



## EH2750 Computer Applications in Power Systems, Advanced Course.

### Lecture 4

Professor Lars Nordström, Ph.D.  
 Dept of Industrial Information & Control systems, KTH  
[larsn@ics.kth.se](mailto:larsn@ics.kth.se)



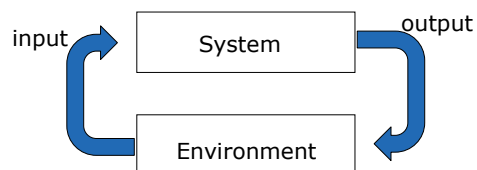
## Outline of the Lecture

- Repeating where we are right now
  - Intelligent Agents of various types
  - How does this appear in JACK?
- Searching for solutions (AI book - Ch 3)
- Informed Searches (Excerpt)
- Planning



## What is an Intelligent Agent?

- The main point about agents is they are *autonomous*: capable of acting independently, exhibiting control over their internal state
- Thus: *an intelligent agent is a computer system capable of flexible autonomous action in some environment in order to meet its design objectives*



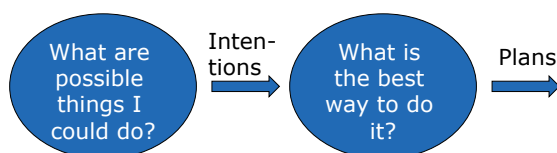
## The discussion so far

- Chapter 2 describes the idea of agents that perform tasks in an environment and sets some definitions
- Chapters 3, 4, & 5 describe three different approaches to describing and developing the apparent Intelligence in the agents.
  - Chapter 3 - Deductive Reasoning Agents
  - Chapter 4 - Practical Reasoning Agents
  - Chapter 5 - Reactive (and Hybrid Agents)
- Today, we take a deeper look at searching & planning

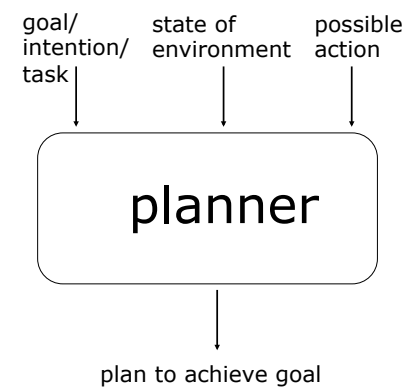


## Practical Reasoning

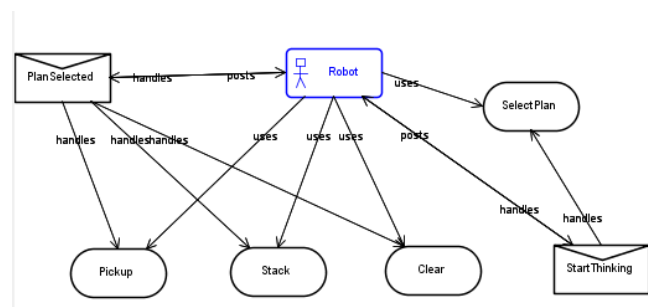
- Human practical reasoning consists of two activities:
  - *deliberation*  
deciding *what* state of affairs we want to achieve
  - *means-ends reasoning*  
deciding *how* to achieve these states of affairs
- The outputs of deliberation are *intentions*



## Planning is a big thing in AI



## How this can look in JACK



## Outline of the Lecture

- Repeating where we are right now
  - Intelligent Agents of various types
  - How does this appear in JACK?
- Searching for solutions (AI book - Ch 3)
- Informed Searches (Excerpt)
- Planning

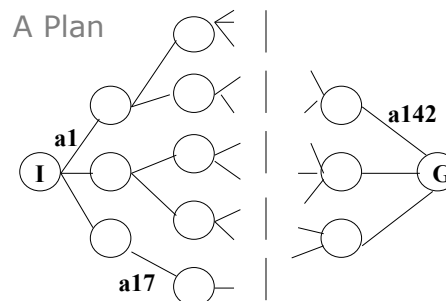


## Tree Search Algorithms

- Tree searching is a classic structure for finding solutions to a problem.
- The program searches through a Tree (graph) to find a solutions
- States are the **nodes** in the tree and actions are the **edges**
- Nodes are expanded into sucessor nodes using a successor function
- Which nodes to expand are determined by which search strategy the program has implemented.



## A Plan



- What is a plan?  
A sequence (list) of actions, with variables replaced by constants.



## Practical Reasoning Agent

```

function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  inputs: percept, a percept
  static: 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 do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
  action ← FIRST(seq)
  seq ← REST(seq)
  return action

```



## First some assumptions: The agent and the environment

- In Lecture 2, we discussed the characteristics of the environment the agent exists within
  - Accessible vs Inaccessible
  - Deterministic vs non-deterministic
  - Static vs Dynamic
  - Continuous vs Discrete
- For the searching & planning discussion we assume:
  - Accessible, Deterministic, Static & Discrete



## Environments *Accessible vs. inaccessible*

- An accessible environment is one in which the agent can obtain complete, accurate, up-to-date information about the environment's state
- Most moderately complex environments (including, for example, the everyday physical world and the Internet) are inaccessible
  - Subsets of the real-world can of course be made accessible
  - Measurements in a Power grid (U,I,P,Q, states,  $\phi$  etc)
- The more accessible an environment is, the simpler it is to build agents to operate in it



## Environments – *Deterministic vs. non-deterministic*

- A deterministic environment is one in which any action has a single guaranteed effect — there is no uncertainty about the state that will result from performing an action
- The physical world can to all intents and purposes be regarded as non-deterministic
  - Again, subsets of the real world can appear deterministic
- Non-deterministic environments present greater problems for the agent designer



## Environments *Static vs. dynamic*

- A static environment is one that can be assumed to remain unchanged except by the performance of actions by the agent
- A dynamic environment is one that has other processes operating on it, and which hence changes in ways beyond the agent's control
- Other processes can interfere with the agent's
- The real world is obviously a highly dynamic environment
  - But is a distribution grid a highly dynamic environment?



## Environments *Discrete vs. continuous*

- An environment is discrete if there are a fixed, finite number of actions and percepts in it
- A chess game is an example of a discrete environment, and taxi driving an example of a continuous one
- Continuous environments have a certain level of mismatch with computer systems
- Discrete environments could *in principle* be handled by a kind of “lookup table”



## Problem Formulation

- Before starting the search for a solution, we need to define the problem we are trying to solve
- A Problem formulation has the following parts:
  - An initial state
  - Actions possible in terms of **successor** function, that is a list of tuples:
    - (Action, Successor)
  - A goal state and a test if we are at the goal
  - A path cost related to the cost of a path/action\*

\*It is easy to think of the steps along the path as separate actions, this is OK, but formally not correct at this stage.



## Example - Searching in Romania

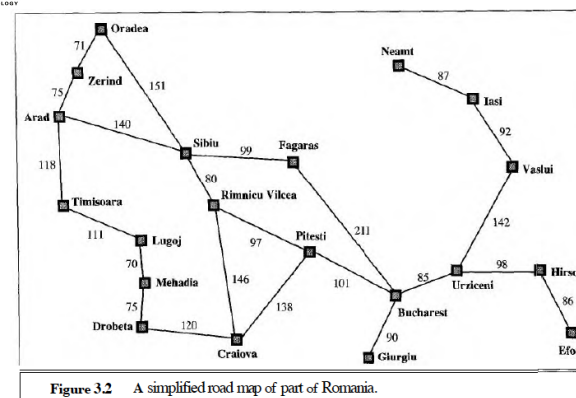


Figure 3.2 A simplified road map of part of Romania.



## Problem Formulation - Romania

- Initial State            In(Arad)
- Actions possible        Successor function F(state)  
For example:
  - F(Arad) = ((Go(Sibiu), In(Sibiu)), (Go(Timisoara), In(Timisoara)), (Go(Zerind), In(Zerind)))
- The Goal test           In(Bucharest)
- Path cost                Distances in Kilometers.



## General Idea of Search algorithm

```
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
```

Figure 3.7 An informal description of the general tree-search algorithm.

So, which search strategy should we use?



## Quality of Problem Solutions Strategies

- How do we rate one strategy over another
- Completeness
  - Is the strategy guaranteed to find a solution?
- Optimality
  - Does the strategy find the solution with the lowest path cost?
- Space complexity
  - How much memory is needed by the strategy
- Time complexity
  - How long time does it take to find the goal using the strategy



## Measuring Complexity

- The complexity of the solution in time & space represents the CPU processing time, and memory needs for the algorithm.
- Measurement (indices for complexity) are:
  - $b$  – branching factor, maximum number of successors to any node.
  - $d$  – depth, number of layers to reach the first optimal solution
  - $m$  – maximum length that a path can have.



## Some typical (uninformed) strategies

- Breadth First Search
- Uniform cost (breadth first) Search
- Depth First Search
- Backtracking Search
- Depth Limited Depth First Search
- Iterative Deepening search



## General tree search algorithm

```

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
    PARENT-NODE[s] ← node
    ACTION[s] ← action
    PATH-COST[s] ← PATH-COST[node] + STEP-COST(STATE[node], action, result)
    DEPTH[s] ← DEPTH[node] + 1
    add s to successors
  return successors
  
```

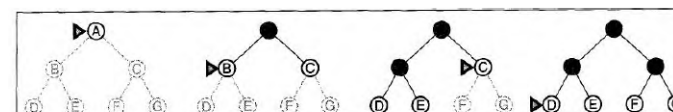


## Where....

- We implement the nodes in the tree as a queue.
- And implement the following functions to work on the queue.
  - **MAKE-QUEUE(*element*, ...)** creates a queue with the given element(s).
  - **EMPTY?(*queue*)** returns true only if there are no more elements in the queue.
  - **FIRST(*queue*)** returns the first element of the queue.
  - **REMOVE-FIRST(*queue*)** returns **FIRST(*queue*)** and removes it from the queue.
  - **INSERT(*element*, *queue*)** inserts an element into the queue and returns the resulting queue. (We will see that different types of queues insert elements in different orders.)
  - **INSERT-ALL(*elements*, *queue*)** inserts a set of elements into the queue and returns the resulting queue.



## Breadth First Search



**Figure 3.10** Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

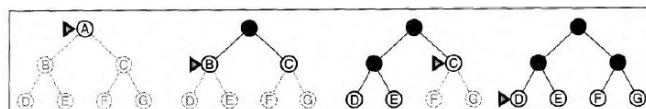
- The queue of Nodes is a FIFO queue (First in First Out)
- If  $d$  and  $b$  are limited, then BFS is Complete
- Optimal only if all Path costs are similar at same level.
- Unfortunately very memory and time-consuming, i.e. Complex
  - Number of nodes generated (memory need)

$$b + b^2 + b^3 + \dots + b^d + (b^{d+1} - b) = O(b^{d+1}).$$



## Uniform Cost Search

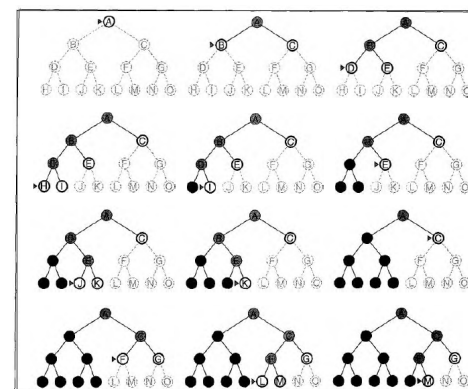
- Utilising the information about Path cost to select which path to follow.



- If all Path costs are equal, this is equal to Breadth First Search



## Depth First Search - I





## Depth First Search - II

- The fringe is implemented as a LIFO (Last in First Out) or commonly known as stack.
- Very modest memory requirements, only one path needs to be stored, since paths can be discarded after search to end.
  - Memory need is  $b \cdot m + 1 \ll O(b^{d+1})$
- DFS is **not complete**, since it can get stuck in loops
- DFS is **not optimal**, since it can find a solution, deep down one part of the tree, even if optimal solution is higher.



## Backtracking Search

- Variant of Depth First Search, where **only one** of a nodes successors is generated before moving on to that successor.
- Additionally, we do not keep the pre-decessor states in memory either, they are regenerated as we go back.
- This leaves un-expanded Nodes higher up, that must be recorded.
- Even less memory requirements –  $O(m)$



## Depth limited search

- By setting an  $l$  (length), that limits the maximum depth that a DFS can go.
- Basically, when the path length reaches  $l$ , we do not expand further successors
- Basic DFS can be considered as having infinite  $l$
- Basing  $l$  on some knowledge about the problem can be useful, this is an example of heuristics



## Iterative Deepening DFS

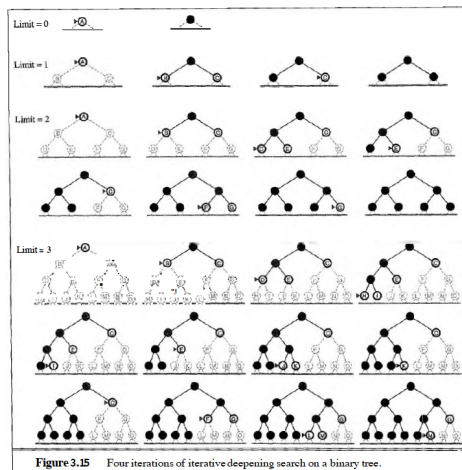
```

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
  
```

**Figure 3.14** The iterative deepening search algorithm, which repeatedly applies depth-limited search with increasing limits. It terminates when a solution is found or if the depth-limited search returns *failure*, meaning that no solution exists.

- Do a DFS with  $l = 1$
- If No solution found, set  $l = 2$  do same thing again.
- Repeated creation of states at higher levels in the tree is a small cost compared to the benefits gained by combining DFS and BFS.
- Preferred uninformed method, if state space is unknown





## Comparison of Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes <sup>a</sup>	Yes <sup>a,b</sup>	No	No	Yes <sup>a</sup>	Yes <sup>a,d</sup>
Time	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes <sup>c</sup>	Yes	No	No	Yes <sup>c</sup>	Yes <sup>c,d</sup>

**Figure 3.17** Evaluation of search strategies.  $b$  is the branching factor;  $d$  is the depth of the shallowest solution;  $m$  is the maximum depth of the search tree;  $l$  is the depth limit. Superscript caveats are as follows: <sup>a</sup> complete if  $b$  is finite; <sup>b</sup> complete if step costs  $\geq \epsilon$  for positive  $\epsilon$ ; <sup>c</sup> optimal if step costs are all identical; <sup>d</sup> if both directions use breadth-first search.



## How to avoid repeated states?

*"If an algorithm forgets its past, it is doomed to repeat it"*

- Simple answer is, keep track if a state has been expanded previously.



## Graph Search algorithm

```

function GRAPH-SEARCH(problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
    node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

```

**Figure 3.19** The general graph-search algorithm. The set *closed* can be implemented with a hash table to allow efficient checking for repeated states. This algorithm assumes that the first path to a state  $s$  is the cheapest (see text).



## Outline of the Lecture

- Repeating where we are right now
  - Intelligent Agents of various types
  - How does this appear in JACK?
- Searching for solutions (AI book - Ch 3)
- Informed Searches (Excerpt)
- Planning



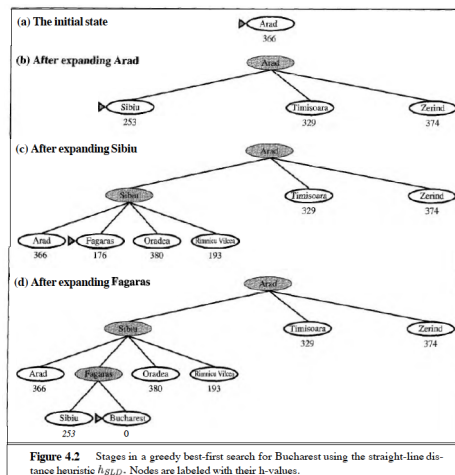
## Heuristics

- Often, we (the programmer) has **some knowledge about** the problem we are asking the agent (the computer) to solve.
- We can add different sorts of clever *heuristics* to our algorithm.
- Essentially, we use an evaluation function  $f(n)$  to select which successor node to expand, creating a priority queue, where  $f(\text{state})$  is the ranking of the nodes.
- Normally node the lowest value (distance to goal) is expanded first.



## Greedy First

- GFS always selects the node with apparent cheapest solution to reach goal.
- In Romania, we set for example:
- $h_{SLD}$  = shortest line distance
- Always expand node with lowest  $h_{SLD}$



## A\*

- A variant of Greedy First Search is A\*
- Uses the evaluation function  $f(n) = h(n) + g(n)$
- Where  $g(n)$  is the cost to get to where we are
- And  $h(n)$  is the estimated cost to reach goal.



## A\* example

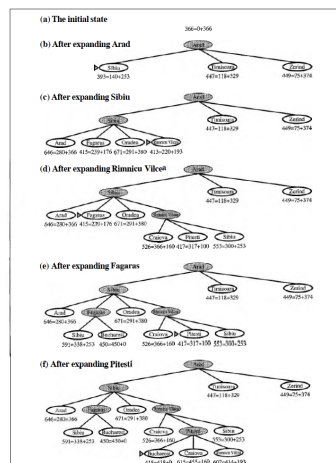


Figure 4.3 Stages in an A\* search for Bucharest. Nodes are labeled with  $f = g + h$ . The  $h$  values are the straight-line distances to Bucharest taken from Figure 4.1.



## Outline of the Lecture

- Repeating where we are right now
  - Intelligent Agents of various types
  - Some words on JACK development
- Searching for solutions (AI book - Ch 3)
- Informed Searches (Excerpt)
- Planning



## Planning approaches

- STRIPS based effort at a switching problem
- We need a problem definition



## Problem Formulation

- Before starting the search for a solution, we need to define the problem we are trying to solve
- A Problem formulation has the following parts:
  - An initial state
  - Actions possible in terms of **successor** function, that is a list of tuples:
    - (Action, Successor)
  - A goal state and a test if we are at the goal
  - A path cost related to the cost of a path/action\*

\*It is easy to think of the steps along the path as separate actions, this is OK, but formally not correct at this stage.



## The Switching Ontology

- To represent this environment, need an *ontology*
  - Conducting(x)*    Circuit Breaker x is conducting
  - Breaking(x)*    CB x is breaking
  - LightsOn(y)*    Load y is on
- *The closed world assumption is implicitly valid.*



## Representing Actions

- *Actions* are represented using a technique that was developed in the STRIPS planner
  - Each action has:
    - a *name*  
which may have arguments
    - a *pre-condition list*  
list of facts which must be true for action to be executed
    - a *delete list*  
list of facts that are no longer true after action is performed
    - an *add list*  
list of facts made true by executing the action
- Each of these may contain *variables*



## Actions in the problem

- Using STRIPS notation
- Closing Breaker x description is:
  - Name:            Close (x)
  - Pre:             Breaking(x)
  - Add:            Conducting (x)
  - Del:             Breaking (x)



So lets try this!



## Outline of the Lecture

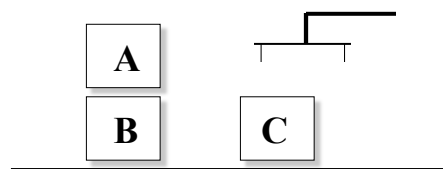
- Repeating where we are right now
  - Intelligent Agents of various types
  - Some words om JACK development
- Searching for solutions (AI book - Ch 3)
- Informed Searches (Excerpt)
- Planning



## Backup slides



## The Blocks World



- We'll illustrate the techniques with reference to the *blocks world*. Contains a robot arm, 3 blocks (A, B, and C) of equal size, and a table-top

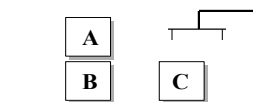


## The Blocks World

- Here is a representation of the blocks world described above:

$Clear(A)$   
 $On(A, B)$   
 $OnTable(B)$   
 $OnTable(C)$

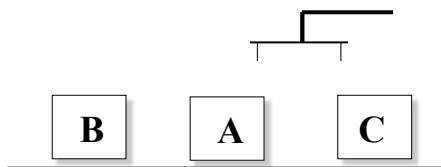
- Use the *closed world assumption*: anything not stated is assumed to be *false*





## The Blocks World

- A *goal* is represented as a set of formulae
- Here is a goal:  
 $OnTable(A) \wedge OnTable(B) \wedge OnTable(C)$



4:53

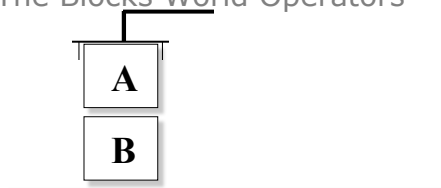


## The Blocks World

- *Actions* are represented using a technique that was developed in the STRIPS planner
  - Each action has:
    - a *name* which may have arguments
    - a *pre-condition list* list of facts which must be true for action to be executed
    - a *delete list* list of facts that are no longer true after action is performed
    - an *add list* list of facts made true by executing the action
- Each of these may contain *variables*



## The Blocks World Operators



- Example 1:  
The *stack* action occurs when the robot arm places the object  $x$  it is holding is placed on top of object  $y$ .

$Stack(x, y)$   
 pre  $Clear(y) \wedge Holding(x)$   
 del  $Clear(y) \wedge Holding(x)$   
 add  $ArmEmpty \wedge On(x, y)$



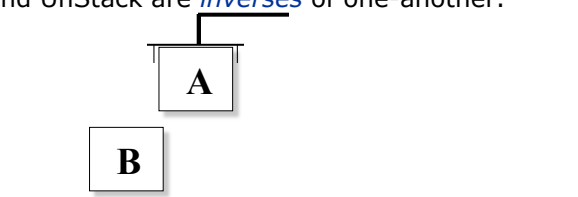
## The Blocks World Operators

### Example 2:

The *unstack* action occurs when the robot arm picks an object  $x$  up from on top of another object  $y$ .

$UnStack(x, y)$   
 pre  $On(x, y) \wedge Clear(x) \wedge ArmEmpty$   
 del  $On(x, y) \wedge ArmEmpty$   
 add  $Holding(x) \wedge Clear(y)$

Stack and UnStack are *inverses* of one-another.





## The Blocks World Operators

- Example 3:

The *pickup* action occurs when the arm picks up an object  $x$  from the table.

```
Pickup(x)
pre  Clear(x)  $\wedge$  OnTable(x)  $\wedge$  ArmEmpty
del  OnTable(x)  $\wedge$  ArmEmpty
add  Holding(x)
```

- Example 4:

The *putdown* action occurs when the arm places the object  $x$  onto the table.

```
Putdown(x)
pre  Holding(x)
del  Holding(x)
add  Clear(x)  $\wedge$  OnTable(x)  $\wedge$  ArmEmpty
```

---