CogSysI Lecture 2: Basic Aspects of Action Planning

Intelligent Agents
WS 2004/2005

Part I: Acting Goal-Oriented

I.2 Basic Definitions, Uninformed Search
Types of Problems

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Deficit</th>
<th>Problem Type</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial,</td>
<td>incomplete/fuzzy</td>
<td>ill-defined</td>
<td>McCarthy (1956)</td>
</tr>
<tr>
<td>Final State</td>
<td>representation</td>
<td>open</td>
<td>Minsky (1965)</td>
</tr>
<tr>
<td>Problem states</td>
<td>relations between states</td>
<td>Transformation problem</td>
<td>Greeno (1978)</td>
</tr>
<tr>
<td></td>
<td>unknown</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Operators</td>
<td>unknown sequence</td>
<td>Interpolation problem</td>
<td>Greeno (1976)</td>
</tr>
<tr>
<td></td>
<td>not available</td>
<td>Synthesis problem</td>
<td>Klix (1971)</td>
</tr>
</tbody>
</table>

Further criteria for classification:

- Complexity (typical AI problems are NP-hard)
- Structure of a problem: decomposable (independent subproblems; commutativity of operators, ...)

CogSysI Lecture 2: Basic Aspects of Action Planning – p. 36
Examples for Problems

- **Open Problem**: Achieve high quality of life
- **Transformation Problem**: Anagram
- **Interpolation Problem**: Towers of Hanoi, Rewrite logical formulars
- **Synthesis Problem**: mathematical proofs, numerical series, nine dots problem

→ classical domain of problem solving in AI: transformation and interpolation problems ("puzzles")
Example Problems

Anagram: centilegelin
intelligence

Series of numbers: 1 1 2 3 5 8 ...
13

Cryptarithmetic

```
CROSS
+ROADS
DANGER
```

96233
+62513
158746

Nine dots

```
O O O O
O O O O
O O O O
```

Hobbits and Orcs

```
left bank
O O O O O
H H H H H

right bank
O O O O O
H H H H H
```

Boat

CogSysI Lecture 2: Basic Aspects of Action Planning – p. 38
Basic Definitions

Problem: \( P = (S_i, S_e, O) \) with

- \( S_i \): Set of initial states,
- \( S_e \): Set of final/goal states,
- \( O \): Set of operator/actions

Problem space: Set of states \( S \) with \( S_i \subset S \), \( S_e \subset S \)

Operator: \( o \in O \) with \( O : S' \rightarrow S, S' \subset S \)

\( \leftarrow \) Operators are typically partial, they have application conditions and cannot be applied in any state in \( S \)

Problem solving: Transformation of an initial state into a final state by applying a (minimal) sequence of operators.

Minimal: least number of steps, minimal costs

Problem solving can be performed in a real environment (acting human or robot) or in a simulation (mental process, manipulation of descriptions)
Problem Representation

- **Representation of the problem**: formal description of states and operators. States are typically represented partially (not all attributes and relations of the real world states). Good representation is crucial for performance!

- **In problem solving**: problem specific representations, knowledge about the problem structure (heuristic function) can be incorporated. Typically, operators are defined without variables (simple rewrite rules).

- **In planning**: states as conjunctions of literals (subset of 1st order logic), generic representation of a domain as set of operator *schemes*, instantiated operators (called actions) are applied to states (see special chapter on planning).
Representation of States and Operators

- **States** can be represented as:
  - Arbitrary data structures: Tower of Hanoi states as list of lists \(((1,2,3),(),())\)
  - Sets (conjunctions) of facts: \{on(A, B), on(B, C), cleartop(A), ontable(C)\}

- **Operators** are represented as production rules:
  - IF \langle condition \rangle THEN \langle action \rangle
  - \langle condition \rangle \rightarrow \langle action \rangle

- The *effect* of an action must be specified somehow: How is the given state changed into a successor state by operator application?
Coffee Can Problem

(Gries, 1981)

- **States** \( S = \{ s \mid s \in \{ W, B \}^* \} \)
  \( S_i = \{ s_i \} \) with \( s_i = WWBBWWBB \)
  \( S_e = \{ s_e \} \) with \( s_e = W \)

- **Operators**: \( O = \{ BW \rightarrow B, WB \rightarrow B, BB \rightarrow W \} \)

The left-hand side represents the application condition (e.g. “BW must appear as substring in \( s \)”), the right-hand side represents the action (e.g. “replace BW by B”)

```
W W B B W W B W
W W B B W W B
W W B W W B W
W W B W W B
W W B W B
W B W B
W B B
W B
```

CogSysI Lecture 2: Basic Aspects of Action Planning – p. 42
Tower of Hanoi

States: List of three lists, representing pegs A, B, C; discs represented as natural numbers (1, 2, 3) which correspond to the size of the discs. The three discs can be arbitrarily distributed over the three lists, such that the following constraint holds: $d_i$ is in front of $d_j \rightarrow d_i < d_j$.

Initial State: ((123)()()), Final State: (()()(123))

Operators:

- $((123)()()) \rightarrow ((23)(1)())$
- $((123)()()) \rightarrow ((23)()(1))$
- $((23)(1)()) \rightarrow ((3)(1)(2))$
- $((23)(1)()) \rightarrow ((123)()())$ cycles may appear in the solution!
Blocksworld Problems

Objects: blocks
Attributes (1-ary relations): `cleartop(x)`, `ontable(x)`
Relations: `on (x, y)`
Operators: `puttable(x)` where `x` must be cleartop; `put(x, y)`, where `x` and `y` must be cleartop, `x` might be ontable or on another block `x`. 
Representing Blocksworld Problems

- States as set (conjunction) of facts:
  e.g. \( s_i = \) 
  \{on(C,A), cleatop(B), cleartop(C), ontable(A), ontable(B)\}

- Operators as simple production rules:
  put(A, B): \{ontable(A), ontable(B), ontable(C), cleartop(A), cleartop(B), cleartop(C)\} 
  \rightarrow \{on(A,B), ontable(B), ontable(C), cleartop(A), cleartop(C)\}
  put(B, A): ... 
  puttable(A): ...

  \leftrightarrow \text{very inconvenient!}

Instead of replacing complete state-descriptions, manipulate the description of the current state to construct the description of the successor state!
Operator Schemes

Application condition PRE (precondition),
action effect as ADD/DEL lists

:action put (?X ?Y)
:ADD ((on ?X ?Y))
:DEL ((cleartop ?Y) (ontable ?X))

:action put (?X ?Y)

:action puttable (?X)
:precondition ((cleartop ?X) (on ?X ?Y))
:ADD ((ontable ?X) (cleartop ?Y))
:DEL ((on ?X ?Y))
Operator Application

In planning: If we describe states as sets of facts and operator effects as ADD- and DEL-lists and if we rely on the closed-world assumption, a state transformation can be calculated syntactically by means of set-theoretical operations:

\[ s' = \text{Res}(s, o) = s \setminus (DEL \cup ADD) \text{ if } PRE \subseteq s \]

In problem solving: there are typically used more straightforward/specialized techniques. In the most simple case (e.g. GPS), there might even be a predefined state-operator table.
State Space

Formally, the state space is a graph $(S, A)$ with:
- $S$: the set of nodes (all states of a problem)
- $A \subseteq S \times S$: the set of arcs, with $(s, s') \in A$ iff state $s'$ can be reached from state $s$ by applying an operator $o \in O$.

State-space semantics: represents the structure of the problem. Each path in the graph from an initial state to a final state is an admissible solution.

Also known as problem space (Newell & Simon, 1972).

Remember: on the syntactical side we have descriptions of states which are partial! We represent only such information which is relevant for the problem (e.g. we are not interested in the material or in the color of blocks).
State Space for Blocksworld
Problem Solving as Search

If we can transform an initial state into a final state by applying one operator, we solve a task and not a problem. (We have the specialized program/the skill to reach our goal directly.)

For interpolation problems we do not know, which sequence of operators is suitable (“admissible”) to transform an initial state into a goal state.

Human problem solvers typically use additional knowledge (besides the legal operators) to solve a problem: choose always that operator which makes the difference between the current state and the desired state as small as possible (heuristics used by GPS, see chapter on human problem solving).

Algorithmic solution: search
Problem Solving as Search cont.

There exist different search strategies:

- Basic, uninformed ("blind") methods: random search, systematic strategies (depth-first, breadth-first)
- Search algorithms for operators with different costs
- Heuristic search: use assumptions to guide the selection of the next candidate operator

In the following: We are not concerned how a single state transformation is calculated. We represent problems via problem graphs. Please note: In general, such a graph is not explicitly given. We do not know the complete set of states $S$ (which can be very large). A part of the problem graph is constructed during search (the states which we explore).
**Search Tree**

- During search for a solution, starting from an initial state, a search tree is generated.
  - **Root**: initial state
  - Each **path from the root to a leaf**: (partial) solution
  - **Intermediate nodes**: intermediate states
  - **Leafs**: Final states or dead ends

- If the same state appears more than once on a path, we have a **cycle**. Without cycle-check search might not terminate! (infinite search tree)

- Again: A search tree represents *a part* of the problem graph
Problem Solving Strategy

As long as no final state is reached or there are still reachable, yet unexplored states:

- Collect all operators which can be applied in the current state (Match state with application conditions)

- Select on applicable operator.
  In our example: alphabetic order of the labels of the resulting node.
  In general: give a preference order

- Apply the operator, generating a successor state

Remark: The Match-Select-Apply Cycle is the core of “production systems” (see chapter human problem solving)
Example

In the following: abstract problem graph with nodes representing states and arcs representing operator applications. Numerical labels of the arcs: costs.

Illustration: Navigation problem with states as cities and arcs as traffic routes; Blocksworld problem with states as constellations of blocks and arcs as put/puttable operators (might have different costs for different blocks); etc.
Example Search Tree

Initial State

Operator cost

Finale state

Cycle deadend

deadend

Depth-First Search

- Construct *one* path from initial to final state.
- Backtracking: Go back to predecessor state and try to generate another successor state (if none exists, backtrack again etc.), if:
  - the reached state is a deadend, or
  - the reached state was already reached before (cycle)
- Data structure to store the already explored states: Stack; depth-first is based on a “last in first out” (LIFO) strategy
- Cycle check: does the state already occur in the path.
- Result: in general not the shortest path (but the first path)
Effort of Depth-First Search

In the best-case, depth-first search finds a solution in linear time $O(d)$, for $d$ as average depth of the search tree: Solution can be found on a path of depth $d$ and the first path is already a solution.

In the worst-case, the complete search-tree must be generated: The problem has only one possible solution and this path is created as the last one during search; or the problem has no solution and the complete state-space must be explored.
Remark: Depth-First Search

The most parsimonious way to store the (partial) solution path is to push always only the current state on the stack. Problem: additional infrastructure for backtracking (remember which operators were already applied to a fixed state)

In the following: push always the partial solution path to the stack.
Depth-First Algorithm

Winston, 1992
To conduct a depth-first search,

- Form a one-element stack consisting of a zero-length path that contains only the root node.
- Until the top-most path in the stack terminates at the goal node or the stack is empty,
  - Pop the first path from the stack; create new paths by extending the first path to all neighbors of the terminal node.
  - Reject all new paths with loops.
  - Push the new paths, if any, on the stack.
- If the goal node is found, announce success; otherwise announce failure.
Depth-First Example

((S))
((S A) (S B))
((S A B) (S A F) [(S A S)] (S B))
((S A B C) (S A B D) [(S A B S)] (S B))
([(S A B C B)] (S A B C F) (S A B D) (S B))
Breadth-First Search

- The search tree is expanded levelwise.
- No backtracking necessary.
- Data structure to store the already explored states: **Queue**
  breadth-first is based on a “first in first out” (FIFO) strategy
- Cycle check: for finite state-spaces noty necessary for termination (but for efficiency)
- Result: shortest solution path
- Effort: If a solution can be first found on level $d$ of the search tree and for an average branching factor $b$:
  $O(b^d)$
Breadth-First Algorithm

*Winston, 1992*

To conduct a breadth-first search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.
- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the back of the queue.
- If the goal node is found, announce success; otherwise announce failure.
Breath-First Example

((S))
((S A) (S B))
((S B) (S A B) (S A F) [(S A S)])
((S A B) (S A F) (S B A) (S B C) (S B D) [(S B S)])
((S A F) (S B A) (S B C) (S B D) (S A B C) (S A B D) [(S A B S)])
Evaluation of Search Strategies

- **Completeness:** The strategy is guaranteed to find a solution whenever there exists one.
- **Termination:** The algorithm terminates with a solution or with an error message if no solution exists.
- **Soundness:** The strategy only returns admissible solutions.
- **Correctness:** Soundness + Termination
- **Optimality:** The strategy finds the “highest-quality” solution (minimal number of operator applications or minimal cost)
- **Effort:** How many time and/or how many memory is needed to generate an output?
Complexity of Blocksworld Problems

Remember: Problems can be characterized by their complexity, most problems considered in AI are NP-hard.

<table>
<thead>
<tr>
<th># blocks</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td># states</td>
<td>1</td>
<td>3</td>
<td>13</td>
<td>73</td>
<td>501</td>
</tr>
<tr>
<td>approx.</td>
<td>$1.0 \times 10^0$</td>
<td>$3.0 \times 10^0$</td>
<td>$1.3 \times 10^1$</td>
<td>$7.3 \times 10^1$</td>
<td>$5.0 \times 10^2$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># blocks</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td># states</td>
<td>4051</td>
<td>37633</td>
<td>394353</td>
<td>4596553</td>
<td>58941091</td>
</tr>
<tr>
<td>approx.</td>
<td>$4.1 \times 10^3$</td>
<td>$3.8 \times 10^4$</td>
<td>$3.9 \times 10^5$</td>
<td>$4.6 \times 10^6$</td>
<td>$5.9 \times 10^7$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># blocks</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td># states</td>
<td>824073141</td>
<td>12470162233</td>
<td>202976401213</td>
<td>3535017524403</td>
<td>65573803186921</td>
</tr>
<tr>
<td>approx.</td>
<td>$8.2 \times 10^8$</td>
<td>$1.3 \times 10^{10}$</td>
<td>$2.0 \times 10^{11}$</td>
<td>$3.5 \times 10^{12}$</td>
<td>$6.6 \times 10^{13}$</td>
</tr>
</tbody>
</table>

Blocksworld problems are **PSpace-complete**: even for a polynomial time algorithm, an exponential amount of memory is needed!
# Time and Memory Requirements

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 ms</td>
<td>100 Byte</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>0.1 sec</td>
<td>11 KiloByte</td>
</tr>
<tr>
<td>4</td>
<td>11.111</td>
<td>11 sec</td>
<td>1 MegaByte</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>18 min</td>
<td>111 MegaByte</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>31 h</td>
<td>11 GigaByte</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>128 days</td>
<td>1 TeraByte</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>35 years</td>
<td>111 TeraByte</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3500 years</td>
<td>11.111 TeraByte</td>
</tr>
</tbody>
</table>

Breadth-first search with branching factor $b = 10$, 1000 nodes/sec, 100 bytes/node. Memory requirements are the bigger problem!
Evaluation of DFS and BFS

**Soundness:** A node $s$ is only expanded to such a node $s'$ where $(s, s')$ is an arc in the state space (application of a legal operator whose preconditions are fulfilled in $s$).

**Termination:** For finite sets of states guaranteed.

**Completeness:** If a finite length solution exists.

**Optimality:** Depth-first no, breadth-first yes

Worst case $O(b^d)$ for both, average case better for depth-first

$\leftrightarrow$ If you know that there exist many solutions, that the average solution length is rather short and if the branching factor is rather high, use depth-first search, if you are not interested in the optimal but just in some admissible solution.

- Prolog is based on a depth-first search-strategy.
- Typical planning algorithms are depth-first.
Uniform Cost Search

- Variation of breadth-first search for operators with different costs.
- Path-cost function $g(n)$: summation of all costs on a path from the root node to the current node $n$.
- Costs must be positive, such that $g(n) < g(\text{successor}(n))$. Remark: This restriction is stronger then necessary. To omit non-termination when searching for an optimal solution it is enough to forbid negative cycles.
- Always sort the paths in ascending order of costs.
- If all operators have equal costs, uniform cost search behaves exactly like breadth-first search.
- Uniform cost search is closely related to branch-and-bound algorithms (cf. operations research).
Uniform Cost Algorithm

To conduct a uniform cost search,

- Form a one-element queue consisting of a zero-length path that contains only the root node.

- Until the first path in the queue terminates at the goal node or the queue is empty,
  - Remove the first path from the queue; create new paths by extending the first path to all neighbors of the terminal node.
  - Reject all new paths with loops.
  - Add the new paths, if any, to the queue and sort the queue with respect to costs.

- If the goal node is found, announce success; otherwise announce failure.
Uniform Cost Example

(omitting cycles)

((S).0)

((S A).3 (S B).4)

((S A B).5 (S A F).6 (S B).4)

sort


sort


sort


((S B C F).7 (S A B C).6 (S A F).6 (S B A).6 (S B D).6 (S A B D).7)
Uniform Cost Example cont.

\[ ((S \ B \ C \ F).7 \ (S \ A \ B \ C).6 \ (S \ A \ F).6 \ (S \ B \ A).6 \ (S \ B \ D).6 \ (S \ A \ B \ D).7) \]

\[ \text{sort} \]

\[ ((S \ A \ B \ C).6 \ (S \ A \ F).6 \ (S \ B \ A).6 \ (S \ B \ D).6 \ (S \ A \ B \ D).7 \ (S \ B \ C \ F).7) \]

\[ \text{sort} \]

\[ ((S \ A \ B \ C \ F).8 \ (S \ A \ F).6 \ (S \ B \ A).6 \ (S \ B \ D).6 \ (S \ A \ B \ D).7 \ (S \ B \ C \ F).7) \]

\[ \text{sort} \]

\[ ((S \ A \ F).6 \ (S \ B \ A).6 \ (S \ B \ D).6 \ (S \ A \ B \ D).7 \ (S \ B \ C \ F).7 \ (S \ A \ B \ C \ F).8) \]

Note: Termination if first path in the queue (i.e. shortest path) is solution, only then it is guaranteed that the found solution is optimal!
Further Search Algorithms

**Depth-limited search:** Impose a cut-off (e.g. $n$ for searching a path of length $n - 1$), expand nodes with max. depth first until cut-off depth is reached (LIFO strategy, since variation of depth-first search).

**Bidirectional search:** forward search from initial state & backward search from goal state, stop when the two searches meet. Average effort $O(b^{d/2})$ if testing whether the search fronts intersect has constant effort $O(1)$.

In AI, the problem graph is typically not known. If the graph is known, to find all optimal paths in a graph with labeled arcs, **standard graph algorithms** can be used. E.g., the Dijkstra algorithm, solving the single source shortest paths problem (calculating the minimal spanning tree of a graph).