Introduction to Artificial Intelligence

Problem Solving and Search

Bernhard Beckert

Universität Koblenz-Landau

Winter Term 2004/2005
Outline

- Problem solving
- Problem types
- Problem formulation
- Example problems
- Basic search algorithms
Problem solving

Offline problem solving
Acting only with complete knowledge of problem and solution

Online problem solving
Acting without complete knowledge

Here
Here we are concerned with offline problem solving only
Example: Travelling in Romania

Scenario

On holiday in Romania; currently in Arad
Flight leaves tomorrow from Bucharest
Example: Travelling in Romania

Scenario

On holiday in Romania; currently in Arad
Flight leaves tomorrow from Bucharest

Goal

Be in Bucharest
Example: Travelling in Romania

Scenario
On holiday in Romania; currently in Arad
Flight leaves tomorrow from Bucharest

Goal
Be in Bucharest

Formulate problem
States: various cities
Actions: drive between cities
Example: Travelling in Romania

Scenario

On holiday in Romania; currently in Arad
Flight leaves tomorrow from Bucharest

Goal

Be in Bucharest

Formulate problem

*States*: various cities
*Actions*: drive between cities

Solution

Appropriate sequence of cities
e.g.: Arad, Sibiu, Fagaras, Bucharest
Example: Travelling in Romania
Problem types

Single-state problem
- observable  (at least the initial state)
- deterministic
- static
- discrete

Multiple-state problem
- partially observable  (initial state not observable)
- deterministic
- static
- discrete

Contingency problem
- partially observable  (initial state not observable)
- non-deterministic
Example: vacuum-cleaner world

Single-state

Start in: 5

Solution:
Example: vacuum-cleaner world

**Single-state**

Start in: 5

**Solution:** \([right, suck]\)
Example: vacuum-cleaner world

Single-state

Start in: 5

Solution: \[ \text{right, suck} \]

Multiple-state

Start in: \{1, 2, 3, 4, 5, 6, 7, 8\}

Solution:
Example: vacuum-cleaner world

Single-state

Start in: 5

Solution: \([\text{right, suck}]\)

Multiple-state

Start in: \{1, 2, 3, 4, 5, 6, 7, 8\}

Solution: \([\text{right, suck, left, suck}]\)

- \text{right} \rightarrow \{2, 4, 6, 8\}
- \text{suck} \rightarrow \{4, 8\}
- \text{left} \rightarrow \{3, 7\}
- \text{suck} \rightarrow \{7\}
Example: vacuum-cleaner world

Contingency

Murphy’s Law: 
*suck* can dirty a clean carpet

Local sensing: 
*dirty/not dirty* at location only

Start in: \( \{1, 3\} \)

Solution:
Example: vacuum-cleaner world

Contingency

Murphy’s Law: *suck* can dirty a clean carpet

Local sensing: *dirty/not dirty* at location only

Start in: \{1, 3\}

Solution: \[suck, right, suck\]

- *suck* \(\rightarrow\) \{5, 7\}
- *right* \(\rightarrow\) \{6, 8\}
- *suck* \(\rightarrow\) \{6, 8\}

Improvement: \[suck, right, if \text{dirt} \text{then} suck\] (decide whether in 6 or 8 using local sensing)
Single-state problem formulation

Defined by the following four items

1. Initial state
   Example: \textit{Arad}

2. Successor function $S$
   Example: $S(\textit{Arad}) = \{ \langle \text{goZerind}, \text{Zerind} \rangle, \langle \text{goSibiu}, \text{Sibiu} \rangle, \ldots \}$

3. Goal test
   Example: $x = \textit{Bucharest}$ (explicit test)
   $\textit{noDirt}(x)$ (implicit test)

4. Path cost (optional)
   Example: sum of distances, number of operators executed, etc.
Single-state problem formulation

Solution

A sequence of operators
leading from the initial state to a goal state
Selecting a state space

Abstraction

Real world is absurdly complex
State space must be abstracted for problem solving

(Abstract) state

Set of real states

(Abstract) operator

Complex combination of real actions
Example: Arad → Zerind represents complex set of possible routes

(Abstract) solution

Set of real paths that are solutions in the real world
Example: The 8-puzzle

States

Actions

Goal test

Path cost
Example: The 8-puzzle

States: integer locations of tiles

Actions

Goal test

Path cost
Example: The 8-puzzle

States: integer locations of tiles

Actions: \textit{left, right, up, down}

Goal test

Path cost
Example: The 8-puzzle

States: integer locations of tiles
Actions: left, right, up, down
Goal test: = goal state?
Path cost

Start State

Goal State
Example: The 8-puzzle

**States**
integer locations of tiles

**Actions**
*left, right, up, down*

**Goal test**
= goal state?

**Path cost**
1 per move
Example: Vacuum-cleaner

States

Actions

Goal test

Path cost
Example: Vacuum-cleaner

States  integer dirt and robot locations

Actions

Goal test

Path cost
Example: Vacuum-cleaner

States  integer dirt and robot locations

Actions  \textit{left}, \textit{right}, \textit{suck}, \textit{noOp}

Goal test

Path cost
Example: Vacuum-cleaner

**States** integer dirt and robot locations

**Actions** *left, right, suck, noOp*

**Goal test** *not dirty?*

**Path cost**
Example: Vacuum-cleaner

**States**  integer dirt and robot locations

**Actions**  *left, right, suck, noOp*

**Goal test**  *not dirty?*

**Path cost**  1 per operation  (0 for *noOp*)
Example: Robotic assembly

States

Actions

Goal test

Path cost
Example: Robotic assembly

States  real-valued coordinates of
robot joint angles and parts of the object to be assembled

Actions

Goal test

Path cost
Example: Robotic assembly

States: real-valued coordinates of robot joint angles and parts of the object to be assembled

Actions: continuous motions of robot joints

Goal test

Path cost
Example: Robotic assembly

**States**
real-valued coordinates of
robot joint angles and parts of the object to be assembled

**Actions**
continuous motions of robot joints

**Goal test**
assembly complete?

**Path cost**
Example: Robotic assembly

**States**
- real-valued coordinates of robot joint angles and parts of the object to be assembled

**Actions**
- continuous motions of robot joints

**Goal test**
- assembly complete?

**Path cost**
- time to execute
Tree search algorithms

- Offline
- Simulated exploration of state space in a search tree by generating successors of already-explored states

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

B. Beckert: KI für IM – p.15
Tree search: Example

[Diagram of a tree search example with cities such as Arad, Sibiu, Timisoara, Zerind, and Oradea.]
Tree search: Example
Tree search: Example
Implementation: States vs. nodes

State
A (representation of) a physical configuration

Node
A data structure constituting part of a search tree (includes parent, children, depth, path cost, etc.)
Implementation of search algorithms

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-First(fringe)
  if Goal-Test[problem] applied to State(node) succeeds then
    return node
  else
    fringe ← Insert-All(Expand(node, problem), fringe)
  end
end

fringe queue of nodes not yet considered
State gives the state that is represented by node
Expand creates new nodes by applying possible actions to node
Search strategies

Strategy

Defines the order of node expansion

Important properties of strategies

- **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 measured in terms of

- $b$: maximum branching factor of the search tree
- $d$: depth of a solution with minimal distance to root
- $m$: maximum depth of the state space (may be $\infty$)
Uninformed search strategies

Uninformed search

Use only the information available in the problem definition

Frequently used strategies

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
Breadth-first search

Idea

Expand shallowest unexpanded node

Implementation

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

Idea
Expand shallowest unexpanded node

Implementation

fringe is a FIFO queue, i.e. successors go in at the end of the queue
Breadth-first search

Idea

Expand shallowest unexpanded node

Implementation

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

Idea

Expand shallowest unexpanded node

Implementation

fringe is a FIFO queue, i.e. successors go in at the end of the queue
Breadth-first search: Example Romania
Breadth-first search: Example Romania
Breadth-first search: Example Romania

- Arad
- Oradea
- Zerind
- Sibiu
- Timisoara

Arad

Sibiu

Timisoara

Zerind

Oradea

Arad
Breadth-first search: Example Romania
Breadth-first search: Properties

Complete

Time

Space

Optimal
**Breadth-first search: Properties**

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

**Time**

**Space**

**Optimal**
Breadth-first search: Properties

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

**Time**

$$1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) \in O(b^{d+1})$$

i.e. exponential in $d$

**Space**

**Optimal**

Yes (if cost = 1 per step), not optimal in general

Disadvantage

Space is the big problem (can easily generate nodes at 5MB/sec so 24hrs = 430GB)
# Breadth-first search: Properties

| **Complete** | Yes  
( if \( b \) is finite) |
|--------------|-------------------------------|
| **Time**     | \[ 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) \in O(b^{d+1}) \]  
i.e. exponential in \( d \) |
| **Space**    | \( O(b^{d+1}) \)  
keeps every node in memory |
| **Optimal**  |                               |
## Breadth-first search: Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes (if ( b ) is finite)</td>
</tr>
<tr>
<td>Time</td>
<td>( 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) \in O(b^{d+1}) )</td>
</tr>
<tr>
<td></td>
<td>i.e. exponential in ( d )</td>
</tr>
<tr>
<td>Space</td>
<td>( O(b^{d+1}) )</td>
</tr>
<tr>
<td></td>
<td>keeps every node in memory</td>
</tr>
<tr>
<td>Optimal</td>
<td>Yes (if cost = 1 per step), not optimal in general</td>
</tr>
</tbody>
</table>
Breadth-first search: Properties

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

**Time**  
\[1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) \in O(b^{d+1})\]

i.e. exponential in $d$

**Space**  
\[O(b^{d+1})\]

keeps every node in memory

**Optimal**  Yes (if cost = 1 per step), not optimal in general

**Disadvantage**

Space is the big problem
(can easily generate nodes at 5MB/sec so 24hrs = 430GB)
Romania with step costs in km

Straight-line distance to Bucharest

- Arad: 366 km
- Bucharest: 0 km
- Craiova: 160 km
- Dobrogea: 242 km
- Eforie: 161 km
- Fagaras: 178 km
- Giurgiu: 77 km
- Hirsova: 151 km
- Iasi: 226 km
- Lugoj: 244 km
- Mehadia: 241 km
- Neamt: 234 km
- Oradea: 380 km
- Pitesti: 98 km
- Rimnicu Vilcea: 193 km
- Sibiu: 253 km
- Timisoara: 329 km
- Urziceni: 80 km
- Vaslui: 199 km
- Zerind: 374 km
Uniform-cost search

Idea

Expand least-cost unexpanded node
(costs added up over paths from root to leafs)

Implementation

*fringe* is queue ordered by increasing path cost

Note

Equivalent to depth-first search if all step costs are equal
Uniform-cost search
Uniform-cost search
Uniform-cost search
Uniform-cost search
Uniform-cost search: Properties

- Complete
- Time
- Space
- Optimal
Uniform-cost search: Properties

**Complete**  Yes  (if step costs positive)

**Time**

**Space**

**Optimal**
Uniform-cost search: Properties

Complete  Yes  (if step costs positive)

Time  # of nodes with past-cost less than that of optimal solution

Space

Optimal
Uniform-cost search: Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes (if step costs positive)</td>
</tr>
<tr>
<td>Time</td>
<td># of nodes with past-cost less than that of optimal solution</td>
</tr>
<tr>
<td>Space</td>
<td># of nodes with past-cost less than that of optimal solution</td>
</tr>
<tr>
<td>Optimal</td>
<td></td>
</tr>
</tbody>
</table>
# Uniform-cost search: Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes (if step costs positive)</td>
</tr>
<tr>
<td>Time</td>
<td># of nodes with past-cost less than that of optimal solution</td>
</tr>
<tr>
<td>Space</td>
<td># of nodes with past-cost less than that of optimal solution</td>
</tr>
<tr>
<td>Optimal</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Depth-first search

Idea

Expand deepest unexpanded node

Implementation

*fringe* is a LIFO queue (a stack), i.e. successors go in at front of queue

Note

Depth-first search can perform infinite cyclic excursions

Need a finite, non-cyclic search space (or repeated-state checking)
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search
Depth-first search: Example Romania

Arad
Depth-first search: Example Romania
Depth-first search: Example Romania
Depth-first search: Example Romania
Depth-first search: Properties

**Complete**

**Time**

**Space**

**Optimal**
Depth-first search: Properties

**Complete**
- Yes: if state space finite
- No: if state contains infinite paths or loops

**Time**

**Space**

**Optimal**
Depth-first search: Properties

**Complete**
- Yes: if state space finite
- No: if state contains infinite paths or loops

**Time**
$O(b^m)$

**Space**

**Optimal**
No
## Depth-first search: Properties

| **Complete** | Yes: if state space finite  
No: if state contains infinite paths or loops |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td>$O(b^m)$</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>$O(bm)$ (i.e. linear space)</td>
</tr>
<tr>
<td><strong>Optimal</strong></td>
<td>No</td>
</tr>
</tbody>
</table>
# Depth-first search: Properties

| **Complete** | Yes: if state space finite  
|              | No: if state contains infinite paths or loops |
| **Time**     | $O(b^m)$ |
| **Space**    | $O(bm)$ (i.e. linear space) |
| **Optimal**  | No |
Depth-first search: Properties

**Complete**
- Yes: if state space finite
- No: if state contains infinite paths or loops

**Time**
- $O(b^m)$

**Space**
- $O(bm)$ (i.e. linear space)

**Optimal**
- No

**Disadvantage**
- Time terrible if $m$ much larger than $d$

**Advantage**
- Time may be much less than breadth-first search if solutions are dense
Iterative deepening search

Depth-limited search

Depth-first search with depth limit
Iterative deepening search

Depth-limited search

Depth-first search with depth limit

Iterative deepening search

Depth-limit search with ever increasing limits

```plaintext
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
end
```
Iterative deepening search with depth limit 0

Limit = 0
Iterative deepening search with depth limit

Limit = 1

B. Beckert: KI für IM – p.34
Iterative deepening search with depth limit 2

Limit = 2

Diagram showing iterative deepening search with depth limit 2.
Iterative deepening search with depth limit 3

Limit = 3

B. Beckert: KI für IM – p.36
Iterative deepening search: Example Romania with $l = 0$
Iterative deepening search: Example Romania with $l = 1$
Iterative deepening search: Example Romania with $l = 1$
Iterative deepening search: Example Romania with $l = 2$
Iterative deepening search: Example Romania with $l = 2$
Iterative deepening search: Example Romania with $l = 2$
Iterative deepening search: Example Romania with $l = 2$
Iterative deepening search: Example Romania with $l = 2$
Iterative deepening search: Properties

- **Complete**

- **Time**

- **Space**

- **Optimal**

Optimal (if step cost = 1)
Iterative deepening search: Properties

Complete: Yes

Time: \(d + 1 + \sum_{i=0}^{d} b + O(bd + 1)\)

Space: \(O(bd)\)

Optimal: Yes (if step cost = 1)

(Depth-First) Iterative-Deepening Search often used in practice for search spaces of large, infinite, or unknown depth.

B. Beckert: KI für IM – p.40
## Iterative deepening search: Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Complete</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>$(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d \in O(b^{d+1})$</td>
</tr>
<tr>
<td><strong>Space</strong></td>
<td>$O(bd)$</td>
</tr>
<tr>
<td><strong>Optimal</strong></td>
<td>(if step cost = 1)</td>
</tr>
</tbody>
</table>

(Depth-First) Iterative-Deepening Search often used in practice for search spaces of large, infinite, or unknown depth.
Iterative deepening search: Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d \in O(b^{d+1})$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(bd)$</td>
</tr>
<tr>
<td>Optimal</td>
<td>(if step cost = 1)</td>
</tr>
</tbody>
</table>

(B. Beckert: KI für IM – p.40)
Iterative deepening search: Properties

**Complete**  Yes

**Time**  $$(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d \in O(b^{d+1})$$

**Space**  $O(bd)$

**Optimal**  Yes  (if step cost = 1)
Iterative deepening search: Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$(d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d \in O(b^{d+1})$</td>
</tr>
<tr>
<td>Space</td>
<td>$O(b^d)$</td>
</tr>
<tr>
<td>Optimal</td>
<td>Yes (if step cost = 1)</td>
</tr>
</tbody>
</table>

(Depth-First) Iterative-Deepening Search often used in practice for search spaces of large, infinite, or unknown depth.
### Comparison

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-first</th>
<th>Uniform-cost</th>
<th>Depth-first</th>
<th>Iterative deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>$b^{d+1}$</td>
<td>$\approx b^d$</td>
<td>$b^m$</td>
<td>$b^d$</td>
</tr>
<tr>
<td>Space</td>
<td>$b^{d+1}$</td>
<td>$\approx b^d$</td>
<td>$b^m$</td>
<td>$bd$</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

B. Beckert: KI für IM – p.41
Comparison

Breadth-first search

Iterative deepening search
Summary

- Problem formulation usually requires abstracting away 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.