Classical Planning

Radek Mařík

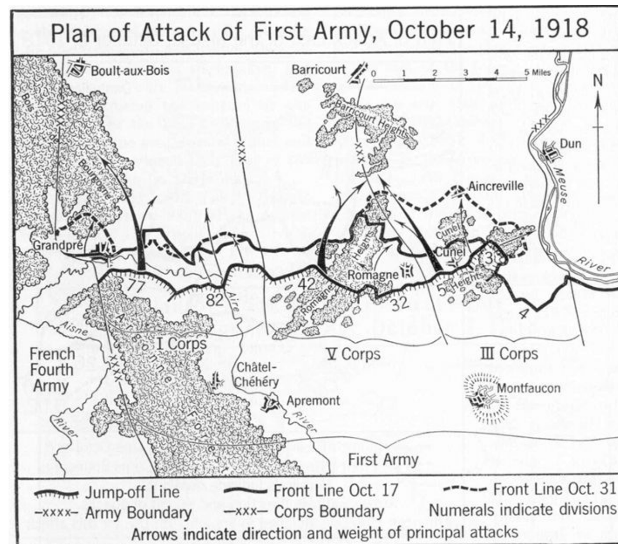
CVUT FEL, K13132

16. dubna 2014

Content

1. Concept of AI Planning
   - Definition
   - Conceptual Model
   - Methodology of Planners
2. Representation
   - STRIPS
   - PDDL
3. Planning Methods
   - Logics and Searching
   - State Space
   - Plan Space
   - Planning Graphs

Concept of Plan [Nau09]



Plan

- many definitions and aspects ....
- A scheme, program, or method worked out beforehand for the accomplishment of an objective.

Plan Definition [Nau09, Pec10]

Planning

- Reasoning about about hypothetical interaction among the agent and the environment with respect to a given task.
- Motivation of the planning process is to reason about possible course of actions that will change the environment in order to reach the goal (task)

Planning and Scheduling [Nau09]

- **Scheduling** . . . assigns in time resources to separate processes,
- **Planning** . . . considers possible interaction among components of plan.

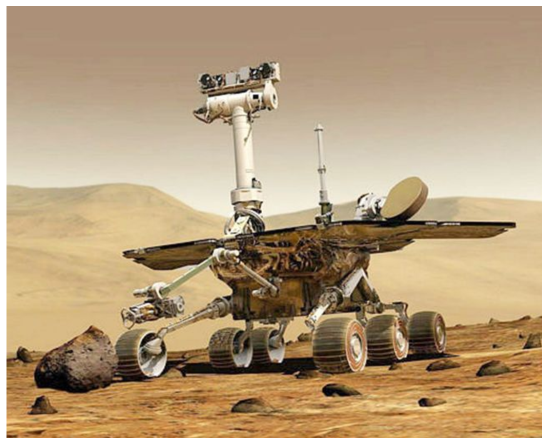
Planning

- Given: the initial state, goal state, operators.
- Find a sequence of operators that will reach the goal state from the initial state
  - Select appropriate actions, arrange the actions and consider the causalities

Scheduling

- Given: resources, actions and constraints.
- Form an appropriate schedule that meets the constraints
  - Arrange the actions, assign resources and satisfy the constrains.

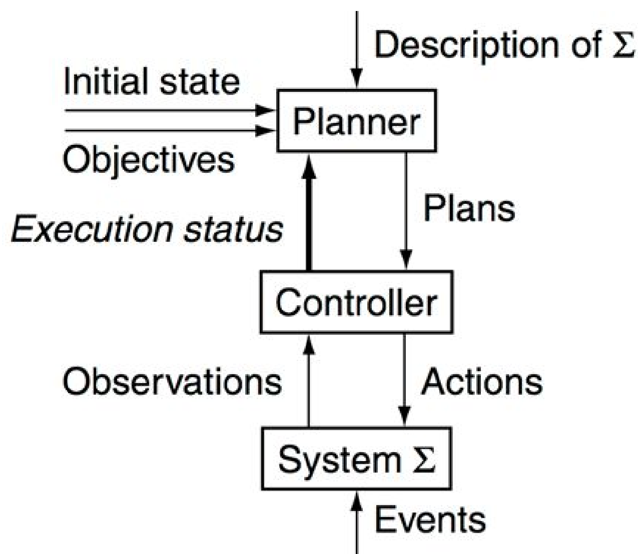
Real Applications - Space Exploration [Nau09]



Projects

- Autonomous planning, scheduling, control
  - NASA: JPL and Ames.
- Remote Agent Experiment (REX)
  - Deep Space 1.
- Mars Exploration Rover (MER)

Conceptual Model of Planning I [Nau09]



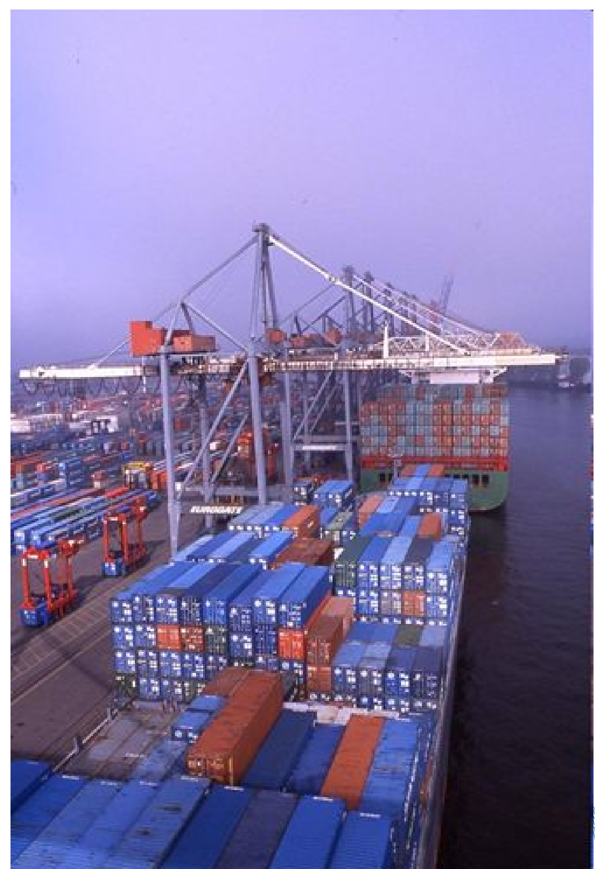
Environment

- State transition system
- $\Sigma = (S, A, E, \gamma)$
- $S = \{\text{states}\}$
- $A = \{\text{actions}\}$
- $E = \{\text{exogenous events}\}$
- $\gamma = \{\text{state-transition function}\}$

The Dock-Worker Robots (DWR) Domain [Wic11]

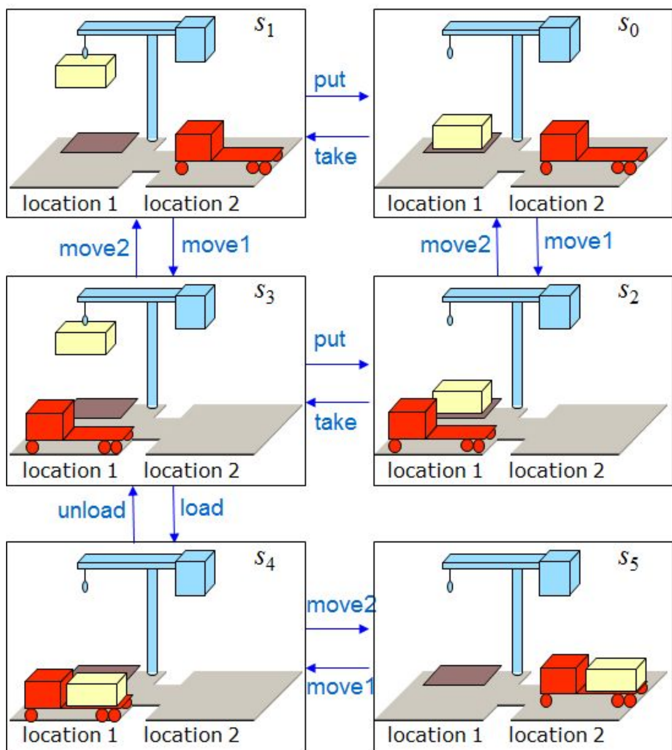
Planning procedure illustration

- harbour with several locations (docks),
- docked ships,
- storage areas for containers,
- parking areas for
  - trains,
  - trucks
- Goal:
  - cranes to load and unload ships.
  - robot carts to move containers around



Port of Hamburg

State Transition System Example [Nau09]



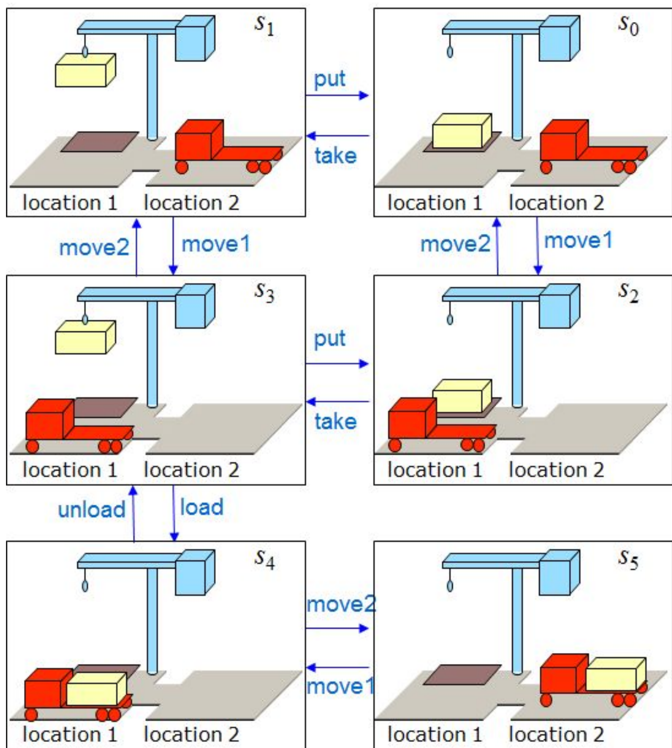
State Trainsition System

- $\Sigma = (S, A, E, \gamma)$
- $S = \{\text{states}\}$
- $A = \{\text{actions}\}$
- $E = \{\text{exogenous events}\}$
- $\gamma = S \times (A \cup E) \rightarrow 2^S$

System Instance

- $S = \{s_0, s_1, \dots, s_5\}$
- $A = \{move_1, move_2, put, take, load, unload\}$
- $E = \{\}$
- $\gamma = \{\text{see arrows}\}$

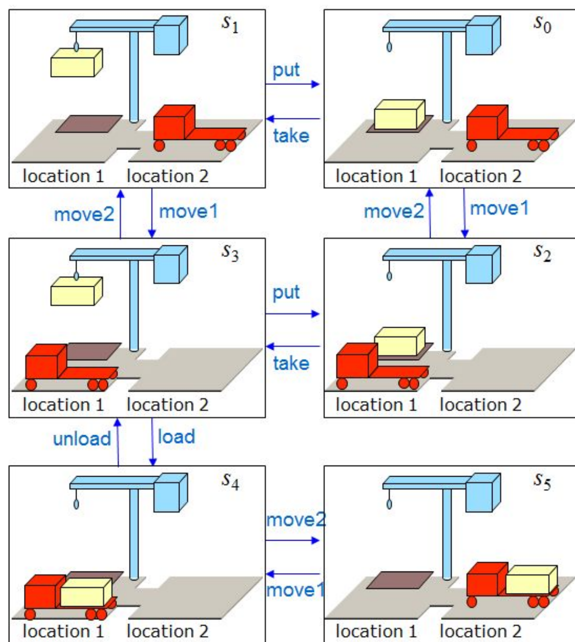
Planning Task [Nau09]



Planning problem

- System description Σ
- Initial state or set of states
  - Initial state $= s_0$
- Objective
  - Goal state,
  - set of goal states,
  - set of tasks,
  - "trajectory" of states,
  - objective function
  - Goal state $= s_5$

Plan [Nau09]



Classical plan

- a sequence of actions
- $< take, move_1, load, move_2 >$

Policy:

- partial function from S into A
- $\{(s_0, take), (s_1, move_1), (s_3, load), (s_4, move_2)\}$

Types of Planners [Nau09]

Domain-specific

- Made or tuned for a specific domain
- Won't work well (if at all) in any other domain
- Most successful real-world planning systems work this way

Domain-independent

- Works in any planning domain, in principle,
- Uses no domain-specific knowledge except the definitions of the basic actions
- In practice, not feasible to develop domain-independent planners that work in every possible domain,
- Make simplifying assumptions to restrict the set of domains
  - *Classical planning*
  - Historical focus of most automated-planning research

Restrictive Assumptions [Nau09]

- **A0: Finite system**
  - finitely many states, actions, events
- **A1: Fully observable**
  - the controller always Σ's current state
- **A2: Deterministic**
  - each action has only one outcome
- **A3: Static** (no exogenous events)
  - no changes but the controller's actions
- **A4: Attainment goals**
  - existency a set of goal states S_g
- **A5: Sequential plans**
  - a plan is a linearly ordered sequence of actions $< a_0, a_1, \ldots, a_n >$
- **A6: Implicit time**
  - no time durations; linear sequence of instantaneous states
- **A7: Off-line planning**
  - planner doesn't know the execution status

Classical Planning [Nau09]

- Requires all 8 restrictive assumptions
  - Offline generation of action sequences for a deterministic, static, finite system, with complete knowledge, attainment goals, and implicit time.
- Reduces to the following problem:
  - Given (Σ, s_0, S_g)
  - Find a sequence of actions $\pi =< a_0, a_1, \ldots, a_n >$, that produces a sequence of state transitions $< s_0, s_1, \ldots, s_n >$ such that $s_n \in S_g$.
- This is just path-searching in a graph
  - Nodes = states
  - Edges = actions
- **Is this trivial?**

Classical Planning - example [Nau09]

Cargo Transportation by Planes

- 10 airports
- 50 aircrafts
- 200 pieces of cargo
- number of states $10^{50} \times (50+10)^{200} \approx 10^{405}$
- minimum number of actions $50 \times 9 = 450$
  all cargo located on airpots with no planes
- maximum number of actions $50^{200} \times 9 \approx 10^{340}$
  all cargo and aircrafts in one airport

Reality

- The number of particles in the universe is about 10^{87}

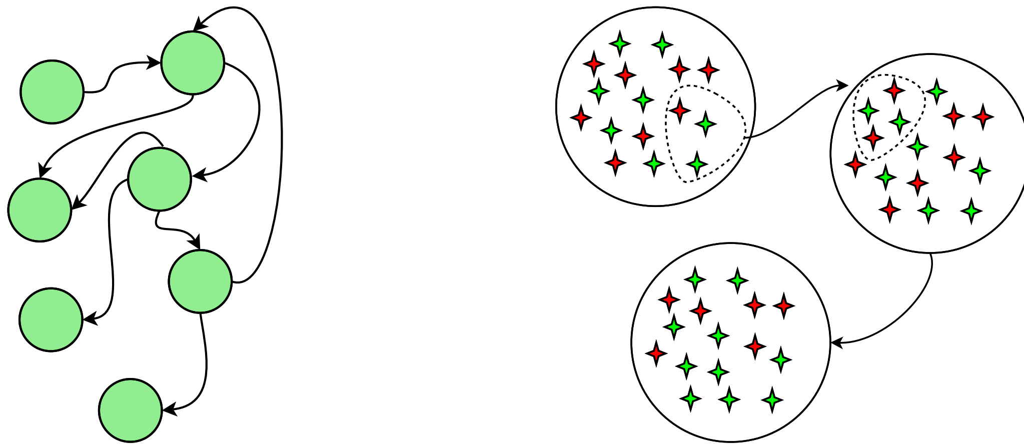
Automated planning research

- classical planning mostly
- dozens (hundreds) of different aglorithms

Classical Representatons [Wic11]

- **Planning as Theorem Proving**
  - world state is a set of propositions
  - actions contains applicability conditions as a set of formulas and effects in a form of formulas added or removed if a given action is applied,
- **STRIPS representation**
  - similar to the propositional representation
  - literals of the first order are used instead of propositions
- **a representation using state variables**
  - state is k-tuple of state variables $\{x_1, \ldots, x_k\}$
  - action is a partial function over states

Factored State Representation



World State Representation

- atomic . . . state is a single indivisible entity
- factored . . . state is a collection of variables

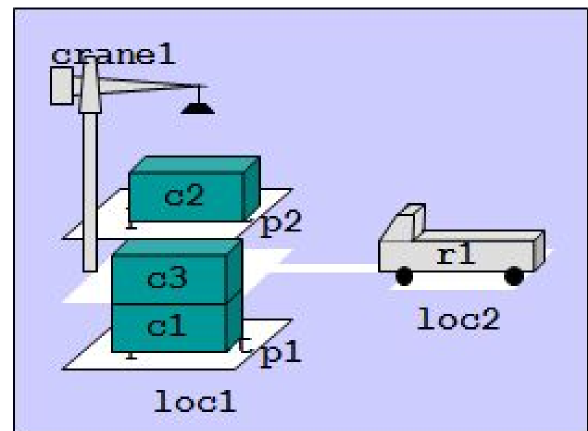
STRIPS - state representation [Wic11]

- Let $\mathcal{L}$ be a first-order language
  - with finitely many predicate symbols,
  - with finitely many constant symbols,
  - and no function symbols.
- **A state in a STRIPS planning domain** is a set of ground atoms of $\mathcal{L}$:
  - (ground) atom p holds in state $s \Leftrightarrow p \in s$
  - s satisfies a set of (ground) literals g ($s \models g$) if
    - every positive literal in g is in s
    - every negative literal in g is not in s

STRIPS State: example [Wic11]

State in DWR Domain

$$state = \{attached(p_1, loc_1), attached(p_2, loc_1), in(c_1, p_1), in(c_3, p_1), top(c_3, p_1), on(c_3, c_1), on(c_1, pallet), in(c_2, p_2), top(c_2, p_2), on(c_2, pallet), belong(crane_1, loc_1), empty(crane_1), adjacent(loc_1, loc_2), adjacent(loc_2, loc_1), at(r_1, loc_2), occupied(loc_2), unloaded(r_1)\}$$



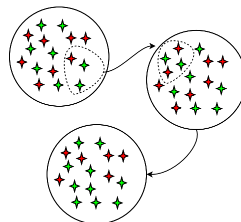
STRIPS - Operator and Action Representations [Wic11]

- **A planning operator** in a STRIPS planning domain is a triple
  - $o = (name(o), precond(o), effects(o))$,
    - the name of the operator $name(o)$
    - is a syntactic expression of the form $n(x_1, \ldots, x_k)$,
    - where n is a (unique) symbol
    - and $x_1, \ldots, x_k$ are all the variables,
    - that appear in o, and
  - the preconditions $precond(o)$ and the effects $effects(o)$ of the operator are sets of literals.
- **An action** in a STRIPS planning domain is a ground instance of a planning operator.

STRIPS Operator: example [Wic11]

- $move(r, l, m)$
  - robot r moves from location l to neighboring location m
  - **precond:** $adjacent(l, m), at(r, l), \neg occupied(m)$
  - **effects:** $at(r, m), occupied(m), \neg occupied(l), \neg at(r, l)$
- $load(k, l, c, r)$
  - crane k in location l loads container c on robot r
  - **precond:** $belong(k, l), holding(k, c), at(r, l), unloaded(r)$
  - **effects:** $empty(k), \neg holding(k, c), loaded(r, c), \neg unloaded(r)$
- $put(k, l, c, d, p)$
  - crane k in location l puts container c onto d in pile p
  - **precond:** $belong(k, l), attached(p, l), holding(k, c), top(d, p)$
  - **effects:** $\neg holding(k, c), empty(k), in(c, p), top(c, p), on(c, d), \neg top(d, p)$

Applicability and State Transitions [Wic11]



- Let L be a set of literals
  - L^+ is the set of atoms that are positive literals in L,
  - L^- is the set of all atoms whose negations are in L
- Let a be an action and s a state.
- Then a is **applicable** in $s \Leftrightarrow$:
  - $precond^+(a) \subseteq s$; and
  - $precond^-(a) \cap s == \{\}$
- The state transition function γ for an applicable action a in state s is defined as:
  - $\gamma(s, a) = (s - effects^-(a)) \cup effects^+(a)$

STRIPS: Planning Domain [Wic11]

- Let $\mathcal{L}$ be a function-free first-order language.
- **STRIPS planning domain** on $\mathcal{L}$ is a restricted state-transition system $\Sigma = (S, A, \gamma)$ such that:
  - S is a set of STRIPS states, i.e. sets of ground atoms,
  - A is a set of ground instances of some STRIPSplanning operators O
  - $\gamma : S \times A \to S$ where
    - $\gamma(s, a) = (s - \mathit{effects}^-(a)) \cup \mathit{effects}^+(a)$ if a is applicable in s
    - $\gamma(s, a) =$ undefined otherwise
  - S is closed under γ

STRIPS: Planning Problem [Wic11]

- **A STRIPS planning problem** is a triple $\mathcal{P} = (\Sigma, s_i, g)$ where:
  - $\Sigma = (S, A, \gamma)$ is a STRIPS planning domain on some first-order language $\mathcal{L}$
  - $s_i \in S$ is the initial state
  - g is a set of ground literals describing the goal such that the set of goal states is $S_g = \{s \in S | s \models g\}$

STRIPS Planning Problem: DWR Example [Wic11]

DWR planning problem

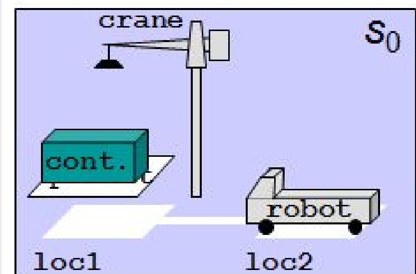
- Σ: STRIPS planning domain DWR
- s_i: any state

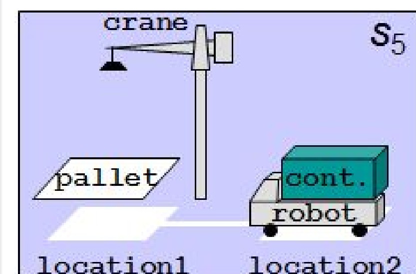
$$s_0 = \{attached(pile, loc_1), in(cont, pile), top(cont, pile), on(cont, pallet), belong(crane, loc_1), empty(crane), adjacent(loc_1, loc_2), adjacent(loc_2, loc_1), at(robot, loc_2), occupied(loc_2), unloaded(robot)\}$$

- $g \subset L$

$$g = \{\neg unloaded(robot), at(robot, loc_2)\}$$

tj. $S_g = \{s_5\}$





Overview of PDDL [Wic11]

Planning Domain Definition Language (PDDL)

- http://cs-www.cs.yale.edu/homes/dvm/
- language features (verze 1.x):
  - basic STRIPS-style actions
  - various extensions as explicit requirements
- used to define:
  - planning domains:
    - requirements,
    - types,
    - predicates,
    - possible actions.
  - planning problems:
    - objects,
    - rigid and fluent relations,
    - initial situation,
    - goal description.
- the current version is 3.x

Monkey Planning Domain

```
(define (domain MONKEY)
  (:requirements :strips :typing)
  (:types monkey box location fruit)
  (:predicates
    (isClear ?b - box)                (onBox ?m - monkey ?b - box)
    (onFloor ?m - monkey)             (atM ?m - monkey ?loc - location)
    (atB ?b - box     ?loc - location)   (atF ?f - fruit  ?loc - location)
    (hasFruit ?m - monkey ?fruit))
  (:action GOTO
    :parameters (?m - monkey ?loc1 ?loc2 - location)
    :precondition (and (onFloor ?m) (atM ?m ?loc1))
    :effect (and (atM ?m ?loc2) (not (atM ?m ?loc1))))
  (:action PUSH
    :parameters (?m - monkey ?b - box ?loc1 ?loc2 - location)
    :precondition (and (onFloor ?m) (atM ?m ?loc1) (atB ?b ?loc1) (isClear ?b))
    :effect (and (atM ?m ?loc2) (atB ?b ?loc2)
                 (not (atM ?m ?loc1))
                 (not (atB ?b ?loc1))))
  (:action CLIMB
    :parameters (?m - monkey ?b - box ?loc1 - location)
    :precondition (and (onFloor ?m) (atM ?m ?loc1) (atB ?b ?loc1) (isClear ?b))
    :effect (and (onBox ?m ?b) (not (isClear ?b)) (not (onFloor ?m)))  )
  (:action GRAB-FRUIT
    :parameters (?m - monkey ?b - box ?f - fruit ?loc1 - location)
    :precondition (and (onBox ?m ?b) (atB ?b ?loc1) (atF ?f ?loc1))
    :effect (and (hasFruit ?m ?f))))
```

Monkey Planning Problem

```
(define (problem MONKEY1)
  (:domain MONKEY)
  (:objects monkeyJudy - monkey
            bananas - fruit
            boxA - box
            locX locY locZ  - location)
  (:init (and
    (onFloor monkeyJudy)
    (atM monkeyJudy locX)
    (atB boxA locY)
    (atF bananas locZ)
    (isClear boxA)
    )
  )

  (:goal (and (hasFruit monkeyJudy bananas))
)
```

)

Monkey Planning Problem Solution

```
Begin plan
1 (goto monkeyjudy locx locy)
2 (push monkeyjudy boxa locy locz)
3 (climb monkeyjudy boxa locz)
4 (grab-fruit monkeyjudy boxa bananas locz)
End plan
```

STRIPS Representation - Example

- initial state: *start*
- *goal: writeValue(a, 3)*
- *action:*

```
startPlan:     PAR []
               PRE [start]
               ADD [declared(server, a), declared(server, b)]
               DEL [start]
connectServer: PAR [A]
               PRE [declared(server, A)]
               ADD [bound(server,A)]
               DEL [declared(server, A)]
writing:       PAR [a, 3]
               PRE [bound(server,a)]
               ADD [writeValue(a, 3)]
               DEL []
```

Example Result

- goal: *writeValue(a,3);*
- *result sequence (plan):*
  1. *start*
  2. *connectServer(a)*
  3. *writing(a, 3)*

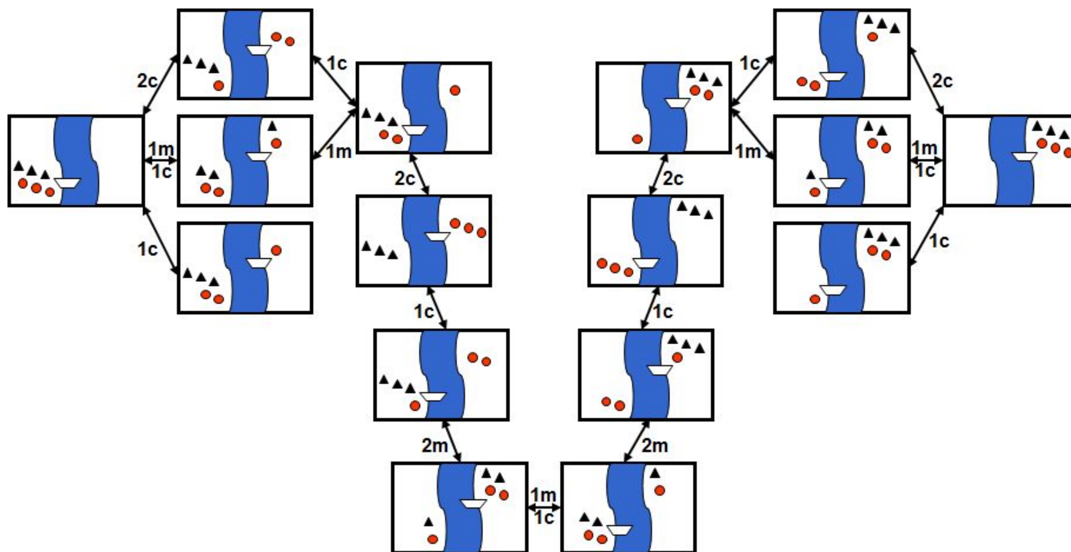
Planning - depth-first search through logical formulea

```
% Depth first search
% ==================
depthFirstSearch(AnsPath) :-
  initialState(Init),
  depthFirst([Init], AnsPath).

depthFirst([S|_], [S]) :-
  finalState(S), !.
depthFirst([S|Path], [S|AnsPath]) :-
  extend([S|Path], S1),
  depthFirst([S1, S |Path], AnsPath).

extend([S|Path], S1) :-
  nextState(S, S1),
  not(memberState(S1, [S|Path])).

memberState(S, Path) :-
  member(S,Path).
```

Planning - a problem specification in Prolog

```
% Farmer, Wolf, Goat, Cabbage
% ===========================
initialState([n,n,n,n]).
finalState([s,s,s,s]).

nextState(S, S1) :-  move(S, S1), safe(S1).

move([F, W, G, C], [F1,  W,  G,  C]) :- cross(F, F1).
move([F, F, G, C], [F1, F1,  G,  C]) :- cross(F, F1).
move([F, W, F, C], [F1,  W, F1,  C]) :- cross(F, F1).
move([F, W, G, F], [F1,  W,  G, F1]) :- cross(F, F1).

safe([F, W, G, C]) :- F=G, !; F=W, F=C.

cross(n,s).
cross(s,n).
%-------
t1(AnsPath) :- depthFirstSearch(AnsPath).
```

State-Transition Graph Example [Wic11]



Game of Missionaries and Cannibals

- move 3 cannibals and 3 missionaries across the river
- Whenever the number of cannibals is higher than the number of missionaries somewhere, the missionaries are cooked and eaten.

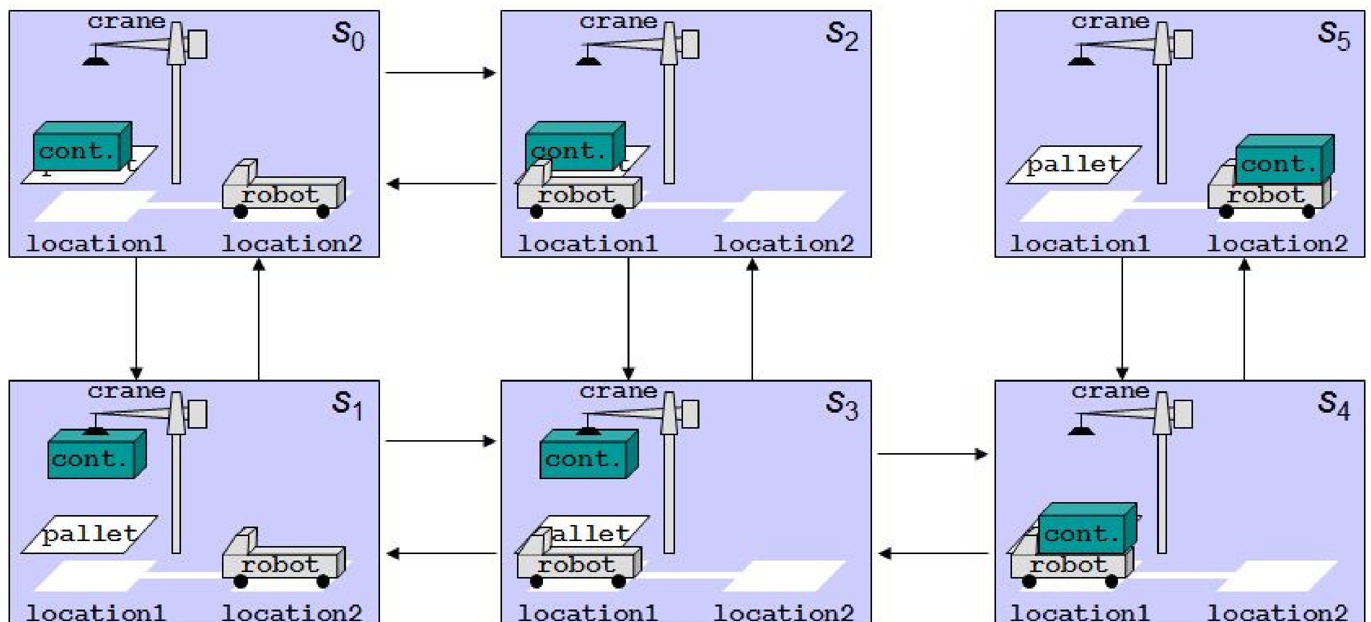
Preliminaries [Nau09, Wic11]

- Propositional logics
- Hill-climbing searching
- A^*, but a suitable heuristics was missing for decades

Idea:

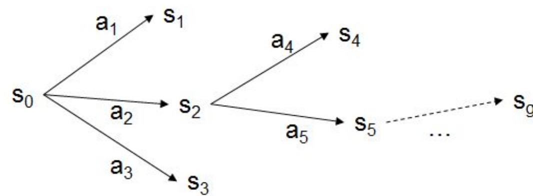
- use of standard searching algorithms
  - breadth-first,
  - depth-first,
  - A^*
  - etc.
- a planning problem task
  - searching space is a subspace of state space
  - nodes represent states of the environment
  - edges correspond to state transitions
  - a path through the state space determines a plan

Planning in State Space - an example [Wic11]



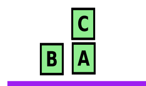
- nodes: closed atoms
- edges: actions (i.e. closed instances of operators)

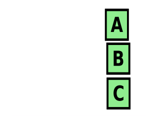
Forward State-Space Search Algorithm [Wic11]



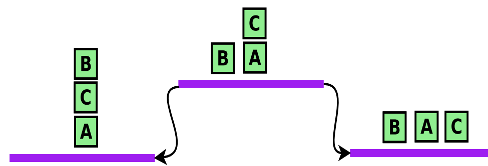
1. Forward-search(O, s_0, g)
   1. $s \leftarrow s_0$
   2. $\pi \leftarrow$ the empty plan
   3. loop
      1. if $s \models g$ then return π
      2. $E \leftarrow \{a | a$ is a ground instance $\in O$ and $precond(a)$ is satisfied in $s\}$
      3. if $E == \emptyset$ then return $FAILURE$
      4. a non-deterministic choice of action $a \in E$
      5. $s \leftarrow \gamma(s, a)$
      6. $\pi \leftarrow \pi.a$

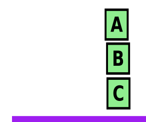
State-Space Searching Example 1 [Wic11]



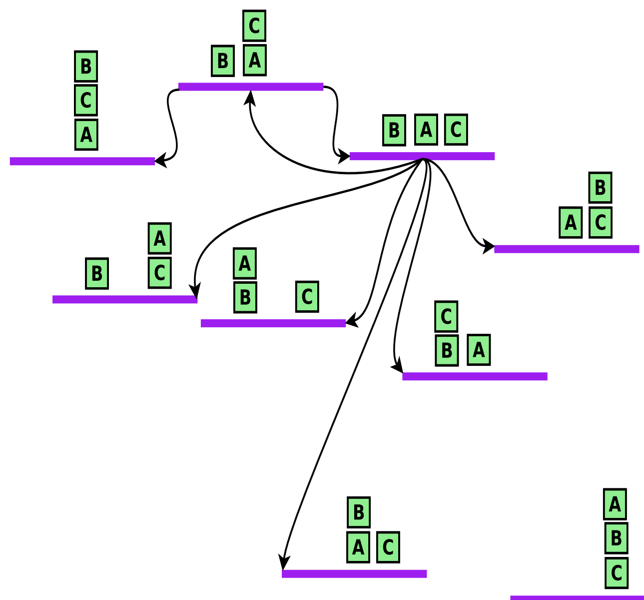


State-Space Searching Example 2 [Wic11]

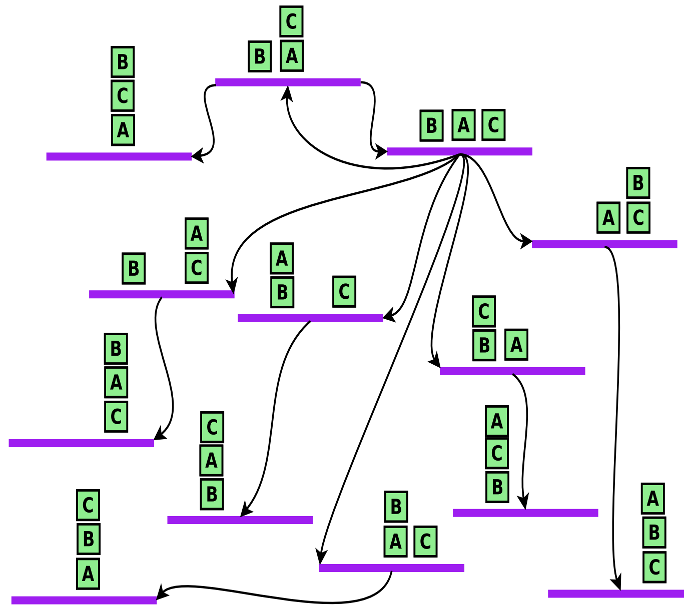




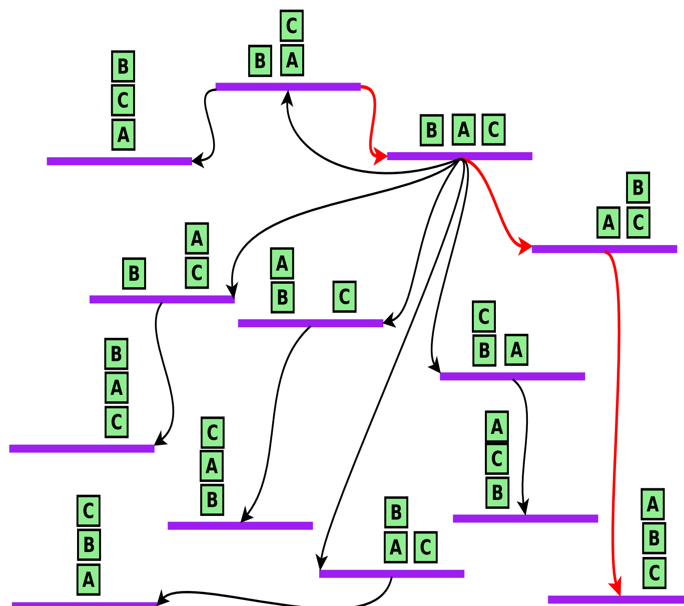
State-Space Searching Example 3 [Wic11]



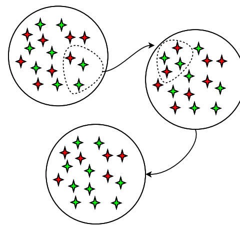
State-Space Searching Example 4 [Wic11]



State-Space Searching Example 5 [Wic11]

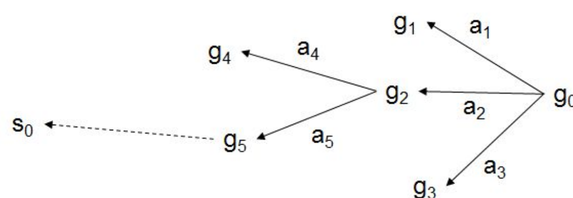


Relevant Actions [Nau09, Wic11]



- Let $\mathcal{P} = (\Sigma, s_i, g)$ be a STRIPS planning problem.
- An action a is **relevant** for g if
  - a causes that at least one of literals of g is satisfied
    - $g \cap \mathit{effects}(a) \neq \emptyset$
  - a does not make any of g's literals false
    - $g^+ \cap \mathit{effects}^-(a) = \emptyset \bigwedge g^- \cap \mathit{effects}^+(a) = \emptyset$
- The **Regression Set** of goal g for a relevant action $a \in A$ is:
  - $\gamma^{-1}(g, a) = (g - \mathit{effects}(a)) \cup \mathit{precond}(a)$

Backward Search [Wic11]

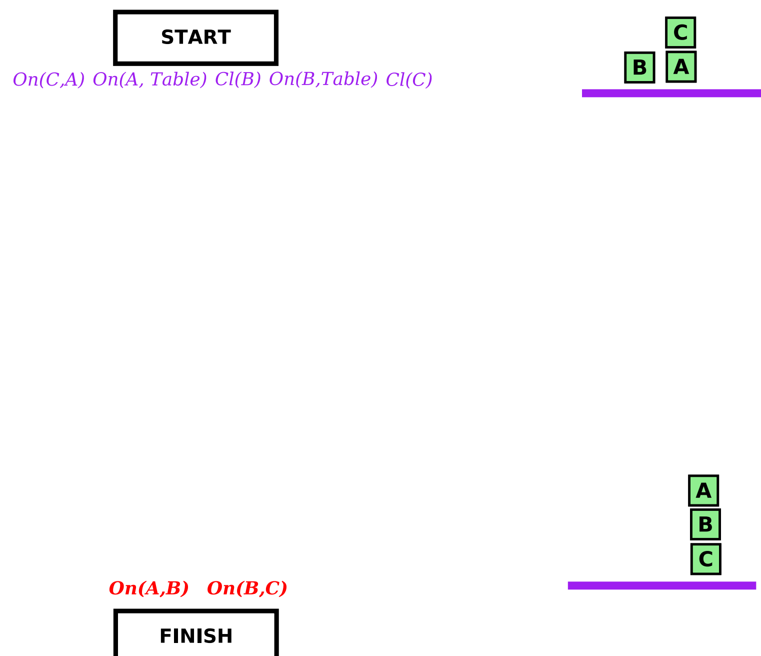


1. Backward-search(O, s_0, g)
   1. $\pi \leftarrow$ the empty plan
   2. loop
      1. if $s_0 \models g$ then return π
      2. $A \leftarrow \{a | a$ is a ground instance of an operator $\in O$ and $\gamma^{-1}(g, a)$ is defined $\}$
      3. if $A == \emptyset$ then return *FAILURE*
      4. nondeterministically choose an action $a \in A$
      5. $\pi \leftarrow a.\pi$
      6. $g \leftarrow \gamma^{-1}(s, a)$

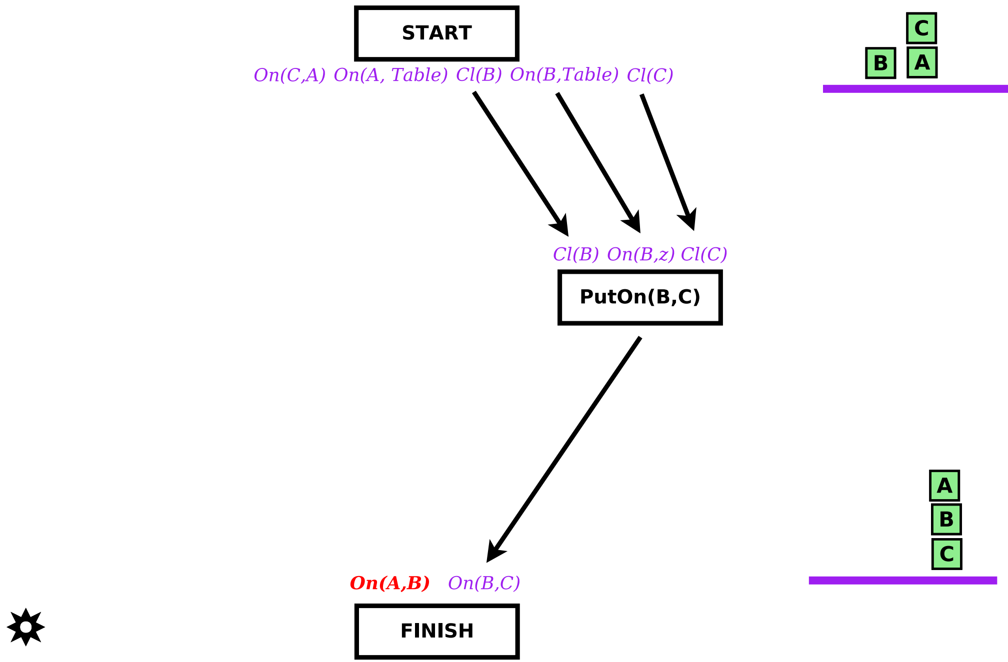
Sussman Anomaly

- an interaction of subgoals so that their solutions must be interleaved to satisfied them all together

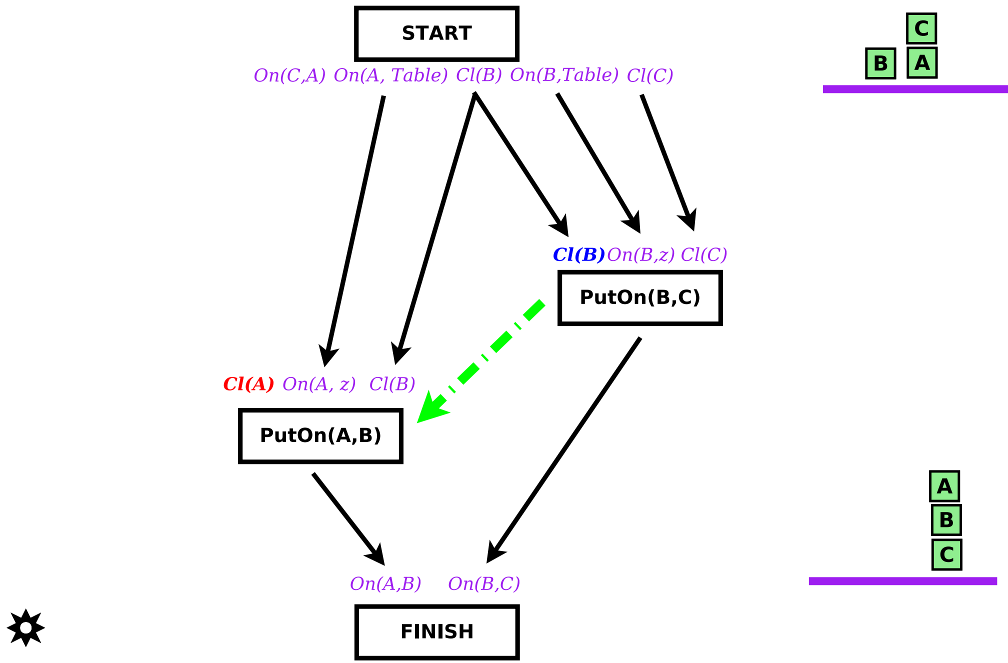
Sussman Anomaly - a Block World Example I [Nau09]



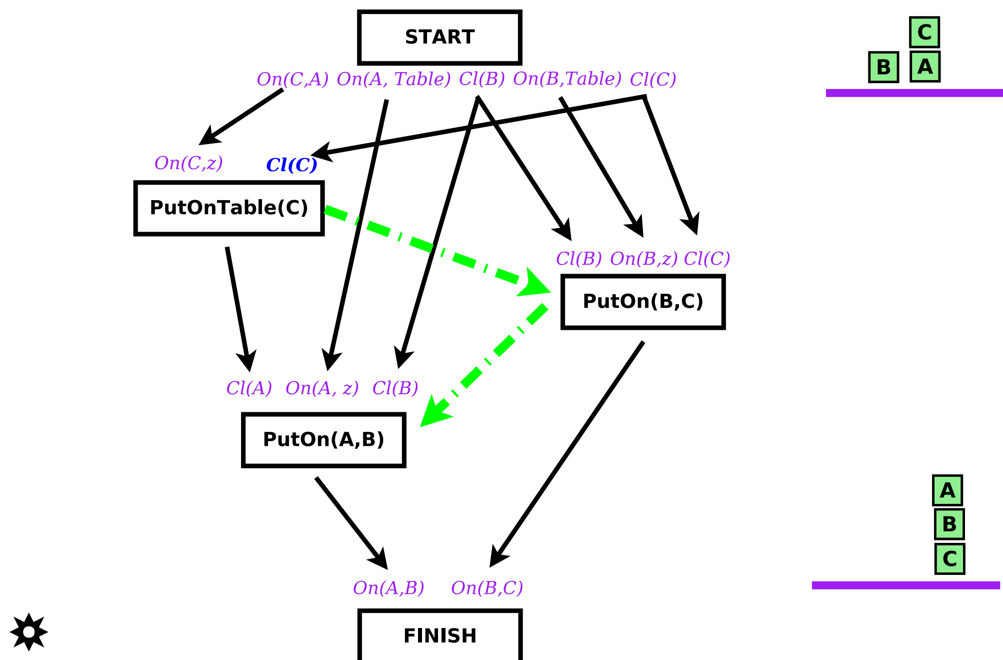
Sussman Anomaly - a Block World Example II [Nau09]



Sussman Anomaly - a Block World Example III [Nau09]



Sussman Anomaly - a Block World Example IV [Nau09]



State-Space vs. Plan-Space Search [Wic11]

state-space search

- search through graph of nodes representing world states

plan-space search

- search through graph of partial plans
- nodes: partially specified plans
- arcs: plan refinement operations
- solutions: partial-order plans
  - temporal ordering of actions
  - rationale: what the action achieves in the plan
  - subset of variable bindings

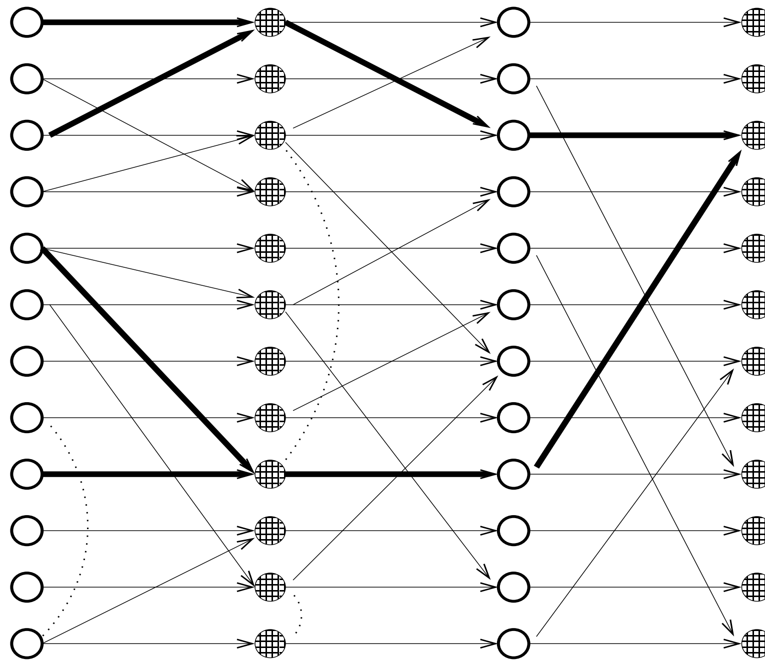
Plan-Space Planning - constraints [Nau09]

- ordering constraints
  - action α must be performed before β ($\alpha \prec \beta$)
- binding constraints
  - inequality constraints, i.e. $v_1 \neq v_2$ or $v_1 \neq c$
  - equality constraints and substitutions, i.e. $v_1 = v_2$ or $v_1 = c$
- causal links
  - use action α to create condition p required by action β

GRAPHPLAN planner

- 1997
- plans are represented as a *planning graph*,
  - the idea is very similar to dynamic programming or network flow solutions
- All plans are constructed concurrently.
  - graph extending (forward run)
  - plan searching (backward run)
- The planner maintains a mutually exclusive relation (*mutex*) between nodes representing applied actions and state propositions.
- The cycling issue is removed.
- Action schemas with parameters cannot be used.
  - It create a huge space of propositions.
- There are many supporting strategies speeding up planning significantly.
- The implementations are capable to create plans with more then 50-100 action calls in minutes.

GraphPlan - Planning Graph



Implementations of planners

Initial attempts

- STRIPS [1971] . . . , the first planner, regressive planning through action preconditions

State/Plan space

- WARPLAN [1973] . . . a linear planner, Sussman anomaly solved using action shifting
- PWEAK, TWEAK [1987], UCPOP [1992] . . . a partial order planner

Planning graphs

- GRAPHPLAN[1997] . . . a breakthrough graphplan planner
- Blackbox [1998] . . . combines GRAPHPLAN and SATPLAN
- FF [2000] . . . a planning graph heuristics with a very fast forward and local search

Summary

- **What is AI planning**
  - reaching a goal or a task problem solution using a sequence of action changing the environment,
  - classical planning as a search for a sequence of actions in state space that transforms the initial state to a goal state.
- **Representations**
  - STRIPS specifies modifications of the world through changes of satisfied closed atoms,
  - PDDL defines a language format that enable to record STRIPS planning domain and STRIPS planning problem
- **Planning Methods**
  - creation of a plan as a searching method through world state/plan/graphplan space

4 Příloha
- PDDL Specification

PDDL Domains [Wic11]

<domain> ::=
  (define (domain <name>)
  [<extension-def>]
  [<require-def>]
  [<types-def>]$^{\text{:typing}}$
  [<constants-def>]
  [<domain-vars-def>]$^{\text{:expression-evaluation}}$
  [<predicates-def>]
  [<timeless-def>]
  [<safety-def>]$^{\text{:safety-constraints}}$
  <structure-def>*)

<extension-def> ::=
  (:extends <domain name>+)
<require-def> ::=
  (:requirements <require-key>+)
<require-key> ::=
  :strips | :typing | ...

<types-def> ::= (:types <typed list (name)>)
<constants-def> ::=
  (:constants <typed list (name)>)
<domain-vars-def> ::=
  (:domain-variables
  <typed list(domain-var-declaration)>)
<predicates-def> ::=
  (:predicates <atomic formula skeleton>+)
<atomic formula skeleton> ::=
  (<predicate> <typed list (variable)>)
<predicate> ::= <name>
<variable> ::= ?<name>
<timeless-def> ::=
  (:timeless <literal (name)>+)
<structure-def> ::= <action-def>
<structure-def> ::=$^{\text{:domain-axioms}}$ <axiom-def>
<structure-def> ::=$^{\text{:action-expansions}}$ <method-def>

PDDL Types [Wic11]

- PDDL types syntax

  <typed list (x)> ::= x*

  <typed list (x)> ::=$^{\text{:typing}}$
    x^+ - <type> <typed list(x)>

  <type> ::= <name>

  <type> ::= (either <type>$^+$)

  <type> ::=$^{\text{:fluents}}$ (fluent <type>)

PDDL Example: DWR Types [Wic11]

```
(define (domain dock-worker-robot)

    (:requirements :strips :typing )

    (:types
        location      ;there are several connected locations
        pile          ;is attached to a location,
                      ;it holds a pallet and a stack of containers
        robot         ;holds at most 1 container,
                      ;only 1 robot per location
        crane         ;belongs to a location to pickup containers
        container )

...)
```

PDDL Example: Predicates [Wic11]

```
(:predicates
    (adjacent ?l1 ?l2 - location)          ;location ?l1 is adjacent to ?l2
    (attached ?p - pile ?l - location)     ;pile ?p attached to location ?l
    (belong ?k - crane ?l - location)      ;crane ?k belongs to location ?l

    (at ?r - robot ?l - location)          ;robot ?r is at location ?l
    (occupied ?l - location)               ;there is a robot at location ?l
    (loaded ?r - robot ?c - container )    ;robot ?r is loaded with container ?c
    (unloaded ?r - robot)                  ;robot ?r is empty

    (holding ?k - crane ?c - container)    ;crane ?k is holding a container ?c
    (empty ?k - crane)                     ;crane ?k is empty

    (in ?c - container ?p - pile)          ;container ?c is within pile ?p
    (top ?c - container ?p - pile)         ;container ?c is on top of pile ?p
    (on ?c1 - container ?c2 - container)   ;container ?c1 is on container ?c2
)
```

PDDL Action [Wic11]

<action-def> ::=
(:action <action functor>
:parameters (<typed list (variable)>)
<action-def body>)
<action functor> ::= <name>
<action-def body> ::=
[:vars (<typed list(variable)>)][:existential-preconditions :conditional-effects]
[:precondition <GD>]
[:expansion <action spec>][:action-expansions]
[:expansion :methods][:action-expansions]
[:maintain <GD>][:action-expansions]
[:effect <effect>]
[:only-in-expansions <boolean>][:action-expansions]

PDDL Goal Specification [Wic11]

<GD> ::= <atomic formula(term)>
<GD> ::= (and <GD>$^+$)
<GD> ::= <literal(term)>
<GD> ::=[:disjunctive-preconditions] (or <GD>$^+$)
<GD> ::=[:disjunctive-preconditions] (not <GD>)
<GD> ::=[:disjunctive-preconditions] (imply <GD> <GD>)
<GD> ::=[:existential-preconditions] (exists (<typed list(variable)>) <GD>)
<GD> ::=[:universal-preconditions] (forall (<typed list(variable)>) <GD>)
<literal(t)> ::= <atomic formula(t)>
<literal(t)> ::= (not <atomic formula(t)>)
<atomic formula(t)> ::= (<predicate> t*)
<term> ::= <name>

PDDL Effects [Wic11]

<effect> ::= (and <effect>$^{+}$)

<effect> ::= <atomic formula(term)>

<effect> ::= (not <atomic formula(term)>)

<effect> ::=[:conditional−effects]
    (forall (<variable>*) <effect>)

<effect> ::=[:conditional−effects]
    (when <GD> <effect>)

<effect> ::=[:fluents] (change <fluent> <expression>)

PDDL Example: Operator [Wic11]

```
;; moves a robot between two adjacent locations
(:action move
   :parameters (?r - robot ?from ?to - location)
   :precondition (and
      (adjacent ?from ?to) (at ?r ?from)
      (not (occupied ?to)))
   :effect (and
      (at ?r ?to) (occupied ?to)
      (not (occupied ?from)) (not (at ?r ?from)) ))
```

PDDL Problem [Wic11]

```
<problem> ::= (define (problem <name>)
      (:domain <name>)
      [<require-def>]
      [<situation> ]
      [<object declaration> ]
      [<init>]
      <goal>+
      [<length-spec> ])
<object declaration> ::= (:objects <typed list (name)>)
<situation> ::= (:situation <initsit name>)
<initsit name> ::= <name>
<init> ::= (:init <literal(name)>+)
<goal> ::= (:goal <GD>)
<goal> ::=:action-expansions (:expansion <action spec(action-term)>)
<length-spec> ::= (:length [(:serial <integer>)] [(:parallel <integer>)])
```

PDDL Problem: DWR example [Wic11]

```
;; a simple DWR problem with 1 robot and 2
locations
(define (problem dwrpb1)
      (:domain dock-worker-robot)
      (:objects
            r1 - robot
            l1 l2 - location
            k1 k2 - crane
            p1 q1 p2 q2 - pile
            ca cb cc cd ce cf pallet - container)
      (:init
            (adjacent l1 l2)
            (adjacent l2 l1)
            (attached p1 l1)
            (attached q1 l1)
            (attached p2 l2)
            (attached q2 l2)
            (belong k1 l1)
            (belong k2 l2)

            (in ca p1) (in cb p1) (in cc p1)
            (on ca pallet) (on cb ca) (on cc cb)
            (top cc p1)

            (in cd q1) (in ce q1) (in cf q1)
            (on cd pallet) (on ce cd) (on cf ce)
            (top cf q1)

            (top pallet p2)
            (top pallet q2)

            (at r1 l1)
            (unloaded r1)
            (occupied l1)

            (empty k1)
            (empty k2))

;; task is to move all containers to locations l2
;; ca and cc in pile p2, the rest in q2
(:goal (and
      (in ca p2) (in cc p2)
      (in cb q2) (in cd q2) (in ce q2) (in cf q2))))
```

References I

Dana Nau.
CMSC 722, ai planning (fall 2009), lecture notes.
http://www.cs.umd.edu/class/fall2009/cmsc722/, 2009.

Michal Pechoucek.
A4m33pah, lecture notes.
http://cw.felk.cvut.cz/doku.php/courses/a4m33pah/prednasky, February 2010.

Gerhard Wickler.
A4m33pah, lecture notes.
http://cw.felk.cvut.cz/doku.php/courses/a4m33pah/prednasky, February 2011.