Einführung in die Künstliche Intelligenz

Uninformierte Suche

- Prof. Michael Schroeder
- Dr. Daniel Eisinger
- Oliver Groth

thanks to Prof. Fürnkranz, Darmstadt for slide templates
Lehrbuch

- Folien basieren auf

Russell und Norvig:
*Künstliche Intelligenz: Ein Moderner Ansatz.*
http://aima.cs.berkeley.edu/
Motivation

Graphs and networks play an important role in AI

Facebook: social network
Dbpedia: graph over millions of subject predicate object triples
Google: the web
GoPubMed: hierarchy of biomedical concepts
IBM deep blue: chess game playing as tree
Navigation of autonomous vehicles: Road network
NASA Deep Space One: Network of system components

How to search over these networks?
Uninformed Search

- Tree search algorithms
  - Breadth-First Search
  - Depth-First Search
  - Limited-Depth Search
  - Iterative Deepening
- Extensions
  - Graph search algorithms
  - Search with Partial Information
Example: Navigate in Romania

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

Formulate goal:
- be in Bucharest

Formulate problem:
- states: various cities
- actions: drive between cities

Find solution:
- sequence of cities, e.g., Arad, Sibiu, Rimnicu Vilcea, Pitesti
Example: Romania
Single-state Problem Formulation

A problem is defined by four items:

- **initial state**
  - e.g., "at Arad"

- **description of actions and their effects**
  - typically as a *successor function* that maps a state $s$ to a set $S(s)$ of action-state pairs
  - e.g., $S(\text{"at Arad"}) = \{<\text{"goto Zerind"}, \text{"at Zerind"}>, \ldots \}$

- **goal test**, can be
  - explicit, e.g., $s =$ "at Bucharest"
  - implicit, e.g., $\text{Checkmate}(s), \text{NoDirt}(s)$

- **path cost** (additive)
  - e.g., sum of distances, number of actions executed, etc.
  - $c(s_1, a, s_2)$ are the costs for one step (one action),
  - assumed to be $\geq 0$
Single-State Problems

Yes
- 8-queens puzzle
- 8-puzzle
- Towers of Hanoi
- Cross-Word puzzles
- Sudoku
- Chess, Bridge, Scrabble puzzles
- Rubik's cube
- Sobokan
- Traveling Salesman Problem

No
- Tetris
  - dynamic not static
- Solitaire
  - only partially observable
State Space of a Problem

- **State Space**
  - the set of all states reachable from the initial state
  - implicitly defined by the initial state and the successor function

![Diagram of State Space](image-url)
State Space of a Problem

- **State Space**
  - the set of all states reachable from the initial state
  - implicitly defined by the initial state and the successor function

- **Path**
  - a sequence of states connected by a sequence of actions

- **Solution**
  - a path that leads from the initial state to a goal state

- **Optimal Solution**
  - solution with the minimum path cost
Example: Romania

1. Head south on 79E671 0.7 km
2. Turn left toward 7/E68 3.8 km
3. Slight left at 7/E68 (signs for E68/DEVA) 231 km
4. Turn right at 1/E68/E81 0.5 km
5. Slight left to stay on 1/E68/E81 32.2 km
6. Turn right at 1/E68 (signs for BRAȘOV/RM. VALCEA) 1.7 km
7. Turn left (signs for BRAȘOV/RM. VALCEA) 0.7 km
8. Turn left toward 7/E81 (signs for BRAȘOV/RM. VALCEA) 2.3 km
9. Turn right at 7/E81 (signs for BRAȘOV/RM. VALCEA) 72.3 km
10. Turn left to stay on 7/E81 6.2 km
11. Turn left at E81 84.3 km
12. Turn left at 65B/E81 1.9 km
13. Turn left at E70/E81 104 km
Selecting a State Space

Real world is absurdly complex

→ state space must be abstracted for problem solving

- (Abstract) state
  - corresponds to a set of real states

- (Abstract) action
  - corresponds to a complex combination of real actions
  - e.g., "go from Arad to Zerind" represents a complex set of possible routes, detours, rest stops, etc.
  - for guaranteed realizability, any real state "in Arad" must get to some real state "in Zerind"
  - each abstract action should be "easier" than the original problem

- (Abstract) solution
  - corresponds to a set of real paths that are solutions in the real world
Example: Romania – State Space
Example: The 8-puzzle

- states?
  - location of tiles
    - ignore intermediate positions during sliding
- goal test?
  - situation corresponds to goal state
- path cost?
  - number of steps in path (each step costs 1)
- actions?
  - move blank tile (left, right, up, down)
    - easier than having separate moves for each tile
  - ignore actions like unjamming slides if they get stuck
Example: The 8-Queens Problem

- states?
  - any configuration of 8 queens on the board
- goal test?
  - no pair of queens can capture each other
- actions?
  - move one of the queens to another square
- path cost?
  - not of interest here

inefficient complete-state formulation

→ $64 \cdot 63 \cdot \ldots \cdot 57 \approx 3 \cdot 10^{14}$ states
Example: The 8-Queens Problem

- states?
  - \(n\) non-attacking queens in the left \(n\) columns
- goal test?
  - no pair of queens can capture each other
- actions?
  - add queen in column \(n + 1\)
  - without attacking the others
- path cost?
  - not of interest here

more efficient incremental formulation → only 2057 states
Tree Search Algorithms

- Treat the state-space graph as a tree
- Expanding a node
  - offline, simulated exploration of state space by generating successors of already-explored states (successor function)
- Search strategy
  - determines which node is expanded next
- General algorithm:

```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
```
Tree Search Example

- **Initial state**: start with node *Arad*
Tree Search Example

- Initial state: start with node \textit{Arad}
- expand node \textit{Arad}
Tree Search Example

- Initial state: start with node Arad
- expand node Arad
- expand node Sibiu
States vs. Nodes

- **State**
  - (representation of) a physical configuration

- **Node**
  - data structure constituting part of a search tree
  - includes
    - state
    - parent node
    - action
    - path cost $g(x)$
    - depth

- **Expand**
  - creates new nodes
  - fills in the various fields
  - uses the successor function to create the corresponding states
Implementation: General Tree Search

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 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
Search Strategies

- A search strategy is defined by picking the order of node expansion
  - implementation in a queue

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - 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 $\infty$)
Search Strategies

- **Uninformed** (blind) search strategies use only the information available in the problem definition
  - Breadth-first search
  - Uniform-cost search
  - Depth-first search
  - Depth-limited search
  - Iterative deepening search

- **Informed** (heuristic) search strategies have knowledge that allows to guide the search to promising regions
  - Greedy Search
  - A* Best-First Search
Breadth-First Strategy

- Expand all neighbors of a node (breadth) before any of its successors is expanded (depth)

Implementation:
- expand the shallowest unexpanded node
- fringe is a FIFO queue (first-in-first-out, new nodes go to end of queue)
Breadth-First Strategy

- Expand all neighbors of a node (breadth) before any of its successors is expanded (depth)

Implementation:
- expand the shallowest unexpanded node
- fringe is a FIFO queue (first-in-first-out, new nodes go to end of queue)
Breadth-First Strategy

- Expand all neighbors of a node (breadth) before any of its successors is expanded (depth)

Implementation:
- expand the shallowest unexpanded node
- fringe is a FIFO queue (first-in-first-out, new nodes go to end of queue)
Breadth-First Strategy

- Expand all neighbors of a node (breadth) before any of its successors is expanded (depth)

Implementation:
- expand the shallowest unexpanded node
- fringe is a FIFO queue (first-in-first-out, new nodes go to end of queue)
Properties of Breadth-First Search

- **Completeness**
  - Yes (if $b$ is finite)

- **Time Complexity**
  - each depth has $b$ times as many nodes as the previous
  - each node is expanded
  - except the goal node in level $d$
    - worst case: goal is last node in this level
      $$\Rightarrow 1 + b + b^2 + b^3 + \ldots + b^d = O\left(b^d\right)$$

- **Space Complexity**
  - every node must remain in memory
    - it is either a fringe node or an ancestor of a fringe node
    - in the end, the goal will be in the fringe, and its ancestors will be needed for the solution path
  $$\Rightarrow O(b^d)$$

- **Optimality**
  - Yes, for uniform costs (e.g., if cost = 1 per step)
Combinatorial Explosion

- Breadth-first search
  - branching factor $b = 10$, 1 million nodes/sec, 1000 bytes/node

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
<td>.11 ms</td>
<td>107 KB</td>
</tr>
<tr>
<td>4</td>
<td>11,100</td>
<td>11 ms</td>
<td>10.6 MB</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>1.1 s</td>
<td>1 GB</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>2 min</td>
<td>103 GB</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>3 h</td>
<td>10 TB</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>13 days</td>
<td>1 PetaBytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3.5 years</td>
<td>99 PetaBytes</td>
</tr>
<tr>
<td>16</td>
<td>$10^{16}$</td>
<td>350 years</td>
<td>10 ExaBytes</td>
</tr>
</tbody>
</table>

- Space is the bigger problem
  - can easily generate nodes at 100MB/sec $\Rightarrow$ 24hrs = 8640 GB
Uniform-Cost Search

- Breadth-first search can be generalized to cost functions
  - each node now has associated costs
  - costs accumulate over path
  - instead of expanding the shallowest path, expand the least-cost unexpanded node
  - breadth-first is special case where all costs are equal

- Implementation
  - fringe = queue ordered by path cost

- Completeness
  - yes, if each step has a positive cost (cost ≥ \( \varepsilon \))
  - otherwise infinite loops are possible

- Space and Time complexity
  \[ b^{1+O\left( \frac{c^*}{\varepsilon} \right)} \]
  - number of nodes with costs < costs of optimal solution \( C^* \)

- Optimality
  - Yes – nodes expanded in increasing order of path costs
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)

**Implementation:**
- expand the deepest unexpanded node
- fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)

![Diagram of a tree structure starting with node A, followed by B, C, D, E, F, G, H, I, J, K, L, M, N, O.]
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)

Implementation:
- expand the deepest unexpanded node
- fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implementation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)

- Implementation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)

Implementation:
- expand the deepest unexpanded node
- fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implementation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implementation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)

- Implementation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implementation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implementation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)
- Implementation:
  - expand the deepest unexpanded node
  - fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Depth-First Strategy

- Expand all successors of a node (depth) before any of its neighbors is expanded (breadth)

Implementation:
- expand the deepest unexpanded node
- fringe is a LIFO queue (last-in-first-out, new nodes at begin of queue)
Properties of Depth-First Search

- **Completeness**
  - No, fails in infinite-depth search spaces and spaces with loops
  - Complete in finite spaces if modified so that repeated states are avoided

- **Time Complexity**
  - Has to explore each branch until maximum depth $m$ $O(b^m)$
  - Terrible if $m > d$ (depth of goal node)
  - But may be faster than breadth-first if solutions are dense

- **Space Complexity**
  - Only nodes in current path and their unexpanded siblings need to be stored
  - $\Rightarrow$ only linear complexity $O(m \times b)$

- **Optimality**
  - No, longer (more expensive) solutions may be found before shorter (cheaper) ones
Backtracking Search

Even more space-efficient variant

- does not store all expanded nodes, but only the current path
  ⇒ $O(m)$
  - if no further expansion is possible, go back to the predecessor
  - each node is able to generate the *next* successor
- only needs to store and modify one state
  - actions can do and undo changes on this one state
Depth-limited Search

- depth-first search is provided with a depth limit \( l \)
  - nodes with depths \( d > l \) are not considered → incomplete
  - if \( d < l \) it is not optimal (like depth-first search)
  - time complexity \( O(b^l) \), space complexity \( O(bl) \)

```plaintext
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 node
    else if Depth[node] = limit then return cutoff
    else for each successor in Expand(node, problem) do
        result ← Recursive-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```
Iterative Deepening Search

- Main problem with depth-limited search is setting of \( l \)

- Simple solution:
  - try all possible \( l = 0, 1, 2, 3, \ldots \)

```plaintext
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
```

- costs are dominated by the last iteration, thus the overhead is marginal
Iterative Deepening Search

Limit = 0

Limit = 1

Limit = 2
Iterative Deepening Search

Limit = 3
Properties of Iterative Deepening Search

- **Completeness**
  - Yes (no infinite paths)

- **Time Complexity**
  - first level has to be searched $d$ times
  - last level has to be searched once
  
  $d \times b + (d - 1) b^2 + \ldots + 1 \times b^d = \sum_{i=1}^{d} (d - i + 1) \times b^i$

- **Space Complexity**
  - only linear complexity $O(bd)$

- **Optimality**
  - Yes, the solution is found at the minimum depth

⇒ combines advantages of depth-first and breadth-first search
Comparison of Time Complexities

Worst-case (goal is in right-most node at level $d$)

- Depth-Limited Search
  \[ N_{DLS} = b + b^2 + \ldots + b^d = \sum_{i=1}^{d} b^i \]

- Iterative Deepening
  \[ N_{IDS} = d \times b + (d - 1) b^2 + \ldots + 1 \times b^d = \sum_{i=1}^{d} (d - i + 1) b^i \]

Example: $b = 10$, $d = 5$

\[
\begin{align*}
N_{DLS} &= 10 + 100 + 1000 + 10,000 + 100,000 = 111,110 \\
N_{IDS} &= 50 + 400 + 3000 + 20,000 + 100,000 = 123,450 \\
\end{align*}
\]

\{ Overhead of IDS only ca. 10% \}
Bidirectional Search

- Perform two searches simultaneously
  - forward starting with initial state
  - backward starting with goal state
- check whether generated node is in fringe of the other search

Properties
- reduction in complexity \((b^{d/2} + b^{d/2} \ll b^d)\)
- only possible if actions can be reversed
- search paths may not meet for depth-first bidirectional search
Summary of Algorithms

- 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.

<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, if ( l \geq d )</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>( b^d )</td>
<td>( b^{C^*/\epsilon} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Space</td>
<td>( b^d )</td>
<td>( b^{C^*/\epsilon} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>
Repeated States

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

Ribbon Example

- two connections from each state to the next

\[ d \text{ states} \quad \text{but state space is } 2^d \]
Repeated States

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

(more realistic) **Grid Example**

- each square on grid has 4 neighboring states
- thus, game tree with repetitions has $4^d$ nodes
- but only about $2d^2$ different states are reachable in $d$ steps
Graph Search

- remembers the states that have been visited in a list \textit{closed}
- Note: the fringe list is often also called the \textit{open list}

```
function \textsc{Graph-Search}( problem, fringe) returns a solution, or failure

\textit{closed} \leftarrow \text{an empty set}

fringe \leftarrow \text{Insert(Make-Node(Initial-State[problem]), fringe)}

\textbf{loop do}

\hspace{1em} \textbf{if fringe is empty then return failure}

\hspace{1em} \textbf{node} \leftarrow \text{Remove-Front(fringe)}

\hspace{1em} \textbf{if Goal-Test(problem, State[node]) then return node}

\hspace{1em} \textbf{if State[node] is not in closed then}

\hspace{1em} \hspace{1em} \textbf{add State[node] to closed}

\hspace{1em} \hspace{1em} \textbf{fringe} \leftarrow \text{InsertAll(Expand(node, problem), fringe)}

\textbf{end}
```

- Example:
  - \textbf{Dijkstra's algorithm} is the graph-search variant of uniform cost search