Basic plan generation systems – Strips

- Its name is derived from **ST**anford **R**esearch **I**nstitute **P**roblem **S**olver.
- An automated planner.
- Planning system for a robotics project: SHAKEY.
- Developed by Richard Fikes and Nils Nilsson in 1971 at SRI International.
- Classical Planning System.
- The STRIPS representation is used to determine the values of primitive features in a state based on the previous state and the action taken by the agent.
- The STRIPS representation is based on the idea that most things are not affected by a single action.
- For each action, STRIPS models:
  - when the action is possible and what primitive features are affected by the action.
  - The effect of the action relies on the STRIPS assumption.
  - All of the primitive features not mentioned in the description of the action stay unchanged.
- The STRIPS representation for an action consists of:
  - Precondition, which is a set of assignments of values to features that must be true for the action to occur.
  - the effect, which is a set of resulting assignments of values to those primitive features that change as the result of the action.

.
Example: action of Rob to pick up coffee (puc) has the following STRIPS representation:
precondition

[cs, ¬rhc] where **cs** – coffee shop and **rhc** – robo is holding coffee

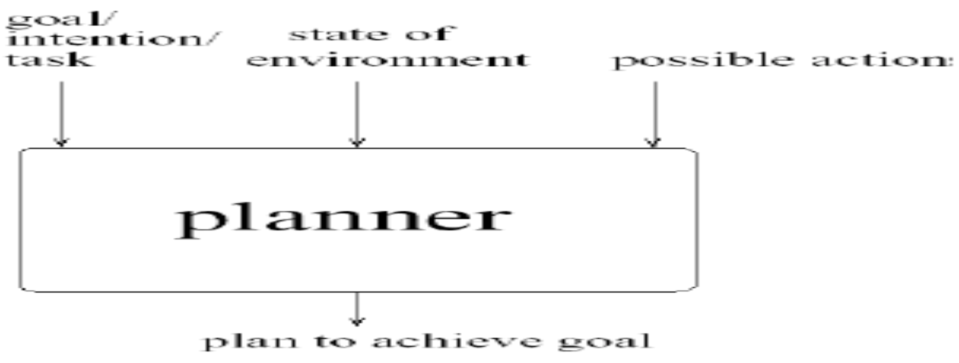
effect

[rhc]

What is a Plan?

- A sequence (list) of actions, with variables replaced by constants (specific objects in the environment)

Planner



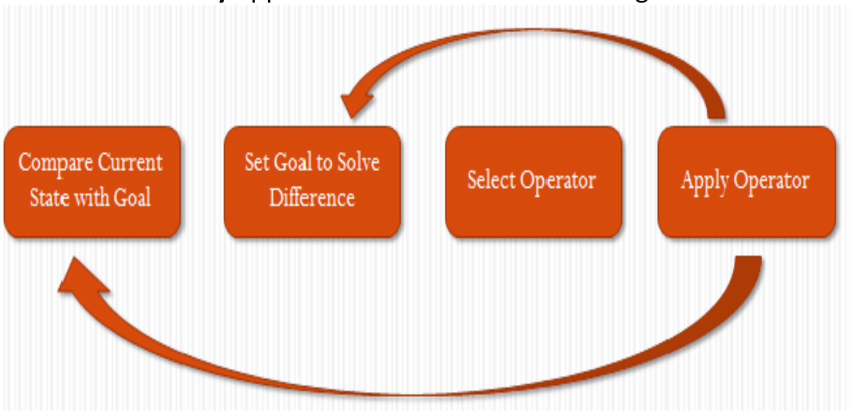
Means-Ends Analysis (Reasoning)

- **Means-Ends Analysis** (MEA) is a problem solving technique used commonly in AI for limiting search in AI programs.
- Idea is to give an agent:
  - representation of goal/intention to achieve;

  - representation of actions it can perform;
  - and representation of the environment; ƒ
- Then have the agent generate a plan to achieve the goal. ƒ
- The plan is generated entirely by the planning system, without human intervention.

How MEA Works?

- The MEA technique is a strategy to control search in problem-solving.
- Given a current state and a goal state, an action is chosen which will reduce the *difference* between the two.
- The action is performed on the current state to produce a new state, and the process is **recursively** applied to this new state and the goal state.



STRIPS-Like Planning Formulation

- A finite, nonempty set of *instances*.
- A finite, nonempty set of *predicates*, which are binary-valued (partial) functions of one of more instances. Each application of a predicate to a specific set of instances is called a *positive literal*. A logically negated positive literal is called a *negative literal*.
- A finite, nonempty set of *operators*, each of which has: 1) *preconditions*, which are positive or negative literals that must hold for the operator to apply, and 2) *effects*, which are positive or negative literals that are the result of applying the operator.
- An *initial set* which is expressed as a set of *positive literals*. Negative literals are implied. For any positive literal that does not appear in , its corresponding negative literal is assumed to hold initially.
- A *goal set* which is expressed as a set of both *positive* and *negative literals*.

STRIPS Planner

- STRIPS maintains **two additional data structures**:
  - **State List** - all currently true predicates.
  - **Goal Stack** - a push down stack of goals to be solved, with current goal on top of stack. ƒ
- If the current goal is not satisfied by present state,
  Find goal in the add list of an operator, and
  push operator and preconditions list on stack.   (=Sub goals) ƒ
- When a current goal is satisfied, POP it from stack. ƒ
- When an operator is on top of the stack,
  - record the application of that operator – update the plan sequence and
  - use the operator's add and delete lists to update the current state.

Planning in STRIPS

- Uses MEA (actions = means, goals = ends) ƒ
- States of the world and goals are represented as a set/list of predicates that are true (e.g. *On(X,Y)* ..)
  - ✓ *The current state is initialized to the start state.*

- ✓ *The goal is placed on the goal stack.*
- ✓ *Loop through the following steps to produce a plan*

Reasoning Loop

- If the top item on the goal stack is:
  - *empty (the goal stack is empty), return the **actions** executed*
    - *they form the plan to achieve the goal*
  - *a **goal**, and it is **satisfied** in the current state, remove it from the stack (no replacement necessary)*
  - *a complex goal, break it into sub goals, placing all sub goals on the goal stack (the original goal is pushed down into the goal stack)*
  - *a predicate, find an action that will make it true, then place that action (with variables bound appropriately) and its preconditions on the goal stack (preconditions first)*
  - *an action and its preconditions are satisfied, perform the action, updating the world state using the delete and add lists of the action (if the pre-conditions are not satisfied, add them to the goal list without removing the action).*
  - *Add this action to the partial plan*

Example-Blocks World (BW) Planning

- **What is the Blocks World?**
- The world consists of:
  - **A flat surface** such as a tabletop
  - An adequate **set of identical blocks** which are **identified by letters**.
  - The blocks can be **stacked** one on one to form towers of apparently unlimited height.

Why Use the Blocks World as an Example?

- Sufficiently simple and well behaved.
- Easily understood.
- Good sample environment to study planning.
  - problems can be broken into nearly distinct sub problems.
  - we can show how partial solutions need to be combined to form a realistic complete solution.

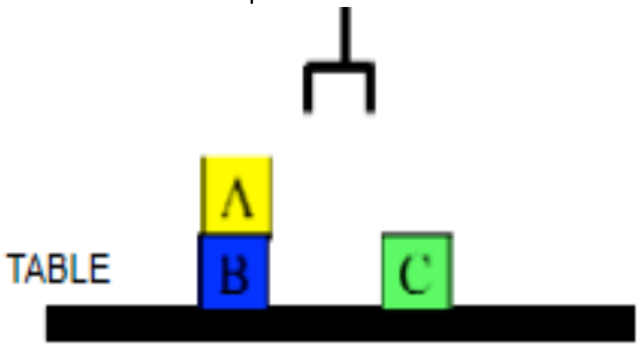
Blocks World Planning Contd...

- The stacking is achieved using a **robot arm** which has fundamental operations and states which can be assessed using logic and combined using logical operations.
- ***The robot can hold one block at a time and only one block can be moved at a time.***
- *Any number of blocks can be on the table.*

How to do We Represent the Blocks World Problem?

- State of environment.
- Goal to be achieved.

- Actions available to agent.
- Plan itself.
- This world contains
  - a robot arm with gripper,
  - 3 blocks (A, B and C) of equal size,
  - a table-top

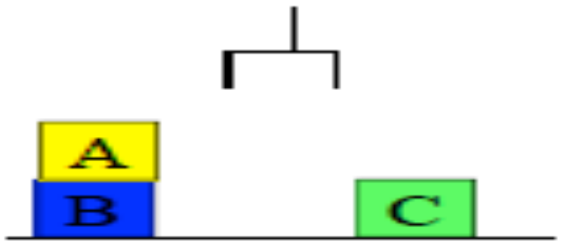


How to do We Represent the Blocks World Problem?
Need of Ontology/ Predicates to represent this environment.

- *On(X,Y)* --- means block X is on top of block Y
- *OnTable(X)* --- block X is on the table ƒ
- *Clear(X)* --- nothing is on top of block X ƒ
- *Holding(X)* --- robot arm is holding block X
- *ArmEmpty()* --- robot arm/hand is not holding anything (block in this world)

State Representation = Environment

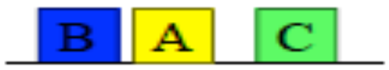
- A representation of one state of the blocks world.
- The state in the figure is:
  - Clear(A)
  - Clear(C)
  - On(A,B)
  - OnTable(B)
  - OnTable(C)
  - ArmEmpty()
  - Holding(A)
- Use the <u>closed world assumption</u>: anything not stated is assumed to be false
- Also called "<u>**Negation by failure**</u>"



Goal Representation

- A *goal* is represented as a set of formulae.
- Here is a goal.
  - *OnTable(A) ƒ*
  - *OnTable(B) ƒ*

- *OnTable(C)*



Actions

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

Action/Operator Representation

- **Basic operations**
  - *stack(X,Y)*: put block X on block Y
  - *unstack(X,Y)*: remove block X from block Y
  - *pickup(X)*: pickup block X from the table
  - *putdown(X)*: put block X on the table ƒ
- Each operator is represented by facts that describe the state of the world before and changes to the world after an action is performed.
  - a list of preconditions
  - a list of new facts to be added (add-effects)
  - a list of facts to be removed (delete-effects)
  - optionally, a set of (simple) variable constraints

Precondition/Delete/Add Lists

- Preconditions
  - $P_1 \ldots\ldots P_i$
- Additions
  - $A_1 \ldots\ldots A_k$
- Deletions
  - $D_1 \ldots\ldots D_j$
- **Meaning:**
  - All P_i must be true before an action is performed (otherwise it can't be accomplished)
  - After the action, all A_k are added to the agent's memory/state
  - After the action, all D_j ƒ are removed from the agent's memory/state
  - Subject to constraints imposed on the state of the world
    - e.g. a block can't be stacked on top of itself!!

Stack Operator

- The **stack action** occurs when the robot arm places the object it is holding [X] on top of another object [Y].
- **Form:** *Stack(X,Y) ƒ*
- **Pre:** $Clear(Y) \wedge Holding(X)$
- **Add:** $ArmEmpty() \wedge On(X,Y) \wedge Clear(X)$
- **Del:** $Clear(Y) \wedge Holding(X)$
- **Constraints:** $(X \neq Y), X \neq Table, Y \neq Table$

Unstack Operator

- The **unstack action** occurs when the robot arm picks up an object X from on top of another object Y.

- **Form:** *UnStack(X,Y)*
- **Pre:** *On(X,Y) ∧ Clear(X) ∧ ArmEmpty()*
- **Add:** *Holding(X) ∧ Clear(Y)*
- **Del:** *On(X,Y) ∧ Clear(X) ∧ ArmEmpty()*
- **Constraints:** *X ≠ Y, X ≠ Table, Y ≠ Table*

Pickup Operator

- The pickup action occurs when the arm ***picks up an object (block) X from the table***.
- **Form:** *Pickup(X)*
- **Pre:** *OnTable(X) ∧ Clear(X) ∧ ArmEmpty()*
- **Add:** *Holding(X)*
- **Del:** *OnTable(x) ∧ Clear(x) ∧ ArmEmpty()*
- **Constraints:** *X ≠ Table*

Putdown Operator

- The putdown action occurs when the ***arm places the object X onto the table.***
- **Form:** *PutDown(X)*
- **Pre:** *Holding(X)*
- **Add:** *OnTable(X) ∧ ArmEmpty()∧ Clear(X)*
- **Del:** *Holding(X)*
- **Constraints:** *X ≠ Table*

Components of a planning System:

- Choose the best rule to apply next based on the best available heuristic information.
- Apply the chosen rule to compute the new problem state that arises from its application.
- Detect when a solution has been found.
- Detect dead ends so that they can be abandoned and the system's effort directed in more fruitful directions.

- Detect when an almost correct solution has been found and employ special techniques to make it totally correct.

  - ➢ Choosing Rules to Apply
  - ➢ Applying Rules
  - ➢ Detecting a solution
  - ➢ Detecting Dead Ends
  - ➢ Repairing an almost correct Solution

Choosing Rules to Apply:

Widely used technique for selecting appropriate rules to apply is to isolate a set of differences between the desired goal state and the current state and then to identify those rules that are relevant to reducing those differences.
If several rules are found, variety of heuristic infn. Can be ex[loited to choose among them.

Applying Rules

STACKS(x,y)

P: CLEARC(y) $\wedge$ HOLDING(x)
D: CLEAR(y) $\wedge$ HOLDING(x)
A: ARMEMPTY $\wedge$ ON(x, y)

UNSTACK(x,y)

P: ON(x, y) $\wedge$ CLEAR(x) $\wedge$ ARMEMPTY
D: ON(x, y) $\wedge$ ARMEMPTY
A: HOLDINGS(x) $\wedge$ CLEARC(y)

PICKUP(x)

P: CLEAR(x) $\wedge$ ONTABLE(x) $\wedge$ ARMEMPTY
D: ONTABLE(x) $\wedge$ ARMEMPTY
A: HOLDING(x)

PUTDOWN(x)

P: HOLDING(x)
D: HOLDING(x)
A: ONTABLE(x) $\wedge$ ARMEMPTY

Detecting a solution

- A planning system has succeeded in finding a solution to a problem when it has found a sequence of operators that transforms the initial problem state into the goal state.
- For simple problem-solving systems, it involves straightforward match of the state descriptions.
- Otherwise, it can be solved using some representational scheme. One such representational technique is predicate logic.

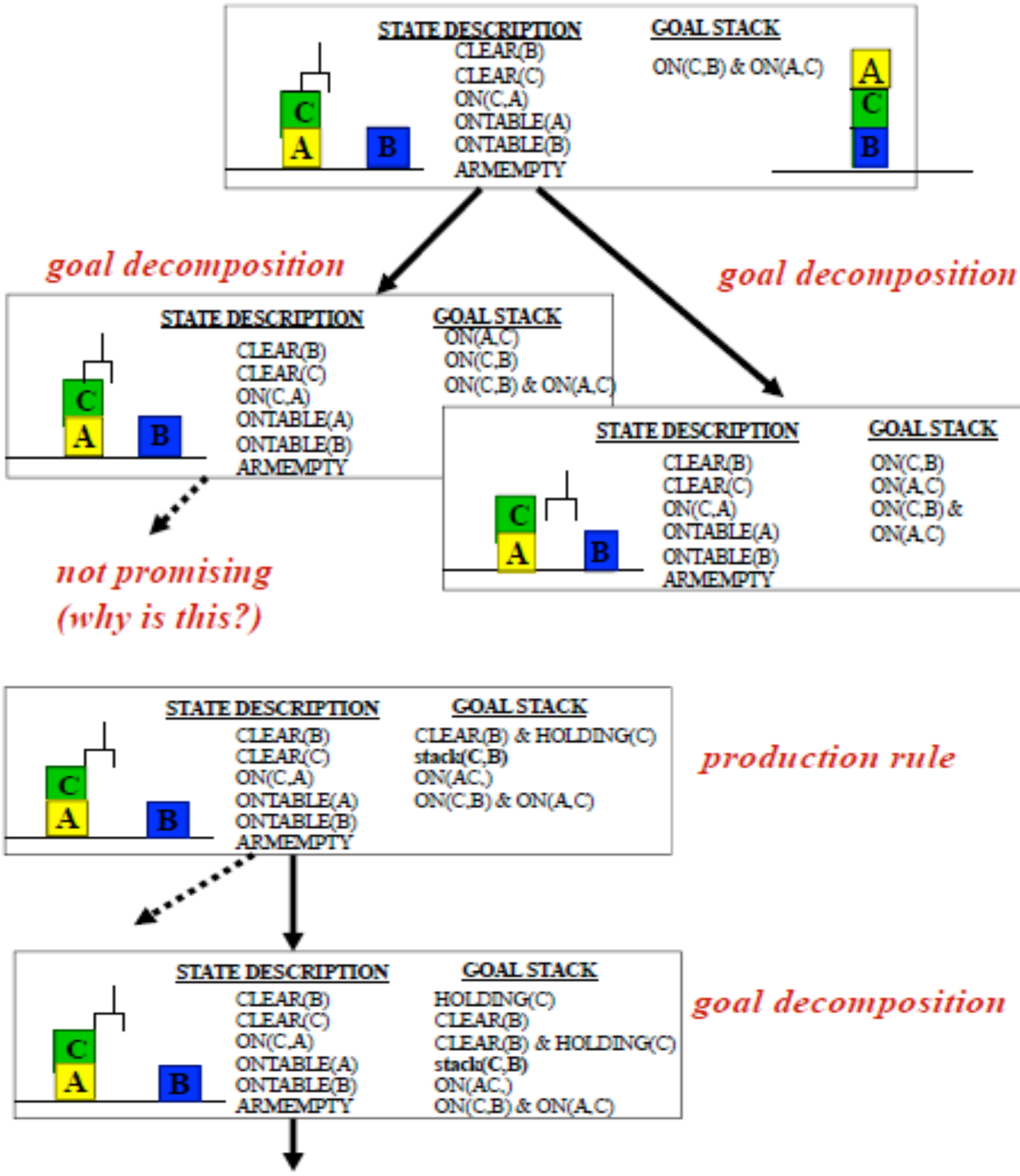
Detecting dead ends

- As a planning system is searching for a sequence of operators to solve a particular problem, it must be able to detect when it is exploring a path that can never lead to a solution.
- Same reasoning mechanisms can be used to detect the dead end.
- If the search process is reasoning forward, it can prune any path that leads to a state from which the goal state cannot be reached.
- If the search process is reasoning back]ward from the goal state, it can also terminate a path either sure that initial state cannot be reached or little progress has been made.

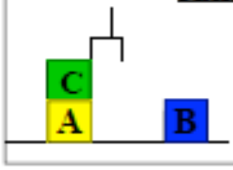
Repairing an almost correct solution

- One way of solving decomposable problems is to assume that thay are completely decomposable, proceed to solve the subproblems separately, then combine the subsolutions, then verify whether they yeild a solution to the original problem.
- If they do not, just throw away the solution, look for another one and hope that it is better.
- A slightly better approach is to look at the situation that results when the sequence of operations corresponding to the proposed solution is executed and to compare that situation to the desired goal. In most cases, the difference between the two will be smaller than the diff. between the initial state and the goal . now the problem-solving system can be called again and asked to find a way of eliminating this new difference. The first solution can then be combined
- Even better way is to patch up an almost correct solution is to appeal to specific knowledge about what went wrong and then to apply a direct path.

Goal Stack Planning


STATE DESCRIPTION
CLEAR(B)
CLEAR(C)
ON(C,A)
ONTABLE(A)
ONTABLE(B)
ARMEMPTY
GOAL STACK
ON(C,B) & ON(A,C)
C
A
B
A
C
B
goal decomposition
goal decomposition
STATE DESCRIPTION
CLEAR(B)
CLEAR(C)
ON(C,A)
ONTABLE(A)
ONTABLE(B)
ARMEMPTY
GOAL STACK
ON(A,C)
ON(C,B)
ON(C,B) & ON(A,C)
C
A
B
STATE DESCRIPTION
CLEAR(B)
CLEAR(C)
ON(C,A)
ONTABLE(A)
ONTABLE(B)
ARMEMPTY
GOAL STACK
ON(C,B)
ON(A,C)
ON(C,B) &
ON(A,C)
C
A
B
not promising
(why is this?)
STATE DESCRIPTION
CLEAR(B)
CLEAR(C)
ON(C,A)
ONTABLE(A)
ONTABLE(B)
ARMEMPTY
GOAL STACK
CLEAR(B) & HOLDING(C)
stack(C,B)
ON(AC,)
ON(C,B) & ON(A,C)
C
A
B
production rule
STATE DESCRIPTION
CLEAR(B)
CLEAR(C)
ON(C,A)
ONTABLE(A)
ONTABLE(B)
ARMEMPTY
GOAL STACK
HOLDING(C)
CLEAR(B)
CLEAR(B) & HOLDING(C)
stack(C,B)
ON(AC,)
ON(C,B) & ON(A,C)
C
A
B
goal decomposition

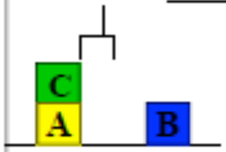
Solution = {}



STATE DESCRIPTION	GOAL STACK
CLEAR(B)	HOLDING(C)
CLEAR(C)	CLEAR(B)
ON(C,A)	CLEAR(B) & HOLDING(C)
ONTABLE(A)	**stack(C,B)**
ONTABLE(B)	ON(A,C,)
ARMEMPTY	ON(C,B) & ON(A,C)

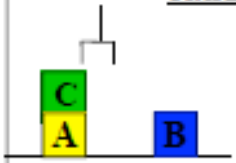
production rule

- **unstack(x,y)**
  - P&D: HANDEMPTY, CLEAR(x), ON(x,y)
  - A: HOLDING(x), CLEAR(y)



STATE DESCRIPTION	GOAL STACK
CLEAR(B)	HANDEMPTY & CLEAR(C) & ON(C, y)
CLEAR(C)	**unstack(C, y)**
ON(C,A)	CLEAR(B)
ONTABLE(A)	CLEAR(B) & HOLDING(C))
ONTABLE(B)	**stack(C,B)**
ARMEMPTY	ON(A,C,)
	ON(C,B) & ON(A,C)

Solution = {}

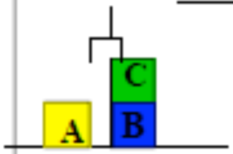


STATE DESCRIPTION	GOAL STACK
CLEAR(B)	HANDEMPTY & CLEAR(C) & ON(C, y)
CLEAR(C)	**unstack(C, y)**
ON(C,A)	CLEAR(B)
ONTABLE(A)	CLEAR(B) & HOLDING(C)
ONTABLE(B)	**stack(C,B)**
ARMEMPTY	ON(A,C,)
	ON(C,B) & ON(A,C)

- **unstack(x,y)**
  - P&D: ARMEMPTY, CLEAR(x), ON(x,y)
  - A: HOLDING(x), CLEAR(y)

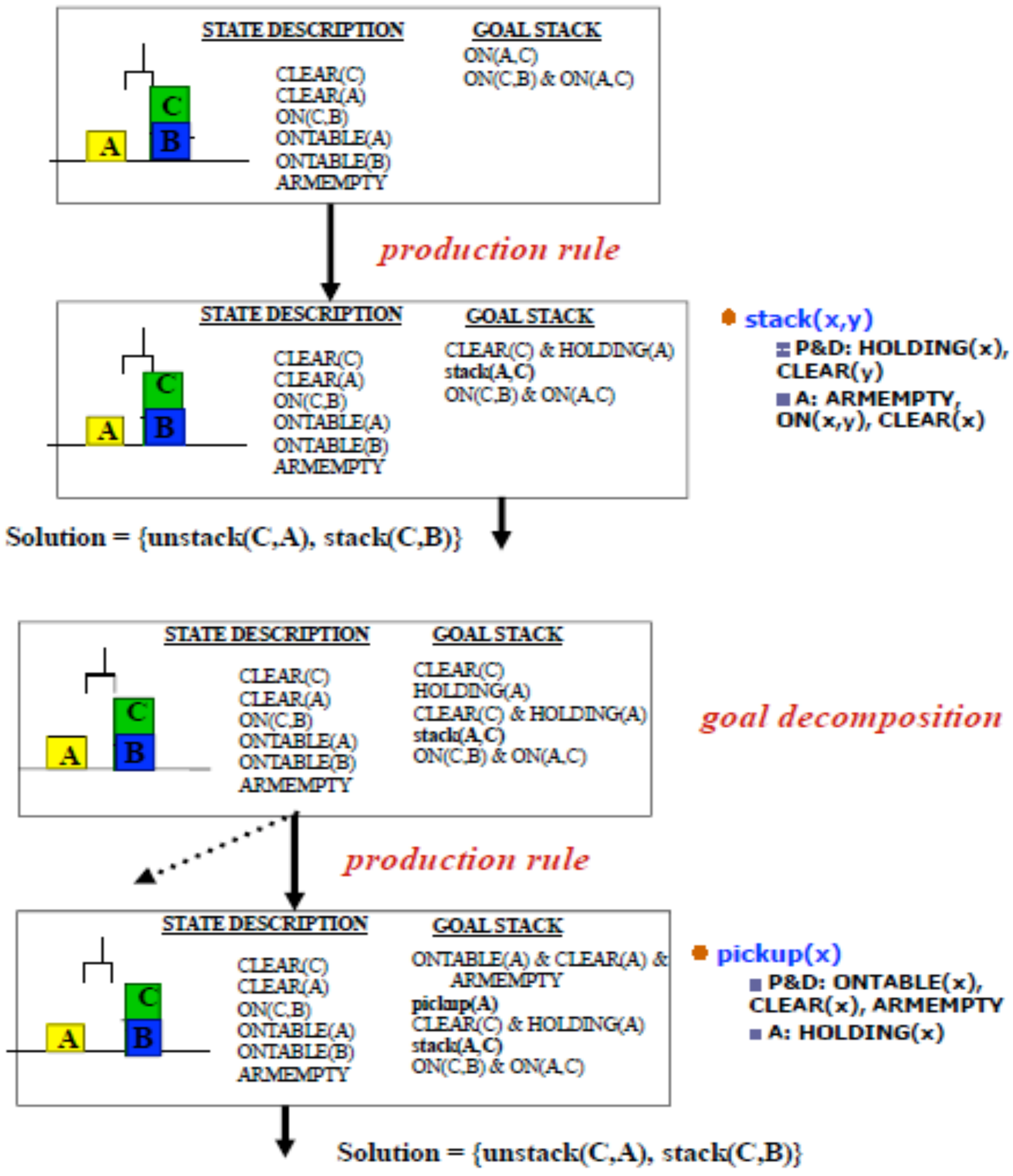
Substitute {A/y}, then apply unstack(C,A) then stack(C,B)

- **stack(x,y)**
  - P&D: HOLDING(x), CLEAR(y)
  - A: ARMEMPTY, ON(x,y), CLEAR(x)



STATE DESCRIPTION	GOAL STACK
CLEAR(C)	ON(A,C)
CLEAR(A)	ON(C,B) & ON(A,C)
ON(C,B)	
ONTABLE(A)	
ONTABLE(B)	
ARMEMPTY	

Solution = {unstack(C,A), stack(C,B)}



<u>STRIPS Examples:</u>

<u>Air cargo transport:</u>

Init(At(C1, SFO) ∧ At(C2,JFK) ∧ At(P1,SFO) ∧ At(P2,JFK) ∧ Cargo(C1) ∧ Cargo(C2) ∧ Plane(P1) ∧ Plane(P2) ∧ Airport(JFK) ∧ Airport(SFO))

Goal(At(C1,JFK) ∧ At(C2,SFO))

Action(Load(c,p,a)

PRECOND: *At(c,a) ∧At(p,a) ∧Cargo(c) ∧Plane(p) ∧Airport(a)*

EFFECT: *¬At(c,a) ∧In(c,p))*

Action(Unload(c,p,a)

PRECOND: *In(c,p) ∧At(p,a) ∧Cargo(c) ∧Plane(p) ∧Airport(a)*

EFFECT: *At(c,a) ∧ ¬In(c,p))*

Action(Fly(p,from,to)

PRECOND: *At(p,from) ∧Plane(p) ∧Airport(from) ∧Airport(to)*

EFFECT: *¬ At(p,from) ∧ At(p,to))*

[Load(C1,P1,SFO), Fly(P1,SFO,JFK), Load(C2,P2,JFK), Fly(P2,JFK,SFO)]

Spare tire problem

Init(At(Flat, Axle) ∧ At(Spare,trunk))

Goal(At(Spare,Axle))

Action(Remove(Spare,Trunk)

PRECOND: *At(Spare,Trunk)*

EFFECT: *¬At(Spare,Trunk) ∧ At(Spare,Ground))*

Action(Remove(Flat,Axle)

PRECOND: *At(Flat,Axle)*

EFFECT: *¬At(Flat,Axle) ∧ At(Flat,Ground))*

Action(PutOn(Spare,Axle)

PRECOND: *At(Spare,Groundp) ∧¬At(Flat,Axle)*

EFFECT: *At(Spare,Axle) ∧ ¬At(Spare,Ground))*

Action(LeaveOvernight

PRECOND:

EFFECT: *¬ At(Spare,Ground) ∧ ¬ At(Spare,Axle) ∧ ¬ At(Spare,trunk) ∧ ¬ At(Flat,Ground) ∧ ¬ At(Flat,Axle))*

This example goes beyond STRIPS: negative literal in pre-condition (ADL description)

Learning:

- Learning is essential for unknown environments, i.e., when designer lacks omniscience
- Learning is useful as a system construction method, i.e., expose the agent to reality rather than trying to write it down.
- Learning modifies the agent's decision mechanisms to improve performance.
- The agents percepts should be used not only for acting, but also for improving future performance.
- Tasks to learn for an agent: –
    - What state will be the result of an action?
    - How will the changing world evolve?
    - What is the value of each state?
    - Which kind of states has high (low) value?
    - Which percepts are relevant?

Learning task – estimations of functions y=f(x): x → y

Types of Learning:

- Supervised learning:

  Given a value x, f(x) is immediately provided by a "supervisor". f(x) is learned from a number of examples: (x1 ,y1), (x2 ,y2), …, (xn ,yn)

- Reinforcement leaning:

  A correct answer y is not provided for each x. Rather a general evaluation is proved after a sequence of actions (occasional rewards)

- Unsupervised learning:

  The agent learns relationships among its percepts. I.e. it perform clustering.

Machine Learning:

- ■ Learn from past experiences
- ■ Improve the performances of intelligent programs

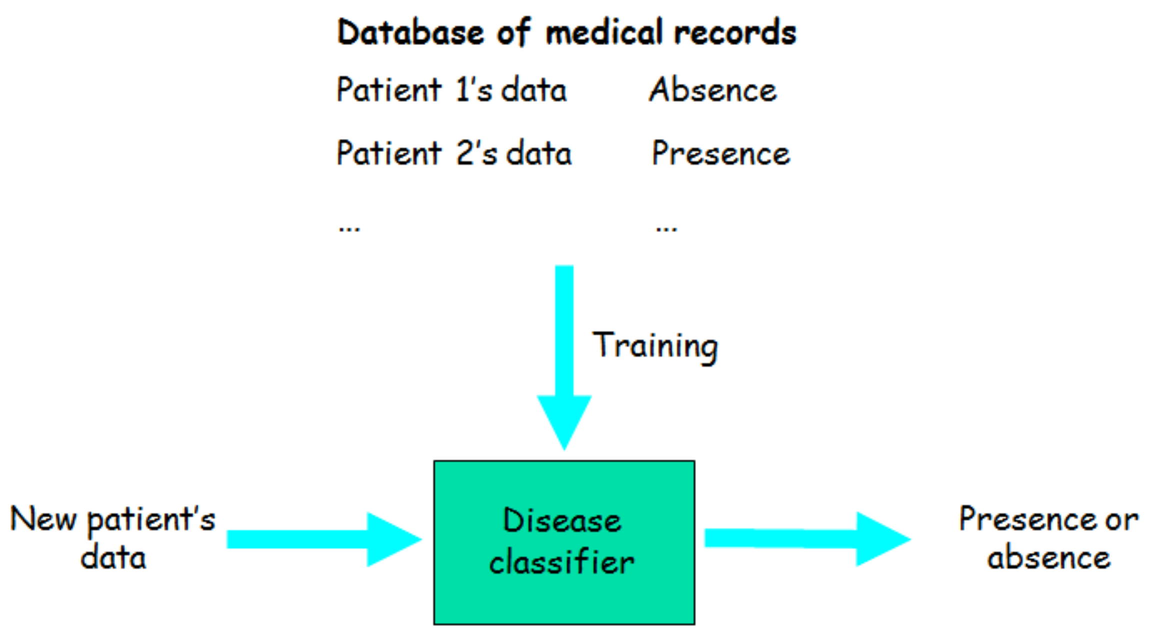
Definitions:

- "Learning denotes changes in a system that ... enable a system to do the same task … more efficiently the next time." - Herbert Simon

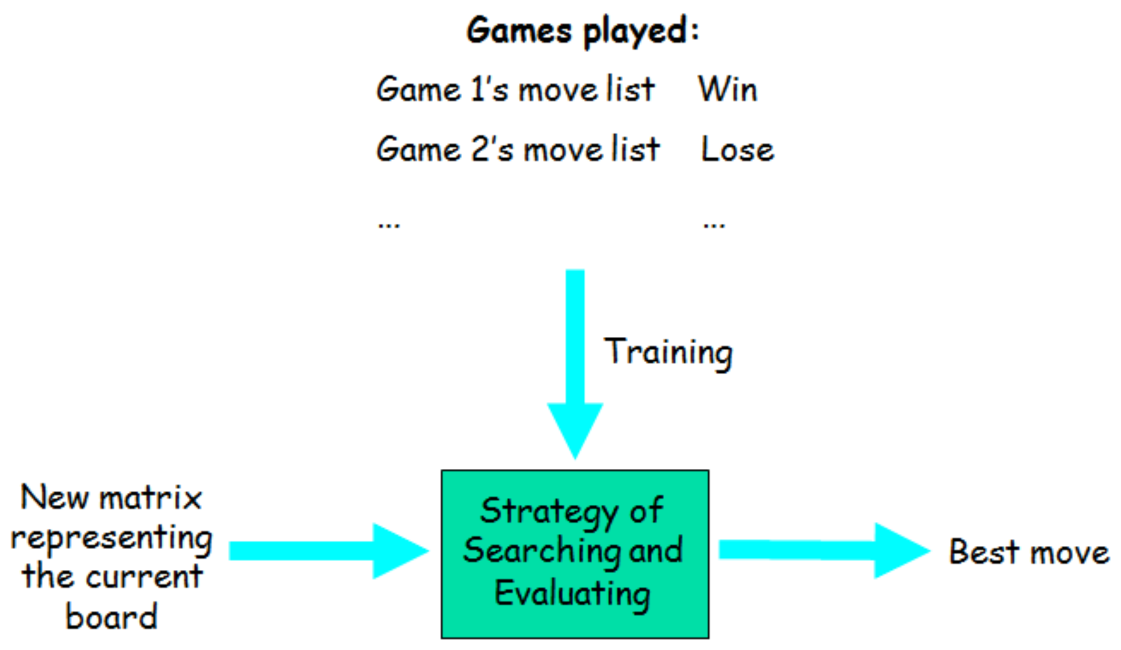
- "Learning is constructing or modifying representations of what is being experienced." - Ryszard Michalski

- "A computer program is said to learn from experience *E* with respect to some class of tasks and performance measure *P*, if its performance at the tasks improves with the experiences"

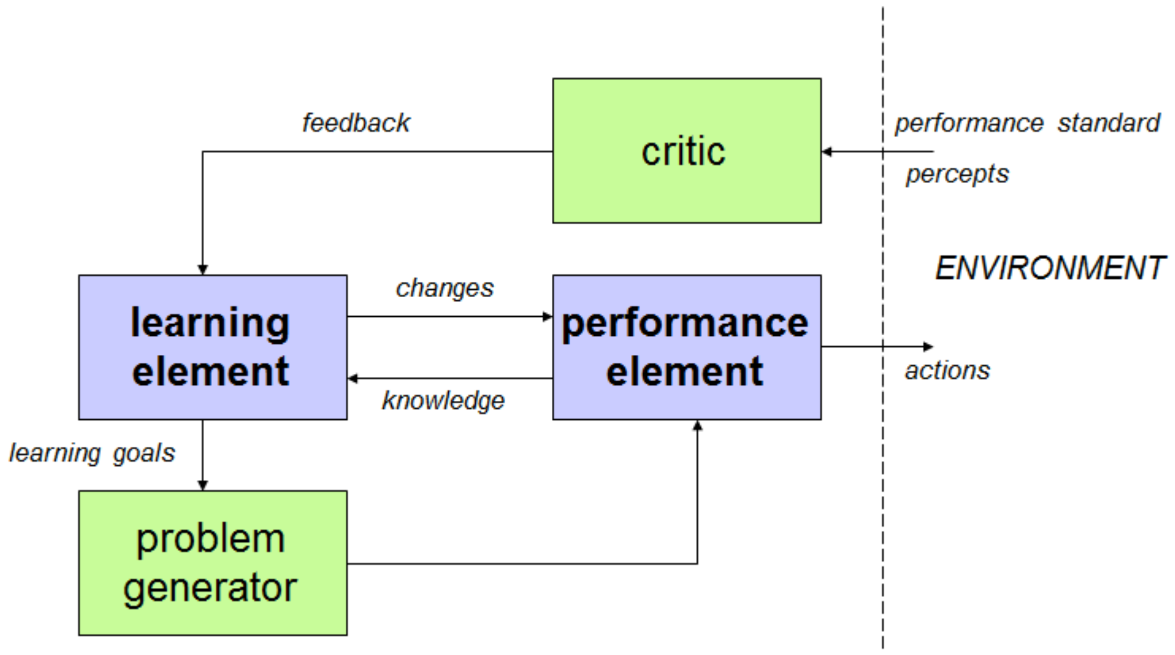
Example 1: Disease Diagnosis



Example 2: Chess Playing



Architecture of a Learning System:



Dimensions of Learning Systems:

- **Type of feedback**
  - **supervised (labeled examples)**
  - **unsupervised (unlabeled examples)**
  - **reinforcement (reward)**
- **Representation**
  - **attribute-based (feature vector)**
  - **relational (first-order logic)**
- **Use of knowledge**
  - **empirical (knowledge-free)**
  - **analytical (knowledge-guided)**