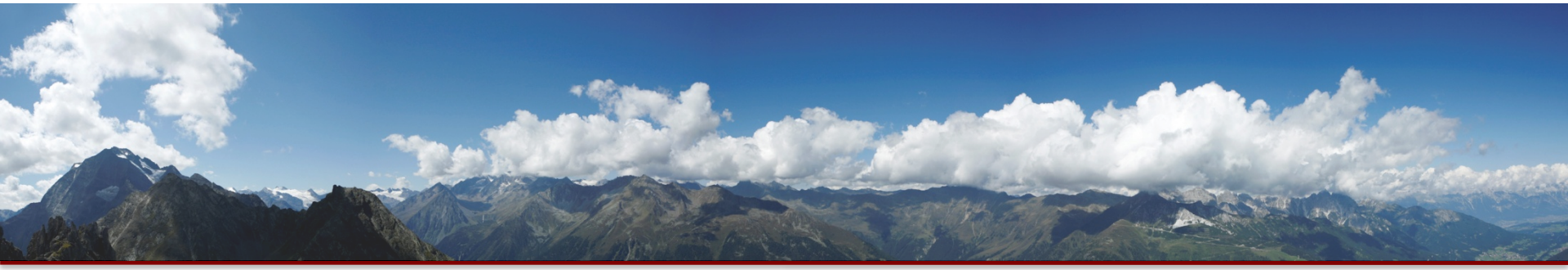
Artificial Intelligence
WS 2015/16

Planning

Anna Fensel



Where are we?

STI · INNSBRUCK

#	Title
1	Introduction
2	Propositional Logic
3	Predicate Logic
4	Reasoning
5	Search Methods
6	CommonKADS
7	Problem-Solving Methods
8	**Planning**
9	Software Agents
10	Rule Learning
11	Inductive Logic Programming
12	Formal Concept Analysis
13	Neural Networks
14	Semantic Web and Services

Agenda

STI · INNSBRUCK

- Motivation
- Technical Solution
- Illustration by a Larger Example
- Extensions
- Summary
- References

MOTIVATION

Motivation

STI · INNSBRUCK

- What is Planning?
  - *"Planning is the process of thinking about the activities required to create a desired goal on some scale"* [Wikipedia]
- We take a more pragmatic view – *planning is a flexible approach for taking complex decisions:*
  - decide about the schedule of a production line;
  - decide about the movements of an elevator;
  - decide about the flow of paper through a copy machine;
  - decide about robot actions.
- By "flexible" we mean:
  - the problem is described to the planning system in some generic language;
  - a (good) solution is found fully automatically;
  - if the problem changes, all that needs to be done is to change the description.
- We look at methods to solve any problem that can be described in the language chosen for the particular planning system.

An ‘easy’ planning example

- **Goal:** Buy 1 liter of milk
- It may sound like a simple task, but if you break it down, there are many small tasks involved:
  - obtain car keys,
  - obtain wallet,
  - exit home,
  - start car,
  - drive to store,
  - find and obtain milk,
  - purchase milk,
  - …






Logistics

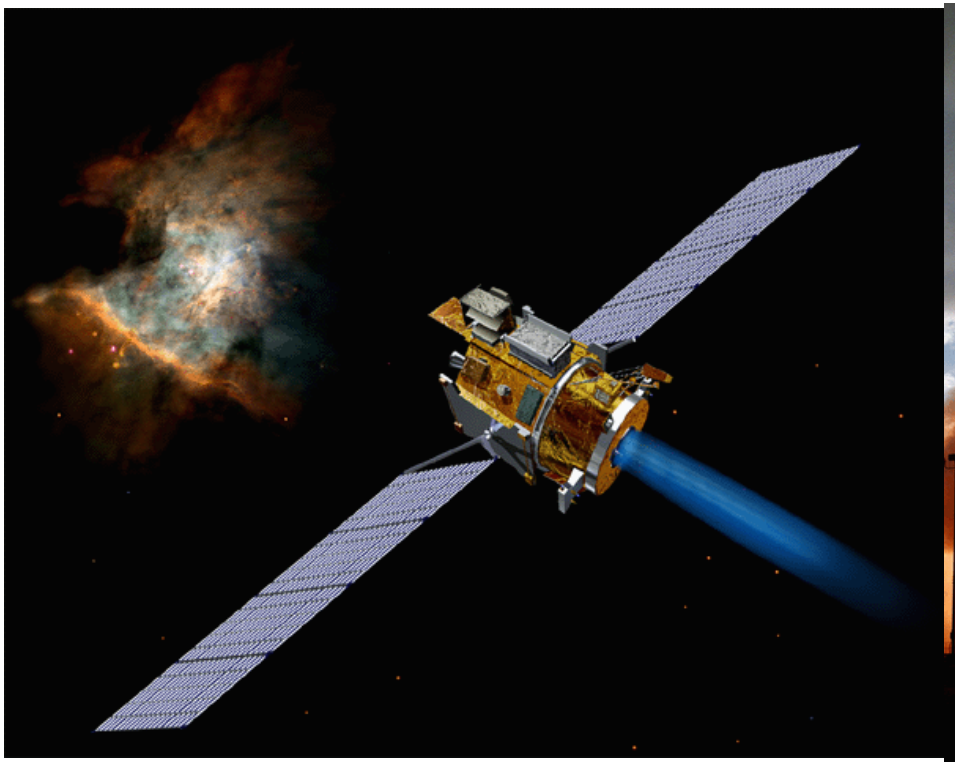
- During the 1991 Gulf War, US forces deployed an AI logistics planning and scheduling program (*Dynamic Analysis and Replanning Tool*) that involved up to 50,000 vehicles, cargo, and people
- This one application alone reportedly more than offset all the money DARPA had funneled into AI research in the last 30 years.

https://en.wikipedia.org/wiki/Dynamic_Analysis_and_Replanning_Tool

Space Exploration

STI · INNSBRUCK

- **Deep Space 1**
- NASA's on-board autonomous planning program controlled the scheduling of operations for a spacecraft



Space Exploration (cont'd)

STI · INNSBRUCK

- **Mars Exploration Rover (MER)**
- Autonomous planning, scheduling, control for Mars Exploration Rover.
- MER uses a combination of planning techniques including:
  - *Mixed-initiative planning* that involves significant human participation
  - *Constraint-based* - quantitative reasoning with metric time and other numerical quantities

TECHNICAL SOLUTIONS

- This lecture will consider planning as state-space search, for which there are several options:
  - **Forward** from I (initial state) until G (goal state) is satisfied;
  - **Backward** from G until I is reached;
  - Use a **Heuristic** function to estimate G- or I-distance, respectively – prefer states that have a lower heuristic value.
- It will also introduce partial-order planning as a more flexible approach
- In all cases there are three implicit concerns to consider:
  - **Representation** – how the problem/state space and goal is defined;
  - **Algorithm** – e.g. which (combination) of these approaches is used;
  - **Complexity** and **decidability** – the feasibility of the approach.

REPRESENTATION

Planning – basic terminology

STI · INNSBRUCK

- A problem to be solved by a planning system is called a **planning problem**
- The world in which planning takes place is often called the **domain** or **application domain.**
- A **state** is a point in the search space of the application
- A planning problem is characterized by an **initial state** and a **goal state**.
- The **initial state** is the state of the world just before the plan is executed
- The **goal state** is the desired state of the world that we want to reach after the plan has been executed. A goal state is also refer as simply **goal**

STRIPS

- Stanford Research Institute Problem Solver (STRIPS) is an automated planner developed by Richard Fikes and Nils Nilsson in 1971 [Nilsson & Fikes, 1970]
- STRIPS is also the name of the planning language used for the STRIPS planner

STRIPS

- A **state** (excluding the goal state) is represented in STRIPS as a conjunction of function-free ground literals

  e.g. on(A, Table) ^ on (B, A) ^ ¬ clear(A) ^ clear(B)

  (block A is on the Table, block B is on block A, block A has something on top, block B has nothing on top)

- A conjunction of function-free ground literals is also know as a **fact**

- A **goal** is represented in STRIPS as a conjunction of literals that may contain variables

  e.g. on(B, Table) ^ on (A, B) ^ clear(A) ^ ¬ clear(B)

  (block B is on the Table, block A is on block B, block B has something on top, block A has nothing on top)

Definition – Action

Let P be a set of facts. An **action** a is a triple $a = (pre(a), add(a), del(a))$ of subsets of P, where $add(a) \setminus del(a) = \emptyset$

$pre(a)$, $add(a)$, and $del(a)$ are called the action *precondition*, *add list*, and *delete list*, respectively

- In other words, an **action** is defined in terms of:
  - ***Preconditions*** - conjunction of literals that need to be satisfiable before the action is performed
  - ***Effects*** - conjunction of literals that need to satisfiable after the action is performed. An effect description includes:
    - **Add List:** list of formulas to be added to the description of the new state resulting after the action is performed
    - **Delete List:** list of formulas to be deleted from the description of state before the action is performed

- The action *pickup* in the Blocks World domain can be modeled as follows in STRIPS

pickup(x)
Preconditions: on(x, Table) ^ handEmpty ^ clear(x)
Add List: holding(x)
Delete List: on(x, Table) ^ handEmpty ^ clear(x)

Definition – Task

A task is a quadruple (P,A,I,G) where P is a (finite) set of facts, A is a (finite) set of actions, and I and G are subsets of P, I being the initial state and G the goal state.

- In other words, a **task** is defined in terms of:
  - A (finite) set of facts
  - A (finite) set of actions that the planner can perform. Each action is defined in terms of preconditions and effects (i.e. add list and delete list)
  - An initial state i.e. the state of the world before the execution of the task
  - A goal state i.e. the desired state of the world that we want to reach

Definition – Result

Let P be a set of facts, $s \subseteq P$ a state, and a an action. The **result** $result(s, \langle a \rangle)$ of applying a to s is:

$result(s, \langle a \rangle) = (s \cup add(a)) \setminus del(a)$ *iff* $pre(a) \subseteq s$; undefined otherwise

In the first case, the action *a* is said to be applicable in *s*. The result of applying an action sequence $\langle a_1, \ldots, a_n \rangle$ of arbitrary length to s is defined by $result(s, \langle a_1, \ldots, a_n \rangle) = result(result(s, \langle a_1, \ldots, a_{n-1} \rangle), \langle a_n \rangle)$ and $result(s, \langle\rangle) = s$.

- In other words, the **result** of applying an action *a* in a given state *s* is a new state of the world that is described by the initial set of formulas describing state *s* to which the formulas from the *add list* are added and the formulas from *delete list* are removed. Action *a* can be performed only if action's *preconditions* are fulfilled in state *s*

- e.g.

  state *s* = on(A, B) ^ handEmpty ^ clear(A)

  action *a* = *unstack*(x,y)

  Preconditions: on(x, y) ^ handEmpty ^ clear(x)

  Add List: holding(x) ^ clear(y)

  Delete List: on(x, y) ^ handEmpty ^ clear(x)

$result(s, \langle a \rangle) = (s \cup add(a)) \setminus del(a)$

$= (on(A, B) \wedge handEmpty \wedge clear(A) \cup holding(x) \wedge clear(y)) \setminus on(x, y) \wedge handEmpty \wedge clear(x)$

$= holding(A) \wedge clear(B)$

Precondition $on(x, y) \wedge handEmpty \wedge clear(x)$ is fulfilled in state $s = on(A, B) \wedge handEmpty \wedge clear(A)$

For convenience let's denote $result(s, \langle a \rangle)$ with $s1$

Given a second action *a1*

action *a1* = *stack*(x,y)

Preconditions: hoding(x) ^ clear(y)

Add List: on(x,y) ^ handEmpty ^ clear(x)

Delete List: holding(x) ^ clear(y)

$$result(s, \langle a, a1 \rangle) = result(result(s, \langle a \rangle), \langle a1 \rangle) = result(s1, \langle a1 \rangle) = s1 \cup add(a1)) \setminus del(a1)$$

$$= ((holding(A) \wedge clear(B) \cup holding(x) \wedge on(x,y) \wedge handEmpty \wedge clear(x)) \setminus holding(x) \wedge clear(y)$$

$$= on(A,B) \wedge handEmpty \wedge clear(A)$$

Precondition $hoding(x) \wedge clear(y)$ is fulfilled in state $s1 = holding(A) \wedge clear(B)$

Definition – Plan

Let (P,A,I,G) be a task. An action sequence $\langle a_1, \ldots, a_n \rangle$ is a plan for the task iff $G \subseteq \text{result}(I, \langle a_1, \ldots, a_n \rangle)$.

- In other words, a **plan** is an organized collection of actions.
- A plan is said to be a **solution** to a given problem if the plan is applicable in the problem's initial state, and if after plan execution, the goal is true.
- Assume that there is some action in the plan that must be executed first. The plan is applicable if all the preconditions for the execution of this first action hold in the initial state.

- A task is called **solvable** if there is a plan, **unsolvable** otherwise

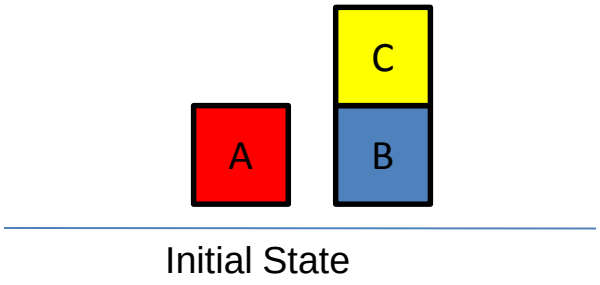
Definition – Minimal Plan

Let (P,A,I,G) be a task. A plan $\langle a_1, \ldots, a_n \rangle$ for the task is **minimal** if $\langle a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n \rangle$ is **not** a plan for any i.

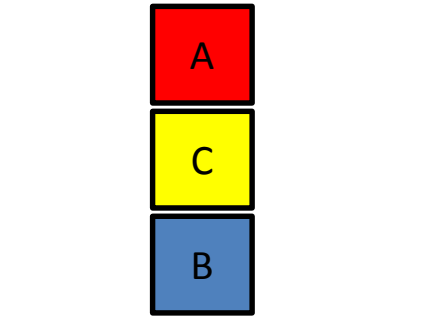
Definition – Optimal Plan

Let (P,A,I,G) be a task. A plan $\langle a_1, \ldots, a_n \rangle$ for the task is **optimal** if there is **no** plan with less than n actions

- Example: “The Block World”
  - Domain:
    - Set of cubic blocks sitting on a table
  - Actions:
    - Blocks can be stacked
    - Can pick up a block and move it to another location
    - Can only pick up one block at a time
  - Goal:
    - To build a specified stack of blocks

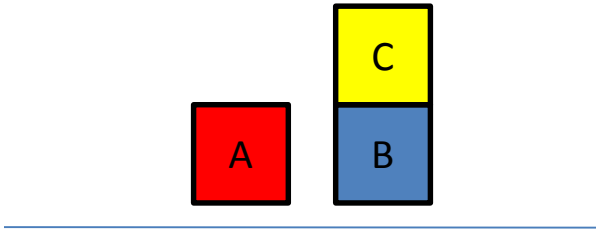


Initial State

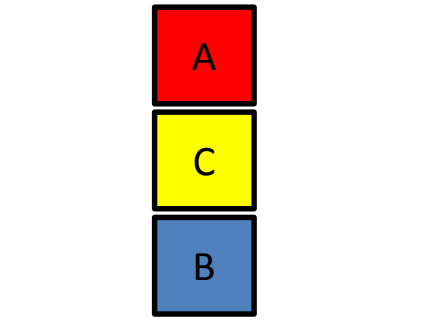


Goal State

- Representing states and goals
  - Initial State:
    - on(A, Table) ^
    - on(B, Table) ^
    - on(C, B) ^
    - clear(A) ^
    - clear(C) ^
    - handEmpty
  - Goal:
    - on(B, Table) ^
    - on(C, B) ^
    - on(A, C) ^
    - clear(A) ^
    - handEmpty



Initial State



Goal State

- Actions in “Block World”
  - pickup(x)
    - picks up block ‘*x’ from table*
  - *putdown(x)*
    - *if holding block ‘x’, puts it down on table*
  - *stack(x,y)*
    - *if holding block ‘x’, puts it on top of block ‘y’*
  - *unstack(x,y)*
    - *picks up block ‘x’ that is currently on block ‘y’*
- Action *stack(x,y)* in STRIPS

  stack(x,y)

  Preconditions: hoding(x) ^ clear(y)

  Add List: on(x,y) ^ handEmpty ^ clear(x)

  Delete List: holding(x) ^ clear(y)

PDDL

STI · INNSBRUCK

- Planning Domain Description Language (PDDL)
- Based on STRIPS with various extensions
- Created, in its first version, by Drew McDermott and others
- Used in the biennial International Planning Competition (IPC) series
- The representation is lifted, i.e., makes use of variables these are instantiated with objects
- **Actions** are instantiated **operators**
- **Facts** are instantiated **predicates**
- A task is specified via two files: the **domain file** and the **problem file**
- The **problem file** gives the objects, the initial state, and the goal state
- The **domain file** gives the predicates and the operators; these may be re-used for different problem files
- The **domain file** corresponds to the transition system, the **problem files** constitute instances in that system

Blocks World Example **domain file**:

```
(define (domain blocksworld)
(:predicates (clear ?x)
(holding ?x)
(on ?x ?y))
(    :action stack
     :parameters (?ob ?underob)
     :precondition (and (clear ?underob) (holding ?ob))
     :effect (and (holding nil) (on ?ob ?underob)
                    (not (clear ?underob)) (not (holding ?ob)))
)
...
```

Blocks World Example **problem file**:

```
(define (problem bw-xy)
    (:domain blocksworld)
    (:objects nil table a b c d e)
    (:init (on a table) (clear a)
           (on b table) (clear b)
           (on e table) (clear e)
           (on c table) (on d c) (clear d)
           (holding nil)
    )
    (:goal (and (on e c) (on c a) (on b d))))
```

ALGORITHMS

- We now present particular algorithms related to (direct) state-space search:
  - **Progression**-based search also know as **forward search**
  - **Regression**-based search also know as **backward search**
  - Heuristic Functions and **Informed Search Algorithms**

STI · INNSBRUCK

Definition – Search Scheme

A **search scheme** is a tuple $(S, s_0, succ, Sol)$:

1. the set S of all states $s \in S$,
2. the start state $s_0 \in S$,
3. the successor state function $succ : S \rightarrow 2^S$, and
4. the solution states $Sol \subseteq S$.

Note:

- Solution paths $s_0 \rightarrow, \ldots, \rightarrow s_n \in Sol$ correspond to solutions to our problem

STI · INNSBRUCK

Definition – Progression

Let (P,A,I,G) be a task. Progression is the quadruple $(S, s_0, succ, Sol)$:

- $S = 2^P$ is the set of all subsets of P,
- $s_0 = I$,
- $succ : S \rightarrow 2^S$ is defined by $succ(s) = \{s_0 \in S \mid \exists a \in A: pre(a) \subseteq s, s_0 = result(s, \langle a \rangle)\}$
- $Sol = \{s \in S \mid G \subseteq s\}$

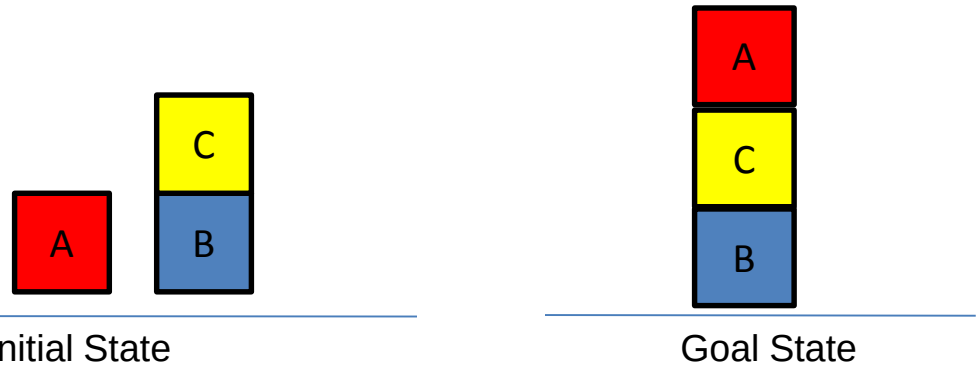
Progression algorithm:

1. Start from initial state
2. Find all actions whose preconditions are true in the initial state (applicable actions)
3. Compute effects of actions to generate successor states
4. Repeat steps 2-3, until a new state satisfies the goal

Progression explores only states that are reachable from I; they may not be relevant for G

STI · INNSBRUCK

Progression in “Blocks World“ – applying the progression algorithm



Step 1:

Initial state $s_0 = I$ = on(A, Table) ^ on(B, Table) ^ on(C, B) ^ clear(A) ^ clear(C) ^ handEmpty

Step 2:

Applicable actions: $\text{unstack}(C,B)$, $\text{pickup}(A)$

Step 3:

$\text{result}(s_0, \langle \text{unstack}(C,B) \rangle)$ = on(A, Table) ^ on(B, Table) ^ on(C, B) ^ clear(A) ^ clear(C) ^ handEmpty $\cup$holding(x) ^ clear(y) \ on(x, y) ^ handEmpty ^ clear(x) = on(A, Table) ^ on(B, Table) ^ clear(A) ^ clear(B) ^ holding(C)

$\text{result}(s_0, \langle \text{pickup}(A) \rangle)$ = on(A, Table) ^ on(B, Table) ^ on(C, B) ^ clear(A) ^ clear(C) ^ handEmpty $\cup$holding(x) \ on(x, Table) ^ handEmpty ^ clear(x) = on(B, Table) ^ on(C, B) ^ clear(C) ^ holding(A)

STI · INNSBRUCK

Definition – Regression

Let P be a set of facts, $s \subseteq P$, and a an action. The regression $regress(s, a)$ of s through a is:

$regress(s, a) = (s \setminus add(a)) \cup pre(a)$ if $add(a) \cap s \neq \emptyset$, $del(a) \cap s = \emptyset$; undefined otherwise

In the first case, s is said to be regressable through a.

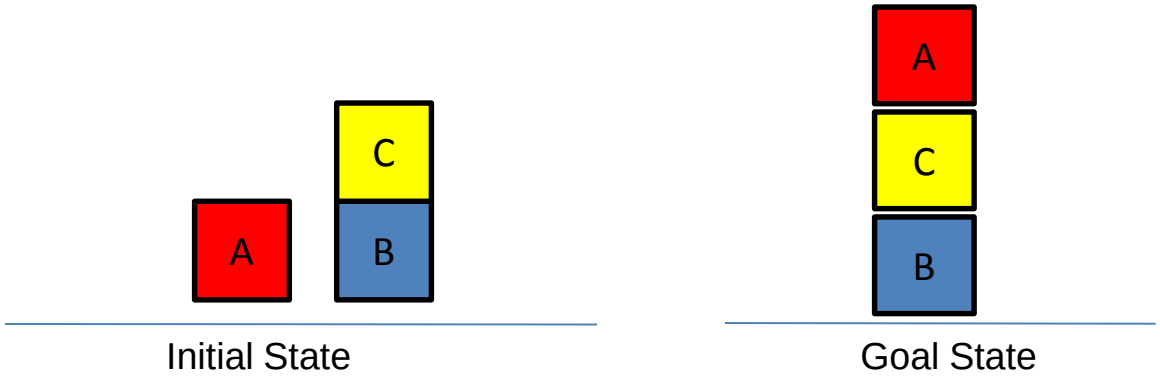
- If $s \setminus add(a) = \emptyset$, then a contributes nothing
- If $s \setminus del(a) \neq \emptyset$, then we can't use a to achieve $\bigwedge_{p \in s} p$

Regression algorithm:

1. Start with the goal
2. Choose an action that will result in the goal
3. Replace that goal with the action's preconditions
4. Repeat steps 2-3 until you have reached the initial state

- Regression explores only states that are relevant for G; they may not be reachable from I
- Regression is not as non-natural as you might think: if you plan a trip to Thailand, do you start with thinking about the train to Frankfurt?

Regression in “Blocks World“ – applying the regression algorithm



Step 1:

Goal state s_n = G = on(B, Table) ^ on(C, B) ^ on(A, C) ^ clear(A) ^ handEmpty

Step 2:

Applicable actions: stack(A,C)

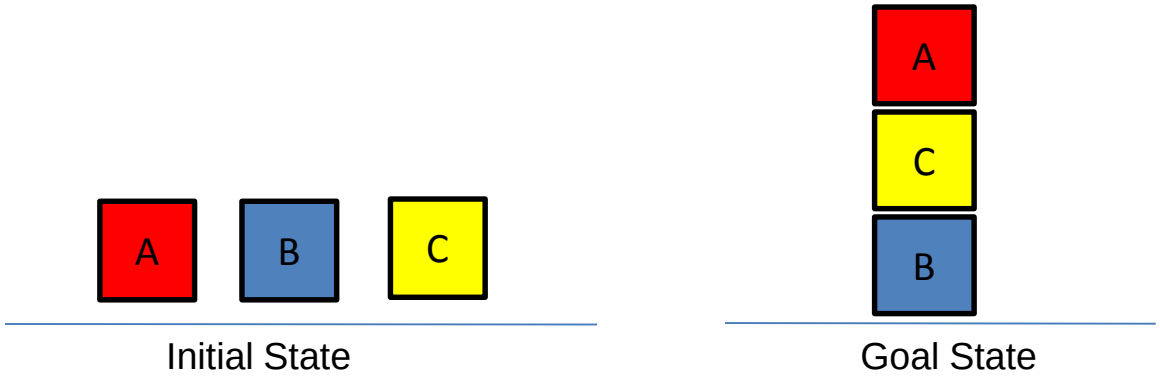
Step 3:

regress(G, stack(A,C)) = $G \setminus add(stack) \cup pre(stack)$ = on(B, Table) ^ on(C, B) ^ on(A, C) ^ clear(A) ^ handEmpty \ on(A,C) ^ handEmpty ^ clear(A) ∪ hoding(A) ^ clear(C) = on(B, Table) ^ on(C, B) ^ holding(A) ^ clear(C)

STI · INNSBRUCK

Definition – Heuristic Function

Let $(S, s_0, succ, Sol)$ be a search scheme. A heuristic function is a function $h : S \rightarrow N_0 \cup \{\infty\}$ from states into the natural numbers including 0 and the ∞ symbol. **Heuristic functions**, or **heuristics**, are efficiently computable functions that estimate a state's "solution distance".



A heuristic function for Blocks World domain could be:

h1(s) = n – sum(b(i)), *i=1…n* , where

- *b(i)=0* in state *s* if the block *i* is resting on a wrong thing
- *b(i)=1* in state *s* if the block *i* is resting on the thing it is supposed to be resting on;
- *n* is the total number of blocks

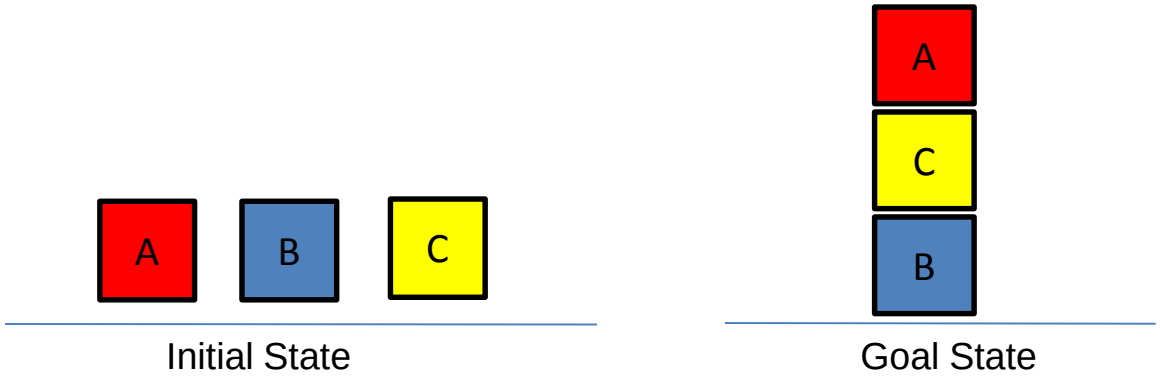
h1(I) = 3 – 1 (because of block B) – 0 (because of block C) – 0 (because of block A) = 2

h1(G) = 3 – 1 (because of block B) – 1 (because of block C) – 1 (because of block A) = 0

STI · INNSBRUCK

Definition – Solution Distance

Let $(S, s_0, succ, Sol)$ be a search scheme. The solution distance $sd(s)$ of a state $s \in S$ is the length of a shortest $succ$ path from s to a state in Sol, or 1 if there is no such path. Computing the real solution distance is as hard as solving the problem itself.



Heuristic Functions

STI · INNSBRUCK

In Progression search:

sd(I) = 4, sd(result(I, ⟨pickup(C)⟩)) = 3, sd(result(result(I, ⟨pickup(C)⟩)), ⟨stack(C,B)⟩) = 2, … for all other $s \in succ(I)$

In Regression search:

sd(G) = 4, sd(regress(G, stack (A,C))) = 3, sd(regress(regress(G, stack (A,C))), pickup(A)) = 2, …

STI · INNSBRUCK

Definition – Admissible Heuristic Function

Let $(S, s_0, succ, Sol)$ be a search scheme. A heuristic function h is admissible if $h(s) \leq sd(s)$ for all $s \in S$.

- In other words, an **admissible heuristic function** is a heuristic function that never overestimates the actual cost i.e. the solution distance.

- e.g. h1 heuristic function defined below is admissible

  $h1(s) = n - sum(b(i))$, $i=1 \ldots n$, where

  - $b(i)=0$ in state s if the block i is resting on a wrong thing
  - $b(i)=1$ in state s if the block i is resting on the thing it is supposed to be resting on;
  - n is the total number of blocks

COMPLEXITY AND DECIDABILITY

- Its *time complexity*: in the worst case, how many search states are expanded?
- Its *space complexity*: in the worst case, how many search states are kept in the open list at any point in time?
- Is it *complete*, i.e. is it guaranteed to find a solution if there is one?
- Is it *optimal*, i.e. is it guaranteed to find an optimal solution?

Definition - PLANSAT

Let PLANSAT denote the following decision problem. Given a STRIPS task (P, A, I, G), is the task solvable?

Theorem

PLANSAT is PSPACE-complete. Proof in see [Bylander, 1994]

Definition – Bounded-PLANSAT

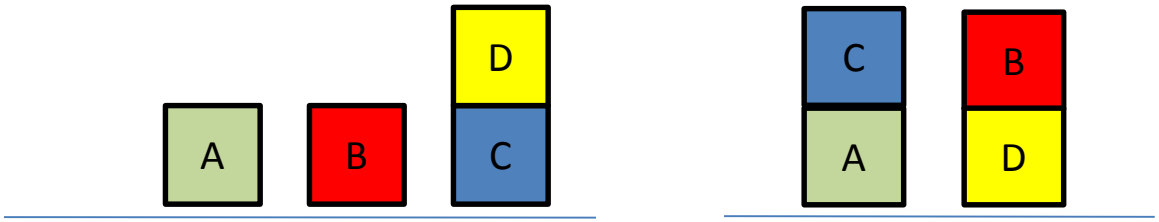
Let Bounded-PLANSAT denote the following decision problem. Given a STRIPS task *(P, A, I, G)* and an integer ***b***. Is there a plan with at most *b* actions?

ILLUSTRATION BY A LARGER EXAMPLE

PROGRESSION

STI · INNSBRUCK

Progression in “Blocks World“ – applying the progression algorithm



1st iteration Initial State Goal State

Step1:

Initial state s_0 = I = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(B) ^ clear(D) ^ handEmpty

Step 2:

Applicable actions: pickup(A), pickup(B), unstack(D,C)

Step 3:

s_{11} = result(s_0, ⟨pickup(A)⟩) = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(B) ^ clear(D) ^ handEmpty ∪holding(x) \ on(x, Table) ^ handEmpty ^ clear(x) = on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(B) ^ clear(D) ^ holding(A)

$s_{12} = \text{result}(s_0, \langle \text{pickup}(B) \rangle) =$ on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(B) ^ clear(D) ^ handEmpty $\cup$ holding(x) \ on(x, Table) ^ handEmpty ^ clear(x) = on(A, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(D) ^ holding(B)

$s_{13} = \text{result}(s_0, \langle \text{unstack}(D,C) \rangle) =$ on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(B) ^ clear(D) ^ handEmpty $\cup$ holding(x) ^ clear(y) \ on(x, y) ^ handEmpty ^ clear(x) = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ clear(A) ^ clear(B) ^ clear(C) ^ holding(D)

STI · INNSBRUCK

2nd iteration

Step1:

s_{11} = on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(B) ^ clear(D) ^ holding(A)

Step 2:

Applicable actions: **putdown (A)**, stack(A,D)

Step 3:

$s_{111} = \text{result}(s_{11}, \langle \text{putdown (A)} \rangle)$ = on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(B) ^ clear(D) ^ holding(A) $\cup$ on(x, Table) ^ handEmpty ^ clear(x) \ holding(x) = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(B) ^ clear(D) ^ handEmpty

we are back in the initial state!

STI · INNSBRUCK

3rd iteration

Step1:

s_{11} = on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(B) ^ clear(D) ^ holding(A)

Step 2:

Applicable actions: putdown (A), **stack(A,D)**

Step 3:

$s_{112} = \text{result}(s_{11}, \langle \text{stack (A,D)} \rangle)$ = on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(B) ^ clear(D) ^ holding(A) $\cup$ on(x, y) ^ handEmpty ^ clear(x) \ holding(x) ^ clear(y)= on(B, Table) ^ on(C, Table) ^ on(D, C) ^ on(A,D) ^ handEmpty ^ clear(A) ^ clear(B)

STI · INNSBRUCK

4th iteration

Step1:

s_{12} = on(A, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(D) ^ holding(B)

Step 2:

Applicable actions: **putdown (B)**, stack(B,D)

Step 3:

$s_{121} = \text{result}(s_{12}, \langle \text{putdown (B)} \rangle)$ = on(A, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(D) ^ holding(B) $\cup$ on(x, Table) ^ handEmpty ^ clear(x) \ holding(x) = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(B) ^ clear(D) ^ handEmpty

we are back in the initial state!

STI · INNSBRUCK

5th iteration

Step1:

s_{12} = on(A, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(D) ^ holding(B)

Step 2:

Applicable actions: putdown (B), **stack(B,D)**

Step 3:

$s_{122} = \text{result}(s_{11}, \langle \text{stack (B,D)} \rangle)$ = on(A, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(D) ^ holding(B) $\cup$ on(x, y) ^ handEmpty ^ clear(x) \ holding(x) ^ clear(y)= on(A, Table) ^ on(C, Table) ^ on(D, C) ^ on(B,D) ^ handEmpty ^ clear(A) ^ clear(B)

STI · INNSBRUCK

6th iteration

Step1:

s_{13} = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ clear(A) ^ clear(B) ^ clear(C) ^ holding(D)

Step 2:

Applicable actions: **putdown(D)**, stack(D,A), stack(D,B), stack(D,C)

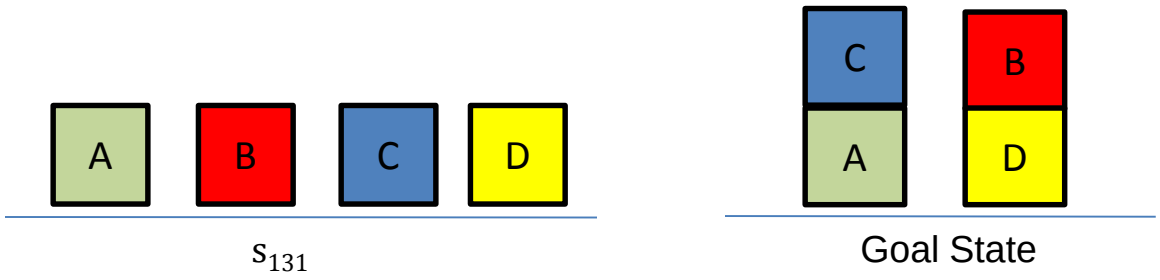
Step 3:

$s_{131} = \text{result}(s_{13}, \langle \text{putdown(D)} \rangle)$ = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ clear(A) ^ clear(B) ^ clear(C) ^ holding(D) $\cup$ on(x, Table) ^ handEmpty ^ clear(x) \ holding(x) = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, Table) ^ clear(A) ^ clear(B) ^ clear(C) ^ clear(D) ^ handEmpty

In iterations 7,8,9, new states are obtained from s_{13} by applying the actions stack(D,A), stack(D,B), stack(D,C)

STI · INNSBRUCK

10th iteration



Step1:

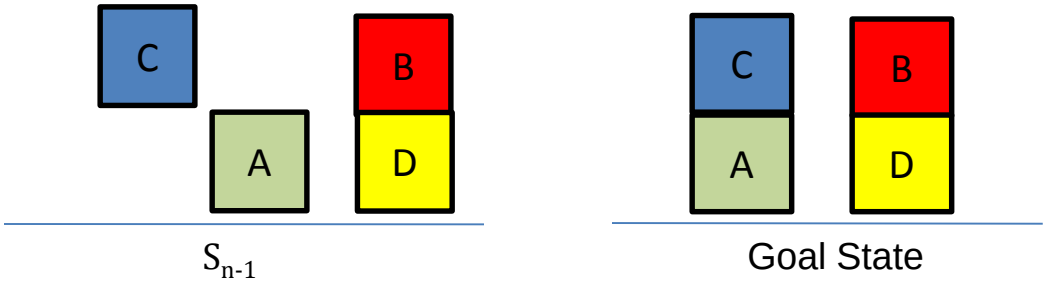
s_{131} = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, Table) ^ clear(A) ^ clear(B) ^ clear(C) ^ clear(D) ^ handEmpty

Step 2:

Applicable actions: pickup(A), pickup(B), **pickup(C)**, pickup(D)

Step 3:

$s_{1311} = \text{result}(s_{131}, \langle \text{putdown}(C) \rangle)$ = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, Table) ^ clear(A) ^ clear(B) ^ clear(C) ^ clear(D) ^ handEmpty $\cup$ holding(x) \ on(x, Table) ^ clear(x) ^ handEmpty = on(A, Table) ^ on(B, Table) ^ on(D, Table) ^ clear(A) ^ clear(B) ^ clear(D) ^ holding(C)



n^{th} iteration

Step1:

s_{n-1} = on(A, Table) ^ on(D, Table) ^ on(B,C) ^ clear(A) ^ clear(B) ^ holding(C)

Step 2:

Applicable actions: putdown(C), **stack(C,A)**, stack(C,B)

Step 3:

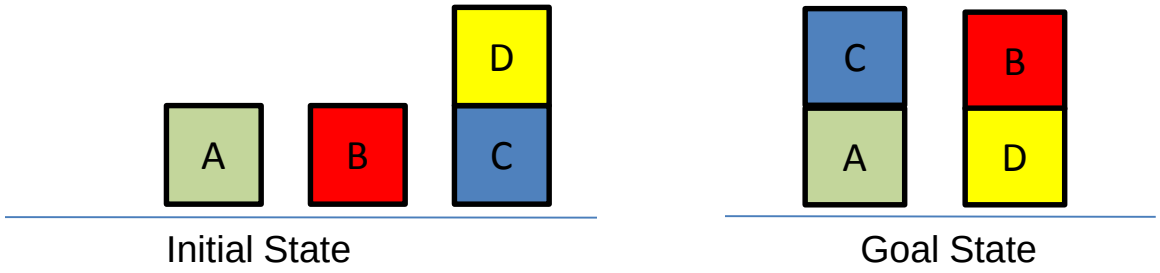
$s_n = \text{result}(s_{n-1}, \langle \text{stack(C,A)} \rangle)$ = on(A, Table) ^ on(D, Table) ^ on(B,C) ^ clear(A) ^ clear(B) ^ holding(C) $\cup$ on(x,y) ^ clear(x) ^ handEmpty \ clear(y) ^ holding(x) = on(A, Table) ^ on(D, Table) ^ on(C, A) ^ on(B,D) ^ clear(C) ^ clear(B) ^ handEmpty

s_n is the goal state G

REGRESSION

STI · INNSBRUCK

Regression in “Blocks World“ – applying the regression algorithm



1st iteration

Step 1:

Goal state s_n = G = on(A, Table) ^ on(D, Table) ^ on(C, A) ^ on(B,D) ^ clear(C) ^ clear(B) ^ handEmpty

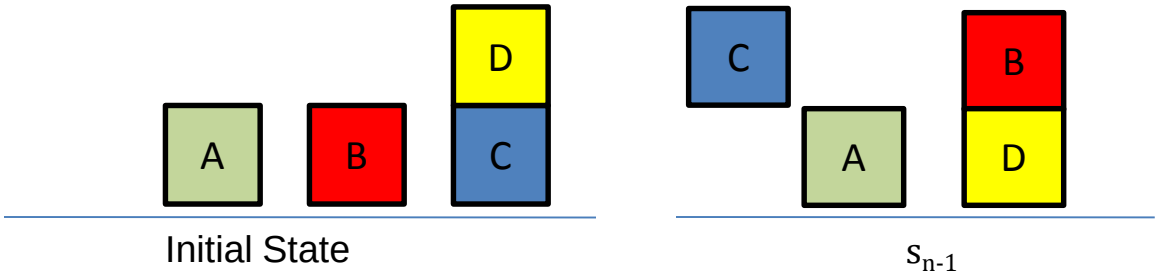
Step 2:

Applicable actions: stack(C,A), stack(B,D)

Step 3:

s_{n-1}=regress(G, stack(C,A)) = $G \mid add(stack) \cup pre(stack)$ = on(A, Table) ^ on(D, Table) ^ on(C, A) ^ on(B,D) ^ clear(C) ^ clear(B) ^ handEmpty \ on(C,A) ^ handEmpty ^ clear(C) ∪holding(C) ^ clear(A) = on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear(A) ^ holding(C)

s_{n-2}=regress(G, stack(B,D)) = $G \mid add(stack) \cup pre(stack)$ = on(A, Table) ^ on(D, Table) ^ on(C, A) ^ on(B,D) ^ clear(C) ^ clear(B) ^ handEmpty \ on(B,D) ^ handEmpty ^ clear(B) ∪holding(B) ^ clear(D) = on(A, Table) ^ on(D, Table) ^ on(C, A) ^ clear(C) ^ clear(D) ^ holding(B)



2nd iteration

Step 1:

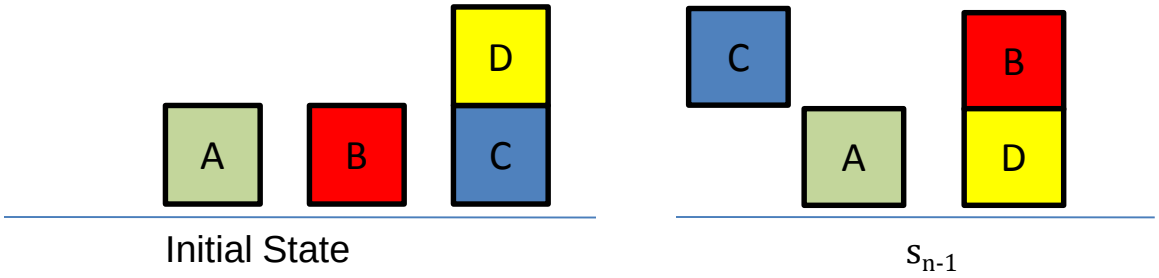
s_{n-1} = on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear(A) ^ holding(C)

Step 2:

Applicable actions: **pickup(C)**, unstack(C,A)

Step 3:

s_{n-2}=regress(s_{n-1}, pickup(C)) = s_{n-1} \ *add(pickup)* ∪ *pre(pickup)* =on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear(A) ^ holding(C) \ holding(C) ∪clear(C) ^ on(C, Table) ^ handEmpty = on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear(A) ^ clear (C) ^ on(C, Table) ^ handEmpty



3rd iteration

Step 1:

s_{n-1} = on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear(A) ^ holding(C)

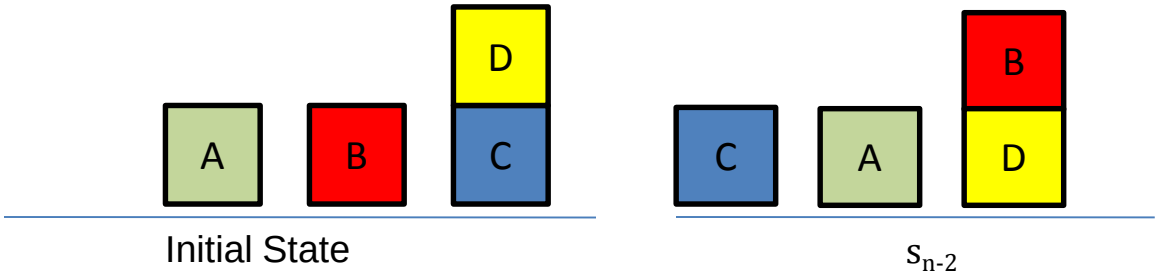
Step 2:

Applicable actions: pickup(C), **unstack(C,A)**

Step 3:

s_{n-3}=regress(s_{n-1}, unstack(C,A)) = s_{n-1} \ *add(unstack)* ∪ *pre(unstack)* = on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear(A) ^ holding(C) \ holding(C) ^ clear(A) ∪clear(C) ^ on(C, A) ^ handEmpty = on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear (C) ^ on(C, A)

we are back in the goal state!



4th iteration

Step 1:

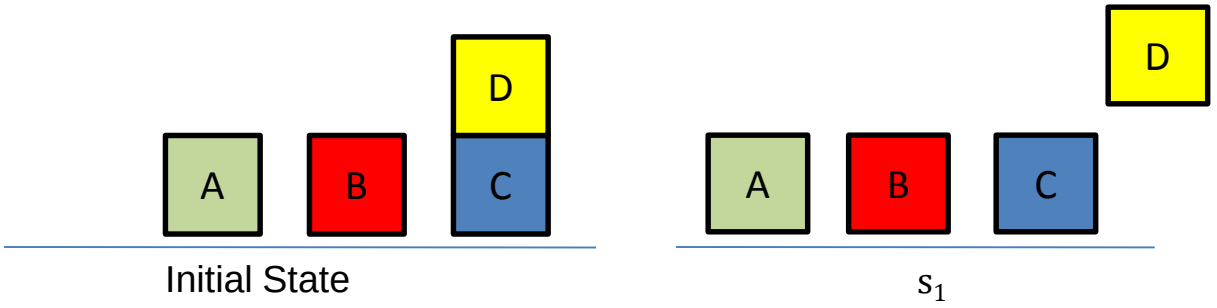
s_{n-2} = on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear(A) ^ clear (C) ^ on(C, Table) ^ handEmpty

Step 2:

Applicable actions: putdown(C), putdown(A), **stack(B,D)**

Step 3:

s_{n-4}=regress(s_{n-2}, stack(B,D)) = s_{n-2} \ *add(stack)* ∪ *pre(stack)* = on(A, Table) ^ on(D, Table) ^ on(B, D) ^ clear(B) ^ clear(A) ^ clear (C) ^ on(C, Table) ^ handEmpty \ on(B,D) ^ clear(B) ^ handEmpty ∪clear(D) ^ holding(B) = on(A, Table) ^ on(D, Table) ^ on(C, Table) ^ clear(A) ^ clear (C) ^ clear(D) ^ holding(B)



n^{th} iteration

Step 1:

s_1 = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ clear(A) ^ clear(B) ^ clear (C) ^ holding(D)

Step 2:

Applicable actions: pickup(D), **unstack(D,C)**

Step 3:

s_0=regress(s_1, unstack(D,C)) = s_1 \ *add(unstack)* $\cup$ *pre(unstack)* =on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ clear(A) ^ clear(B) ^ clear (C) ^ holding(D) \ holding(D) ^ clear(C) $\cup$on(D,C) ^ clear(D) ^ handEmpty = on(A, Table) ^ on(B, Table) ^ on(C, Table) ^ on(D, C) ^ clear(A) ^ clear(B) ^ clear(D) ^ handEmpty

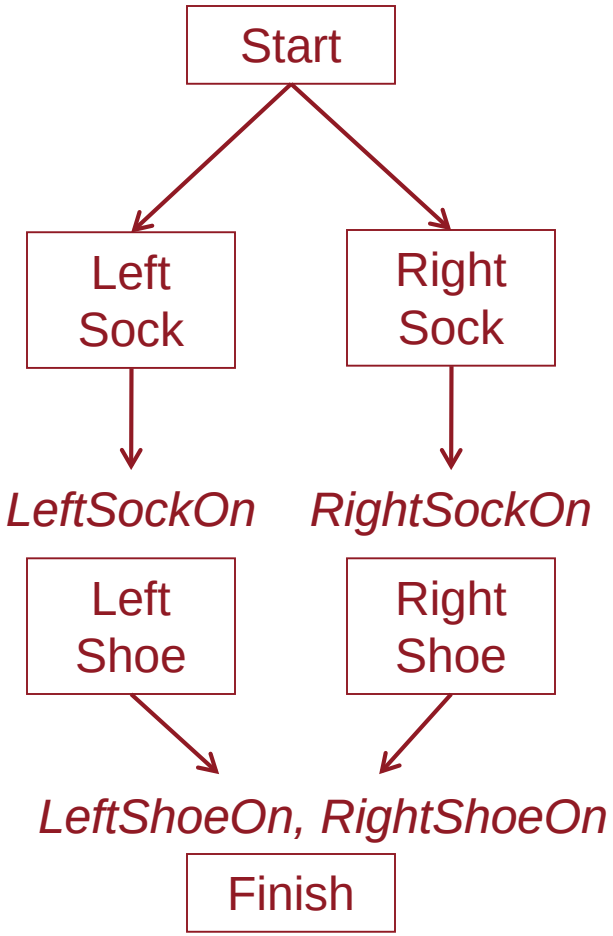
EXTENSIONS

- Forward and backward state-space search are both forms of *totally-ordered* plan search – explore linear sequences of actions from initial state or goal.
- As such they cannot accomodate *problem decomposition* and work on subgraphs separately.
- A *partial-order planner* depends on:
  - A set of **ordering constraints**, wherein $A \prec B$ means that A is executed some time before B (not necessarily directly before)
  - A set of **causal links**, A –p-> B, meaning A achieves p for B
  - A set of **open preconditions** not achieved by any action in the current plan

Partial-Order Plans

STI · INNSBRUCK

- Partial Order Plan:



- Total Order Plans:

- Compared with a classical search procedure, viewing a problem as one of constraint satisfaction can reduce substantially the amount of search.
- Constrain-based search involved complex numerical or symbolic variable and state constraints

Constraint-Based Search:

1. Constraints are discovered and propagated as far as possible.
2. If there is still not a solution, then search begins, adding new constraints.

- C*onstrain* the task by a time horizon, b
- Formulate this as a *constraint satisfaction problem* and use *backtracking* methods to solve it: “do *x* at time *t* yes/no”, “do *y* at time *t’* yes/no”,
- If there is no solution, increment b

- Inside *backtracking*: constraint propagation, conflict analysis are used
- Constraint-Based Searcg is an “undirected” search i.e. there is no fixed time-order to the decisions done by backtracking

SUMMARY

Planning Summary

STI · INNSBRUCK

- Planning is a flexible approach for taking complex decisions:
  - the problem is described to the planning system in some generic language;
  - a (good) solution is found fully automatically;
  - if the problem changes, all that needs to be done is to change the description.
- Following concerns must be addressed:
  - Syntax - we have examined representations in PDDL and STRIPS
  - Semantics – we have examined progression, regression and the use of heuristics in algorithms
  - Complexity and decidability

REFERENCES

References

STI · INNSBRUCK

- Mandatory reading:
  - D. McDermott. “PDDL – the planning domain definition language”, Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
  - N. Nilsson and R. Fikes. “STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving”, SRI Technical Note 43, 1970. http://www.ai.sri.com/pubs/files/tn043r-fikes71.pdf
- Further reading:
  - S. Russell and P. Norvig. “AI: A Modern Approach” (2nd Edition), Prentice Hall, 2002
  - G. Malik; D.S. Nau, P. Traverso (2004), *Automated Planning: Theory and Practice, Morgan Kaufmann, ISBN 1-55860-856-7*
  - T .Bylander,The Computationa lComplexity of Propositional STRIPS Planning, Artificial Intelligence Journal, 1994
- Wikipedia links:
  - http://en.wikipedia.org/wiki/Automated_planning_and_scheduling
  - http://en.wikipedia.org/wiki/STRIPS
  - http://en.wikipedia.org/wiki/Hierarchical_task_network
  - http://en.wikipedia.org/wiki/Planning_Domain_Definition_Language

Next Lecture

STI · INNSBRUCK

#	Title
1	Introduction
2	Propositional Logic
3	Predicate Logic
4	Reasoning
5	Search Methods
6	CommonKADS
7	Problem-Solving Methods
8	Planning
9	**Software Agents**
10	Rule Learning
11	Inductive Logic Programming
12	Formal Concept Analysis
13	Neural Networks
14	Semantic Web and Services

Questions?