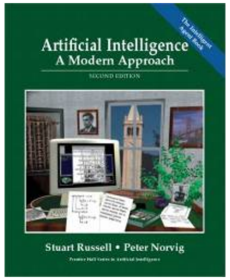
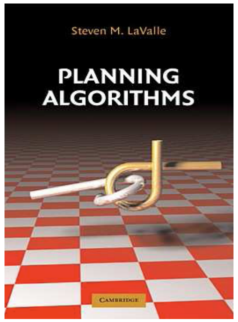
Literature

Illustrations and content presented in this lecture where taken from:



Artificial Intelligence – A Modern Approach, 2nd Edition

by Stuart Russell - Peter Norvig



Planning Algorithms

By Steven M. LaValle

Available for downloading at: http://planning.cs.uiuc.edu/

Problem-Solving Agents

→ Goal-based agents

Formulation: *goal* and *problem*

Given: *initial state*

Task: To reach the specified goal (a state) through the *execution of appropriate actions.*

→ Search for a suitable action sequence and execute the actions

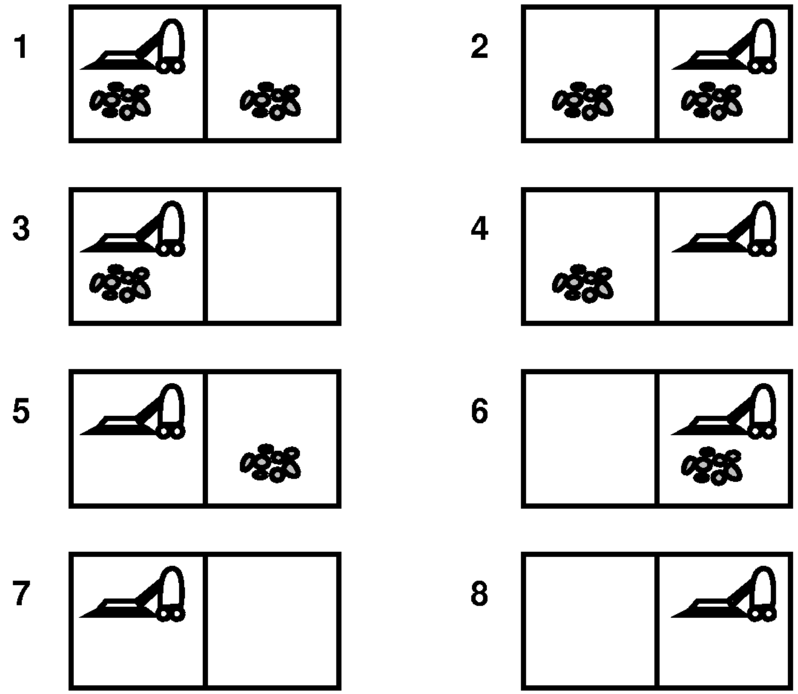
Problem Formulation

- Goal formulation
  World states with certain properties
- Definition of the state space
  important: only the relevant aspects → abstraction
- Definition of the actions that can change the world state
- Determination of the search cost (search costs, offline costs) and the execution costs (path costs, online costs)

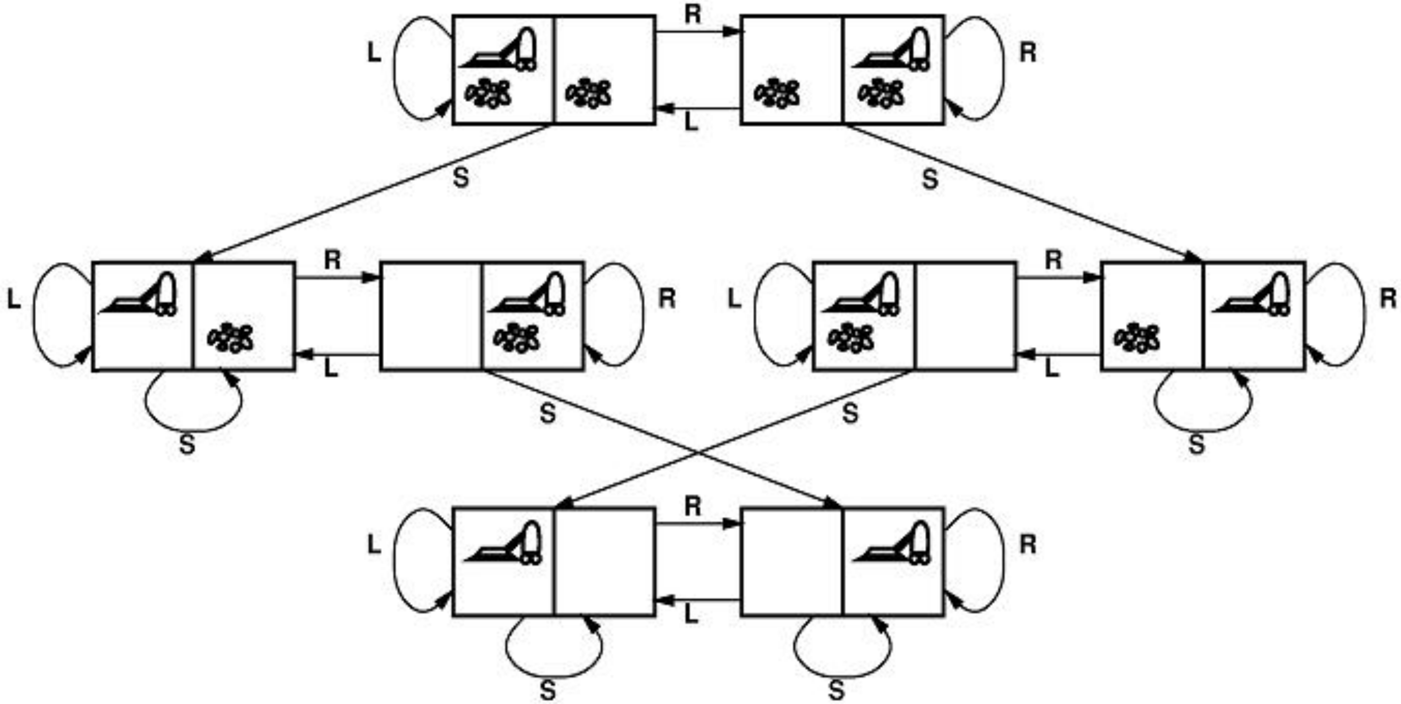
Note: The type of problem formulation can have a big influence on the difficulty of finding a solution.

Problem Formulation for the Vacuum Cleaner World

- World state space:
  2 positions, dirt or no dirt
  → 8 world states
- Successor function (Actions):
  Left (L), Right (R), or Suck (S)
- Goal state:
  no dirt in the rooms
- Path costs:
  one unit per action



The Vacuum Cleaner State Space



States for the search: The world states 1-8.

Implementing the Search Tree

Data structure for nodes in the search tree:

State: state in the state space

Node: Containing a state, pointer to predecessor, depth, and path cost, action

Depth: number of steps along the path from the initial state

Path Cost: Cost of the path from the initial state to the node

Fringe: Memory for storing expanded nodes. For example, stack or a queue

General functions to implement:

Make-Node(state): Creates a node from a state

Goal-Test(state): Returns true if state is a goal state

Successor-Fn(state): Implements the successor function, i.e. expands a set of new nodes given all actions applicable in the state

Cost(state,action): Returns the cost for executing action in state

Insert(node, fringe): Inserts a new node into the fringe

Remove-First(fringe): Returns the first node from the fringe

General Tree-Search Procedure

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
     if EMPTY?(fringe) then return failure
     node ← REMOVE-FIRST(fringe)
     if GOAL-TEST[problem] applied to STATE[node] succeeds
        then return SOLUTION(node)
     fringe ← INSERT-ALL(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
          STATE[s] ← result
 Make-    PARENT-NODE[s] ← node
 Node     ACTION[s] ← action
          PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
          DEPTH[s] ← DEPTH[node] + 1
     add s to successors
  return successors
```

Search Strategies

Uninformed or blind searches:

No information on the length or cost of a path to the solution.

- breadth-first search, uniform cost search, depth-first search,
- depth-limited search, Iterative deepening search, and
- bi-directional search

In contrast: informed or heuristic approaches

Criteria for Search Strategies

Completeness:

Is the strategy guaranteed to find a solution when there is one?

Time Complexity:

How long does it take to find a solution?

Space Complexity:

How much memory does the search require?

Optimality:

Does the strategy find the best solution (with the lowest path cost)?