Automated Planning

Theory and practice

Damien Pellier
Damien.Pellier@math-info.univ-paris5.fr
http://www.math-info.univ-paris5.fr/~pellier/

MASTER II Informatique
UFR de Mathématiques et d'Informatique

PARIS DESCARTES

Part I

Introduction and Overview

Outline of Introduction

Intuitive Planning Definition

What is Planning ?

- Planning is the reasonning side of acting. It is an abstract, explicite deliberation process that chooses and organizes action by anticipating their outcomes.
- This deliberation aims at achieving as best as possible some pretated objectives.
- Automated planning is an area of Artificial Intelligence (AI) that studies this deliberation process computationally.

Automated Planning Motivations

1. *Practical Motivations:* Designing information processing tools that give access to affordable and efficient planning ressources.

Example

Imagine a rescue operation after a natural disaster.

- That operation involves a large number of actors and require transportation infrastuctures.
- It relies on careful planning and assessment of several alternate plans.
- It is also time constrainted and it demands immediate decisions that must be supported with a planning tool.

2. *Theorical Motivations:* Planning is an important component of rational behavior.

Path and Motion Planning

Path and Motion Planning is concerned with the synthesis of a geometric path from a starting position in space to a goal and a control trajectory along that path that specifies the state variables in the configuration space of mobile systems, such as a truck, a mechanical arm, a robot, *etc.*

Motion planning takes into account:

- the model of the environment
- the kinematic constraints
- the dynamic contraints

Perception Planning

Perception Planning is concerned with plans involving senseing actions for gathering informations. It arises in tasks such as modeling environements or objects, identifying objects, localizing through sensing a mobile system, or more generally identifying the current state of the environment.

- Perception planning addresses question such as information needed and when it is needed, where to look for it, which sensors are most adequate for a particular task and how to use them.
- It relies on decision theory for problems of which and when information is needed, on mathematical programming and constraint satisfaction for viewpoint selection and the sensor modalities.

Navigation Planning

Navigation Planning combines the two previous problems of motion and perception planning in order to reach a goal or to explore an area. the purpurse of navigation planning is to synthetize a policy that combines localization primitives and sensor-based motion primitives.

Example

- visually following a road until reaching some landmark
- moving along some heading while avoiding obstacles
- *etc.*

Manipulation Planning

Manipulation Planning is concerned with handling objects, *e.g.*, to build assemblies.

- The actions include sensory information.
- A plan might involve picking up an object from its marked sides, returning it if needed, inserting it into an assembly, and pushing lightly till it clips mechanically into position.

Manipulation Planning

Manipulation Planning arises in dialog and in cooperation problems between several agents, human or artificial. It addresses issues such as when and how to query needed information and which feedback should be provided.

Domain Specific Approaches

Domain specific approaches to specific forms of planning are certainly well justified. However, they are frustrating for several reasons.

1. Some commonalities to all these forms of planning are not adressed in the domain specific approaches. The study of these commonalities is needed for understanding the process of planning.
2. It is more costly to adress each planning problem anew instead of relying on and adapting some general tools.
3. Domain specific approaches are not satisfactory for studying and designing an autonomous intelligent machine. Its deliberative capabilities will be limited to areas for which it has a domain specific planner.

Domain Independent Approaches

Domain independent approaches relies on abstract, general models of actions. These models range from simple ones that allow only for limited forms of reasoning to models with richer prediction capabilities. There are in particular the following forms of models and planning capabilities.

1. *Project Planning* in which models of actions are reduced mainly to temporal and precedence constraints, *e.g.*, the earliest and the latest start times of an action or its latency with respect to another action. Project planning is used for interactive plan edition and verification.
2. *Scheduling and resources allocation* in which the action models include the above types of constraints plus constraints on the resources to be used by each action.
3. *Plan synthesis* in which the action models enrich the precedent models with the conditions needed for applicability of an action and the effects of the action on the state of the world.

State Transition System

The conceptual model of planning can be represented as a *state transition system*. Formally, a state transition system is a 4-tuple $\Sigma = (S, A, E, \gamma)$, where:

- $S = \{s_1, s_2, \ldots, s_n\}$ is a finite or recursively enumerable set of states
- $A = \{a_1, a_2, \ldots, a_n\}$ is a finite or recursively enumerable set of actions
- $E = \{e_1, e_2, \ldots, e_n\}$ is a finite or recursively enumerable set of events
- $\gamma : S \times A \times E \rightarrow 2^S$ is a state transition function

A state transition system may be represented by a directed graph whose nodes are the state in S

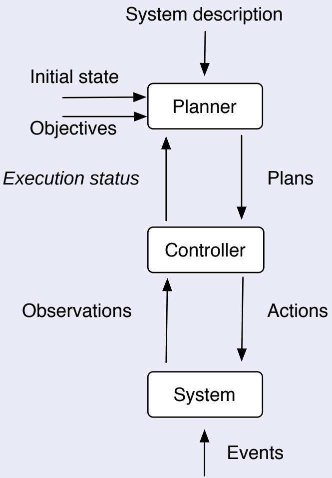
Planning Objectives

Given a state transition Σ, the purpose planning is to find which actions to apply to which states in order to achieve some objective when starting from a given situation. A *plan* is a structure that gives the appropriate actions. The objective can be specified in several different ways.

1. The simplest specification consists of a *goals state* s_g or a set of goal states S_g. In this case, the objective is achieved by any sequence of state transition that ends at one of the goal states.
2. The objective can be also expressed by the satisfaction of some conditions over the sequence of state followed by the system.
3. The objective can be expressed by an utility function attached to each states, with penalties and rewards. The goal is to optimize some compound function of these utilities.
4. The objective can be expressed as a tasks that the system should perform.

Graphical Representation of Planning Model

Planning Conceptual Model



It is convenient to depict conceptual planning model through the interaction between three components:

- A *state transition system* Σ evolves as specified by its state transition function γ, according to the events and actions that it receives.
- A *controller*, given as input the state s of the system, provides as output an action a according to some plan.
- A *planner*, given as input a description of the system Σ, an initial situation, and some objective, synthesizes a plan for the controller in order to achieve the objectives.

Crane and robot transportation example I

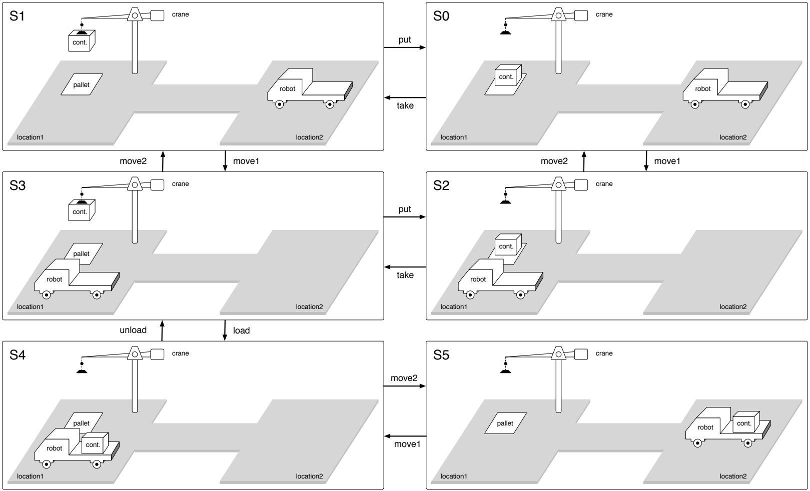


Figure: shows a state transition system involving a container in a pile, a crane that pick up and put down the container and a robot that can carry the container and move it from one location to another.

Crane and robot transportation example II

In this example:

- the set of states is $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$
- the set of actions is $A = \{take, put, load, unload, move1, move2\}$
- the set of events E is empty
- the transition function γ is defined by: if a is an action and $\gamma(s, a)$ is not empty then a is *applicable* to state s

Restricted Model

Planning model puts forward various restrictive assumptions, particularly the following ones.

- *Finite* Σ. The system Σ has a finite set of states.
- *Fully Observable* Σ. The system Σ is *fully observable*, *i.e.,* one has complete knowledge about the state of Σ.
- *Deterministic* Σ. The system Σ is *deterministic*, *i.e.,* , for every states s and for every event of action u, $|\gamma(s, u)| \leq 1$. If an action is applicable to a state, its application brings a deterministic system to a single other state.
- *Static* Σ. The system Σ is *static*, *i.e.,* the set of event E is empty. Σ has no internal dynamics.

Restricted Model

- *Restricted Goals*. The planner handles only *restricted goals* that are specified as an explicit goal state s_g or a set of goal states S_g.
- *Sequential Plans*. A solution plan to a planning problem is a linearly ordered finite sequence of actions.
- *Implicit time*. Actions and events have no duration. They are instantaneous, state transitions. This assumptions is embedded in state transition systems, a model that does not represent time explicitly.
- *Offline Planning*. The planner is not concerned with any change that may occur in Σ while it is planning. It plans for a given initial and goal states regardless of the current dynamics, if any.

Bibliography

M. Ghallab, D. Nau and P. Traverso
Automated Planning Theory and Practice
Morgan Kaufmann, 2004

P. Régnier and V. Vidal
Algorithmes de la planification en IA
Cépaduès, 2004

Part II

Classical Representation for Planning

Outline of Part II

Introduction

We discuss three different ways to represent classical planning problems. Each of them is equivalent in expressive power.

- *Set theoric representation*, each state of the world is a set of propositions and each action is a syntactic expression specifying which propositions belong to the state in order for the action to be applicable and which propositions the action will add or remove to change the state of the world.
- *Classical representation*, the states and the actions are like the ones described for set theoric representation except that first order literals and logical connectives are used instead propositions.
- *State variable representation*, each state is represented by a tuple of value n state variables $\{x_1, \ldots, x_n\}$ and each action is represented by a partial function that map this tuple into some other tuple of values of the n states.

Planning Domains, Problems and Solutions

A *set theoric representation* relies on a finite set of proposition symbols that are intented to represent various propositions about the world. We need to define the basic notion of

- Planning Domain
- Planning Problem
- Planning Solution

Planning Domains Definition

Definition (Set Theoretic Planning Domain)

Let $L = \{p_1, \ldots, p_n\}$ be a finite set of *proposition symbols*. A *set theoretic planning domain* on L is a restricted state transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq 2^L$, *i.e.*, each state s is a subset of L. If $p \in s$ then p holds in s. Otherwise p does not hold in s (Closed World Assumption).
- Each action $a \in A$ is a triple of subset of L written $a = (\text{precond}(a),$ $\text{effect}^-(a), \text{effect}^+(a))$ and $\text{effect}^-(a)$ $\text{effect}^+(a)$ are disjoint.
- S has the property that if $s \in S$, then, for every action a that is applicable to s, the set $(s - \text{effect}^-(\text{a})) \cup \text{effect}^+(a) \in S$.
- The state transition function is $\gamma(s, a) = (s - \text{effect}^-(\text{a})) \cup \text{effect}^+(a)$ if $a \in A$ is applicable to $s \in S$.

Planning Problem Definition

Definition (Set Theoric Planning Problems)

A *set theoric planning problem* is a triple $\mathcal{P} = (\Sigma, s_0, g)$ where

- s_0, the *initial state*, is a member of S
- $g \subseteq L$ is a set of propositions called *goal propositions* that give the requirements that a state must be satisfy in order to be a goal state. The set of goal states is $S_g = \{s \in S \mid g \subseteq s\}$.

Planning Problem Example

Here is one possible set theoric representation of the domain described in figure 2.

Example (Set of propositions)

$L = \{$onground, onrobot, holding,at1,at2$\}$ where

- onground means that the container is on the ground
- onrobot means that the container is on the robot
- holding means that the crane is holding the container
- at1 means that the robot is at location1
- at2 means that the robot is at location2

Planning Problem Example

Example (Set of states)

$S = \{s_0, \ldots, s_5\}$ where

- $s_0 = \{onground, at2\}$; $s_1 = \{holding, at2\}$; $s_2 = \{onground, at1\}$
- $s_3 = \{holding, at1\}$; $s_4 = \{onrobot, at1\}$; $s_5 = \{onrobot, at2\}$

Example (Set of actions)

$A = \{$take, put, load, unload, move1, move2$\}$ where

- take $= (\{$onground$\}, \{$onground$\}, \{$holding$\})$
- put $= (\{$holding$\}, \{$holding$\}, \{$onground$\})$
- load $= (\{$holding, at1$\}, \{$holding$\}, \{$onrobot$\})$
- unload $= (\{$onrobot, at1$\}, \{$onrobot$\}, \{$holding$\})$
- move1 $= (\{$at2$\}, \{$at2$\}, \{$at1$\})$
- move2 $= (\{$at1$\}, \{$at1$\}, \{$at2$\})$

Plan Definition

Definition (Plan)

A *plan* is any sequence of action $\pi = \langle a_1, \ldots, a_k \rangle$, where $k \geq 0$. The *length* of the plan $|\pi| = k$, the number of actions. If $\pi_1 = \langle a_1, \ldots, a_k \rangle$ and $\pi_2 = \langle a'_1, \ldots, a'_j \rangle$ are plans, then their *concatenation* is a plan $\pi_1 \cdot \pi_2 = \langle a_1, \ldots, a_k, a'_1, \ldots, a'_j \rangle$.

The state produced by applying π to a state s is the state that is produced by applying the action of π in the order given. We will denote this by extending the state transition function γ as follows:

$$\gamma(s, \pi) = \begin{cases} s & \text{if} k = 0 \\ \gamma(\gamma(s, a_1, \langle a_2, \ldots, a_k \rangle) & \text{if} k > \text{and } a_1 \text{ is applicable to } s \\ \text{undefined} & \text{otherwise} \end{cases}$$

Plan Solution Definition

Definition (Plan Solution)

Let $\mathcal{P} = (\Sigma, s_0, g)$ be a planning problem. A plan π is a *solution* for $\mathcal{P}$ if $g \subseteq \gamma(s_0, \pi)$.

A solution can have two proprieties:

1. A solution plan π is *redundant* if there is a proper subsequence of π that is also a solution of $\mathcal{P}$.
2. A solution plan π is *minimal* if no other solution plan for $\mathcal{P}$ contains fewer actions that π.

Plan Solution Example

Example

In the planning domain described previously, suppose the initial state is s_0 and $g = \{onrobot, at2\}$. Let

- $\pi_1 = \langle move2, move2 \rangle$
- $\pi_2 = \langle take, move1 \rangle$
- $\pi_3 = \langle take, move1, put, move2, take, move1, load, move2 \rangle$
- $\pi_4 = \langle take, move1, load, move2 \rangle$
- $\pi_5 = \langle move1, take, load, move2 \rangle$

Then π_1 is not a solution because it is not applicable to s_0; π_2 is not a solution because although it is applicable to s_0, the resulting state is not a goal state; π_3 is a redundant solution; π_4 and π_5 are the only minimal solutions.

Properties of the Set Theoric Representation

1. **Readability.** On advantage of the set theoric representation is that it provides a more concise and readable representation of the state transition system than we would get by enumerating all of the states and transition explicitly.
2. **Computation.** A propositions in a state s is assumed to *persist* in $\gamma(s, a)$ unless explicitly mentioned in the effects of a. The effects are defined with two subsets: $\text{effect}^-(a)$ and $\text{effect}^+(a)$. Hence, the transition function γ and the applicability conditions of actions rely on very early computable set operations: if $\text{precond}(a) \subseteq s$;, then $\gamma(s, a) = (s - \text{effect}^-(a)) \cup \text{effect}^+(a)$.
3. **Expressibility.** A significant problem is that not every state transition system Σ has a set theoric representation.

Classical Representation

The classical representation scheme generalize the set theoric representation scheme using notation derived from first order logic.

- *States* are represented as set of logicals atoms that ere true or false within some interpretation.
- *Actions* are represented by *planning operators* that change the truth values of theses atoms.

States Representation

The classical planning language is built on a first order language $\mathcal{L}$.

Definition (State)

A state is a set of ground atoms of $\mathcal{L}$. $\mathcal{L}$ has no function symbols. Thus the set S of all possible states is guaranteed to be finite. As in the set of theoric representation scheme, an atom p holds in s iff $p \in s$. If g is a set of literals, we will say that s satisfies g (denoted $s \models g$) when there is a substitution σ such that every positive literal of $\sigma(g)$ is in s and no negated literal of $\sigma(g)$ is in s.

States Representation Example

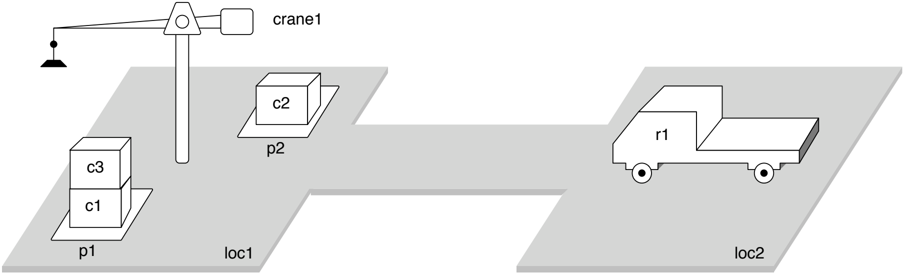


Figure: Initial state $s_0 = \{$ attached(p1, loc1), attached(p2, loc1); in(c1, p1, in(c3, p1), top(c3, p1), on(c3, c1), on(c1, pallet) in(c2, p2), top(c2, p2), on(c2,pallet), belong(crane1, loc1), empty(crane1), adjacent(loc1, loc2), adjacent(loc2, loc1), at(r1, loc2), occupied(loc2), unloaded(r1)$\}$.

Planning Operator Definition

The planning operators define the transition function γ of the state transition system.

Definition (Planning Operator)

A planning operator is a triple $o = (\text{name}(o), \text{precond}(o), \text{effects}(o))$ whose elements are follows:

- name(o), the name of the operator, is a syntactic expression of the form $n(x_1, \ldots, x_k)$ where n is a symbol called an operator symbol (n is unique in $\mathcal{L}$) and $x_1, \ldots, x_k$ are all variable symbols that appear anywhere in o.
- precond(o) and effects(o), the preconditions and effects of o, respectively are generalizations of the preconditions and the effects of the set theory action, *i.e.*, instead of being sets of proposition they are sets of literals.

Planning Operator Example

Example (Take operator)

The planning operator take(k,l,c,d,p) can be defined as follow:

;; crane k at location l takes c off of d in pile p

take(k,l,c,d,p)

precond: belong(k,l), attached(p,l), empty(k), top(k), on(c,d)

effects: holding(k,c), ¬empty(k), ¬in(c,p), ¬top(c,p), ¬on(c,d), top(d,p)

Action Definition

Definition (Action)

An *action* is any ground instance of planning operator. If a is an action and s is a state such that $\text{precond}^{+}(a) \subseteq s$ and $\text{precond}^{-}(a) \cap s = \emptyset$, then a is applicable to s, and the result of applying a to s is the state:

$$\gamma(s, a) = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a)$$

Thus, like in set theoric planning, state transitions can easily be computed using set operations.

Action Example

Example

The action take(*crane*1,*loc*1,*c*3,*c*1,*p*1) is applicable to the state s_0 of the figure 2. The result is the state $s_5 = \gamma(s_0, \text{take}(crane1, loc1, c3, c1, p1))$ shown by the figure below.

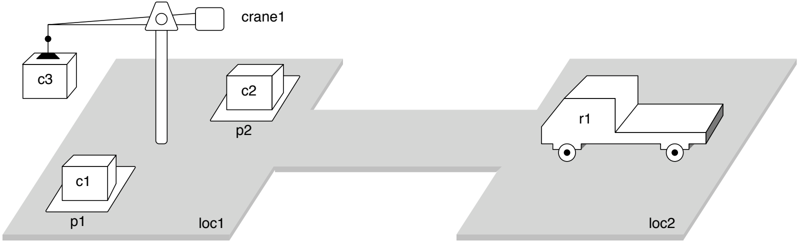


Figure: $s_5 = \{$ attached(p1, loc1), in(c1, p1), top(c1,p1), on(c1, pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), holding(crane1,c3), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)$\}$.

Classical Planning Domains Definition

Definition (Classical Planning Domain)

Let $\mathcal{L}$ be a first order language that has finitely many predicate symbols and constraint symbols. A classical planning domain in $\mathcal{L}$ is a restricted state transition system $\Sigma = (S, A, \gamma)$ such that:

- $S \subseteq 2^{\text{all ground atoms of} \mathcal{L}}$
- $A = \{\text{all ground instances of the operators in } \mathcal{O}\}$ where $\mathcal{O}$ is a set of operators as defined earlier
- $\gamma(s, a) = (s - \text{effects}^-(a)) \cup \text{effects}^+(a)$ if $a \in A$ is applicable to $s \in S$ and otherwise $\gamma(s, a)$ is undefined
- S is closed under γ, *i.e.*, if $s \in S$, then for every action a that is applicable to s, $\gamma(s, a) \in S$.

Classical Planning Problems Definition

Definition (Classical Planning Problem)

A classical planning problem is a triple $\mathcal{P} = (\mathcal{O}, s_0, g)$ where:

- $\mathcal{O}$ is the set of planning operators
- s_0, the initial state, is any state in S
- g, the goal, is any set of ground literals
- $S_g = \{s \in S \mid s \text{ satisfies } g\}$

Plan Example

Example

Consider the following plan:

$$\pi_1 = \langle \text{take(crane1, loc1,c3,c1,p1)}, \text{move(r1,loc2,loc1)}, \text{load(crane1,loc1,c3,r1)} \rangle$$

This plan is applicable to the state s_0 shown in figure 2 producing the state s_6. We verify that

$$g_1 = \{\text{loaded(r1,c3), at(r1,loc1)}\}$$

is included in s_6.

Action Example

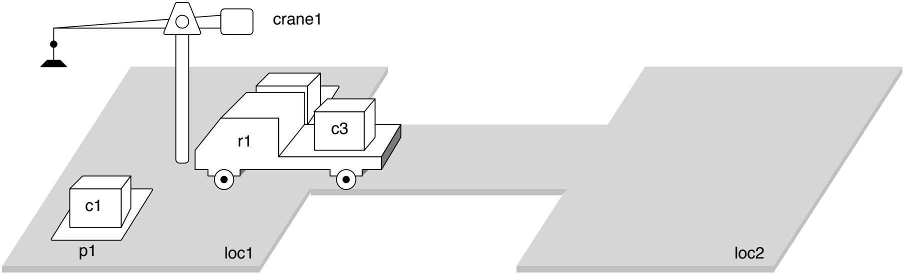


Figure: $s_6 = \{$ attached(p1, loc1), in(c1, p1), top(c1,p1), on(c1, pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), empty(crane1), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc1), occupied(loc1), loaded(r1)$\}$.

Extending the Classical Representation

Classical planning formalism is very restricted, extensions to it are needed in order to describe interesting domains. The most important extensions are :

- Typing variables
- Conditional Planning Operators
- Quantified expression
- Disjunctive preconditions
- Axiomatic Inference
- *etc.*

A planning langage, called PDDL, has been developed to express all these extensions (PDDL stands for Planning Domain Description Langage).

PDDL Example

Example (Crane robot transportation domain)

```
(define (domain dwr)
    (:requirements :strips :typing)
    (:types location pile robot crane container)
    (:predicates
        (adjacent ?l1 ?l2 - location) (attached ?p - pile ?l -location)
        (belong ?k - crane ?l - location) (at ?r - robot ?c container)
        (occupied ?l - location) etc.)
    (:action move
        (:parameters (?r - robot ?from ?to - location))
        (:precondition (and (adjacent .from ?to) (at ?r ?from)
            (not (occupied ?to))))
        (:effect (and (at ?r ?to) (not (occupied ?from)) (occupied ?to)
            (not (at ?r ?from)))))
    (:action load
        etc.))
```

Further readings

V. Lifschitz
On the semantics of STRIPS.
Reasoning about actions and plans 1-9, Morgan Kaufmann, 1987

B. Nebel
On the compatibility and expressive power of propositional planning formalism.
Journal of Artificial Intelligence Research 12:271-315, 2000

D. McDermott
PDDL, the Planning Domain Definition Language.
Technical report. Yale Center for Computational Vision and Control, 1998
ftp://ftp.cs.yale.edu/pub/mcdermott/software/pddl.tar.gz

Part III

State Space Planning

Outline of Part IV

Introduction

What is State Space Planning ?

- The simplest classical planning algorithms.
- Search algorithms in which the search space is a subset of the state space:
  - Each node corresponds to a state of the world.
  - Each arc corresponds to a state transition.
  - The current plan corresponds to the current path in the seach space.

Forward Search Principle

- The forward search algorithm is nondeterministic
- The forward search algorithm is sound and complete
- The forward search algorithm takes as input the statement $P = (\mathcal{O}, s_0, g)$ of a planning problem $\mathcal{P}$. If $\mathcal{P}$ is solvable, then Forward-search($\mathcal{O}$, s_0, g) returns a solution plan. Otherwise it returns failure.
- The plan returned by each recursive invocation of the algorithm is called a *partial solution* because it is part of the final solution returned by the top level invocation.

Forward Search Algorithm

Algorithm (ForwardSearch($\mathcal{O}$, s_0, g))

if *s satisfies g* **then return** *an empty plan* π
active $\leftarrow \{a \mid$ *a is a ground instance of an operator* $\mathcal{O}$
and precond(a) is true in s $\}$
if active $= \emptyset$ **then return** Failure
nondeterministically choose an action $a_1 \in$ active
$s_1 \leftarrow \gamma(s, a_1)$
$\pi \leftarrow$ `ForwardSearch(`$\mathcal{O}$, s_1, g`)`
if $\pi \neq$ Failure **then return** $a_1 \cdot \pi$
else return Failure

Forward Search Example

Take the state s_5 defined in figure 4:

$s_5 = \{$ attached(p1, loc1), in(c1, p1), top(c1,p1), on(c1, pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), holding(crane1,c3), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)$\}$.

and the goal:

$g = \{$ at(r1,loc1), loaded(r1,c3)$\}$.

If the ForwardSearch algorithm chooses the action $a =$ move(r1,loc2,loc1) in the first invocation and $a =$ load(crane1, loc1,c3,r1) in the second invocation producing the state s_6. s_6 staisfies g, the execution returns:

$\pi = \langle$move(r1,loc2,loc1), load(cran1,loc1,c3,r1)$\rangle$

Forward Search Example

Warning

There are many other execution traces, some of which are infinite. For instance, one of them makes the following infinite sequence of choices for *a*:

- move(r1,loc2,loc1)
- move(r1,loc1,loc2)
- move(r1,loc2,loc1)
- move(r1,loc1,loc2)
- *etc.*

Backward Search Principle and Algorithm

The idea is to start at the goal and apply inverses of the planning operator to produce subgoals, stopping if we produce a set of subgoals satisfied by the initial state. The backward search algorithm is sound and complete.

Algorithm (BackwardSearch($\mathcal{O}$, s_0, g))

if s_0 *satisfies* g **then return** *an empty plan* π
revelant $\leftarrow \{a \mid$ *a is a ground instance of an operator* $\mathcal{O}$
that is revelant for g $\}$
if revelant $= \emptyset$ **then return** Failure
nondeterministically choose an action $a_1 \in$ revelant
$s_1 \leftarrow \gamma^{-1}(s, a_1)$
$\pi \leftarrow$ `BackwardSearch`($\mathcal{O}$, s_1, g)
if $\pi \neq$ Failure **then return** $a_1 \cdot \pi$
else return Failure

Backward Search Example

Recall that the initial state is the state s_5:

$s_5 = \{$ attached(p1, loc1), in(c1, p1), top(c1,p1), on(c1, pallet), attached(p2,loc1), in(c2,p2), top(c2,p2), on(c2,pallet), belong(crane1,loc1), holding(crane1,c3), adjacent(loc1,loc2), adjacent(loc2,loc1), at(r1,loc2), occupied(loc2), unloaded(r1)$\}$.

and the goal:

$g = \{$ at(r1,loc1), loaded(r1,c3)$\}$.

which is a subset of the state s_6.

Backward Search Example: first invocation

In the first invocation of the BackwardSearch algorithm, it chooses $a =$ load(crane1, loc1,c3,r1) and then assigns:

First Invocation

$$
\begin{array}{lll}
g & \leftarrow & \gamma^{-1}(g, a) \\
 & = & (g - \text{effects}^{+}(a)) \cup \text{precond}(a) \\
 & = & (\{\text{at(r1,loc1), loaded(r1,c3)}\} - \{\text{empty(crane1), loaded(r1,c3)}\}) \\
 & \cup & \{\text{belong(crane1,loc1), holding(crane1,c3), at(r1,loc1),} \\
 & & \text{unloaded(r1)}\} \\
 & = & \{\text{at(r1,loc1), belong(crane1,loc1), holding(crane1,c3),} \\
 & & \text{unloaded(r1)}\}
\end{array}
$$

Backward Search Example: second invocation

In the second invocation of the BackwardSearch algorithm, it chooses $a =$ move(r1, loc2,loc1) and then assigns:

Second Invocation

$$
\begin{aligned}
g &\leftarrow \gamma^{-1}(g, a) \\
&= (g - \text{effects}^{+}(a)) \cup \text{precond}(a) \\
&= (\{\text{at(r1,loc1), belong(crane1,loc1), holding(crane1,c3),} \\
&\quad \text{at(r1,loc1), unloaded(r1)}\} - \{\text{at(r1,loc1), occupied(loc1)}\}) \\
&\cup \{\text{adjacent(loc2,loc1), at(r1,loc2),} \neg\text{occupied(loc1)}\} \\
&= \{\text{belong(crane1,loc1), holding(crane1,c3), unloaded(r1),} \\
&\quad \text{adjacent(loc2,loc1), at(r1,loc2), occupied(loc1)}\}
\end{aligned}
$$

Backward Search Example: result

This time g is satisfied by s_5, so the execution trace terminates and returns the plans:

$\pi = \langle$(move(r1,loc2,loc1), load(crane1,loc1,c3,r1)$\rangle$

Warning

Like ForwardSearch algorithm, there are many other execution traces, some of which are infinite. For instance, one of them makes the following infinite sequence of choices for a:

- load(crane1,loc1,c3,r1)
- unload(crane1,loc1,c3,r1)
- load(crane1,loc1,c3,r1)
- unload(crane1,loc1,c3,r1)
- *etc.*

STRIPS Algorithm Principle

- The biggest problem of the previous approaches is how improve efficiency by reducing the size of the search space.
- STRIPS is somewhat similar to the BackwardSearch but differs from it in the following ways:
  1. In each recursive call of the STRIPS algorithm, the only subgoals eligible to be worked on are the preconditions of the last operator added to the plan. This reduce the branching factor substantially. However, it makes STRIPS incomplete.
  2. If the current state satisfies all of on operator's preconditions, STRIPS commits to executing that operator and will not backtrack over this commitment. This prune a large portion of the search space but again make STRIPS incomplete.

STRIPS Algorithm

Algorithm (STRIPS($\mathcal{O}$, s, g))

$\pi \leftarrow$ *the empty plan*
while true **do**
  if *s satisfies g* **then return** π
  revelant $\leftarrow \{a \mid$ *a is a ground instance of an operator* $\mathcal{O}$
    that is revelant for g $\}$
  if revelant $= \emptyset$ **then return** Failure
  nondeterministically choose an action $a \in$ revelant
  $\pi' \leftarrow$ `STRIPS`($\mathcal{O}$, s, *precond(a)*)
  if $\pi' =$ Failure **then return** Failure
  ;; if we get here, then π' *achieves precond(a) from s*
  $s \leftarrow \gamma(s, \pi')$
  ;; s now satisfies precond(a)
  $s \leftarrow \gamma(s, a)$
  $\pi' \leftarrow \pi \cdot \pi \cdot a$
end

Sussman Anomaly

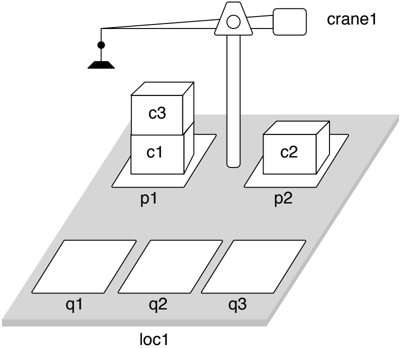


Figure: $s_0 = \{$ in(c3,p1), top(c3,p1), in(c1,p1), on(c3, c1), on(c1,pallet), in(c2,p2), top(c2,p2), on(c2,pallet), top(pallet,q1), top(pallet,q2), top(pallet, q3), empty(crane1) $\}$

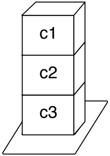


Figure: $g = \{$ on(c1,c2), on(c2,c3) $\}$

STRIPS result for the Sussman Anomaly

The shortest solutions that STRIPS can find are all similar to the following:

take(c3,loc1,crane1,c1)	
put(c3,loc1,crane1,q1)	
take(c1,loc1,crane1,p1)	
put(c1,loc1,crane1,c2)	*STRIPS has achieved on(c1,c2)*
take(c1,loc1,crane1,c2	
put(c1,loc1,crane1,p1)	
take(c2,loc1,crane1,p2)	
put(c2,loc1,crane1,c3)	*STRIPS has achieved on(c2,c3)* *but needs to reachieved on(c1,c2)*
take(c1,loc1,crane1,p1)	
put(c1,loc1,crane1,c2)	*STRIPS has now achieved both goals*

STRIPS result for the Sussman Anomaly

- STRIPS does not always find the best solution.
- STRIPS's difficulty involves *deleted condition interaction*.

Example

The action take(c1,loc1,crane1,c2) is necessary in order to help achieve on(c2,c3) but it deletes the previous acheived condition on(c1,c2).

- One way to find the shortest plan for Sussman anomaly is to interleave plans for different goals.

Note

This observation such as these led to the development of a technique called *plan space planning*, in which the planning system searches thought a space whose nodes are partial plans rather that states of the world.

Exercice

Consider the Sussman anomaly shown in figures 1 and 1. The shortest plan π_1 for achieving on(c1,c2) from the initial state is:

$$\pi_1 = \langle \text{take(c3,loc1,crane1,c1)} \\ \text{put(c3,loc1,crane1,q1)} \\ \text{take(c1,loc1,crane1,p1)} \\ \text{put(c1,loc1,crane1,c2)} \rangle$$

and the the shortest plan π_2 for achieving on(c2,c3) from the initial state is:

$$\pi_2 = \langle \text{take(c2,loc1,crane1,p2)} \\ \text{put(c2,loc1,crane1,c3)} \rangle$$

How to interleave π_1 and π_2 to find the shortest plan for the Sussman anomaly ?

Further readings

R. Fikes and N. Nilsson
STRIPS: A new approach to the application of theorem proving to problem solving.
Artificial Intelligence 2(3-4):189-208, 1971

G. Sussman
A computational model of skill acquisition.
New York: Elsevier, 1975

J. Hoffmann
FF: The fast forward planning system.
Artificial Intelligence Magazine 22(3):57-62, 2001

Part IV

Plan Space Planning

Outline of Part IV

Introduction

- The search space is no more a states space but a plans space.
  - Nodes are partially specified plans.
  - Arcs are *plan refinement operations* intended to further complete a partial plan, *i.e.,* to achieve an open goal or to remove a possible inconsistency.
- Solution plan definition changes. Planning is considered as two separate operations:
  1. the choice of actions
  2. the ordering of the chosen actions so to achieve the goal.

Plan Space Example

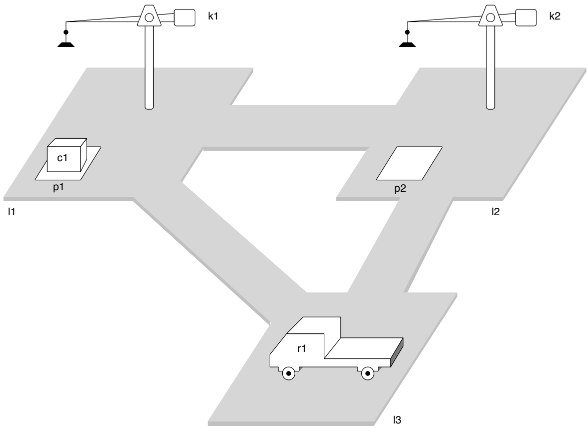


Figure: A robot r1 has to move a container c1 from pile p1 at location l1 to pile p2 and location l2. Initially r1 is unloaded at location l3. There are empty cranes k1 and k2 at locations l1 and l2. Pile p1 at location l1 contains only container c1; pile p2 at location l2 is empty. All location are adjacent.

Plan Space Example

Consider we have a partial plan that contains only the two following actions

- take(k1,c1,p1,l1): crane k1 picks up container c1 from pile 1 at location l1
- load(k1,c1,r1,l1): crane k1 loads container c1 on robot r1 at location l1

Let us refine it by adding a new action and let us analyse how the partial plan should be updated. We will come up with four ingredient:

1. adding actions
2. adding ordering constraints
3. adding causal relationship
4. adding variable binding constraints

Adding Actions Example

Nothing in this partial plan guarantees that robot r1 is already at location l1. Proposition at(r1,l1), required as a precondition by action load, is a *subgoal* in this partial plan. We need to add the following action:

move(r1,l,l1): robot r1 moves from location l to the required location l1.

Adding Ordering Constraints

This additional move action achieves its purpose only if it is constrained to come *before* the load action. But should the move action come *before* or *after* the take action? Both are possible.

Least Commitment principle

Not add a constraint to a partial plan unless it is strictly needed. May permit to run actions concurrently.

Adding Causal Links

In partial plan, we have added one action and an ordering constraint to another action already in the plan. Is that enough? No quite. Because

there is no explicit notion of a current state (*e.g.*, an ordering constraint does not say that the robot stays at location l1 until load action is performed). Hence, we will be encoding explicitly in the partial plan the reason why action move was added: to satisfy the subgoal at(r1,l1) required by action load.

This relationship between the two actions move and load with respect to proposition at(r1,l1), is called a *causal link*.

Note

The former action is called the *provider* of the proposition, the later the *consumer*. The role of a causal link is to state that a precondition is *supported* by another.

Adding Variable Binding Constraints

A final item in the partial plan that goes with refinement we are considering is that variable binding constraints.

- Operators are added in the partial plan with systematic variables renaming.
- We should make sure that the new operator move concerns the same robot r1 and the same location l1 as those in the operator take and load.
- What about l the robot will be come from? At this stage there is no reason to bind this variable to a constant. The variable l is kept unbounded.

Partial Plan Definition

Definition (Partial Plan)

A *partial plan* is a tuple $= (A, \prec, B, L)$ where:

- $A = \{a_1, \ldots, a_k\}$ is a set of partially instantiated planning operators.
- $\prec$ is a set of ordering constraints on A of the form $(a_i \prec a_j)$.
- B is a set of binding constraints on the variables of actions in A of the form $x = y$, $x \neq y$, or $x \in D_x$, D_x being a subset of the domain of x.
- L is a set of causal links of the form $\langle a_i \xrightarrow{p} a_j \rangle$, such that a_i and a_j are actions in A, the constraint $(a_i \prec a_j)$ is in $\prec$, proposition p is an effect of a_i and a precondition of a_j, and the binding constraints for variables of a_i and a_j appearing in p are in B.

Partial Plan Example

Let us illustrate two partial plans corresponding figure 1. The goal of having container c1 in pile p2 can be expressed simply as in(c1,p2). The initial state is:

{ adjacent(l1,l2), adjacent(l1,l3), adjacent(l2,l3), adjacent(l2,l3), adjacent(l3,l1), adjacent(l3,l2), attached(p1,l1), attached(p2,l2), belong(k1,l1), belong(k2,l2), empty((k1), empty(k2), at(r1,l3), unloaded(r1), occupied(l3), in(c1,p1), on(c1,pallet), top(c1,p1), top(pallet,p2) }

Partial Plan Example

A graphical representation of the initial plan π_0 is shown in figure 3. Each box is an action preconditions above and effects below the box. Solid arrows are ordering constraints; dashed arrows are causal links; and binding constraint are implicit or shown directly in the arguments.

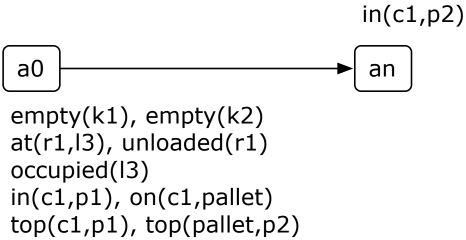


Figure: Initial plan π_0.

Partial Plan Example

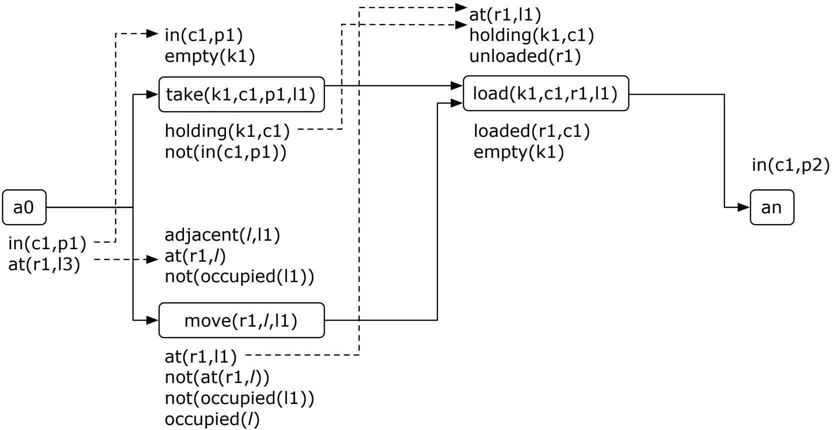


Figure: A partial plan

Solution Plan Definition

Definition (Solution Plan)

A partial plan $\pi = (A, \prec, B, L)$ is a *solution plan* for a problem $P = (\Sigma, s_0, g)$ if:

- its ordering constraints $\prec$ and binding constraints B are consistent.
- every sequence of totally ordered and totally instantiated actions of A satisfying $\prec$.
- B is a sequence that defines a path in the state transition system Σ from the initial state s_0 corresponding to effects of the action a_0 to state containing all goal proposition in g given by preconditions of a_n.

Example: Plan with incorrect sequence

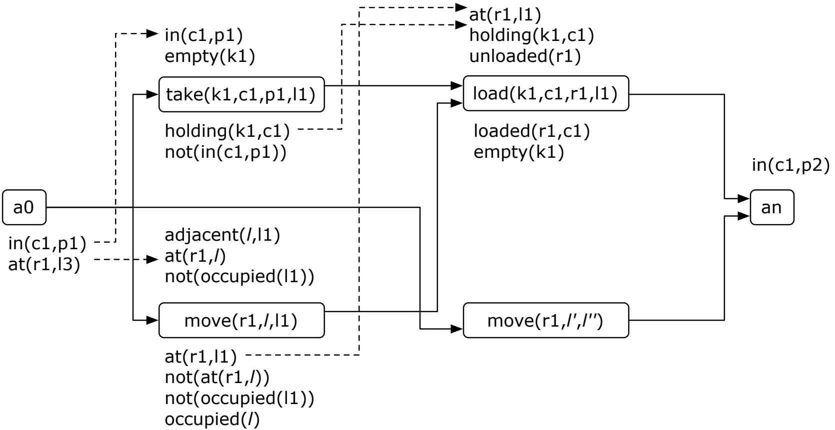


Figure: A plan containing an incorrect sequence

Flaw and Threat

Definition (Threat)

An action a_k in a plan π is a *threat* on a causal link $(a_i \xrightarrow{p} a_j)$ iff:

- a_k has an effect $\neg q$ that is possible inconsistent with p.
- the ordering constraints $(a_i \prec a_k)$ and $(a_k \prec a_j =$ are consistent with B.
- the binding constraints from the unification of q and p are consistent with B.

Definition (Flaw)

A flaw in a plan $\pi = (A, \prec, B, L)$ is either:

- a subgoal, *i.e.*, a precondition of an action in A with out a causal link
- a threat, *i.e.*, an action that may interfere with causal link.

Example: Solution Plan

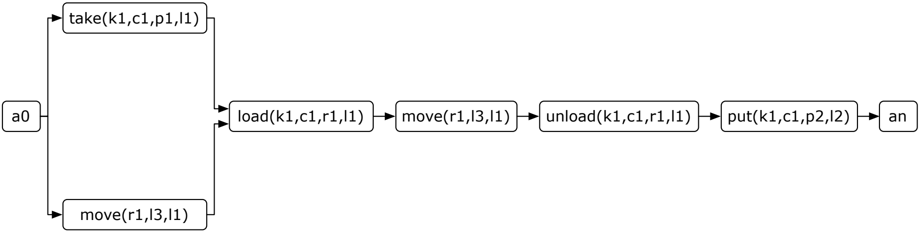


Figure: A solution plan

PSP Principle

A plan π is a solution when it has no flaw, the main principle is to refine π, while maintaining $\prec$ and B consistent, until it has no flaw. The basic operations for refining a partial plan π toward a solution plan are the following:

- Find the flaws of π, *i.e.,* its subgoals and its threats.
- Select on such flaw.
- Find ways to resolve it.
- Choose a resolver for the flaw.
- Refine π according to that resolver.

PSP Algorithm

Algorithm (PSP(π))

flaws $\leftarrow$ `OpenGoals(`π`)` $\cup$ `Threat(`π`)`
if flaws $= \emptyset$ **then return** π
select any flaw sigma $\in$ flaws
resolvers $\leftarrow$ `Resolve(`σ,π`)`
if resolvers $= \emptyset$ **then return** Failure
nondeterministically choose a resolver $\rho \in$ resolvers
$\pi' \leftarrow$ `Refine(`ρ, π`)`
return `PSP(`π'`)`

Attached Procedures

OpenGoals(π). This procedure find all subgoals in π.

Threat(π). This procedure find every action a_k that is a threat on some causal link $(a_i \xrightarrow{p} a_j)$.

Resolve(σ, π). This procedure finds all ways to solve a flaw σ.

Refine(ρ, π). This procedure refines the partial plan π with le elements in the resolver, adding to π on ordering constraint, on or several binding constraints, a causal link, and/or a new action.

Exercice

1. Trace the PSP procedure step-by-step on the Sussman anomaly (see figure 1).
2. Draw the complete graph to compute the solution plan of the figure 4:
   - How many threats are there ?
   - How many plans can be found ?

Further readings

E. Sacerdoti
Planning in a hierarchy of abstraction spaces.
Artificial Intelligence 5:115-135, 1974

J. Penberthy and D.S. Weld
UCPOP: A sound, complete, partial order planner for ADL.
In *Proceedings of the International Conference on Knowledge Representation and Reasoning* 103-114, 1992

Part V

Heuristics in Planning

Outline of Part V

Introduction

- Why heuristics are interested for planning ?
  - Although planning systems have become much more efficient, they still suffer from combinatorial complexity. Even restrited planning domains, the complexity can be intractable in the worst case
- Approache to study heuristics
  - Define a nondeterministic abstract search procedure in a space in which each node u, (*i.e.*, structured collection of actions and constraints) represents a set of solution Π_u, (*i.e.*, the set of all solution reachable from u), For instance, u is
    - in state-space planning, a simple sequence of actions
    - in plan-space planning, a set of actions, causal links, orderig constraints and bindings constraints
    - in graph based planning, a subgraph of the planning graph
    - *etc.*

Abstract Search Procedure (1/2)

- The abstract search procedure involves three main steps in addition to a terminaison step:
  1. A refinement step consists of modifying the collection of actions and/or constraints associated with a node u. In a refinement of u, the set of soltion Π_u remains *unchanged*
     - For instance, if we find out there is only one action a that meets a constraint in u, a is maked an explicit part of u and the constraints is removed
  2. A branching setp generates on or more children of u. These nodes will be the next *candidates* for the next node to visit
     - For instance, in forward state-space seach, each child corresponds to appending a different action to the end of a partial plan
  3. A pruning step consists of removing from the set of candidates nodes some nodes that appear to be unpromising for the search
     - For instance, a node migth be considered to be unpromising if we have a record of having already visited that node

Abstract Search Procedure (2/2)

Algorithm (Abstract-search(u))

```
if Terminal(u) then
    return u
else
    u ← Refine(u)
    B ← Branch(u)
    C ← Prune(B)
    if C = ∅ then
        return Failure
    else
        nondeterministically choose any v ∈ C
        return Abstract-search(v)
    end
end
```

Abstract Search Procedure for Plan-Space Planning

The different steps of the abstract search procedure for plan-space planning are the following:

1. **Branching** consists of selecting flaws and finding its resolvers
2. **Refinement** consists of applying a resolver to the current partial plan
3. **Pruning** : there is no pruning step
4. **Terminaison** occurs when no flaws are left in the partial plan

Note

Since paths in the plan space are likely to be infinite, a control strategy such as best-first search or iterative deepening should be used

Abstract Search Procedure for State-Space Planning

The different steps of the abstract search procedure for state-space planning are the following:

1. **Branching** are defined by actions
2. **Refinement** : there is no branching step
3. **Pruning** removes candidate nodes corresponding to cycle
4. **Terminaison** occurs when the plan goes all the way from the initial state to a goal

Note

A control strategy such as A*, branch-and-bound search or iterative deepening should be used

Abstract Search Procedure for Graph-Based Planning

The different steps of the abstract search procedure for graph-based planning are the following:

1. **Branching** idendifies possible actions that achieve subgoals
2. **Refinement** consists of propaging constraints for actions chosen in the branching step
3. **Pruning** uses the recorded nogood tuples of subgoals that failed in some layer
4. **Terminaison** occurs if the solution-extraction process succeeds

Note

Graph-based planning correspond to using abstract search procedure with iterative deepening control strategy.

Deterministic versus undeterministic search

- To implement a deterministic search procedure a node selection function (`Select(`C`)`) is needed to choose which node u to visite next from a set of candidates C
- Often the deterministic search is done in a depth-first manner

$\Rightarrow$

Algorithm (Depth-first-search(u))

if `Terminal(`u`)` **then** **return** u
else
    $u \leftarrow$ `Refine(`u`)`
    $B \leftarrow$ `Branch(`u`)`
    $C \leftarrow$ `Prune(`B`)`
    while $C = \emptyset$ **do**
        $v \leftarrow$ `Select(`C`)`
        $C \leftarrow C - \{v\}$
        $\pi \leftarrow$ `Depth-first-search(`v`)`
        if $\pi \neq$ Failure **then** **return** π
        return Failure
    end
end

Design Principle for Heuristics : Relaxation

Node delection heuristique

A *node selection heuritic* is any way of ranking a set of nodes in order of ther relative deirability. We will model this heuristic as function h that can be used to compute a numeric evaluation $h(u)$ for each candidates node $u \in C$, *i.e.*,

$$\texttt{Select}(C) = min\{h(u) \mid u \in C\}$$

Notes

1. Node selection heuristics are used for resolving nondeterministic choices
2. If there isa deterministic technique for choosing at each point the rigth node, this technique is not a heuristic
3. A node selection heuristic not always garantees to be the best choice but often lead to the best solution
4. A node selection heuristic must be easy to compute

Design Principle for Heuristics : Relaxation

Node selection heuristics are often based on *relaxation priciple*:

Relaxation Principle

In order to assess how desirable a node u is, one considers a simpler problem that is obtained from the original one by making simplifying assumptions and by relaxing constraints

- One estimates how desirable u is by using u to solve the simpler relaxed problem and using that solution as a, estimate of the solution one would get if pne used u to solve the original problem
- On the other hand, the more simplified the relaxed problem is, the easier it will be to compute the heuristic

Admissible Node Selection Heuristic

A node selection heuristic h is *admissible* if it is a lower bound estimate cost of a minimal solution reachable from u, *i.e.*, $h(u) \leq h^*(u)$ with $h^*(u)$ the minimum cost of any solution reachable from u

- $h^*(u) = \infty$ if no solution is reachable from u

Notes

1. Admissible node selection heuristic is desirable if one seeks a optimal solution with respect to some cost criterion, *e.g.*, path-finding A^*
2. Heuristic search as iterative-deepening scheme, are usually able to garantee on optimal solution when guided with an admissible node selection heuristic

Heuristics for State-Space Planning

Reminder

- In state-space planning, each node u corresponds to a state s
- At some point the candicates nodes are the sucessor states of the current state s, for the actions applicable to s. For each action a to a state s:
  - in forward search the next state is given by the transition function:

$$\gamma(s, a) = (s - \text{effects}^{-}(a)) \cup \text{effects}^{+}(a)$$

  - In backward search the next state is given by the transition function:

$$\gamma(s, a)^{1} = (s - \text{effects}^{+}(a)) \cup \text{precond}(a)$$

Relaxation principle

In order to choose the most preferable candidate state, we need to assess how close each action may bring us to the goal (forward search) or initial state s_0 (backward search).

State Reachability Relaxation

A Simple Relaxation Heuristic (1/2)

- Simple relaxation heuristic idea
  - A very simple relaxation heuristic is to neglect $\text{effects}^{-}(a)$
- Consequences:
  - $\gamma(s, a)$ involves on a monotonic increase in the number of propositions of s
  - It is easier to compute distance goal with such simplified γ

Definition (Simple Relaxation Heuristic)

Let $s \in S$ be a state, p a proposition and g a set of propositions. The *minimum distance* from s to p, denoted $\Delta^*(s, g)$, is the minimum number of actions required to reach from s a state containing all proposition p g.

State Reachability Relaxation

A Simple Relaxation Heuristic (1/2)

- Δ is given by the following equations:

$$\Delta(s,p) = 0 \qquad \text{if } p \in s$$
$$\Delta(s,p) = \infty \qquad \text{if } \forall a \in A, p \notin \text{effects}^+(a)$$
$$\Delta(s,g) = 0 \qquad \text{if } g \subseteq s$$

otherwise :

$$\Delta(s,p) = \min_a\{1 + \Delta(s, \text{precond}(a)) \mid p \in \text{effects}^+(a)\}$$
$$\Delta(s,g) = \Sigma_{p \in g} \Delta(s,p)$$

Notes

1. These equation gives the distance to g in the relaxed problem and
2. an estimate distance in the unrelaxed problem
3. The heuristic function can be define as $h(s) = \Delta(s,g)$

State Reachability Relaxation

The Δ-Algorithm

- The Δ-algorithm is polynomial in time
- As minimum distance graph searching, the algorithm stops when a fixed point is reached

Algorithm (Delta(s))

foreach p **do**
    if $p \in s$ **then** $\Delta(s,p) \leftarrow 0$ **else** $\Delta(s,p) \leftarrow \infty$
    $U \leftarrow \{s\}$
end
repeat
    foreach *a such that* $\exists u \in U$, $precond(a) \subseteq u$ **do**
        $U \leftarrow \{u\} \cup effects^{+}(a)$
        foreach $p \in effects^{+}(a)$ **do**
            $\Delta(s,p) \leftarrow min\{\Delta(s,p), 1 + \Sigma_{q \in precond(a)} \Delta(s,q)\}$
        end
    end
until *no change occurs in the above updates*

Heuristics Guided Forward Search

Algorithm (Heuristic-forward-Search(π, s, g, A))

if *s satisfies g* **then return** π
options $\leftarrow \{a \in \ \mid a \textit{ applicable to } s\}$
foreach $a \in$ options **do** $\Delta(\gamma(s, a))$
while options $\neq \emptyset$ **do**
    $a \leftarrow min\{\Delta(\gamma(s, a), g) \mid a \in \text{options}\}$
    options $\leftarrow$ options $-\{a\}$
    $\pi' \leftarrow$ `Heuristic-forward-Search`$(\pi, a, \gamma(s, a), g, A)$
    if $\pi' \neq$ Failure **then return** π'
end
return Failure

Heuristics Guided Backward Search

Algorithm (Heuristic-backward-Search(π, s_0, g, A))

if s_0 *satisfies* g **then return** π
options $\leftarrow \{a \in \ \mid a \ \textit{revelant for}\ g\}$
while options $\neq \emptyset$ **do**
    $a \leftarrow min\{\Delta(s, \gamma^{-1}(g, a)) \mid a \in$ options $\}$
    options $\leftarrow$ options $-\{a\}$
    $\pi' \leftarrow$ `Heuristic-backward-Search`$(a \cdot \pi, s_0, \gamma^{-1}(g, a), A)$
    if $\pi' \neq$ Failure **then return** π'
end
return Failure

Notes

1. We suppose that Δ-algorithm is run once initially
2. The backward search is more efficient than forward search because it has to be run less Δ-algorithm

Admissible State-Space Heuristics

- It can be desirable to use admissible heuristic function for two reasons:
  1. It may be interested in getting the shortest plan, *e.g.*, cost may be associated to actions
  2. Admissible permit a safe pruning
     - If Y is the length of a plan and if $h(u) < Y$, h being admissible, then we are sure that non solution plan of length smaller that Y can be obtained from u. $\Rightarrow$ pruning does not affect completeness

Exercice

Is the simple heuristic h previouly introduced admissible ?

No, because $\Delta(s, g)$ is not a lower bound on the true minimal distance $\Delta^*(s, g)$. Assume a problem where there is an action a such that:

- $precond(a) \subseteq s_0$,
- $effects^+(a) = g$ *and*
- $s_0 \cap g = \emptyset$.

The distance to the goal is 1, but $\Delta(s_0, g) = \Sigma_{p \in g} \Delta(s_0, p) = |g|$

Admissible State-Space Heuristics

First Admissible heuristic

Idea

Instead of estimating the distance to a set of propositions g to be the sum of the distances to the elements of g, we estimate it to be the maximum distance to its propositions

- Now, Δ_1 is given by the following equations:

$$
\begin{aligned}
\Delta_1(s,p) &= 0 && \text{if } p \in s \\
\Delta_1(s,p) &= \infty && \text{if } \forall a \in A, p \notin \text{effects}^+(a) \\
\Delta_1(s,g) &= 0 && \text{if } g \subseteq s
\end{aligned}
$$

otherwise :

$$
\begin{aligned}
\Delta_1(s,p) &= \min_a\{1 + \Delta_1(s, \text{precond}(a)) \mid p \in \text{effects}^+(a)\} \\
\Delta_1(s,g) &= \max\{\Delta_1(s,p) \mid p \in g\}
\end{aligned}
$$

- Experience shows that h_1 is not as informative as h even if h_1 is admissible

Admissible State-Space Heuristics

Second Admissible heuristic

Idea

Instead of considering that the distance to a set of propositions g is the maximum distance to propositions $p \in g$, we estimate it to be the maximum distance to a pair of propositions $\{p, q\}$

- Now, Δ_2 is given by the following recusive equations (terminaison cases remain unchanged):

$$
\begin{array}{rcl}
\Delta_2(s, p) & = & \min_a\{1 + \Delta_2(s, \text{precond}(a)) \mid p \in \text{effects}^+(a)\} \\
\Delta_2(s, \{p, q\}) & = & \min\{ \\
 & & \min_a\{1 + \Delta_2(s, \text{precond}(a)) \mid \{p, q\} \in \text{effects}^+(a)\} \\
 & & \min_a\{1 + \Delta_2(s, \{q\} \cup \text{precond}(a)) \mid p \in \text{effects}^+(a)\} \\
 & & \min_a\{1 + \Delta_2(s, \{p\} \cup \text{precond}(a)) \mid q \in \text{effects}^+(a)\}\} \\
\Delta_2(s, q) & = & \max_{p,q}\{\Delta_2(s, \{p, q\}) \mid \{p, q\} \subseteq q\}
\end{array}
$$

Algorithm (GraphPlan(A, s_0, g))

$i \leftarrow 0, \nabla \leftarrow \emptyset, P_0 \leftarrow s_0$

repeat

  $i \leftarrow i + 1,\ G \leftarrow \texttt{Expand}(G)$

until $[g \subseteq P_i$ *and* $g \cap \mu P_i = \emptyset]$ *or* $\texttt{Fixedpoint}(G)$

if $g \not\subseteq P_i$ *or* $g \cap \mu P_i \neg \emptyset$ **then return** Failure

$\Pi \leftarrow \texttt{Extract}(G, g, i)$

if $\texttt{Fixedpoint}(G)$ **then return** $\eta \leftarrow |\nabla(\kappa)|$ **else** $\eta \leftarrow 0$

while $\Pi =$ Failure **do**

  $i \leftarrow i + 1,\ G \leftarrow \texttt{Expand}(G), \Pi \leftarrow \texttt{Extract}(G, g, i)$

  if $\Pi =$ Failure *and* $\texttt{Fixedpoint}(G)$ **then**

    if $\eta = |\nabla(\kappa)|$ **then** **return** Failure

    $\eta \leftarrow |\nabla(\kappa)|$

  end

end

return Π

Graphplan as Heuristics Search Planner

Comments

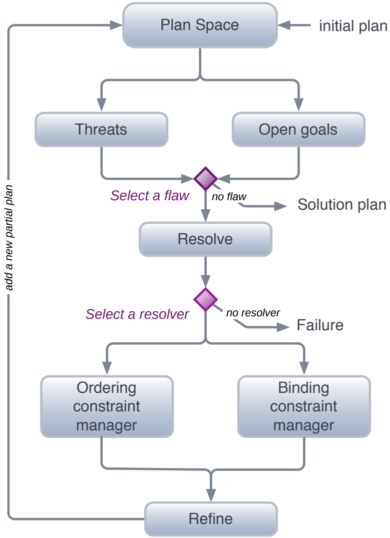
- Graphplan looks like heuritic backward search procedure
  - Δ-procedure and `Expand` procedure in graphplan perform a reachability analysis
  - The main difference :
    - `Expand` builds a data stucture, the planning graph, which provides more information attached to propositions not just distance to s_0
- The planning graph approximate the distance $\Delta^*(s_0, g)$, that is the level of the first layer of the graph that $g \subseteq P_i$ and no pair of g is in μP_i
- Graphplan can be viewed as a heuristic search planner that first computes the distance estimates in a forward propagation manner and then searches backward from the goal using a iterative-deepening strategy augmented with a learning mechanisms (nogoods hashtable)

Heuristics for Plan-Space Planning

Reminder: PSP Procedure

Algorithm (PSP(π))

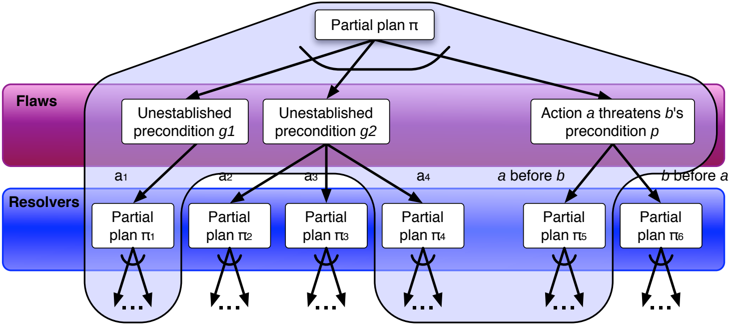
flaws $\leftarrow$ OpenGoals(π) $\cup$
Threat(π)
if flaws $= \emptyset$ **then return** π
select any flaw sigma $\in$ flaws
resolvers $\leftarrow$ Resolve(σ,π)
if resolvers $= \emptyset$ **then return** Failure
nondeterministically choose a resolver $\rho \in$ resolvers
$\pi' \leftarrow$ Refine(ρ, π)
return PSP(π')



Heuristics for Plan-Space Planning

Reminder: Plan-Space

- Plan space can be viewed as AND/OR tree
- The flaw correspond to the AND branches
  - each flaw must be resolved in order to find a solution plan
- The resolver correspond to the OR branches
  - only one resolver is needed in order to a solution plan

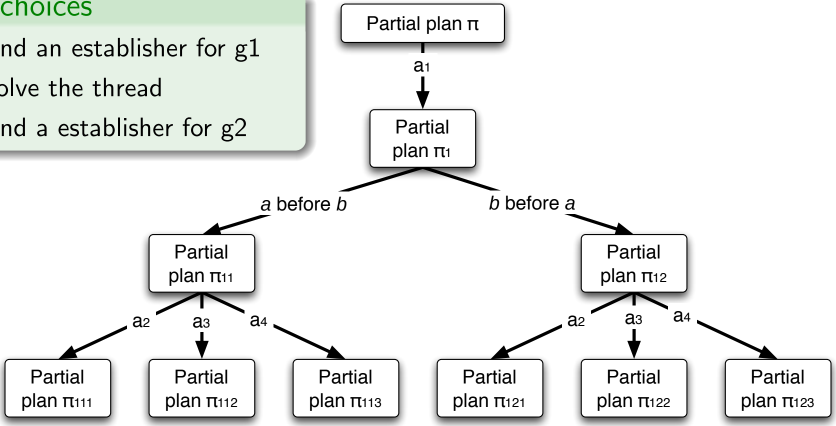


Flaw-Selection Heuristics

Serialization tree example (1/3)

PSP choices

1. find an establisher for g1
2. solve the thread
3. find a establisher for g2

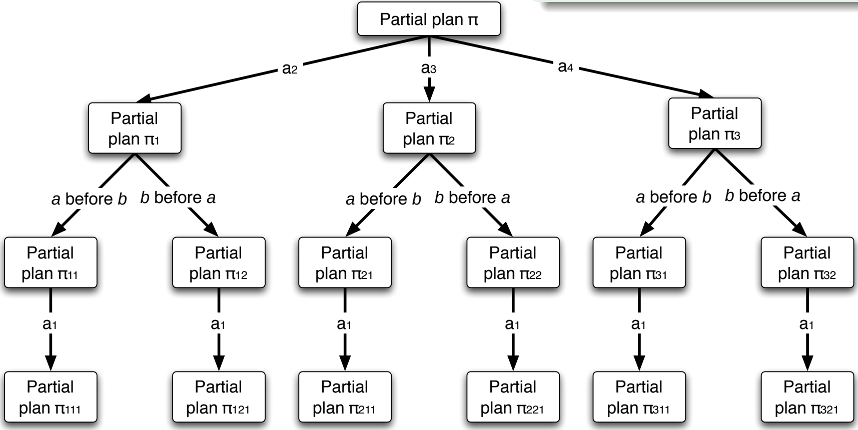


Flaw-Selection Heuristics

Serialization tree example (2/3)

PSP choices

1. find a establisher for g2
2. solve the thread
3. find an establisher for g1



Flaw-Selection Heuristics

Serialization tree example (3/3)

- All serialization trees lead to exactly the same set of solutions
- All serialization trees do not contain the same number of nodes
- The speed of PSP varies significantly depending on the number of node explore. Thus PSP speeds depends on the order in which its selects flaws to resolve

Question

How to choose the flaw to resolve to reduce the number of nodes to explored ?

Flaw-Selection Heuristics

The FAF-Heuristic

Idea

The *fewest alternatives first* (FAF) is to choose the flaw having the smallest branching factor as early as possible in oder to limit the cost of eventual backtracks.

- The FAF-heuristic is easy to compute $\Theta(n)$ where n is the number of flaws in a partial plan
- The FAF-heuristic works relatively well compared with other flaw selection heuristics

Flaw-Selection Heuristics

Other Flaw-Selection Heuristics

- *Zero-commitment:* chooses flaw that has not already been choosen in order to cut as soon as possible unachievable branches (low overhead)
- *Least-commitment:* always selects a open goal which generates the fewest refined plans (higth overhead)
- *Least-cost-flaw-repair :* same as "Least-commitment" applied to the threat too (higth overhead)
- *LIFO:* Last in last out choice of the flaw (low overhead)
- *ZLIFO:* Threat are selected depending "LIFO" strategy and open goal depending "Zero-commitment" (low overhead)

Resolver-Selection Heuristics

- The technics presented for state space planning cannot be applied
  - because they rely on relaxed distances between states, while states are not explicit in the plan space
- Hence, we have to come up with other means to rank the candidate nodes, *i.e.*, partial plan, at a search point

Resolver-Selection Heuristics

Simple Heuristics (1/2)

Idea

The choice of the resolver is based on an A* best-first search strategy with a heuristic

$$f(\pi) = g(\pi) + h(\pi)$$

where

- $g(\pi)$ the cost of the partial plan π and
- $h(\pi)$ estimate of the additionnal cost of the best complete solution that extends π

Resolver-Selection Heuristics

Simple Heuristics (2/2)

- To elaborate the simple heuristic we can used:
  1. the number of actions (S)
  2. the number of open goals (OC)
  3. the number of causal links (CL)
  4. the number of threats (UC)
- For instance UCPOP uses : S + OC + UC
- Experiments show that S + OC works relatively well compared with other heuristic combinaisons

Note

Due to causal links addition refinement mechanism, $f(\pi)$ is not admissible

Regression AND/OR Graph heuristic

For each $OC(\pi)$, the heuristic compute an AND/OR graph along regression steps defined by γ^{-1} down to some fixed level k. Let $\eta_k(OC(\pi))$ be the weighted sum of:

1. the number of actions in this graph that are not in π and
2. the number of subgoals remaining in its leaves that are not in the initial state s_0

Note

- η_k incurs a significant overhead

Resolver-Selection Heuristics

Heuristic based on planning graph

Planning Graph Heuristic

Instead of computing for each $OC(\pi)$ a regression AND/OR graph, this heuristic builds a planning graph once for the planning domain and uses it as follow in order to estimate $\eta_k(OC(\pi))$:

$$\eta_k(OC(\pi)) = \left\{ \begin{array}{ll} 0 & \text{if } OC(\pi) \subseteq s_0 \\ \infty & \text{if } \forall a \in A, a \text{ is not revelant for } OC(\pi) \\ \multicolumn{2}{l}{\max_p \{\delta_\pi(a) + \eta(\gamma^{-1}, a)) \mid p \in OC(\pi) \cap \text{effects}^+(a)} \\ & \text{and } a \text{ is relevant for } OC(\pi)\} \text{ otherwise} \end{array} \right\}$$

with $\delta_\pi(a) = 0$ when a is in π and $\delta_\pi(a) = 1$ otherwise

Exercice

Exercice 1

How many serialization trees are there for the AND/OR tree in slide 1 ?

Further readings

X. Nguyen, S. Kambhampati, and R. Nigenda.
Planning graph as the basis for deriving heuristics for plan synthesis by state space and csp search.
Artificial Intelligence, 135(1-2):73 124, 2002.

A. Gerevini and L. Schubert.
Accelerating partial-order planners: Some techniques for effective search control and pruning.
Journal of Artificial Intelligence Research, 5(1):95-137, 1996.

B. Bonet and H. Geffner.
Planning as heuristic search: New results.
In Proceedings of European Conference on Artificial Intelligence, pages 360 372, 1999.

Part VI

Hierarchical Task Network Planning

Outline of Part VI

Introduction

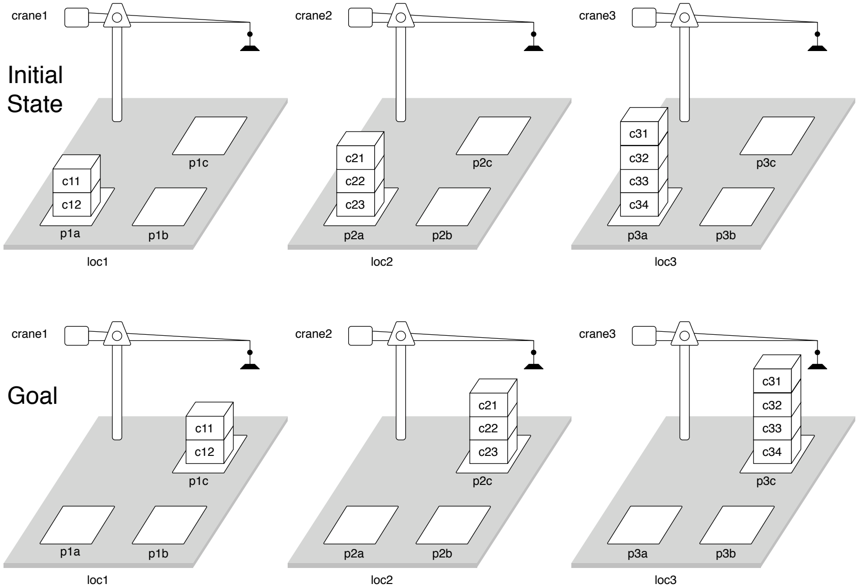
- *Hierarchical Task Network* (HTN) planning is like classical planning:
  - each state of the world is represented by a set of atoms
  - each action corresponds to a deterministic state transition
- In HTN planner, the objective is not to achieve a set of goals but instead to perform some set of tasks
- The imput to the HTN planning system includes
  1. a set of (operators) (similar to classical planning)
  2. a set of methods each of which is a presciption for how to decompose some task into some set of subtasks (smaller tasks)
- HTN planning has been more widely used for practical applications because HTN methods provide a convenient way to write problem-solving "recipes" that correspond to human expertise.

HTN Principle

HTN Principle

HTN planning proceeds by decomposing nonprimitive tasks recursively into smaller and smaller subtasks, until primitive tasks are reached that can be performed directly using the planning operators.

HTN Example (1/2)



HTN Example (2/2)

Example (Take and put method)

take-and-put(*c*,*k*,*l1*,*l2*,*p1*,*p2*,*x1*,*x2*)

precond: top(*p1*,*l1*), on(*c*,*x1*) *;; true if p1 is not empty*
attached(*p1*,*l1*), belong(*k*,*l1*) *;; bind l1 and k*
attached(*p2*,*l2*), top(*x2*,*p2*) *;; bind l2 and x2*

subtasks: ⟨take(*k*,*l1*,*c*,*x1*,*p1*), put(*k*,*l2*,*c*,*x2*,*p2*)⟩

To accomplish the task of moving the topmost container of a pile *p1* to another pile *p2*, we can use :

1. the DWR domain's take operator to remove the container from *p1* and
2. the put operator to put it on the top.

STN Planning

- STN (Simple Task Network) is a simplified version of HTN
- In STN, terms, literals, operators, actions and plans definitions are the same as in classical planning
- However, STN language includes:
  1. tasks
  2. methods
  3. task networks

Tasks and Methods

Tasks Definition

Definition (Task)

A *task* is an expression of the form

$$t(r_1, \ldots, r_k)$$

such

- t is a task symbol, i.e., an operator symbol (primitive task) or a method symbol (nonprimitive task)
- $r_1, \ldots, r_k$ are terms

Notes

1. A task is *ground* is all of the terms are ground; otherwise, it is *unground*
2. An action $a = (name(a), precond(a), effects(a))$ *accomplishes* a ground primitive task t in a state s if $name(a) = t$ and a is applicable to s.

Tasks and Methods

Task Networks Definition

Definition (Simple Task Network)

A *simple task network* is an acyclic digraph

$$w = (U, E)$$

in which

- U is the node set such that each node $u \in U$ contains a task t_u
- E is the edge set that defines a partial ordering of U, *e.g.*, $u \prec v$ iff there is a path from u to v

Notes

1. w is *ground* is all of the tasks $\{t_u \mid u \in U\}$ are ground; otherwise w is *unground*
2. w is *primitive* is all of the tasks $\{t_u \mid u \in U\}$ are primitive; otherwise w is *nonprimitive*

Tasks and Methods

Task Networks Example

Example (Task Network)

In the DWR domain, let three tasks:

- $t_1 =$ take(cran2,loc1,c1,c2,p1) a primitive task
- $t_2 =$ put(cran2,loc2,c3,c4,p2) a primitive task
- $t_3 =$ move-stack(p1,q) a non primitive task

and two task networks such $\forall i, u_i = ti$:

- $w_1 = (\{u1, u2, u3\}, \{(u1, u2), (u2, u3)\})$
- $w_2 = (\{u1, u2\}, \{(u1, u2)\})$

Since w_2 is totally ordered, we would usually write $w_2 = \langle t_1, t_2 \rangle$
Since w_2 is ground and primitive, it corresponds to the plan
⟨take(cran2,loc1,c1,c2,p1), put(cran2,loc2,c3,c4,p2)⟩

Tasks and Methods

STN Method Definition

Definition

An *STN method* is a 4-tuple

$$m = (name(m), task(m), precond(m), network(m))$$

in which

- $name(m)$, the name of the method, *i.e.*, , a expression if the form $m(x_1, \ldots, x_2)$ where n is an unique method symbol and $x_1, \ldots, x_2$ are all of the variables symbols that occurs anywhere in m
- $task(m)$ is a non primitive task
- $precond(m)$ is a set of literals call method's preconditions
- $network(m)$ is a task network whose tasks are called the *subtasks* of m

Tasks and Methods

STN Method Example

Example (DWR methods)

recursive-move(p,q,c,x)

task: move-stack(p,q)
precond: top(c,p), on(c,x) *;; true if p is not empty*
subtasks: ⟨move-topmost-container(p,q), move-stack(p,q)⟩
;; the second subtask recursively moves the rest of the stack

do-nothing(p,q)

task: move-stack(p,q)
precond: top(pallet,p), on(c,x) *;; true if p is empty*
subtasks: ⟨⟩ *;; no substasks because we are done*

move-each-twice()

task: move-all-stacks()
precond: *;; no preconditions*
network: u_1 = move-stack(p1a,p1b), u_2 = move-stack(p1b,p1c),
u_3 = move-stack(p2a,p2b), u_4 = move-stack(p2b,p2c),
u_5 = move-stack(p3a,p3b), u_6 = move-stack(p3b,p3c),
$\{(u_1, u_2), (u_3, u_4), (u_5, u_6)\}$

Tasks and Methods

Applicale and Relevant Method

Definition (Applicable Method)

A method instance m is *applicable* in a state s if $\text{precond}^{+}(m) \subseteq s$ and $\text{precond}^{-}(m) \cap s = \emptyset$.

Definition (Revelant Method)

Let t be a task and m a method instance, if there is a substitution σ such that $\sigma(t) = task(m)$, then m is revelant for t, and the *decomposition* of t by m under σ is $\delta(t, m, \sigma) = \text{network}(m)$. If m is totally ordered, we may write $\delta(t, m, \sigma) = \text{subtasks}(m)$.

Note

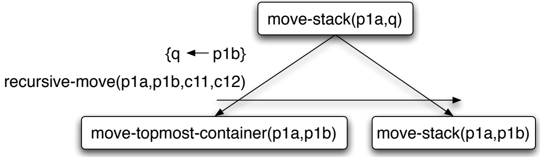
For planning, we will interested in finding method instances that are both applicable in the current state and relevant for some task we are trying to accomplish.

Example (Applicable and Revelant Method)

Let t be the nonprimitive task move-stack(p1a,q), s the state of the world show slide 6, and m be the method instance recursive-move(p1a,p1b,c11,c12). m is applicable to s, revelant for t under substitution $\sigma = \{\text{q} \leftarrow \text{p1b}\}$, and decomposes t into:

$$\delta(t, m, \sigma) = \langle \text{move-topmost-container(p1a,p1b),move-stack(p1a,p1b)} \rangle$$

- Graphical representation of the method decomposition:



Definition (STN Planning Domain)

An *STN planning domain* is a pair

$$\mathcal{D} = (O, M)$$

where

- O is a set of operators
- M is a set of methods.

$\mathcal{D}$ is a *total-order planning domain* if every $m \in M$ is totally ordered.

Problems and Solutions

STN Planning Problem Definition

Definition (STN Planning Problem)

An *STN planning problem* is a 4-tuple

$$\mathcal{P} = (s_0, w, O, M)$$

where

- s_0 is the initial state
- w is a task network called the *initial task network*
- $\mathcal{D} = (O, M)$ is a STN planning domain

$\mathcal{P}$ is a *total-order planning problem* if w and D are totally ordered.

Problems and Solutions

Solution Plan

Definition (Solution Plan)

Let $\mathcal{P} = (s_0, w, O, M)$ be a planning problem. Here are the cases in which a plan $\pi = \langle a_1, \ldots, a_n \rangle$ is *solution* for $\mathcal{P}$:

- **Case 1:** w is empty. Then π is a solution for $\mathcal{P}$ is π is empty, *i.e.*, $\pi = \langle\rangle$.
- **Case 2:** There is a primitive task node $u \in w$ that has no predessors in w. Then π is a solution for $\mathcal{P}$ is a_1 is applicable to t_u in s_0 and the plan $\pi = \langle a_2, \ldots, a_n \rangle$ is a solution of the planning problem:

$$\mathcal{P}' = (\gamma(s_0, a_1), w - \{u\}, O, M)$$

- **Case 3:** There is a nonprimitive task node $u \in w$ that has no predessor in w. Suppose there is an instance m of some method in M such that m is revelant for t_u and applicable in s_0. Then π is a solution for $\mathcal{P}$ is there is a task network $w' \in \delta(w, u, m, \sigma)$ such that π is a solution for (s_0, w', O, M).

Problems and Solutions

Solution Plan Example (1/2)

Example (DWR Solution Plan)

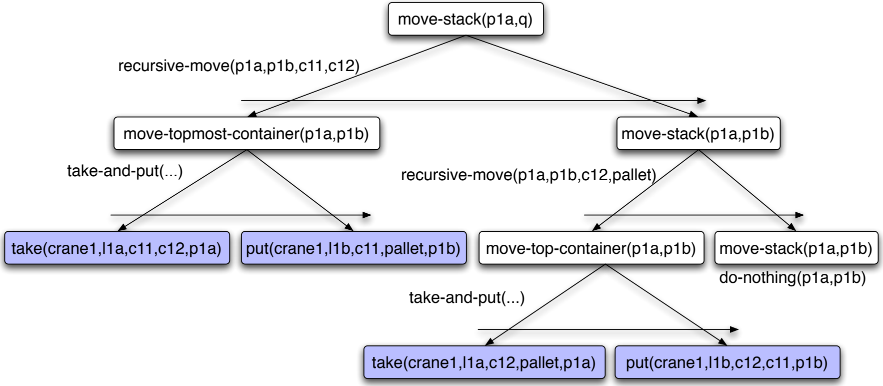
Let $\mathcal{P} = (s_0, w, O, M)$, where s_0 is the state shown slide 6, $w = \langle$move-stack(p1a,p1b)$\rangle$, O is the usual set of operators, and M is the set of methods given slide 5. Then there is only one solution for $\mathcal{P}$:

$$\begin{aligned}\pi \quad = \quad & \langle \text{take(crane1,l1a,c11,c12,p1a)}, \\ & \text{put(crane1,l1b,c11,pallet,p1b)}, \\ & \text{take(crane1,l1a,c12,pallet,p11)}, \\ & \text{put(crane1,l1b,c12,c11,p1b)} \rangle\end{aligned}$$

Problems and Solutions

Solution Plan Example (2/2)

- Example of tree decomposition for the solution plan π:



Total-Order STN Planning

Total-order Forward Decomposition

Algorithm ($\text{TFD}(s, \langle t_1, \ldots, t_2 \rangle, O, M)$)

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

Total-Order STN Planning

TFD Comparaison

1. Like Forward-search, TFD considers only actions whose preconditions are satisfied in the current state. Moreover, like Backward-search, it considers only operators that revelant for the task to achieve

   $\Rightarrow$ greatly increase the efficiency of the search

2. Like Forward-search, TFD generates actions in the same order in which they will be executed

   $\Rightarrow$ it knows the current state of the world

Partial-Order STN Planning

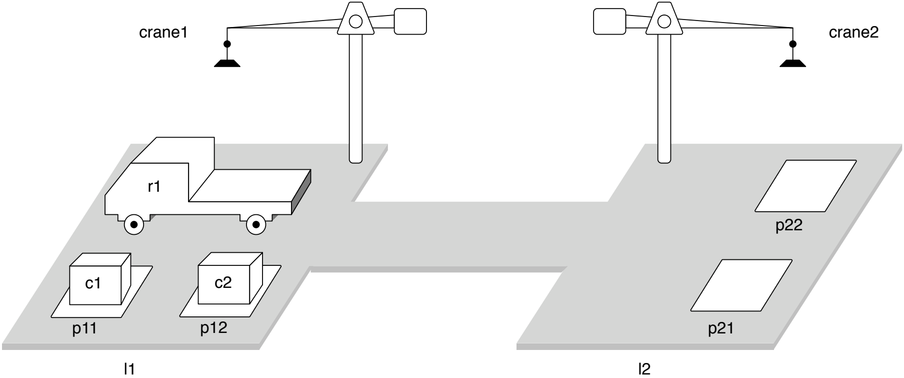
Why partial-order planning is interested to be considered ?

$\Rightarrow$ because not all planning domains can be rewritten into total-order planning

Partial-Order STN Planning

Example (1/5)

- Consider the following initial state for the DWR domain:



Partial-Order STN Planning

Example (2/5)

Example (DWR methods to move two containers at once)

transfer2(*c1,c2,l1,l2,r*) *;; method to transfert c1 and c2*

task: transfer-two-containers(*c1,c2,l1,l2,r*)
precond: *;; no preconditions*
subtasks: ⟨transfer-one-container(*c1,l1,l2,r*), transfer-one-container(*c2,l1,l2,r*)⟩

transfer1(*c,l1,l2,r*) *;; method to transfert c*

task: transfer-one-container(*c,l1,l2,r*)
precond: *;; no preconditions*
network: $u_1 =$ setup(*c,r*), $u_2 =$ move-robot(*l1,l2*), $u_3 =$ finish(*c,r*), $\{(u_1, u_2), (u_2, u_3)\}$

move1(*r,l1,l2*) *;; method to move r if r is not at l2*

task: move-robot(*l1,l2*)
precond: at(*r,l1*)
subtasks: ⟨move(*r,l1,l2*)⟩

Partial-Order STN Planning

Example (3/5)

Example (DWR methods to move two containers at once)

move0(r,$l1$,$l2$) *;; method to move r if r is already at l2*

  task: move-robot($l1$,$l2$)
  precond: at(r,$l2$)
  subtasks: $\langle\rangle$ *;; no subtasks*

do-setup(c,d,k,l,p,r) *method to prepare for moving a container*

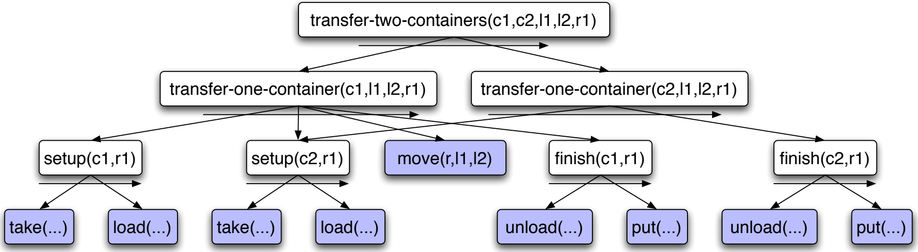
  task: setup(c,r)
  precond: on(c,d), in(c,p), belong(k,l), attached(p,l), at(r,l)
  network: u_1 = take(k,l,c,r), u_2 = put(k,l,c,d,p), $\{(u_1, u_2)\}$

unload-robot(c,d,k,l,p,r) *;; method to finish after moving a container*

  task: finish(c,r)
  precond: attached(p,l), loaded(r,c), top(d,p), belong(k,l), at(r,l)
  network: u_1 = unload(k,l,c,r), u_2 = put(k,l,c,d,p), $\{(u_1, u_2)\}$

Partial-Order STN Planning

Example (4/5)

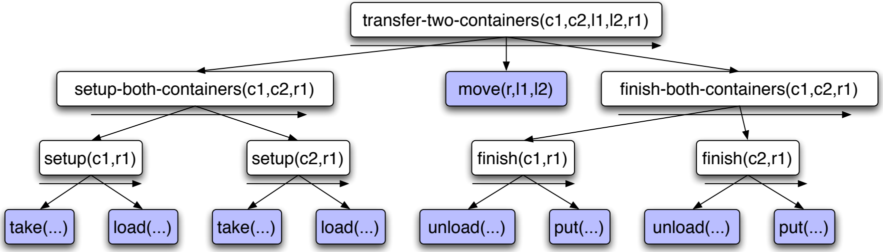


Interleaved Decomposition Tree

- The subtasks of the root are unordered, and their subtasks are interleaved
- Decomposition tree like this cannot occur in total-order STN planning domain

Partial-Order STN Planning

Example (5/5)



Noninterleaved Decomposition Tree

- To obtain a totally ordered tree, the best is to write method that generate a noninterleaved decomposition tree

Partial-Order STN Planning

Partial-order Forward Decomposition

Algorithm (PFD(s, w, O, M))

if $w = \emptyset$ **then return** *an empty plan* $\pi = \langle\rangle$
nondeterministically choose any $u \in w$ *that as no predessors in* w
else if t_1 *is primitive task* **then**
    active $\leftarrow \{(a, \sigma) \mid$ *a is a ground instance of an operator in* O, σ *is a substitution such that a is revelant for* $\sigma(t_1)$, *and a is applicable to s* $\}$
    if active $= \emptyset$ **then return** Failure
    nondeterministically choose any $(a, \sigma) \in$ active
    $\pi \leftarrow \texttt{PFD}(\gamma(s, a), \sigma(w - \{u\}), O, M)$
    if $\pi =$ Failure **then return** Failure **else return** $a \cdot \pi$
else if t_1 *is nonprimitive* **then**
    active $\leftarrow \{(m, \sigma) \mid$ *m is a ground instance of a method in* M, σ *is a substitution such that m is revelant for* $\sigma(t_1)$, *and m is applicable to s* $\}$
    if active $= \emptyset$ **then return** Failure
    nondeterministically choose any $(m, \sigma) \in$ active
    nondeterministically choose any task network $w' \in \delta(w, u, m, \sigma)$
    return $\texttt{PFD}(s, w', O, M)$

HTN Planning

- In STN planning, two kinds of constraints are associated with a method:
  1. preconditions
  2. ordering constraints
- Ordering constraints are explicitely represented in the task network but not preconditions
- HTN planning is a generalization of SNT planning that give the planning procedure more freedom about how to construct the task network

HTN Planning

Task Network Definition

Definition

A *task network* is a pair

$$w = (U, C)$$

where

- U is a set of task nodes and
- C is a set of constraints.

HTN Planning

Task Network Constraints

- HTN Task Network can handle the following kinds of constraints:
  1. A precedence constraint is an expression of the form $u \prec v$, where u and v are task node. Its meaning is identical to the edge (u, v) in STN planning.
  2. A before-constraint is a generalization of the notion of a precondition in STN planning. It is a constraint of the form before(U', l), where $U' \subseteq U$ is a set of task nodes and l is a literal.

Example

For instance, consider the task u is a task node for which $t_u =$ move(r2,l2,l3). Then the constraints before$(\{u\}$, at(r2,l2)) says that r2 must be at l2 just before we move it from l2 to l3.

  3. An after-constraint has the form after(U', l). It is like a before-constraint except that it says that l must be true in the state that occurs just after last (U', π)
  4. A between-constraint has the form between(U', U'', l). It says that literal l must be true in the state just after last (U', π), the state just before first (U'', π) and all of the states in between

Definition (HTN Method)

An *HTN method* is a 4-tuple

$$m = (name(m), task(m), subtasks(m), constr(m))$$

in which the elements are described as follows:

- $name(m)$, the name of the method, *i.e.,* , a expression if the form $m(x_1, \ldots, x_2)$ where n is an unique method symbol and $x_1, \ldots, x_2$ are all of the variables symbols that occurs anywhere in m
- $task(m)$ is a non primitive task
- $(subtasks(m), constr(m))$ is a task network

HTN Methods

Dynamic of HTN Method

Suppose that $w = (U, C)$ is a task network, $u \in U$ is a task node, t_u is it task, m is an instance of a method in M, and task$(m) = t_u$. Then *m decomposes u* into subtasks(m'), producing the task network:

$$\delta(w, u, m) = ((U - \{u\}) \cup subtasks(m'), C' \cup constr(m'))$$

where C' is the following modified version of C:

- For every precedence constraint that constains u, replace it with precedence constraints containing the node of $subtasks(m')$

Example

If $subtasks(m') = \{u_1, u_2\}$, then we would replace $u \prec v$ with $u_1 \prec v$ and $u_2 \prec v$

- For every before, after, between constraints in which there is a set of task nodes U' that contains u, replace U' with $(U' - \{u\}) \cup subtasks(m')$

Example

If $subtasks(m') = \{u_1, u_2\}$, then we would replace before$(\{u, v\}, l)$ with before$(\{u_1, u_2, v\}, l)$

HTN Methods

HTN Method Example (1/2)

Example (DWR HTN Methods of example slide 6)

transfer2(*c1*,*c2*,*l1*,*l2*,*r*) *;; method to move c1 and c2 from pile p1 to pile p2*

task: transfer-two-containers(*c1*,*c2*,*l1*,*l2*,*r*)
substasks: u_1 = transfer-one-container(*c1*,*l1*,*l2*,*r*), u_2 = transfer-one-container(*c2*,*l1*,*l2*,*r*)
constr: $u_1 \prec u_2$

transfer1(*c*,*l1*,*l2*,*r*) *;; method to transfert c*

task: transfer-one-container(*c*,*l1*,*l2*,*r*)
subtasks: u_1 = setup(*c*,*r*), u_2 = move-robot(*l1*,*l2*), u_3 = finish(*c*,*r*)
constr: $u_1 \prec u_2$ $u_2 \prec u_3$

move1(*r*,*l1*,*l2*) *;; method to move r if r is not at l2*

task: move-robot(*l1*,*l2*)
subtasks: move(*r*,*l1*,*l2*)
constr: before($\{u_1\}$, at(*r*,*l1*))

HTN Methods

HTN Method Example (2/2)

Example (DWR HTN Methods of example slide 6)

move0(r,$l1$,$l2$) *;; method to move r if r is already at l2*

- **task:** move-robot($l1$,$l2$)
- **subtasks:** *;; no subtasks*
- **constr:** before($\{u_0\}$, at(r,$l2$))

do-setup(c,d,k,l,p,r) *method to prepare for moving a container*

- **task:** setup(c,r)
- **subtasks:** $u_1 =$ take(k,l,c,r), $u_2 =$ put(k,l,c,d,p)
- **network:** $u_1 \prec u_2$, before($\{u_1\}$, on(c,d)), before($\{u_1\}$, attached(p,l)), before($\{u_1\}$, in(c,p)), before($\{u_1\}$, belong(k,l)), before($\{u_1\}$, at(r,l))

unload-robot(c,d,k,l,p,r) *;; method to finish after moving a container*

- **task:** finish(c,r)
- **subtasks:** $u_1 =$ unload(k,l,c,r), $u_2 =$ put(k,l,c,d,p)
- **network:** $u_1 \prec u_2$, before($\{u_1\}$, attached(p,l)), before($\{u_1\}$, loaded(r,c)), before($\{u_1\}$, top(d,p)), before($\{u_1\}$, belong(k,l)), before($\{u_1\}$, at(r,l))

HTN Planning Domain and Problem Definition

Definition (HTN Planning Domain)

An *HTN planning domain* is a pair $\mathcal{D} = (O, M)$ where

- O is a set of operators
- M is a set of methods.

Definition (HTN Planning Problem)

An *HTN planning problem* is a 4-tuple $\mathcal{P} = (s_0, w, O, M)$ where

- s_0 is the initial state
- w is a task network called the *initial task network*
- $\mathcal{D} = (O, M)$ is a STN planning domain

HTN Solution Plan Definition (1/2)

Definition (HTN Solution Plan)

- **Case 1:** If $w = (U, C)$ is primitive, then a plan $\pi = \langle a_1, \dots, a_k \rangle$ is a *solution* for $\mathcal{P}$ if there is a ground instance (U', C') of (U, C) and a total ordering $\langle u_1, \dots, u_k \rangle$ of the node U' such that all the following condition hold:
  1. The action in π are the ones named by the node $u_1, \dots, u_k$, *i.e.*, $\text{name}(a_i) = t_{u_i}$ for $i = 1, \dots k$
  2. The plan π is executable from s_0
  3. The total ordering $\langle u_1, \dots, u_k \rangle$ satisfies the precedence constraints in C', *i.e.*, C' contains no constraint $u_i \prec u_j$ such that $j \leq i$
  4. For every constraints $\text{before}(U', l)$ in C', l holds in the state s_{i-1} that immediately precedes action a_i, where a_i is the action named by the first node of U'.
  5. For every constraints $\text{after}(U', l)$ in C', l holds in the state s_j produced by the action a_j, where a_j is the action named by the last node of U'.
  6. For every constraints $\text{between}(U', U'', l)$ in C', l holds in every state that comes between a_i and a_j, where a_i is the action named by the last node of U' and a_j the action named by the first node of U''.

HTN Solution Plan Definition (2/2)

Definition (HTN Solution Plan)

- **Case 2:** If $w = (U, C)$ is nonprimitive, ($i.e.$, al least one task in U is nonprimitive), then a plan π is a *solution* for $\mathcal{P}$ if there is a sequence of task decompositions that can be applied to w to produce primitive task network w' such taht π is a solution for w'. In this case, the decomposition tree for π is the tree structure corresponding to these task decompositions.

HTN Planning Procedure

Algorithm (Abstract-HTN(s, U, C, O, M))

if (U, C) *can be shown to have no solution* **then return** Failure
else if U *is primitive* **then**
  if (U, C) *has no solution* **then return** Failure
  else return *nondeterministically a plan* π *from any such solution*
else
  choose a nonprimitive task node $u \in U$
  active $\leftarrow \{m \in M \mid task(m)$ *is unifiable with* $t_u\}$
  if active $\neq \emptyset$ **then** *nondeterministically choose any* $m \in$ active
  $\sigma \leftarrow$ *an mgu for* m *and* t_u *that renames all variables of* m
  $(U', C') \leftarrow \delta(\sigma(U, C), \sigma(u), \sigma(m))$
  return `Abstract-HTN`(s, U', C', O, M)
end

HTN versus Classical Planning

- STN planning and thus HTN planning can be used to encode undecidable problem, but not classical planning
- However STN and HTN language can produce undesirable effects

Example (Recursive method calls)

method1()

task: task1()
precond: *;; no preconditions*
subtasks: op1(), task1(), op2()

op1()

precond: *;; no preconditions*
effects: *;; no effects*

method2()

task: task1()
precond: *;; no preconditions*
subtasks: *;; no subtasks*

op2()

precond: *;; no preconditions*
effects: *;; no effects*

The solutions to this problem are as follows:

$$\pi_0 = \langle \rangle$$

$$\pi_1 = \langle \text{op1}(), \text{op2}() \rangle$$

$$\pi_2 = \langle \text{op1}(), \text{op1}(), \text{op2}(), \text{op2}() \rangle$$

Complexity of plan existance for HTN planning

Restrictions on nonprimitive tasks	Must the HTNs be totally ordered ?	Are variables allowed?	
		No	Yes
None	No Yes	Undecidable[a] In EXPTIME PSPACE-hard	Undecidable[a,b] in DEXPTIME[d] EXPSPACE-hard
"Regularity" (≤ 1 nonprimitive task, which must follow all primitive tasks)	Does not matter	PSPACE-complete	EXPSPACE-complete[c]
No nonprimitive tasks	No Yes	NP-complete Polynomial time	NP-complete NP-complete

[a] Decidable if we impose acyclic restrictions

[b] Undecidable even when the planning domain is fixed in advance

[c] In PSPACE when the planning domain is fixed in advance, and PSPACE-complete for some fixed planning domains

[d] DEXPTIME means double-exponential time

HTN Plannin Extensions

- The main extensions of HTN planning are:
  1. *Function Symbols.* If we allow the planning language to contain function symbols, then aguments of an atom, or task are no longer restricted to being constant symbol of variable symbols.
  2. *Axioms.* To incorporate axiomatic inference, we will need to used theorem prover as a subroutine of the planning procedure.
  3. *Attached Procedures.* We can modify the precondition evaluation algorithm to recognize that certain terms or predicate symbols are to be evaluated by using attached procedure rather that by using the normal theorem prover.
  4. *Time.* It is possible to generalize PFD and Abstract-HTN to certain kinds of temporal planning, *e.g.*, to deal with action that have time durations and may overlap with each other.

Exercices

Exercice 1

Write totally ordered methods to generate the noninterleaved decomposition tree similar to the one shown slide 13.

Exercice 2

Suppose we write a deterministic implementation of TFD that does a depth-first search of its decomposition tree. Is this implementation complete ? Why or why not ?

Exercice 3

In example slide 5, suppose we allow the initial state to contain an atom need-to-move(p,q) for each stack of the containers that needs to be moved from som pile p to some other q. Rewrite the methods and operators so that instead of being restricted to work on three stacks of containers, they will work correctly for an arbitrary number of stacks and containers.

Further readings

D. Nau and T. Au and O. Ilghami and U. Kuter and W. Murdock and D. Wu and Y. Yaman
Shop2: An HTN planning system
Journal of Artificial Intelligence Research 20(1):379-404, 2003

E. Sacerdoti.
The nonlinear nature of plans.
In Proceedings of the International Joint Conference on Artificial Intelligence, pages 206 214, 1975.