AI Planning

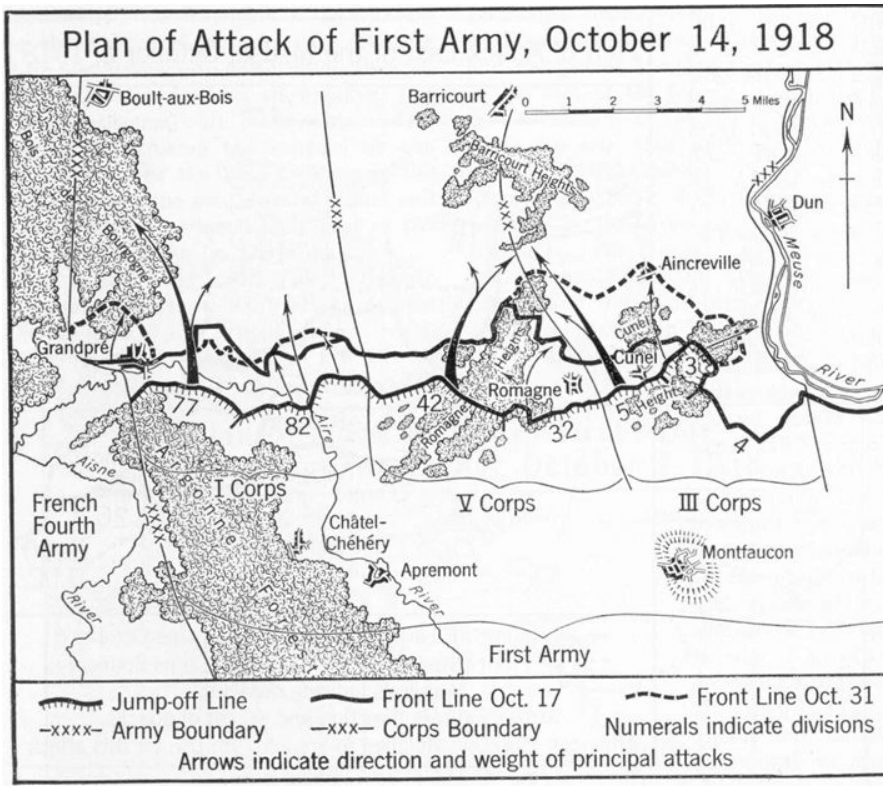
Radek Mařík

FEE CTU, K13132

April 18, 2017

Outline

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.

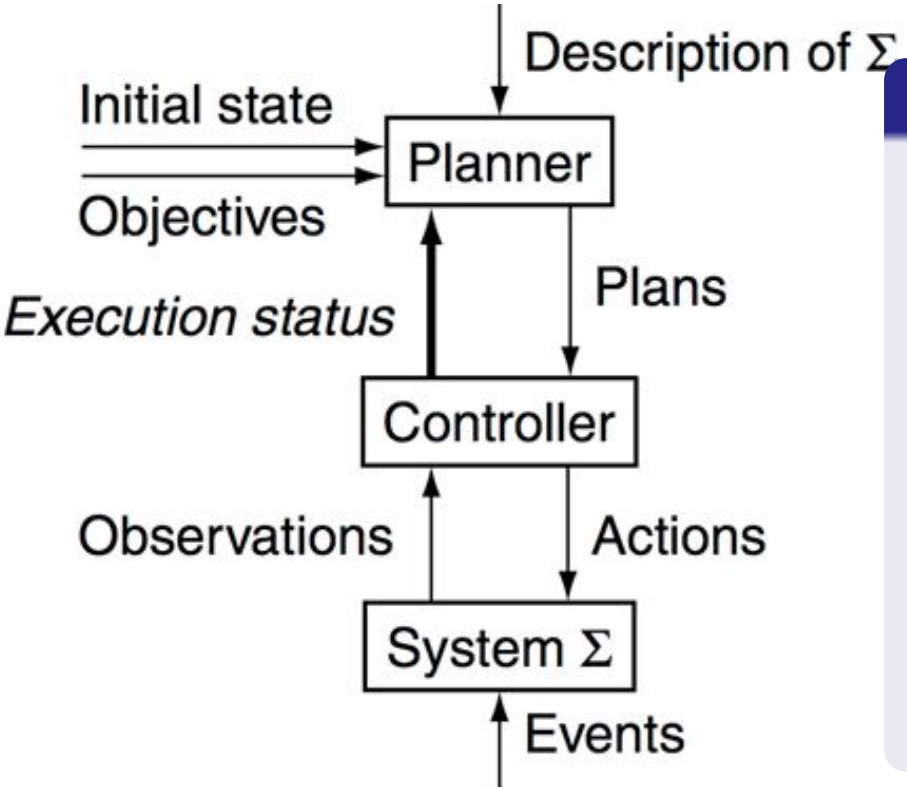
Planning

- 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.

Conceptual Model of Planning I [Nau09]



Environment

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

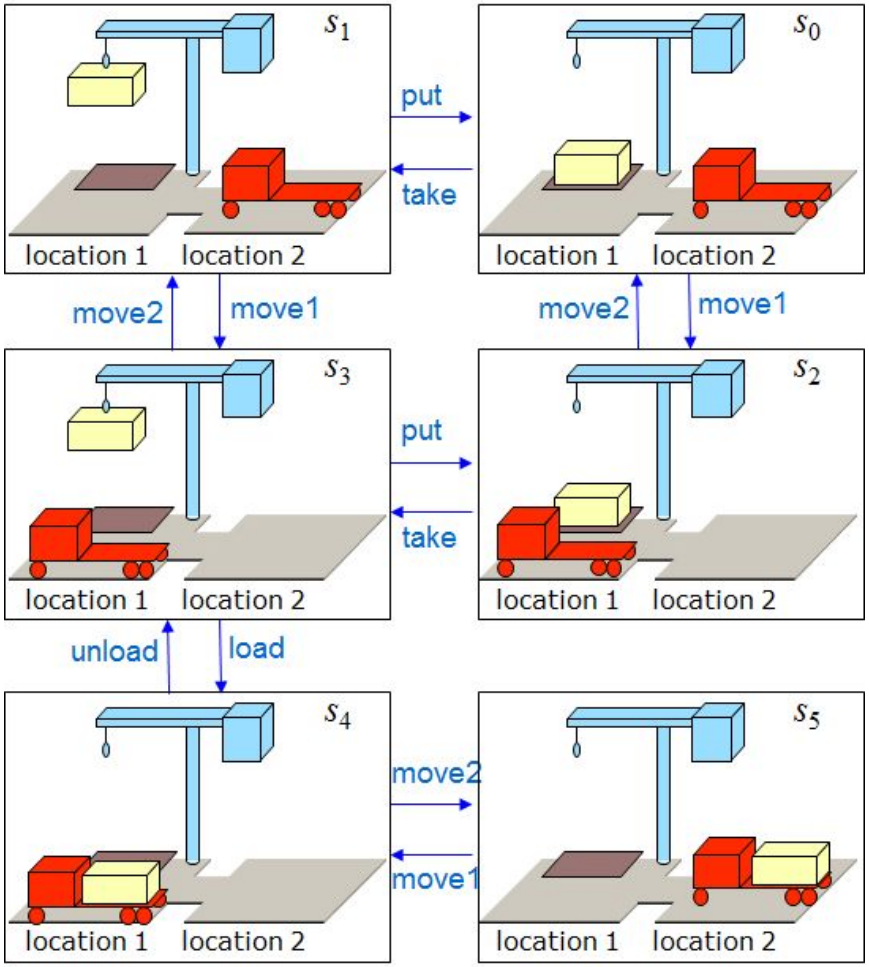
The Dock-Worker Robots (DWR) Domain [Wic11]

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

Planning Task [Nau09]



Planning problem

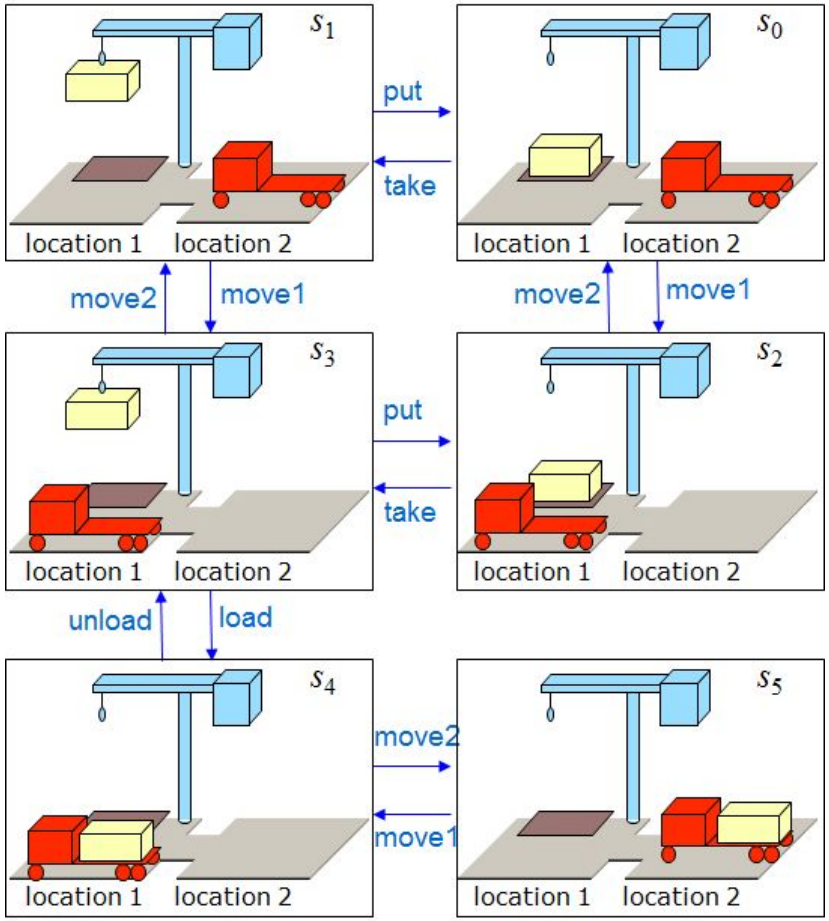
- System description Σ
- Initial state or set of states
  - Initial state $= s_0$
- Objective
  - Goal state,
  - set of goal states,
  - set of tasks,
  - "trajectory" of states,
  - objective function
  - Goal state $= s_5$

Example 1

System Instance

- $S = \{s_0, s_1, \ldots, s_5\}$, $E = \{\}$
- $A = \{move_1, move_2, put, \ldots\}$
- $\gamma = \{$see arrows$\}$

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}

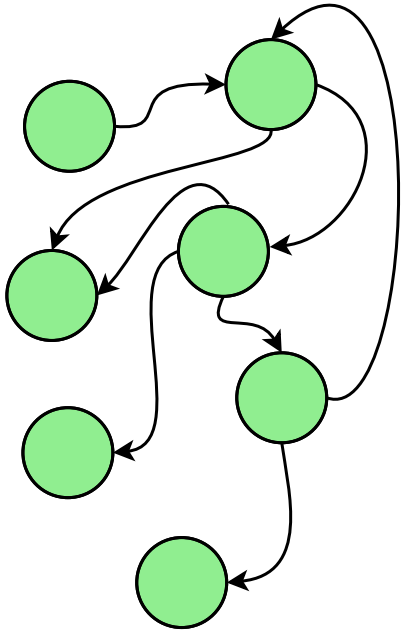
Automated planning research

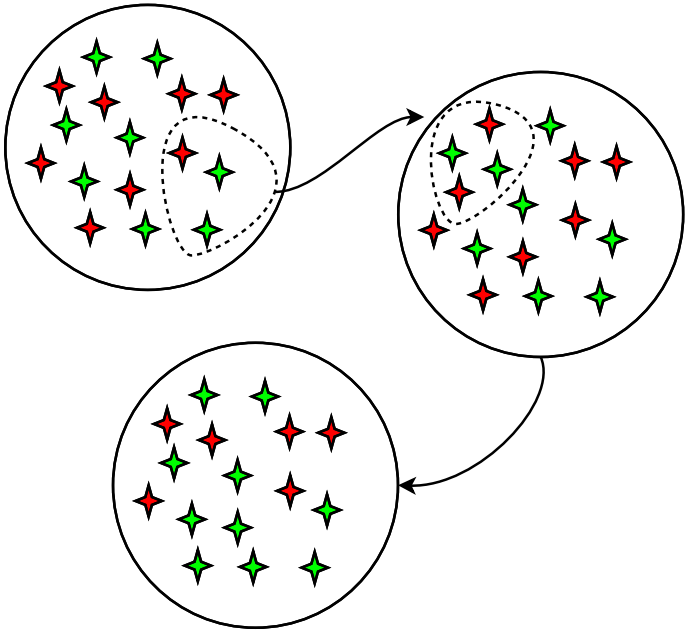
- classical planning mostly
- dozens (hundreds) of different aglorithms

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

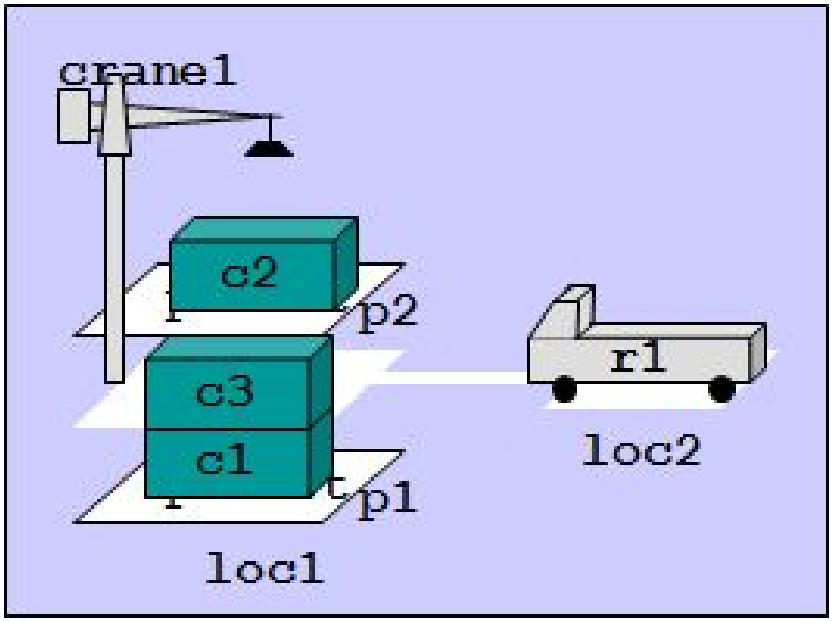
STRIPS - state representation [Wic11]

- 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 ($s \mid g$) 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

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



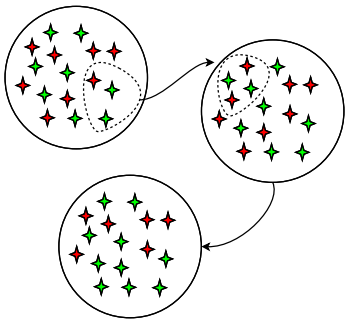
STRIPS - Operator and Action Representations [Wic11]

- **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
  - $\gamma : S \times A \rightarrow S$ where
    - $\gamma(s, a) = (s - effects^{-}(a)) \cup effects^{+}(a)$ 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 $S_g = \{s \in S | s \models g\}$

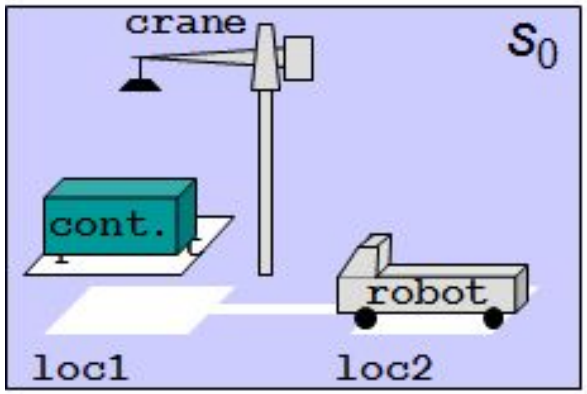
STRIPS Planning Problem: DWR Example [Wic11]

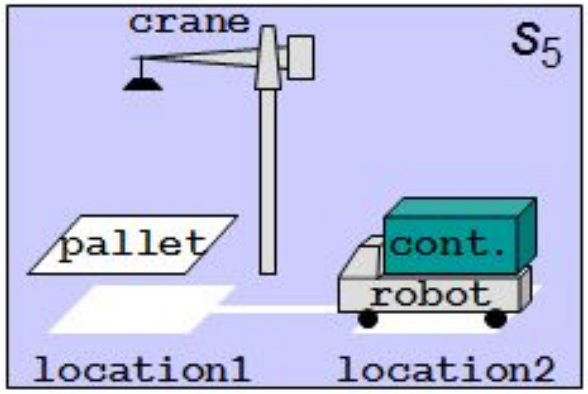
DWR planning problem

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

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

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

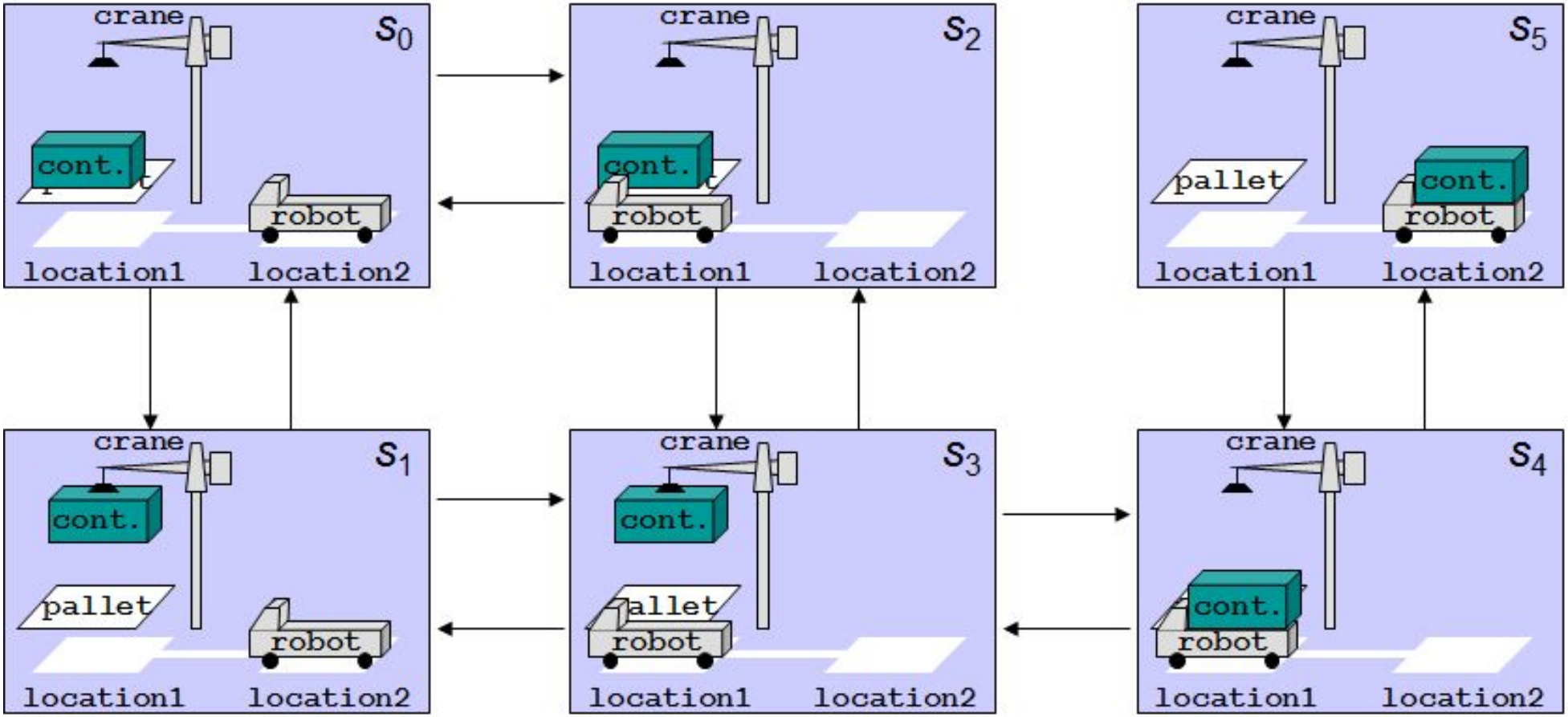
Preliminaries [Nau09, Wic11]

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

Idea:

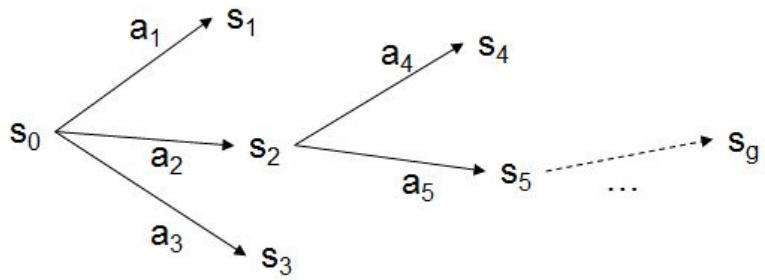
- use of standard searching algorithms
  - breadth-first,
  - depth-first,
  - A^*
  - etc.
- a planning problem task
  - searching space is a subspace of state space
  - nodes represent states of the environment
  - edges correspond to state transitions
  - a path through the state space determines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

Planning in State Space - an example [Wic11]



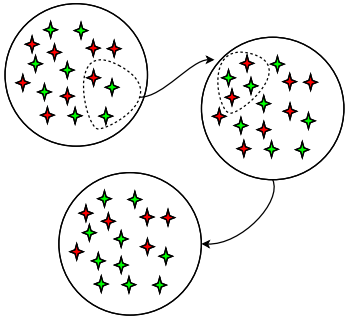
- nodes: closed atoms
- edges: actions (i.e. closed instances of operators)

Forward State-Space Search Algorithm [Wic11]



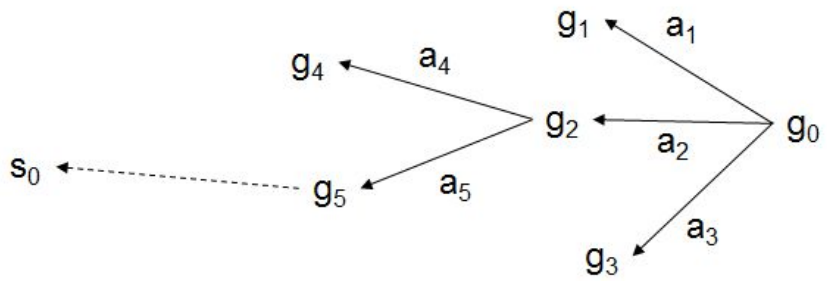
1. Forward-search(O, s_0, g)
   1. $s \leftarrow s_0$
   2. $\pi \leftarrow$ the empty plan
   3. loop
      1. if $s \mid g$ then return π
      2. $E \leftarrow \{a | a$ is a ground instance $\in O$ and $precond(a)$ is satisfied in $s\}$
      3. if $E == \emptyset$ then return $FAILURE$
      4. a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

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
    - $g \cap effects(a) \neq \emptyset$
  - a does not make any of g's literals false
    - $g^+ \cap effects^-(a) = \emptyset \bigwedge g^- \cap effects^+(a) = \emptyset$
- The **Regression Set** of goal g for a relevant action $a \in A$ is:
  - $\gamma^{-1}(g, a) = (g - effects(a)) \cup precond(a)$

Backward Search [Wic11]



1. Backward-search(O, s_0, g)
   1. $\pi \leftarrow$ the empty plan
   2. loop
      1. if $s_0 \mid g$ 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. $g \leftarrow \gamma^{-1}(s, a)$

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]

- precedence 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, e.g. $v_1 = v_2$ or $v_1 = c$
- causal links
  - use action α to create precondition p required by action β

Flaw solution - Causal Link Threats [Nau09]

- *Causal link* formally:
  - regarding to the ordering relation
  - $\forall \alpha_1, \alpha_2 \in \pi : \exists x : x \in \text{pre}(\alpha_2) \wedge x \in \text{eff}(\alpha_1) \Leftrightarrow \alpha_1 \prec \alpha_2$
  - insert a causal link as a satisfiability relation between the operators
  - $\alpha_1 \xrightarrow{x} \alpha_2$, kde $x \in \text{eff}(\alpha_1) \wedge x \in \text{pre}(\alpha_2) \wedge \alpha_1 \prec \alpha_2$
  - to be read as: α_1 supplies x for α_2
- A threat on a causal link
  - **a negative thread to a causal link**: $\alpha_1 \xrightarrow{q} \alpha_3$ is a causal link in a plan and $\alpha_1 \prec \alpha_2$, $\alpha_2 \prec \alpha_3$, are consistent with a plan and $\exists p \in \text{eff}(\alpha_2)$ so that it can remove q
  - **a positive threat to a causal link**: ... defined similarly, adding q
- Causal link threat solution
  - demotion $\alpha_2 \prec \alpha_1$
  - promotion $\alpha_3 \prec \alpha_2$
  - variable binding constraint

TOPLAN [Pec10]

- Total Order Planning

1. initialization: $\boldsymbol{\Pi} \leftarrow \{\{s_{goal}\}\}, \mathbf{S} \leftarrow \{s_{goal}\}$

$\underline{toplan}(s_0, \boldsymbol{\Pi}, \mathbf{S})$

1. if $\exists s_n \in \mathbf{S}, \pi_n \in \boldsymbol{\Pi} : s_{goal} == s_n$, then $return(\pi_n)$
2. if $\mathbf{S} = \emptyset$
   - then $return(failure)$
   - otherwise
     1. $\underline{remove}$ s_i from $\mathbf{S}$ a $\underline{remove}$ π_i from $\boldsymbol{\Pi}$
     2. $A \leftarrow \{\alpha | eff(\alpha) \in s_i\}$
     3. $S \leftarrow \{s | \forall \alpha \in A : successor(\alpha, s) = s_i\}$
     4. $\Omega \leftarrow \{\pi | \forall \alpha \in A : \pi = \alpha \cup \pi_i\}$
     5. $return(\underline{toplan}(s_0, \underline{append}(\boldsymbol{\Pi}, \Omega), \underline{append}(\mathbf{S}, S)))$

POPLAN - Partial Order Planning (simplified) [Pec10]

1. initialization: $\boldsymbol{\Pi} \leftarrow \{\text{actions}, \{s_0 \prec s_{goal}\}, \{\}, \{\text{pre}(s_{goal})\}\}$
2. actions α, β

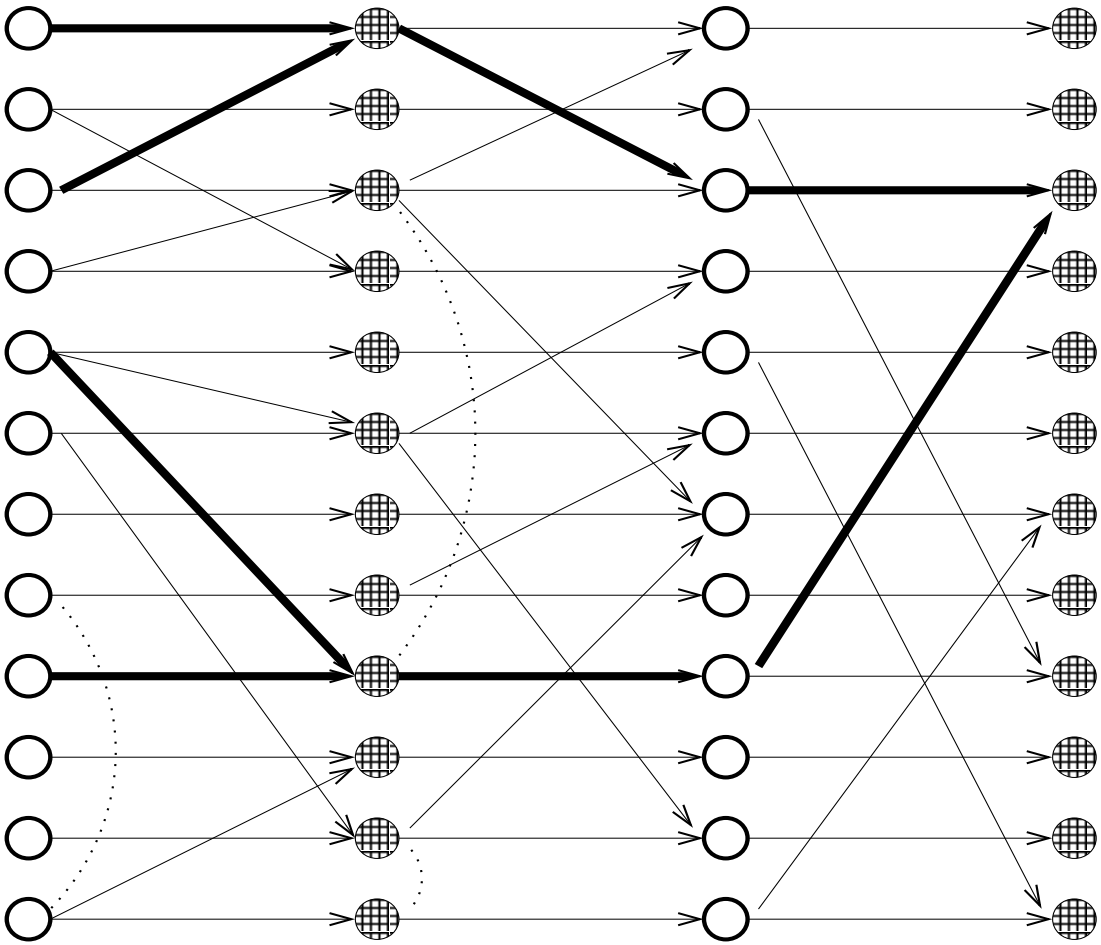
$\underline{poplan}(\boldsymbol{\Pi})$

1. *if* complete$(\boldsymbol{\Pi})$ then $return(\boldsymbol{\Pi})$
2. *if* $\exists p \in \text{eff}(\beta) \wedge \beta \in \text{openGoals}(\boldsymbol{\Pi})$ and $\exists \alpha$, so that $p \in \text{eff}(\alpha)$
   - *then* append$(\boldsymbol{\Pi}, \{\{\alpha \xrightarrow{p} \beta\}, \{\alpha \prec \beta\}\})$ a remove$(p, \beta, \text{openGoals}(\boldsymbol{\Pi}))$
   - *else* $return(fail)$
3. *if* there is a causal link $\alpha_1 \xrightarrow{x} \alpha_2$ threaten by action α_3
   - *then* do one of the following steps
     - Promotion: $return(\underline{poplan}(\boldsymbol{\Pi} \uplus \{\alpha_3 \prec \alpha_1\}))$
     - Demotion: $return(\underline{poplan}(\boldsymbol{\Pi} \uplus \{\alpha_2 \prec \alpha_3\}))$
   - *else* $return(\underline{poplan}(\boldsymbol{\Pi}))$

GRAPHPLAN planner

- 1997
- plans are represented using *planning graph*,
  - the idea is very similar to dynamic programming or network flow solution,
- All plans are build concurrently.
  - graph expansion (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 (only instances).
  - It creates 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



Planning Graphs [Nau09]

Planning graph

is a directed layered graph with 2 types of layers

- S_i contains all literals that may be satisfied at step i
- A_i contains all actions which conditions may be satisfied at step i

Mutex

- between two actions
  - **inconsistent effects** . . . one action negates an effect of the other,
  - **interference** . . . one of the effects of one action is the negation of a precondition of the other,
  - **competing needs** . . . one of the preconditions of one action is mutually exclusive with a precondition of the other.
- between two literals
  - **inconsistent support** . . . one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive.

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

Hierarchical Planning [SV10]

Hierarchical Task Net (HTN) Planning

Basic Ideas

- Complex plans often have identifiable structure,
- That structure can often be captured in the form of hierarchies of abstract subplans.
- Subplans are often (nearly) independent of one another.

Example

- "To get to conference in ?x, get to the airport, take a plane to ?x, then get to the conference hotel"
- "To get to the airport, either drive or take a cab"
- "If you have enough money for the fare: To take a cab to ?y, either call ahead, or flag a cab down, then enter the cab, say “I want to go to ?y”, wait until at ?y, pay the fare, then exit the cab."

HTN vs Classical Planning I [Hoa07]

Like classical planning

- Each **state** of the world is represented by a set of atoms
- Each **action** corresponds to a deterministic state transition

Situation calculus

```
(block b1) (block b2) (block b3) (block b4)
(on-table b1) (on-table b3)
(clear b2) (clear b4) (on b2 b1) (on b4 b3)
```

HTN vs Classical Planning Differences [Hoa07]

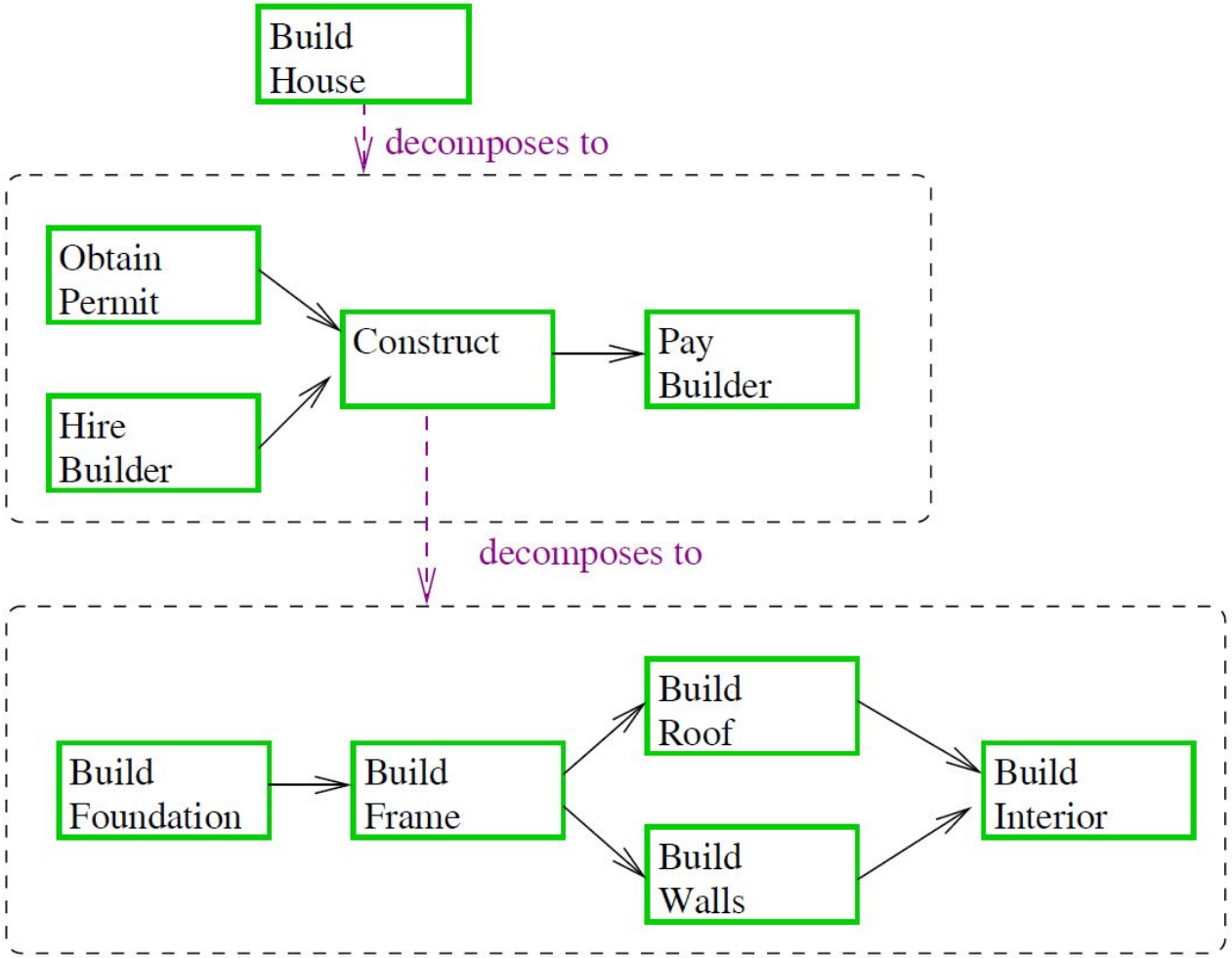
Classical Planning

1. Objective: to perform a set of goals
2. Terms, literals, operators, actions, plans

HTN planning

1. Objective: to perform a set of tasks
2. Terms, literals, operators, actions, plans have same meaning as classical planning
3. Added tasks, methods, task networks
4. Tasks decompose into subtasks
   - Constraints
   - Backtrack if necessary

HTN Plans: Example



HTN Task [Hoa07]

- **Task:** an expression of the form $t(u_1, \ldots, u_n)$
  - t is a task symbol and
  - each u_i is a term
    (variable, constant, function expression $(f \quad t_1 \quad t_2 \quad t_3)$)

Example

```
(move-block ?nomove)
(move-block (list ?x . ?nomove))
```

- Two types of tasks
  1. **Non-primitive** (compound) - decomposed into subtasks
  2. **Primitive**
     - cannot be decomposed,
     - know how to perform directly,
     - task name is the operator name

Example

```
(!drive-truck ?truck ?loc-from ?loc-to)
```

Methods and Operators (SHOP) [Hoa07]

- **Method:**

  `(:method h` $[n_1]$ C_1 T_1 $[n_2]$ C_2 T_2 $\ldots$ $[n_k]$ C_k T_k`)`

  - h method head - task atom with no call terms
  - n_1 OPTIONAL name for succeeding C_1 T_1 pair
  - C_1 conjuct - precondition
  - T_1 task list

- **Operator:**

  `(:operator h P D A)`

  - h head - primitive task atom with no call terms
  - P precondition list (logical atoms)
  - D delete list (logical atoms)
  - A add list (logical atoms)

Methods and Operators Examples (SHOP) [Hoa07]

Method: decomposes into subtasks

```
(:method (drive-truck ?truck ?loc-from ?loc-to)
  ((same ?loc-from ?loc-to))
  ((!do-nothing))
  ()
  ((!drive-truck ?truck ?loc-from ?loc-to)))
```

Operator: achieves PRIMITIVE TASKS

```
(:operator (!drive-truck ?truck ?locfrom ?locto)
  ()
  ((truck-at ?truck ?locfrom))
  ((truck-at ?truck ?locto)))
```

STN and HTN [Hoa07]

STN: Simple Task Network (simplified version of HTN

1. TFD - Total-order Forward Decomposition (SHOP [NCLMA99])
   1. Input: tasks are totally ordered.
   2. Output: totally ordered plan
2. PFD - Partial-order Forward Decomposition (SHOP2)
   1. Input: tasks are partially ordered.
   2. Output: totally ordered plan

HTN: generalization of STN (NONLIN, O-PLAN, SIPE-2, UMCP)

1. More freedom about how to construct the task networks
2. Can use other decomposition procedures than just forward decomposition
3. Like Partial-order planning combined with STN
   1. Input: partial-order tasks
   2. Output: the resulting plan is partially ordered

Task Network [Hoa07]

STN:

- $w = (U, E)$ ... an acyclic graph
- U ... set of task nodes
- E ... set of edges

HTN:

- $w = (U, C)$
- U ... set of task nodes
- C ... set of constraints (allow for generic tasks networks)
- Different planning procedures

Search Space [SV10]

Problem Reduction Search

- *Goal state* is an abstract task to be achieved
- **not** state of the world

Search space operators

- Decompose task into subtask(s)
- Parameterize task
- Solve for conflicts.

Pseudo-code for TFD [Hoa07]

- $TFD(s, \langle t_1, \ldots, t_k \rangle, O, M)$
  - if $k = 0$ then return $\langle\rangle$ (i.e. the empty plan)
  - if t_1 is **primitive** then
    - $active \leftarrow \{(a, \sigma) | a$ is a ground instance of an operator in O, σ is a substitution such that a is relevant for $\sigma(t_1)$, and a is applicable to $s\}$
    - if $active = \emptyset$ then return failure
    - nondeterministically choose any $(a, \sigma) \in active$
    - $\pi \leftarrow TFD(\gamma(s, a), \sigma(\langle t_2, \ldots, t_k \rangle), O, M)$
    - if $\pi =$ failure then return failure
    - else return $a.\pi$
  - else if t_1 is **nonprimitive** then
    - $active \leftarrow \{(m, \sigma) | m$ is a ground instance of a method in M, σ is a substitution such that m is relevant for $\sigma(t_1)$, and m is applicable to $s\}$
    - if $active = \emptyset$ then return failure
    - nondeterministically choose any $(m, \sigma) \in active$
    - $w \leftarrow$ subtasks$(m).\sigma(\langle t_2, \ldots, t_k \rangle)$
    - return $TFD(s, w, O, M)$

HTN Constraints and Plans [SV10]

Constraints

- Precedence (before, after, between)
- Metric temporal
- Resource
- Constraints at one level apply to all pairs of tasks at lower level
  - $U = \{u_1, u_2\}; V = \{v_1, v_2\}$
  - $U < V \Rightarrow u_1 < v_1; u_2 < v_1; u_1 < v_2; u_2 < v_2$

Plans

- An **Abstract Plan** contains compound operators
- An **Instantiated Plan** has only primitive operators
- A **Fully Instantiated Plan** is a totally-ordered instantiated plan with all variables bound.

Abstract HTN Algorithm [SV10]

HTN-plan(tasks, constraints, methods)

1. If (tasks, constraints) has no solution, return($\{\}$)
2. If (tasks, constraints) is an instantiated plan
   - **Select** a fully instantiated plan
   - If such a plan exists, return it; otherwise return($\{\}$)
3. **Select** an abstract task $t \in$ tasks
4. **Choose** an applicable $m \in$ methods
   - where u is the task-list of m and c are the constrains m
   - tasks $=$ (tasks $- \{t\}) \cup u$
   - constraints $=$ constraints $\cup c$
5. (tasks, constraints) $=$ applyCritics(tasks, constraints)
6. return(HTN-plan(tasks, constraints, methods)

Where is the Power? [SV10]

Specification

- Methods encode domain knowledge.
- Methods encode problem solving knowledge
  - "how" rather than "what"
- Abstractions encapsulate patterns of interaction
- $+$ May be easier to specify domain
- $-$ Have to specify all possible goals (and how to achieve them)

Caveats

- HTN planning, in worst case, is still NP-complete
- May not terminate (recursive method expansions – may be hard to detect infinite loops)
- May have to completely expand before finding plan is illegal.

Simple HTN Planning Extensions [SV10]

1. Threat detection
   - Deal with interacting goals,
   - Find ways of reusing operators
2. Methods can include preconditions and effects
   - Find threats earlier in the planning process
3. Methods can indicate resource usage
   - Do some types of scheduling
4. Operators/Methods can include open conditions ("external preconditions")
   - Need to use action-based (refinement) planning
   - Enables planner to find "novel" solutions

Simple Hierarchical Order Planner (SHOP) [SV10]

SHOP Algorithm

- Forward search, linear planner
  - Plans in same order as execution
  - Essentially depth-first search
- Primitive operators have no preconditions
- No concurrent actions
- Highly expressive operator representation (numeric calculations)
- Efficient (but rather inflexible) planning algorithm

SHOP Domain Example [SV10]

```
(:operator  (!putdown ?block)
   ((holding ?block))                    ← Delete list
   ((ontable ?block) (handempty)))       ← Add list

(:method  (make-clear ?y)
   ((clear ?y))                          ← Applicability condition
   nil)                                  ← Task list

(:method  (make-clear ?y)
   ((on ?x ?y))                          ← Applicability condition
   ((make-clear ?x)
    (!unstack ?x ?y) (!putdown ?x))      ← Task list
```

SHOP, SHOP2 Applications [NAI+05]

- Projects in **government** laboratories
  - Evacuation planning,
  - Evaluating terrorist threats,
  - Fighting forest fires,
- **Industry** projects
  - Controlling multiple UAVs,
  - Evaluating of enemy threats,
  - Location-based services,
  - Material selection for manufacturing
- **University** projects
  - Automated composition of Web services,
  - Project planning,
  - Statistical goal recognition in agent systems,
  - Distributed planning.

Example: house building (O-Plan) [SV10]

Method Build (?house)

Precondition: (and (own land) (have money))
Effects: (built ?house)
Applicability: (single-family-home ?house)
Expansion: S1: Build-Foundation(?house)
S2: Build-Frame(?house)
S3: Build-Roof(?house)
S4: Build-Walls(?house)
S5: Build-Interior(?house)
S6: Decorate(?house)
Orderings: S1<S2, S2<S3, S2<S4, S3<S5, S5<S6
Links: S1 causes (foundations laid) for S2
S2 causes (frame erected) for S3 and S4
S3 causes (roof built) for S5
S4 causes (walls built) for S5
S5 causes (interior done) for S6
TimeWindow: start between 11:30 and 14:30 at S3
Resources: bricklayers = between 1 and 2 men at S4

Pyhop Planner [NAI+05]

- A simple HTN planner written in Python
  - Works in both Python 2.7 and 3.2
- Planning algorithm is like the one in SHOP
- Main differences:
  - HTN operators and methods are ordinary Python functions
  - The current state is a Python object that contains variable bindings
    - Operators and methods refer to states explicitly
    - To say c is on a, write s.loc['c']='a' where s is the current state
- Easy to implement and understand
  - Less than 150 lines of code
- Open-source software, Apache license
  - `http://bitbucket.org/dananau/pyhop`

Pyhop Actions

Abstract

```
walk(a: Agents, x: Locations,
     y: Locations)
  Pre: loc(a) = x
  Eff: loc(a) = y
call-taxi(a: Agents, x: Locations)
  Pre: -
  Eff: loc(taxi) = x
ride-taxi(a: Agents, x: Locations,
           y: Locations)
  Pre: loc(a) = x, loc(taxi) = x
  Eff: loc(a) = y, loc(taxi) = y,
       owe(a) = 1.50
                + 1/2 distance(x,y)
pay-driver(a: Agents)
  Pre: owe(a) = r, cash(a) ≥ r
  Pre: owe(a) = r,
  cash(a) = cash(a) - r
```

```
def walk(state,a,x,y):
  if state.loc[a] == x:
    state.loc[a] = y
    return state
  else: return False
def callTaxi(state,a,x):
  state.loc['taxi'] = x
  return state
def rideTaxi(state,a,x,y):
  if state.loc['taxi']==x and state.loc[a]==x:
    state.loc['taxi'] = y
    state.loc[a] = y
    state.owe[a] = 1.5 + 0.5*state.dist[x][y]
    return state
  else: return False
def payDriver(state,a):
  if state.cash[a] >= state.owe[a]:
    state.cash[a] = state.cash[a] - state.owe[a]
    state.owe[a] = 0
    return state
  else: return False
declare_operators(walk,callTaxi,rideTaxi,payDriver)
```

Pyhop Methods

Abstract

```
travel-by-foot(a, x, y)
  Task: travel(a,x,y)
  Pre: loc(a,x), distance(x,y) ≤ 4
  Sub: walk(a,x,y)
travel-by-taxi(a,x,y)
  Task: travel(a,x,y)
  Pre: cash(a) ≥ 1.5 + 0.5*dist(x,y)
  Sub: call-taxi(a,x),
       ride-taxi(a,x,y),
       pay-driver(a)
```

```
def travelByFoot(state,a,x,y):
  if state.dist[x][y] <= 4:
    return [('walk',a,x,y)]
  return False
def travelByTaxi(state,a,x,y):
  if state.cash[a] >= 1.5 +

     0.5*state.dist[x][y]:
    return [('callTaxi',a,x),
            ('rideTaxi',a,x,y),
            ('payDriver',a,x,y)]
  return False
declare_methods('travel',
  travelByFoot, travelByTaxi)
```

Pyhop Travel Planning Problem

Initial state:

```
loc(me) = home, cash(me) = 20, dist(home, park) = 8
```

```
state1 = State('state1')
state1.loc  = 'me':'home'
state1.cash = 'me':20
state1.owe  = 'me':0
state1.dist = 'home':'park':8, 'park':'home':8
```

Task:

```
travel(me,home,park)
```

```
# Invoke the planner park
pyhop(state1,[('travel','me','home','park')])
```

Solution plan:

```
call-taxi(me,home), ride-taxi(me,park), pay-driver(me)
```

```
[('callTaxi','me','home'),('rideTaxi','me','home', 'park'), ('payDriver', 'me')]
```

3 Appendix

- JSHOP2

JSHOP2 [Ilg06]

Atomic task

- ([: immediate] s t_1 t_2 $\ldots$ t_n)
-

Task list

- ([: unordered] [tasklist$_1$ tasklist$_2$ $\ldots$ tasklist$_n$])
-

JSHOP2 [Ilg06]

Operators

- (: operator h P D A $[c]$)
- (: *protection* a)

Methods

- (: method h [name$_1$] L_1 T_1 [name$_2$] L_2 T_2 … [name$_n$] L_n T_n)
-

JSHOP2 [Ilg06]

Planning domain

- (defdomain domain-name$(d_1\ d_2\ \ldots\ d_n)$)

Planning task

- (defproblem problem-name domain-name $([a_{1,1}\ a_{1,2}\ \ldots\ a_{1,n}])\ T_1\ \ldots\ ([a_{m,1}\ a_{m,2}\ \ldots\ a_{m,n}])\ T_m$)
-

References I

Hai Hoang.
Hierarchical task network (HTN) planning, lecture notes.
http://www.cse.lehigh.edu/ munoz/AIPlanning/classes/HTNApril 2007.

Okhtay Ilghami.
Documentation for jshop2.
Technical Report CS-TR-4694, University of Maryland, Department of Computer Science, University of Maryland, College Park, MD 20742, USA, May 2006.

D. Nau, T.-C. Au, O. Ilghami, U. Kuter, D. Wu, F. Yaman, H. Munoz-Avila, and J.W. Murdock.
Applications of shop and shop2.
Intelligent Systems, IEEE, 20(2):34–41, 2005.

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

Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila.
Shop: simple hierarchical ordered planner.
In *Proceedings of the 16th international joint conference on Artificial intelligence - Volume 2*, IJCAI'99, pages 968–973, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

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

References II

Reid Simmons and Manuela Veloso.
Planning, execution, and learning, lecture notes.
http://www.cs.cmu.edu/ mmv/planning/, September 2010.

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