seat class, departure time, etc.

**transition model:** The state resulting from taking a flight, including destination and arrival time.

**goal test:** At the final destination?

**path cost:**
Example: Airline Travel

**states:** Include locations (airports), current time.

- Also perhaps fares, domestic/international, and other “historical aspects”.

**initial state:** Given by a user’s query

**actions:** Flight from current location with attributes such as seat class, departure time, etc.

**transition model:** The state resulting from taking a flight, including destination and arrival time.

**goal test:** At the final destination?

**path cost:** Depends on total cost, time, waiting time, seat type, type of plane, etc.
Others Examples

How about:

- Crosswords?
- n-Queens?
- Propositional Satisfiability?
- Coffee and Mail Delivering Robot?
- Others?
Tree Search Algorithms

Basic idea:

- Offline exploration of the state space
  - So, exploring a *directed graph*
  - Result of exploration is a *tree*

- Generate successors of already-explored states
  (a.k.a. *expanding* states)

⇒ The set of nodes available for expansion is the *fringe* or *frontier*.

- Key issue: Which node should be expanded next?
Tree search example
Tree search example
Tree search example
Implementation: General Tree Search

In outline:

Function Tree-Search(problem) returns a solution or failure
   Initialize the search tree by 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 some strategy)
         - remove the leaf node from the frontier
      if the node satisfies the goal state then return the solution
      expand the node and add the resulting nodes to the search tree
   } 

 Aside: Strategy will most often be implicit in the resulting function.
Implementation: States vs. Nodes

It is important to distinguish the state space and the search tree.

- A state represents a configuration in the problem space.
- A node is part of a search tree.
  - has attributes parent, children, depth, path cost $g(x)$.

States do not have parents, children, depth, or path cost (though one state may be reachable from another).

An **Expand** function creates new nodes, filling in the various fields and using a **SuccessorFn** of the problem to create the corresponding states.
Search strategies

- A *strategy* is defined by picking the *order of node expansion*
- The *fringe* (also *frontier*) is a list of nodes that have been generated but not yet expanded.
Search strategies

- A *strategy* is defined by picking the *order of node expansion*.
- The *fringe* (also *frontier*) is a list of nodes that have been generated but not yet expanded.
- Strategies are evaluated along the following dimensions:
  - *completeness* – does it always find a solution if one exists?
  - *time complexity* – number of nodes generated/expanded
  - *space complexity* – maximum number of nodes in memory
  - *optimality* – does it always find a least-cost solution?

Time and space complexity are measured in terms of:
- $b$ – maximum branching factor
- $d$ – depth of the least-cost solution
- $m$ – maximum depth of the state space (may be $\infty$)
Search strategies

• A **strategy** is defined by picking the **order of node expansion**

• The **fringe** (also **frontier**) is a list of nodes that have been generated but not yet expanded.

• Strategies are evaluated along the following dimensions:
  - **completeness** – does it always find a solution if one exists?
  - **time complexity** – number of nodes generated/expanded

• Time and space complexity are measured in terms of:
  - **b** – maximum branching factor
  - **d** – depth of the least-cost solution
  - **m** – maximum depth of the state space (may be $\infty$)
Search strategies

• A strategy is defined by picking the order of node expansion.
• The fringe (also frontier) is a list of nodes that have been generated but not yet expanded.
• Strategies are evaluated along the following dimensions: completeness – does it always find a solution if one exists?
  time complexity – number of nodes generated/expanded
  space complexity – maximum number of nodes in memory
Search strategies

- A *strategy* is defined by picking the *order of node expansion*.
- The *fringe* (also *frontier*) is a list of nodes that have been generated but not yet expanded.
- Strategies are evaluated along the following dimensions:
  - *completeness* – does it always find a solution if one exists?
  - *time complexity* – number of nodes generated/expanded
  - *space complexity* – maximum number of nodes in memory
  - *optimality* – does it always find a least-cost solution?
Search strategies

- A **strategy** is defined by picking the **order of node expansion**
- The **fringe** (also **frontier**) is a list of nodes that have been generated but not yet expanded.
- Strategies are evaluated along the following dimensions:
  - **completeness** – does it always find a solution if one exists?
  - **time complexity** – number of nodes generated/expanded
  - **space complexity** – maximum number of nodes in memory
  - **optimality** – does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - \( b \) – maximum **branching factor** of the search tree
  - \( d \) – depth of the least-cost solution
  - \( m \) – maximum depth of the state space (may be \( \infty \))
Uninformed search strategies

- *Uninformed* strategies use only the information available in the problem definition.
- I.e. except for the goal state, there is no notion of one state being “better” than another.
- Examples:
Uninformed search strategies

- *Uninformed* strategies use only the information available in the problem definition
- I.e. except for the goal state, there is no notion of one state being “better” than another.
- Examples:
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search
Breadth-first search

Expand the shallowest unexpanded node

*Implementation*

*fringe* is a FIFO queue, i.e., new successors go at end

```
A
B C
D E F G
```
Breadth-first search

Expand the shallowest unexpanded node

*Implementation*

*f fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand the shallowest unexpanded node

*Implementation*

*fringe* is a FIFO queue, i.e., new successors go at end
Breadth-first search

Expand the shallowest unexpanded node

*Implementation*

*fringe* is a FIFO queue, i.e., new successors go at end
Properties of breadth-first search

Complete: ??
Properties of breadth-first search

Complete: Yes (if $b$ is finite)

Time: ??
Properties of breadth-first search

**Complete:** Yes (if $b$ is finite)

**Time:** $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

I.e., exponential in $d$

**Space:** ??
Properties of breadth-first search

Complete: Yes (if \( b \) is finite)

Time: \( 1 + b + b^2 + b^3 + \ldots + b^d = O(b^d) \)

I.e., exp. in \( d \)

Space: \( O(b^d) \) (keeps every node in memory)

Optimal: ??
Properties of breadth-first search

Complete: Yes (if $b$ is finite)

Time: $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$
I.e., exp. in $d$

Space: $O(b^d)$ (keeps every node in memory)

Optimal: Yes (if cost = 1 per step); not optimal in general

*Space* is the big problem; can easily generate nodes at 100MB/sec.
So 24hrs = 8640GB.
Uniform-Cost Search

- Expand the least-cost unexpanded node
- *Implementation*
  \[ \text{fringe} = \text{queue ordered by path cost, lowest first} \]
- Equivalent to breadth-first if step costs all equal
- For the travel-in-Romania example, expand the node on the fringe for that city closest in distance to the city at the root (Arad).
Uniform-Cost Search

Complete: Yes, if step cost $\geq \epsilon$, for $\epsilon$ some small positive constant.

- So NoOps of cost 0 can be a problem.

Time: $O\left(b^{\lceil C^*/\epsilon \rceil}\right)$, where $C^*$ is the cost of the optimal solution

Space: $O\left(b^{\lceil C^*/\epsilon \rceil}\right)$

- Time and space complexity can be worse than $b^d$.

Optimal: Yes

- Nodes expanded in increasing order of $g(n)$ where $g(n)$ is the cost to get to node $n$. 
Depth-First Search

Expand the deepest unexpanded node

Implementation

$fringe = \text{LIFO queue, i.e., put successors at front}$
Depth-first search

Expand the deepest unexpanded node

*Implementation*

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

**Implementation**

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

fringe = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

*fringe =* LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

*fringe* = LIFO queue, i.e., put successors at front
Depth-first search

Expand the deepest unexpanded node

*Implementation*

*fringe* = LIFO queue, i.e., put successors at front
Properties of depth-first search

Complete: ??
Properties of depth-first search

Complete: No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time: ??
Properties of depth-first search

**Complete:** No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

**Time:** $O(b^m)$: terrible if $m$ is much larger than $d$
  - But if solutions are dense, may be much faster than breadth-first

**Space:** ??
Properties of depth-first search

**Complete:** No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

**Time:** $O(b^m)$: terrible if $m$ is much larger than $d$
- But if solutions are dense, may be much faster than breadth-first

**Space:** $O(bm)$, i.e., linear space!

**Optimal:** ??
Properties of depth-first search

Complete: No: fails in infinite-depth spaces, spaces with loops
Modify to avoid repeated states along path
⇒ complete in finite spaces

Time: $O(b^m)$: terrible if $m$ is much larger than $d$
  • But if solutions are dense, may be much faster
    than breadth-first

Space: $O(bm)$, i.e., linear space!

Optimal: No
Depth-Limited Search

Depth-limited search = depth-first search with depth limit \( l \),
- i.e., nodes at depth \( l \) have no successors

Recursive implementation:
The implementation simply calls a “helper” function (described on the next slide):

Function \textbf{Depth-Limited-Search}(\textbf{problem},\textbf{limit})
\begin{verbatim}
    returns soln/fail/cutoff
    Recursive-DLS(Make-Node(Initial-State[problem]),
                   problem,limit)
\end{verbatim}
Depth-Limited Search

Recursive implementation:

Function **Recursive-DLS**(node, problem, limit) returns soln/fail/cutoff

- cutoff-occurred? ← false
- if Goal-Test(problem, State[node]) then return node
- else if Depth[node] = limit then return cutoff
- else for each successor in Expand(node, problem) do
  - result ← Recursive-DLS(successor, 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

• Note: second edition has a bug in the recursive call!
Iterative Deepening Search

Function **Iterative-Deepening-Search**(problem) returns a solution

inputs: problem a problem

for depth ← 0 to ∞ do
  result ← Depth-Limited-Search(problem, depth)
  if result ≠ cutoff then return result

end
Iterative deepening search $l = 0$
Iterative deepening search \( l = 1 \)
Iterative deepening search $l = 2$

Limit = 2

Diagram showing the iterative deepening search process with a limit of 2.
Iterative deepening search $l = 3$

Limit = 3
Properties of iterative deepening search

Complete: ??
Properties of iterative deepening search

Complete: Yes
Time: ??
Properties of iterative deepening search

Complete: Yes

Time: \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)

Space: ??
Properties of iterative deepening search

Complete: Yes

Time: \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)

Space: \(O(bd)\)

Optimal:
Properties of iterative deepening search

Complete: Yes

Time: \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)

Space: \(O(bd)\)

Optimal: Yes, if step cost = 1
Properties of iterative deepening search

- Comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  \[ N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \]
  \[ N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 111,100 \]

- For a large search space with unknown depth of solution, IDS is usually best.

- For BFS, we have the following ratio of IDS to BFS:

  \[
  \begin{array}{c|c}
  b & 2 \quad 3 \quad 5 \\
  2 & 1.5 & 1.2
  \end{array}
  \]
Properties of iterative deepening search

- Comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  \[ N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \]
  \[ N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 111,100 \]

- For a large search space with unknown depth of solution, IDS is usually best.
Properties of iterative deepening search

- Comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  \[ N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 + 100,000 = 123,450 \]
  \[ N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 111,100 \]

- For a large search space with unknown depth of solution, IDS is usually best.

- For BFS, we have the following ratio of IDS to BFS:

<table>
<thead>
<tr>
<th>$b$</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1.5</td>
</tr>
<tr>
<td>10</td>
<td>1.2</td>
</tr>
</tbody>
</table>
Properties of iterative deepening search

- Comparison for $b = 10$ and $d = 5$, solution at far right leaf:
  \[ N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450 \]
  \[ N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 111,100 \]

- For a large search space with unknown depth of solution, IDS is usually best.

- For BFS, we have the following ratio of IDS to BFS:

<table>
<thead>
<tr>
<th>$b$</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1.5</td>
</tr>
<tr>
<td>10</td>
<td>1.2</td>
</tr>
</tbody>
</table>

- Can be modified to explore uniform-cost tree
## Summary of algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$b^{d+1}$</td>
<td>$b^\lceil C^*/\epsilon \rceil$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</td>
<td>$b^\lceil C^*/\epsilon \rceil$</td>
<td>$b^m$</td>
<td>$b^l$</td>
<td>$bd$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>

*: If $b$ is finite.
Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!

- If we detect repeated states, then our search algorithm amounts to searching a graph rather than a tree.
  - Keep a list of encountered nodes, called the *closed* list.
Function $\text{Graph-Search}(\text{problem}, \text{fringe})$ returns a solution, or failure

closed $\leftarrow$ an empty set
fringe $\leftarrow$ $\text{Insert}(\text{Make-Node(Initial-State[problem])}, \text{fringe})$

loop do
    if fringe is empty then return failure

    node $\leftarrow$ $\text{Remove-Front}(\text{fringe})$

    if $\text{Goal-Test(}\text{problem}, \text{State[node]}\text{)}$ then return node

    if State[node] is not in closed then
        add State[node] to closed
        fringe $\leftarrow$ $\text{InsertAll}(\text{Expand(node,problem)}, \text{fringe})$
    end

end
Summary

- Problem formulation usually requires abstracting from real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- Graph search can be exponentially more efficient than tree search