Hierarchical Task Networks

Planning to perform tasks rather than to achieve goals

Literature

HTN Planning

• HTN planning:
  • objective: perform a given set of tasks

• input includes:
  • set of operators
  • set of methods: recipes for decomposing a complex task into more primitive subtasks

• planning process:
  • decompose non-primitive tasks recursively until primitive tasks are reached

Overview

♫ Simple Task Networks
• HTN Planning
• Extensions
• State-Variable Representation
STN Planning

- STN: Simple Task Network
- what remains:
  - terms, literals, operators, actions, state transition function, plans
- what’s new:
  - tasks to be performed
  - methods describing ways in which tasks can be performed
  - organized collections of tasks called task networks

DWR Stack Moving Example

- task: move stack of containers from pallet p1 to pallet p3 in a way the preserves the order

- (informal) methods:
  - move via intermediate: move stack to intermediate pile (reversing order) and then to final destination (reversing order again)
  - move stack: repeatedly move the topmost container until the stack is empty
  - move topmost: take followed by put action
Tasks

- **task symbols**: $T_S = \{t_1, \ldots, t_n\}$
  - operator names $\subseteq T_S$: primitive tasks
  - non-primitive task symbols: $T_S - \text{operator names}$

- **task**: $t(r_1, \ldots, r_k)$
  - $t$: task symbol (primitive or non-primitive)
  - $r_1, \ldots, r_k$: terms, objects manipulated by the task
  - ground task: are ground

- **action $a$ accomplishes** ground primitive task $t_i(r_1, \ldots, r_k)$ in state $s$ iff
  - $\text{name}(a) = t_i$ and
  - $a$ is applicable in $s$

Simple Task Networks

- A simple task network $w$ is an acyclic directed graph $(U, E)$ in which
  - the node set $U = \{t_1, \ldots, t_n\}$ is a set of tasks and
  - the edges in $E$ define a partial ordering of the tasks in $U$.

- A task network $w$ is **ground/primitive** if all tasks $t_u \in U$ are ground/primitive, otherwise it is unground/non-primitive.
Totally Ordered STNs

- ordering: $t_u < t_v$ in $w = (U, E)$ iff there is a path from $t_u$ to $t_v$
- STN $w$ is totally ordered iff $E$ defines a total order on $U$
  - $w$ is a sequence of tasks: $\langle t_1, \ldots, t_n \rangle$
- Let $w = \langle t_1, \ldots, t_n \rangle$ be a totally ordered, ground, primitive STN. Then the plan $\pi(w)$ is defined as:
  - $\pi(w) = \langle a_1, \ldots, a_n \rangle$ where $a_i = t_i$, $1 \leq i \leq n$

STNs: DWR Example

- tasks:
  - $t_1 = \text{take}(\text{crane}, \text{loc}, c1, c2, p1)$: primitive, ground
  - $t_2 = \text{take}(\text{crane}, \text{loc}, c2, c3, p1)$: primitive, ground
  - $t_3 = \text{move-stack}(p1, q)$: non-primitive, unground
- task networks:
  - $w_1 = \{(t_1, t_2, t_3), (t_1, t_2), (t_1, t_3)\}$
    - partially ordered, non-primitive, unground
  - $w_2 = \{(t_1, t_2), (t_1, t_2)\}$
    - totally ordered: $w_2 = \langle t_1, t_2 \rangle$, ground, primitive
    - $\pi(w_2) = \langle \text{take}(\text{crane}, \text{loc}, c1, c2, p1), \text{take}(\text{crane}, \text{loc}, c2, c3, p1) \rangle$
STN Methods

- Let $M_S$ be a set of method symbols. An STN method is a 4-tuple $m=(\text{name}(m), \text{task}(m), \text{precond}(m), \text{network}(m))$ where:
  - $\text{name}(m)$:
    - the name of the method
    - syntactic expression of the form $n(x_1, ..., x_k)$
      - $n \in M_S$: unique method symbol
      - $x_1, ..., x_k$: all the variable symbols that occur in $m$;
  - $\text{task}(m)$: a non-primitive task;
  - $\text{precond}(m)$: set of literals called the method's preconditions;
  - $\text{network}(m)$: task network $(U,E)$ containing the set of subtasks $U$ of $m$.

STN Methods: DWR Example (1)

- move topmost: take followed by put action
- take-and-put($c,k,l,p_o,p_d,x_o,x_d$)
  - task: move-topmost($p_o,p_d$)
  - precond: top($c,p_o$), on($c,x_o$), attached($p_o,l$), belong($k,l$), attached($p_d,l$), top($x_d,p_d$)
  - subtasks: $\langle$take($k,l,c,x_o,p_o$), put($k,l,c,x_d,p_d$)$\rangle$
STN Methods: DWR Example (2)

- move stack: repeatedly move the topmost container until the stack is empty
- recursive-move($p_o, p_d, c, x_o$)
  - task: move-stack($p_o, p_d$)
  - precond: top($c, p_o$), on($c, x_o$)
  - subtasks: $\langle$move-topmost($p_o, p_d$), move-stack($p_o, p_d$)$\rangle$
- no-move($p_o, p_d$)
  - task: move-stack($p_o, p_d$)
  - precond: top(pallet, $p_o$)
  - subtasks: $\langle\rangle$

STN Methods: DWR Example (3)

- move via intermediate: move stack to intermediate pile (reversing order) and then to final destination (reversing order again)
- move-stack-twice($p_o, p_i, p_d$)
  - task: move-ordered-stack($p_o, p_d$)
  - precond: -
  - subtasks: $\langle$move-stack($p_o, p_i$), move-stack($p_i, p_d$)$\rangle$
Hierarchical Task Networks

**Applicability and Relevance**

- A method instance $m$ is **applicable** in a state $s$ if
  - $\text{precond}^+(m) \subseteq s$ and
  - $\text{precond}^-(m) \cap s = \emptyset$.
- A method instance $m$ is **relevant** for a task $t$ if
  - there is a substitution $\sigma$ such that $\sigma(t) = \text{task}(m)$.
- The **decomposition** of a task $t$ by a relevant method $m$ under $\sigma$ is
  - $\delta(t,m,\sigma) = \sigma(\text{network}(m))$ or
  - $\delta(t,m,\sigma) = \sigma(\langle\text{subtasks}(m)\rangle)$ if $m$ is totally ordered.

**Method Applicability and Relevance: DWR Example**

- task $t = \text{move-stack}(p1,q)$
- state $s$ (as shown)

- method instance $m_i = \text{recursive-move}(p1,p2,c1,c2)$
  - $m_i$ is applicable in $s$
  - $m_i$ is relevant for $t$ under $\sigma = \{q\leftarrow p2\}$
Method Decomposition: DWR

Example

- $\delta(t, m, \sigma) = \langle \text{move-topmost}(p1, p2), \text{move-stack}(p1, p2) \rangle$

Decomposition of Tasks in STNs

- Let
  - $w = (U, E)$ be a STN and
  - $t \in U$ be a task with no predecessors in $w$ and
  - $m$ a method that is relevant for $t$ under some substitution $\sigma$ with $\text{network}(m) = (U_m, E_m)$.
- The decomposition of $t$ in $w$ by $m$ under $\sigma$ is the STN $\delta(w, u, m, \sigma)$ where:
  - $t$ is replaced in $U$ by $\sigma(U_m)$ and
  - edges in $E$ involving $t$ are replaced by edges to appropriate nodes in $\sigma(U_m)$. 
**STN Planning Domains**

- An STN planning domain is a pair $\mathcal{D}=(O,M)$ where:
  - $O$ is a set of STRIPS planning operators and
  - $M$ is a set of STN methods.

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

**STN Planning Problems**

- An STN planning problem is a 4-tuple $\mathcal{P}=(s_i,w_i,O,M)$ where:
  - $s_i$ is the initial state (a set of ground atoms)
  - $w_i$ is a task network called the initial task network and
  - $\mathcal{D}=(O,M)$ is an STN planning domain.

- $\mathcal{P}$ is a total-order STN planning domain if $w_i$ and $\mathcal{D}$ are both totally ordered.
STN Solutions

A plan $\pi = \langle a_1, \ldots, a_n \rangle$ is a solution for an STN planning problem $\mathcal{P} = (s_i, w_i, O, M)$ if:

- $w_i$ is empty and $\pi$ is empty;
- or:
  - there is a primitive task $t \in w_i$ that has no predecessors in $w_i$ and
  - $a_1 = t$ is applicable in $s_i$, and
  - $\pi' = \langle a_2, \ldots, a_n \rangle$ is a solution for $\mathcal{P}' = (s_i, t, w_i - \{t\}, O, M)$
- or:
  - there is a non-primitive task $t \in w_i$ that has no predecessors in $w_i$ and
  - $m \in M$ is relevant for $t$, i.e. $\sigma(t) = \text{task}(m)$ and applicable in $s_i$ and
  - $\pi$ is a solution for $\mathcal{P}' = (s_i, \delta(w_i, t, m, \sigma), O, M)$.

Decomposition Tree: DWR Example
Ground-TFD: Pseudo Code

function Ground-TFD(s,〈t₁,…,tₖ〉,O,M)
    if k=0 return ()
    if t₁.isPrimitive() then
        actions = {(a,σ) | a=σ(t₁) and a applicable in s}
        if actions.isEmpty() then return failure
        (a,σ) = actions.chooseOne()
        plan ← Ground-TFD(γ(s,a),σ(〈t₂,…,tₖ〉),O,M)
        if plan = failure then return failure
        else return (a) • plan
    else
        methods = {(m,σ) | m is relevant for σ(t₁) and m is applicable in s}
        if methods.isEmpty() then return failure
        (m,σ) = methods.chooseOne()
        plan ← subtasks(m) • σ(〈t₂,…,tₖ〉)
        return Ground-TFD(s,plan,O,M)

TFD vs. Forward/Backward Search

- choosing actions:
  - TFD considers only applicable actions like forward search
  - TFD considers only relevant actions like backward search
- plan generation:
  - TFD generates actions execution order; current world state always known
- lifting:
  - Ground-TFD can be generalized to Lifted-TFD resulting in same advantages as lifted backward search
Ground-PFD: Pseudo Code

function Ground-PFD(s,w,O,M)
    if w.U={} return ∅
    task ← {t∈U | t has no predecessors in w.E}.chooseOne()
    if task.isPrimitive() then
        actions = {(a,σ) | a=σ(t) and a applicable in s}
        if actions.isEmpty() then return failure
        (a,σ) = actions.chooseOne()
        plan ← Ground-PFD(γ(s,a),σ(w-{task}),O,M)
        if plan = failure then return failure
        else return (a) ∙ plan
    else
        methods = {(m,σ) | m is relevant for σ(t) and m is applicable in s}
        if methods.isEmpty() then return failure
        (m,σ) = methods.chooseOne()
        return Ground-PFD(s, δ(w,task,m,σ),O,M)

Overview

- Simple Task Networks
- HTN Planning
- Extensions
- State-Variable Representation
Preconditions in STN Planning

- STN planning constraints:
  - ordering constraints: maintained in network
  - preconditions:
    - enforced by planning procedure
    - must know state to test for applicability
    - must perform forward search

- HTN Planning
  - additional bookkeeping maintains general constraints explicitly

First and Last Network Nodes

- Let
  - \( \pi = \langle a_1, \ldots, a_n \rangle \) be a solution for \( w \),
  - \( U' \subseteq U \) be a set of tasks in \( w \), and
  - \( A(U') \) the subset of actions in \( \pi \) such that each \( a_i \in A(U') \) is a descendant of some \( t \in U' \) in the decomposition tree.

- Then we define:
  - \( \text{first}(U', \pi) = \) the action \( a_i \in A(U') \) that occurs first in \( \pi \); and
  - \( \text{last}(U', \pi) = \) the action \( a_i \in A(U') \) that occurs last in \( \pi \).
Hierarchical Task Networks

- A (hierarchical) task network is a pair \( w = (U, C) \), where:
  - \( U \) is a set of tasks and
  - \( C \) is a set of constraints of the following types:
    - \( t_1 \prec t_2 \): precedence constraint between tasks satisfied if in every solution \( \pi \): \( \text{last}\{t_1, \pi\} < \text{first}\{t_2, \pi\} \);
    - before\((U', l)\): satisfied if in every solution \( \pi \): literal \( l \) holds in the state just before \( \text{first}(U', \pi) \);
    - after\((U', l)\): satisfied if in every solution \( \pi \): literal \( l \) holds in the state just after \( \text{last}(U', \pi) \);
    - between\((U', U'', l)\): satisfied if in every solution \( \pi \): literal \( l \) holds in every state after \( \text{last}(U', \pi) \) and before \( \text{first}(U'', \pi) \).

HTN Methods

- Let \( M_S \) be a set of method symbols. An HTN method is a 4-tuple \( m = (\text{name}(m), \text{task}(m), \text{subtasks}(m), \text{constr}(m)) \) where:
  - \( \text{name}(m) \):
    - the name of the method
    - syntactic expression of the form \( n(x_1, \ldots, x_k) \)
      - \( n \in M_S \): unique method symbol
      - \( x_1, \ldots, x_k \): all the variable symbols that occur in \( m \);
  - \( \text{task}(m) \): a non-primitive task;
  - \( (\text{subtasks}(m), \text{constr}(m)) \): a task network.
HTN Methods: DWR Example (1)

- move topmost: take followed by put action
- take-and-put($c,k,l,p_o,p_d,x_o,x_d$)
  - task: move-topmost($p_o,p_d$)
  - network:
    - subtasks: $t_1$=take($k,l,c,x_o,p_o$), $t_2$=put($k,l,c,x_o,p_d$)
    - constraints: \( t_1 \prec t_2, \text{before}(\{t_1\}, \text{top}(c,p_o)), \text{before}(\{t_1\}, \text{on}(c,x_o)), \text{before}(\{t_1\}, \text{attached}(p_o,l)), \text{before}(\{t_1\}, \text{belong}(k,l)), \text{before}(\{t_2\}, \text{attached}(p_o,l)), \text{before}(\{t_2\}, \text{top}(x_o,p_d)) \)}

HTN Methods: DWR Example (2)

- move stack: repeatedly move the topmost container until the stack is empty
- recursive-move($p_o,p_d,c,x_o$)
  - task: move-stack($p_o,p_d$)
  - network:
    - subtasks: $t_1$=move-topmost($p_o,p_d$), $t_2$=move-stack($p_o,p_d$)
    - constraints: \( t_1 \prec t_2, \text{before}(\{t_1\}, \text{top}(c,p_o)), \text{before}(\{t_1\}, \text{on}(c,x_o)) \)}
- move-one($p_o,p_d,c$)
  - task: move-stack($p_o,p_d$)
  - network:
    - subtasks: $t_1$=move-topmost($p_o,p_d$)
    - constraints: \( \text{before}(\{t_1\}, \text{top}(c,p_o)), \text{before}(\{t_1\}, \text{on}(c,pallet)) \)}
Hierarchical Task Networks

**HTN Decomposition**

- Let \( w=(U,C) \) be a task network, \( t\in\mathcal{U} \) a task, and \( m \) a method such that \( \sigma(\text{task}(m))=t \). Then the decomposition of \( t \) in \( w \) using \( m \) under \( \sigma \) is defined as:

\[
\delta(w,t,m,\sigma) = ((U-\{t\})\cup\sigma(\text{subtasks}(m)), C\cup\sigma(\text{constr}(m)))
\]

where \( C' \) is modified from \( C \) as follows:
- for every precedence constraint in \( C \) that contains \( t \), replace it with precedence constraints containing \( \sigma(\text{subtasks}(m)) \) instead of \( t \); and
- for every before-, after-, or between constraint over tasks \( U' \) containing \( t \), replace \( U' \) with \( (U'\{-t\})\cup\sigma(\text{subtasks}(m)) \).

**HTN Decomposition: Example**

- network: \( w = (\{t_1=\text{move-stack}(p1,q)\}, \{\}) \)

\[
\delta(w, t_1, \text{recursive-move}(p_o,p_d,c,x_o), \{p_o\leftarrow p1,p_d\leftarrow q\}) = w' = \\
\delta(w', t_2, \text{take-and-put}(c,k,l,p_o,p_d,x_o,x_d), \{p_o\leftarrow p1,p_d\leftarrow q\}) =
\]

- \( \delta(w', t_2) = \{(t_2=\text{move-topmost}(p1,q), t_3=\text{move-stack}(p1,q)), \\
\{t_2< t_3, \text{before}(\{t_2, \text{top}(c,p1)\}), \text{before}(\{t_3, \text{on}(c,x_o)\})\}) \\
\{t_4< t_5, \text{attached}(p1,l), \text{before}(\{t_4, \text{belong}(k,l)\}), \text{before}(\{t_5, \text{attached}(q,l)\}), \text{before}(\{t_3, \text{top}(x_o,q)\})\}) \\
\}

Hierarchical Task Networks
HTN Planning Domains and Problems

- An HTN planning domain is a pair $\mathcal{D} = (O, M)$ where:
  - $O$ is a set of STRIPS planning operators and
  - $M$ is a set of HTN methods.
- An HTN planning problem is a 4-tuple $\mathcal{P} = (s_i, w_i, O, M)$ where:
  - $s_i$ is the initial state (a set of ground atoms)
  - $w_i$ is a task network called the initial task network and
  - $\mathcal{D} = (O, M)$ is an HTN planning domain.

Solutions for Primitive HTNs

- Let $(U, C)$ be a primitive HTN. A plan $\pi = \langle a_1, \ldots, a_n \rangle$ is a solution for $\mathcal{P} = (s_i, (U, C), O, M)$ if there is a ground instance $(\sigma(U), \sigma(C))$ of $(U, C)$ and a total ordering $\langle t_1, \ldots, t_n \rangle$ of tasks in $\sigma(U)$ such that:
  - for $i = 1 \ldots n$: $\text{name}(a_i) = t_i$;
  - $\pi$ is executable in $s_i$, i.e. $\gamma(s_i, \pi)$ is defined;
  - the ordering of $\langle t_1, \ldots, t_n \rangle$ respects the ordering constraints in $\sigma(C);
  - for every constraint before $(U', l)$ in $\sigma(C)$ where $t_k = \text{first}(U', \pi)$: $l$ must hold in $\gamma(s_i, \langle a_1, \ldots, a_{k-1} \rangle)$;
  - for every constraint after $(U', l)$ in $\sigma(C)$ where $t_k = \text{last}(U', \pi)$: $l$ must hold in $\gamma(s_i, \langle a_1, \ldots, a_{k} \rangle)$;
  - for every constraint between $(U', U'', l)$ in $\sigma(C)$ where $t_k = \text{first}(U', \pi)$ and $t_m = \text{last}(U'', \pi)$: $l$ must hold in every state $\gamma(s_i, \langle a_1, \ldots, a_j \rangle)$, $j \in \{k \ldots m-1\}$. 
Solutions for Non-Primitive HTNs

- Let \( w = (U,C) \) be a non-primitive HTN. A plan \( \pi = \langle a_1, \ldots, a_n \rangle \) is a solution for \( P = (s_i, w, O, M) \) if there is a sequence of task decompositions that can be applied to \( w \) such that:
  - the result of the decompositions is a primitive HTN \( w' \); and
  - \( \pi \) is a solution for \( P' = (s_i, w', O, M) \).

Abstract-HTN: Pseudo Code

```
function Abstract-HTN(s, U, C, O, M)
    if (U,C).isInconsistent() then return failure
    if U.isPrimitive() then
        return extractSolution(s, U, C, O)
    else
        return decomposeTask(s, U, C, O, M)
```

**extractSolution: Pseudo Code**

```plaintext
function extractSolution(s, U, C, O)
  \langle t_1, \ldots, t_n \rangle \leftarrow U.\text{chooseSequence}(C)
  \langle a_1, \ldots, a_n \rangle \leftarrow \langle t_1, \ldots, t_n \rangle.\text{chooseGrounding}(s, C, O)
  \text{if } \langle a_1, \ldots, a_n \rangle.\text{satisfies}(C) \text{ then}
    \text{return } \langle a_1, \ldots, a_n \rangle
  \text{return failure}
```

**decomposeTask: Pseudo Code**

```plaintext
function decomposeTask(s, U, C, O, M)
  t \leftarrow U.\text{nonPrimitives}().\text{selectOne()}
  methods \leftarrow \{(m, \sigma) \mid m \in M \text{ and } \sigma(\text{task}(m)) = \sigma(t)\}
  \text{if } \text{methods}.\text{isEmpty()} \text{ then return failure}
  (m, \sigma) \leftarrow \text{methods}.\text{chooseOne()}
  (U', C') \leftarrow \delta((U, C), t, m, \sigma)
  (U', C') \leftarrow (U', C').\text{applyCritic()}
  \text{return Abstract-HTN}(s, U', C', O, M)
```
**HTN vs. STRIPS Planning**

- Since
  - HTN is a generalization of STN Planning, and
  - STN problems can encode undecidable problems, but
  - STRIPS cannot encode such problems:
- **STN/HTN formalism is more expressive**
- non-recursive STN can be translated into equivalent STRIPS problem
  - but exponentially larger in worst case
- “regular” STN is equivalent to STRIPS

**Overview**

- Simple Task Networks
- HTN Planning
  - Extensions
- State-Variable Representation
Functions in Terms

- allow function terms in world state and method constraints
- ground versions of all planning algorithms may fail
  - potentially infinite number of ground instances of a given term
- lifted algorithms can be applied with most general unifier
  - least commitment approach instantiates only as far as necessary
  - plan-existence may not be decidable

Axiomatic Inference

- use theorem prover to infer derived knowledge within world states
  - undecidability of first-order logic in general
- idea: use restricted (decidable) subset of first-order logic: Horn clauses
  - only positive preconditions can be derived
  - precondition $p$ is satisfied in state $s$ iff $p$ can be proved in $s$
**Attached Procedures**

- associate predicates with procedures
- modify planning algorithm
  - evaluate preconditions by
    - calling the procedure attached to the predicate symbol if there is such a procedure
    - test against world state (set-relation, theorem prover) otherwise
- soundness and completeness: depends on procedures

**High-Level Effects**

- allow user to declare effects for non-primitive methods
- aim:
  - establish preconditions
  - prune partial plans if high-level effects threaten preconditions
- increases efficiency
- problem: semantics
Other Extensions

- other constraints
  - time constraints
  - resource constraints
- extended goals
  - states to be avoided
  - required intermediate states
  - limited plan length
  - visit states multiple times

Overview

- Simple Task Networks
- HTN Planning
- Extensions
  - State-Variable Representation
State Variables

- some relations are functions
  - example: at(r1,loc1): relates robot r1 to location loc1 in some state
    - truth value changes from state to state
    - will only be true for exactly one location / in each state

- idea: represent such relations using state-variable functions mapping states into objects
  - example: functional representation:
    \[ \text{rloc}: \text{robots} \times S \rightarrow \text{locations} \]

States in the State-Variable Representation

- Let \( X \) be a set of state-variable functions. A \( k \)-ary state variable is an expression of the form \( x(v_1,\ldots,v_k) \) where:
  - \( x \in X \) is a state-variable function and
  - \( v_i \) is either an object constant or an object variable.

- A state-variable state description is a set of expressions of the form \( x_s = c \) where:
  - \( x_s \) is a ground state variable \( x(v_1,\ldots,v_k) \) and
  - \( c \) is an object constant.
DWR Example: State-Variable State Descriptions

- simplified: no cranes, no piles
- state-variable functions:
  - rloc: robots×S → locations
  - rolad: robots×S→containers ∪ {nil}
  - cpos: containers×S → locations ∪ robots
- sample state-variable state descriptions:
  - \{rloc(r1)=loc1, rload(r1)=nil, cpos(c1)=loc1, cpos(c2)=loc2, cpos(c3)=loc2\}
  - \{rloc(r1)=loc1, rload(r1)=c1, cpos(c1)=r1, cpos(c2)=loc2, cpos(c3)=loc2\}

Operators in the State-Variable Representation

- A state-variable planning operator is a triple \( (\text{name}(o), \text{precond}(o), \text{effects}(o)) \) where:
  - \( \text{name}(o) \) 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 object variables that appear in \( o \),
  - \( \text{precond}(o) \) are the unions of a state-variable state description and some rigid relations, and
  - \( \text{effects}(o) \) are sets of expressions of the form \( x_s←v_k+1 \) where:
    - \( x_s \) is a ground state variable \( x(v_1,\ldots,v_k) \) and
    - \( v_{k+1} \) is an object constant or an object variable.
DWR Example: State-Variable Operators

- **move(r,l,m)**
  - precond: \( \text{rloc}(r) = l \), adjacent\((l,m)\)
  - effects: \( \text{rloc}(r) \rightarrow m \)

- **load(r,c,l)**
  - precond: \( \text{rloc}(r) = l \), \( \text{cpos}(c) = l \), \( \text{rload}(r) = \text{nil} \)
  - effects: \( \text{cpos}(c) \leftarrow r \), \( \text{rload}(r) \leftarrow c \)

- **unload(r,c,l)**
  - precond: \( \text{rloc}(r) = l \)
  - effects: \( \text{rload}(r) \leftarrow \text{nil} \), \( \text{cpos}(c) \leftarrow l \)

Applicability and State Transitions

- Let \( a \) be an action and \( s \) a state. Then \( a \) is **applicable** in \( s \) iff:
  - all rigid relations mentioned in \( \text{precond}(a) \) hold, and
  - if \( x_s = c \in \text{precond}(a) \) then \( x_s = c \in s \).

- The state transition function \( \gamma \) for an action \( a \) in state \( s \) is defined as:
  \[
  \gamma(s,a) = \{ x_s = c \mid x \in X \}
  \]
  where:
  - \( x_s = c \in \text{effects}(a) \) or
  - \( x_s = c \in s \) otherwise.
State-Variable Planning Domains

- Let $X$ be a set of state-variable functions. A state-variable planning domain on $X$ is a restricted state-transition system $\Sigma=(S,A,\gamma)$ such that:
  - $S$ is a set of state-variable state descriptions,
  - $A$ is a set of ground instances of some state-variable planning operators $O$,
  - $\gamma:S \times A \rightarrow S$ where
    - $\gamma(s,a) = \{x_s=c \mid x \in X$ and $x,\leftarrow c \in \text{effects}(a)$ or $x_s=c \in s$ otherwise} if $a$ is applicable in $s$
    - $\gamma(s,a) = \text{undefined}$ otherwise,
  - $S$ is closed under $\gamma$

State-Variable Planning Problems

- A state-variable planning problem is a triple $P=(\Sigma,s_i,g)$ where:
  - $\Sigma=(S,A,\gamma)$ is a state-variable planning domain on some set of state-variable functions $X$
  - $s_i \in S$ is the initial state
  - $g$ is a set of expressions of the form $x_s=c$ describing the goal such that the set of goal states is: $S_g = \{s \in S \mid x_s=c \in s\}$
Relevance and Regression Sets

- Let $P=(\Sigma, s_i, g)$ be a state-variable planning problem. An action $a \in A$ is relevant for $g$ if
  - $g \cap \text{effects}(a) \neq \emptyset$ and
  - for every $x \leftarrow c \in g$, there is no $x \leftarrow d \in \text{effects}(a)$ such that $c \neq d$.
- The regression set of $g$ for a relevant action $a \in A$ is:
  - $\gamma^{-1}(g, a) = (g - \theta(a)) \cup \text{precond}(a)$ where
    - $\theta(a) = \{x \leftarrow c \mid x \leftarrow c \in \text{effects}(a)\}$
- definition for all regression sets $\Gamma^<(g)$ exactly as for propositional case

Statement of a State-Variable Planning Problem

- A statement of a state-variable planning problem is a triple $P=(O, s_i, g)$ where:
  - $O$ is a set of planning operators in an appropriate state-variable planning domain $\Sigma=(S,A,\gamma)$ on $X$
  - $s_i$ is the initial state in an appropriate state-variable planning problem $P=(\Sigma, s_i, g)$
  - $g$ is a goal in the same state-variable planning problem $P$
Translation: STRIPS to State-Variable Representation

- Let \( P=(O,s_i,g) \) be a statement of a classical planning problem. In the operators \( O \), in the initial state \( s_i \), and in the goal \( g \):
  - replace every positive literal \( p(t_1,\ldots,t_n) \) with a state-variable expression \( p(t_1,\ldots,t_n)=1 \) or \( p(t_1,\ldots,t_n)\leftarrow 1 \) in the operators’ effects, and
  - replace every negative literal \( \neg p(t_1,\ldots,t_n) \) with a state-variable expression \( p(t_1,\ldots,t_n)=0 \) or \( p(t_1,\ldots,t_n)\leftarrow 0 \) in the operators’ effects.

Translation: State-Variable to STRIPS Representation

- Let \( P=(O,s_i,g) \) be a statement of a state-variable planning problem. In the operators’ preconditions, in the initial state \( s_i \), and in the goal \( g \):
  - replace every state-variable expression \( p(t_1,\ldots,t_n)=v \) with an atom \( p(t_1,\ldots,t_n,v) \), and
  - in the operators’ effects:
    - replace every state-variable assignment \( p(t_1,\ldots,t_n)\leftarrow v \) with a pair of literals \( p(t_1,\ldots,t_n,v), \neg p(t_1,\ldots,t_n,w) \), and add \( p(t_1,\ldots,t_n,w) \) to the respective operators preconditions.
Overview

- Simple Task Networks
- HTN Planning
- Extensions
- State-Variable Representation