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

HTN Planning

• HTN planning:
  • world state represented by set of atoms and actions correspond to deterministic state transitions
  • objective: perform a given set of tasks
    • previously: achieve some goals
  • input includes:
    • set of operators
    • set of methods: recipes for decomposing a complex task into more primitive subtasks
      • methods: at a higher level of abstraction
      • primitive task: can be performed directly by an operator instance
  • planning process:
    • decompose non-primitive tasks recursively until primitive tasks are reached
  • HTN most widely used technique for real-world planning applications
    • methods are a natural way to encode recipes (which should lead to solution plans only; reduces search significantly)
    • methods reflect the way experts think about planning problems
Overview

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

Overview

- Simple Task Networks
  - now: representation and planning algorithms for STNs

- HTN Planning
- Extensions
- State-Variable Representation
STN Planning

• **STN: Simple Task Network**
  - STN: simplified version of the more general HTN case to be discussed later

• **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
  • preserve order: each container should be on same container it is on originally

• (informal) methods:
  • methods: possible subtasks and how they can be accomplished
  • 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

• action: no further decomposition required

• note: abstract concept: stack
Tasks

• **task symbols**: $T_S = \{t_1, \ldots, t_n\}$
  - used for giving unique names to tasks
  - operator names $\not\subset T_S$: primitive tasks
  - non-primitive task symbols: $T_S$ - operator names

• **task**: $t_i(r_1, \ldots, r_k)$
  - $t_i$: task symbol (primitive or non-primitive)
    - tasks: primitive iff task symbol is 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
  - name(a) = $t_i$ and
  - a is applicable in s

**Note**: unique operator names, hence primitive tasks can only be performed in one way – no search!
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.

**simple task network**: shortcut “task network”
Totally Ordered STNs

- ordering: $t_u \prec t_v$ 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: $(t_1, \ldots, t_n)$
- 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:**
  - \( t_1 = \text{take(crane,loc,c1,c2,p1)} \): primitive, ground
    - crane “crane” at location “loc” takes container “c1” of container “c2” in pile “p1”
  - \( t_2 = \text{take(crane,loc,c2,c3,p1)} \): primitive, ground
  - \( t_3 = \text{move-stack(p1,q)} \): non-primitive, unground
    - move the stack of containers on pallet “p2” to pallet “q” (variable)

- **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
    - \( \pi(w_2) = \langle \text{take(crane,loc,c1,c2,p1)}, \text{take(crane,loc,c2,c3,p1)} \rangle \)
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:

- **name($m$):**
  - the name of the method
  - symbolic 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 $m$;
- **task($m$):** a non-primitive task;
- **precond($m$):** set of literals called the method’s preconditions;
- **network($m$):** task network $(U,E)$ containing the set of subtasks $U$ of $m$.

**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:
  - **method symbols:** disjoint from other types of symbols
  - **STN method:** also just called method
  - **name($m$):**
    - **the name of the method**
    - **unique name:** no two methods can have the same name; gives an easy way to unambiguously refer to a method instances
    - **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 $m$;
      - **no “local” variables in method definition (may be relaxed in other formalisms)**
  - **task($m$):** a non-primitive task;
    - **what task can be performed with this method**
    - **non-primitive:** contains subtasks
  - **precond($m$):** set of literals called the method’s preconditions;
    - **like operator preconditions:** what must be true in state $s$ for $m$ to be applicable
    - **no effects:** not needed if problem is to refine/perform a task as opposed to achieving some effect.
STN Methods: DWR Example (1)

• move topmost: take followed by put action
  • simplest method from previous example

• take-and-put\((c, k, l, p_o, p_d, x_o, x_d)\)
  • using crane \(k\) at location \(l\), take container \(c\) from object \(x_o\) (container or pallet) in pile \(p_o\) and put it onto object \(x_d\) in pile \(p_d\) (\(o\) for origin, \(d\) for destination)

• task: move-topmost\((p_o, p_d)\)
  • move topmost container from pile \(p_o\) to pile \(p_d\)

• precond:
  • \(\text{top}(c, p_o), \text{on}(c, x_o):\) pile must be empty with container \(c\) on top
  • \(\text{attached}(p_o, l), \text{belong}(k, l), \text{attached}(p_d, l):\) piles and crane must be at same location
  • \(\text{top}(x_d, p_d):\) destination object must be top of its pile

• subtasks: \(\langle \text{take}(k, l, c, x_o, p_o), \text{put}(\text{take}(k, l, c, x_d, p_d)) \rangle\)
  • simple macro operator combining two (primitive) operators (sequentially)
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**: (move-topmost($p_o,p_d$), move-stack($p_o,p_d$))

- **no-move**($p_o,p_d$)
  - **task**: move-stack($p_o,p_d$)
  - **precond**: top(pallet,$p_o$)
  - **subtasks**: ()

STN Methods: DWR Example (2)

- **move stack**: repeatedly move the topmost container until the stack is empty

- **recursive-move**($p_o,p_d,c,x_o$)
  - move container $c$ which must be on object $x_o$ in pile $p_o$ to the top of pile $p_d$
  - **task**: move-stack($p_o,p_d$)
    - move the remainder of the stack from $p_o$ to $p_d$: more abstract than method
  - **precond**: top($c,p_o$), on($c,x_o$)
    - $p_o$ must be empty; $c$ is the top container
    - method is not applicable to empty piles!
  - **subtasks**: (move-topmost($p_o,p_d$), move-stack($p_o,p_d$))
    - recursive decomposition: move top container and then recursive invocation of method through task

- **no-move**($p_o,p_d$)
  - performs the task by doing nothing
  - **task**: move-stack($p_o,p_d$)
    - as above
  - **precond**: top(pallet,$p_o$)
    - the pile must be empty (recursion ends here)
  - **subtasks**: ()
    - do nothing does nothing
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: \(\langle\text{move-stack}(p_o,p_i),\text{move-stack}(p_i,p_d)\rangle\)

- move the stack of containers in pile $p_o$ first to intermediate pile $p_i$ then to $p_d$, thus preserving the order

- task: move-ordered-stack($p_o$,$p_d$)
  - move the stack from $p_o$ to $p_d$ in an order-preserving way

- precond: -
  - none; should mention that piles must be at same location and different

- subtasks: \(\langle\text{move-stack}(p_o,p_i),\text{move-stack}(p_i,p_d)\rangle\)
  - the two stack moves
Applicability and Relevance

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

\[ \delta(t, m_i, \sigma) = \langle \text{move-topmost}(p1, p2), \text{move-stack}(p1, p2) \rangle \]

[figure]

- graphical representation (called a decomposition tree):
  - view as AND/OR-graph: AND link – both subtasks need to be performed to perform super-task
  - link is labelled with substitution and method instance used
  - arrow under label indicates order in which subtasks need to be performed
  - often leave out substitution (derivable) and sometimes method parameters (to save space)
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 \( \sigma(U_m) \)
  - edges in \( E \) involving \( t \) are replaced by edges to appropriate nodes in \( \sigma(U_m) \).

Decomposition of Tasks in STNs

- idea: applying a method to a task in a network results in another network
- 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 \( \sigma(U_m) \)
  - edges in \( E \) involving \( t \) are replaced by edges to appropriate nodes in \( \sigma(U_m) \).

  - every node in \( \sigma(U_m) \) should come before nodes that came after \( t \) in \( E \)
  - \( \sigma(E_m) \) needs to be added to \( E \) to preserve internal method ordering
  - ordering constraints must ensure that \( \text{precond}(m) \) remains true even after subsequent decompositions
### 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 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 \( \mathcal{P} = (\mathcal{s}_i, \mathcal{w}_i, \mathcal{O}, \mathcal{M}) \) where:
  - \( \mathcal{s}_i \) is the initial state (a set of ground atoms)
  - \( \mathcal{w}_i \) is a task network called the initial task network and
  - \( \mathcal{D} = (\mathcal{O}, \mathcal{M}) \) is an STN planning domain.

- \( \mathcal{P} \) is a total-order STN planning domain if \( \mathcal{w}_i \) and \( \mathcal{D} \) are both totally ordered.

STN Planning Problems

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

- \( \mathcal{P} \) is a total-order STN planning domain if \( \mathcal{w}_i \) and \( \mathcal{D} \) are both totally ordered.
A plan \( \pi = \langle a_1, \ldots, a_n \rangle \) is a solution for an STN planning problem \( P = (s, w, O, M) \) 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 \( P' = (y(s_i,a_1), w_i \setminus \{t\}, O, M) \)
- 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 \( P' = (s, \delta(w_i,t,m,\sigma), O, M) \).

STN Solutions

- if \( \pi \) is a solution for \( P \), then we say that \( \pi \) accomplishes \( P \)
- intuition: there is a way to decompose \( w_i \) into \( \pi \) such that:
  - \( \pi \) is executable in \( s_i \) and
  - each decomposition is applicable in an appropriate state of the world
- \( w_i \) 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 \( P' = (y(s_i,a_1), w_i \setminus \{t\}, O, M) \)
- 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 \( P' = (s, \delta(w_i,t,m,\sigma), O, M) \).

- 2nd and 3rd case: recursive definition
- if \( w_i \) is not totally ordered more than one node may have no predecessors and both cases may apply
**Decomposition Tree: DWR Example**

- choose method: recursive-move(p1,p2,c1,c2) – binds variable `q`
- decompose into two sub-tasks
  - choose method for first subtask: take-and-put: c1 from c2 onto pallet
  - decompose into subtasks – primitive subtasks (grey) cannot be decomposed/correspond to actions
- choose method for second sub-task: recursive-move (recursive part)
- decompose (recursive)
  - choose method and decompose (into primitive tasks): take-and-put: c2 from c3 onto c1
  - choose method and decompose (recursive)
  - choose method and decompose: take-and-put: c3 from pallet onto c2
  - choose method (no-move) and decompose (empty plan)

- note:
  - (grey) leaf nodes of decomposition tree (primitive tasks) are actions of solution plan
  - (blue) inner nodes represent non-primitive task; decomposition results in sub-tree rooted at task according to decomposition function `\( \delta \)`
  - no search required in this example
Ground-TFD: Pseudo Code

• TFD = Total-order Forward Decomposition; direct implementation of definition of STN solution

• 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

**TFD vs. Forward/Backward Search**

**choosing actions:**

- TFD considers only applicable actions like forward search
- TFD considers only relevant actions like backward search
- TFD combines advantages of both search directions – better efficiency

**plan generation:**

- TFD generates actions execution order; current world state always known
  - e.g. good for domain-specific heuristics

**lifting:**

- Ground-TFD can be generalized to Lifted-TFD resulting in same advantages as lifted backward search
- avoids generating unnecessarily many actions (smaller branching factor)
- works for initial task list that is not ground
Ground-PFD: Pseudo Code

• PFD = Partial-order Forward Decomposition; direct implementation of definition of STN solution

• function Ground-PFD(s, w, O, M)
  • if w. U={} return Ø
  • task ← {t∈U | t has no predecessors in w,E}.chooseOne()
  • if task.isPrimitive() then
    • actions = {(a,σ) | a=σ(t_i) 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_i) 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

Overview

- **Simple Task Networks**
  - just done: representation and planning algorithms for STNs
- **HTN Planning**
  - now: generalizing the formalism and algorithm
- **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:
  - \( \text{first}(U', \pi) = \text{the action } a_i \in A(U') \text{ that occurs first in } \pi \); and
  - \( \text{last}(U', \pi) = \text{the action } a_i \in A(U') \text{ that occurs last in } \pi \).

First and Last Network Nodes

• for defining the constraints in an HTN network

• Let
  - \( \pi = \langle a_1, \ldots, a_n \rangle \) be a solution for \( w \),
    - HTN solution will be defined later
  - \( U' \subseteq U \) 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:
  - \( \text{first}(U', \pi) = \text{the action } a_i \in A(U') \text{ that occurs first in } \pi \); and
  - \( \text{last}(U', \pi) = \text{the action } a_i \in A(U') \text{ that occurs last in } \pi \).

• network is partially ordered; solution is totally ordered
  - for a given set of subtasks, one action decomposing \( U' \) must occur first/last in the solution plan
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 \( \pi \): \( \text{last}(\{t_1, \pi\}) < \text{first}(\{t_2, \pi\}) \);
    • \( \text{before}(U', I) \): satisfied if in every solution \( \pi \): literal \( I \) holds in the state just before \( \text{first}(U', \pi) \);
    • \( \text{after}(U', I) \): satisfied if in every solution \( \pi \): literal \( I \) holds in the state just after \( \text{last}(U', \pi) \);
    • \( \text{between}(U', U'', I) \): satisfied if in every solution \( \pi \): literal \( I \) holds in every state after \( \text{last}(U', \pi) \) and before \( \text{first}(U'', \pi) \).
HTN Methods

- extension of the definition of an STN method

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=\text{take}(k,l,c,x_o,p_o), t_2=\text{put}(k,l,c,x_d,p_d)\}
    - constraints: \{t_1 < 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_d,l)),
      \text{before}\{t_2\}, \text{top}(x_d,p_d))\}
  
  • note: before-constraints refer to both tasks; more precise than STN representation of preconditions
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: \{ $t_1$<$t_2$, before({$t_1$}, top($c,p_o$)), before({$t_1$}, 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: \{ before({$t_1$}, top($c,p_o$)), before({$t_1$}, on($c,pallet$)) \}

- move-stack-twice($p_o,p_i,p_d$) trivial; not shown again
HTN Decomposition

Let \( w=(U,C) \) be a task network, \( t \in U \) 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:

\[
\delta(w,t,m,\sigma) = ((U\setminus\{t\})\cup\sigma(\text{subtasks}(m)), C'\cup\sigma(\text{constr}(m)))
\]

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 \( (U'\setminus\{t\})\cup\sigma(\text{subtasks}(m)) \).

new, additional constraints may introduce threats that need to be resolved

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 \( (U'\setminus\{t\})\cup\sigma(\text{subtasks}(m)) \).

example: let subtasks(\( m \))=\{\( t_1, t_2 \)\} and \( t<t'<C \)

- then replace \( t<t' \) with \( t_1<t' \) and \( t_2<t' \)
  - cannot introduce inconsistencies (circles) since subtasks are new nodes

example (other constraints): let subtasks(\( m \))=\{\( t_1, t_2 \)\} and before(\{\( t,t' \},l \})\in C

- then replace before(\{\( t,t' \},l \}) with before(\{\( t_1,t_2,t' \},l \})
  - cannot introduce inconsistencies either
HTN Decomposition: Example

- network: \( w = \{ \{ t_1 = \text{move-stack}(p1,q) \} \} \)
  - initial, single task with no constraints
- \( \delta(w, t_1, \text{recursive-move}(p_o,p_d,c,x_o), \{ p_o \leftarrow p1, p_d \leftarrow q \}) = w' = \)
  - \( \{ t_2 = \text{move-topmost}(p1,q), t_3 = \text{move-stack}(p1,q) \} \),
  - 2 instantiated subtasks from method
  - \{ \( t_2 < t_3 \), before(\{ \( t_2 \) \}, top(c,p1)), before(\{ \( t_2 \) \}, on(c,x_o)) \} 
  - instantiated constraints from method
- \( \delta(w', t_2, \text{take-and-put}(c,k,l,p_o,p_d,x_o,x_d), \{ p_o \leftarrow p1, p_d \leftarrow q \}) = \)
  - \( \{ t_3 = \text{move-stack}(p1,q), t_4 = \text{take}(k,l,c,x_o,p1), t_5 = \text{put}(k,l,c,x_d,q) \} \),
  - \( t_3 \): from input network \( w' \); \( t_4 \) and \( t_5 \) from method
  - \{ \( t_4 < t_3 \), \( t_5 < t_3 \),
  - ordering did involve \( t_2 \) – replace with two constraints for new subtasks \( t_4 \) and \( t_5 \)
  - before(\{ \( t_4, t_5 \) \}, top(c,p1)), before(\{ \( t_4, t_5 \) \}, on(c,x_o)) \} \cup
  - replaced \{ \( t_2 \) \} with \{ \( t_4, t_5 \) \}
  - \{ \( t_4 < t_5 \), before(\{ \( t_4 \) \}, top(c,p1)), before(\{ \( t_4 \) \}, on(c,x_o)),
  - before(\{ \( t_4 \) \}, attached(p1,l)), before(\{ \( t_4 \) \}, belong(k,l)),
  - before(\{ \( t_5 \) \}, attached(q,l)), before(\{ \( t_5 \) \}, top(x_d,q)) \})
  - instantiated constraints from new method
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 \(\mathcal{P} = (s_p, (U, C), O, M)\) 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 \(i = 1, \ldots, n\): name\((a_i)\) = \(t_i\);
- \(\pi\) is executable in \(s_p\), i.e. \(\gamma(s_p, \pi)\) 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 \(\gamma(s_p, \langle a_1, \ldots, a_{k-1} \rangle)\);
- for every constraint after \((U', l)\) in \(\sigma(C)\) where \(t_k = \text{last}(U', \pi)\): \(l\) must hold in \(\gamma(s_p, \langle a_1, \ldots, a_k \rangle)\);
- 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 \(\gamma(s_p, \langle a_1, \ldots, a_j \rangle), j \in \{k \ldots m-1\}\).
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 \( \mathcal{P} = (s, w, O, M) \) 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 \( \mathcal{P}' = (s, w', O, M) \).
Abstract-HTN: Pseudo Code

• general schema for a function that implements HTN planning
• function Abstract-HTN(s,U,C,O,M)
• if (U,C).isInconsistent() then return failure
  • e.g. test for inconsistency of C, or apply other, domain-specific tests
• if U.isPrimitive() then
  • no further decompositions of tasks possible
• return extractSolution(s,U,C,O)
  • compute a total-order, grounded plan; may fail
• else
  • network still contains decomposable tasks
• return decomposeTask(s,U,C,O,M)
  • will recursively call Abstract-HTN function
extractSolution: Pseudo Code

**function** extractSolution(s, U, C, O)

\[ \langle t_1, \ldots, t_n \rangle \leftarrow U.\text{chooseSequence}(C) \]

\[ \langle a_1, \ldots, a_n \rangle \leftarrow \langle t_1, \ldots, t_n \rangle.\text{chooseGrounding}(s, C, O) \]

if \( \langle a_1, \ldots, a_n \rangle.\text{satisfies}(C) \) then

\[ \text{return} \ \langle a_1, \ldots, a_n \rangle \]

return failure

**extractSolution: Pseudo Code**

- **function extractSolution(s, U, C, O)**

- \[ \langle t_1, \ldots, t_n \rangle \leftarrow U.\text{chooseSequence}(C) \]
  
  - non-deterministically choose a serialization of the tasks in \( U \) that respects the ordering constraints in \( C \)

- \[ \langle a_1, \ldots, a_n \rangle \leftarrow \langle t_1, \ldots, t_n \rangle.\text{chooseGrounding}(s, C, O) \]
  
  - non-deterministically choose a grounding of the variables in \( t_1, \ldots, t_n \)
    
    - use \( s \) and \( C \) to ensure constraints hold, and \( O \) for type information if present

- if \( \langle a_1, \ldots, a_n \rangle.\text{satisfies}(C) \) then
  
  - this test can be performed during the grounding

- return \( \langle a_1, \ldots, a_n \rangle \)
  
  - plan is a solution, return it

- return failure
decomposeTask: Pseudo Code

- function decomposeTask(s,U,C,O,M)
  - t ← U.nonPrimitives().selectOne()
    - deterministically select a non-primitive task-node from the network
      - no backtracking required, all tasks must be decomposed eventually; selection important for efficiency
  - methods ← {(m,σ) | m∈M and σ(task(m))= σ(t)}
    - substitution should be mgu for least commitment planner (generates smaller search space)
  - if methods.isEmpty() then return failure
  - (m,σ) ← methods.chooseOne()
    - non-deterministically choose a method that can be applied to decompose the task
  - (U',C') ← δ((U,C),t,m,σ)
    - compute the decomposition
  - (U',C') ← (U',C').applyCritic()
    - optional; may make arbitrary modifications, e.g. application-specific computations
      - soundness and completeness depends on this function
  - return Abstract-HTN(s,U',C',O,M)
HTN vs. STRIPS Planning

• Since
  • HTN is 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

• non-recursive
  • at most one non-primitive subtask per method
  • non-primitive sub-task must be last in sequence
Overview

Simple Task Networks

HTN Planning
  • just done: generalizing the formalism and algorithm

Extensions
  • now: approaches to extending the formalism and algorithm

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 \( s \)

- semantics of negative preconditions: closed world assumption?
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

- applications:
  • perform numeric computations
  • query external data sources
- soundness and completeness: depends on procedures
- attached procedures to function symbols: critics
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

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
  - can be defined in different ways
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
  - just done: approaches to extending the formalism and algorithm
- State-Variable Representation
  - now: different style of representation (used in O-Plan/I-Plan)
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 l in each state

• idea: represent such relations using state-variable functions mapping states into objects
  • example: functional representation:
    rloc:robots×S→locations

• STRIPS state containing at(r1,loc1) and at(r1,loc2) usually inconsistent

• idea: represent such relations using **state-variable functions** mapping states into objects
  • advantage: reduces possibilities for inconsistent states, smaller state space

• example: **functional representation:**
  rloc:robots×S→locations
  • in general: maps objects and state into object
  • rloc is state-variable symbol that denotes state-variable function
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.

---

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.

- As for ground atoms in STRIPS states, state is implicit
- State description will usually give all values of ground state variables
- Values of state variables are not independent
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 (name(o), precond(o), effects(o)) where:
  - 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 \( o \),
  - precond(o) are the unions of a state-variable state description and some rigid relations, and
  - effects(o) are sets of expressions of the form \( x_s \leftarrow v_{k+1} \) 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.

Operators in the State-Variable Representation

- A state-variable planning operator is a triple (name(o), precond(o), effects(o)) where:
  - 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 \( o \),
    - looks like name of a STRIPS planning operator
  - precond(o) are the unions of a state-variable state description and some rigid relations, and
    - set of state variable equals value expressions and some rigid relations (as in STRIPS operators)
    - values of state variables refer to state before the operator is applied
  - effects(o) are sets of expressions of the form \( x_s \leftarrow v_{k+1} \) 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.
    - similar to state but assignment operator instead of equals sign
    - updates in effects refer to state after operator is applied
  - as for STRIPS operators, actions are ground instances of operators
DWR Example: Operators

- simplified domain: no piles, no cranes – only three operators:
  - **move**($r,l,m$)
    - move robot $r$ from location $l$ to adjacent location $m$
    - **precond**: rloc($r$)=$l$, adjacent($l,m$)
    - **effects**: rloc($r$)$\leftarrow m$
  - **load**($r,c,l$)
    - robot $r$ loads container $c$ at location $l$
    - **precond**: rloc($r$)=$l$, cpos($c$)=$l$, rload($r$)=nil
    - **effects**: cpos($c$)$\leftarrow r$, rload($r$)$\leftarrow c$
  - **unload**($r,c,l$)
    - robot $r$ unloads container $c$ at location $l$
    - **precond**: rloc($r$)=$l$, rload($r$)=$c$
    - **effects**: rload($r$)$\leftarrow nil$, 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:

- \( x_s \leftarrow c \in \text{effects}(a) \) or
- \( x_s=c \in s \) otherwise.

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
  - as in STRIPS representation
  - if \( x_s=c \in \text{precond}(a) \) then \( x_s=c \in s \).
    - if values of state variables in preconditions agree with same values in state

- 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:
  - \( x_s \leftarrow c \in \text{effects}(a) \) or
  - \( x_s=c \in s \) otherwise.
    - update the values of state variables in the effects
    - keep other values from previous state
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
    - $\gamma(s,a) = \{x_s=c | x\in X \text{ and } x_s\leftarrow c \in \text{effects}(a) \text{ or } x_s=c \in s \text{ otherwise} \}$ 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 $\mathcal{P}=(\Sigma, s_i, g)$ 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: $S_g = \{s \in S \mid x_s = c \in s\}$

- a goal is a specification of the values of some ground state variables
- goals are like preconditions without rigid relations

- definitions for plan, reachable states, and solutions as for propositional case
Relevance and Regression Sets

Let \( P=(\Sigma, s, g) \) be a state-variable planning problem. An action \( a \in A \) is relevant for \( g \) if

1. \( g \cap \text{effects}(a) \neq \emptyset \) and
2. for every \( x_s=c \in g \), there is no \( x_s\leftarrow d \in \text{effects}(a) \) 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
- \( \theta(a) = \{ x_s=c \mid x_s\leftarrow c \in \text{effects}(a) \} \)
- definition for all regression sets \( \Gamma<(g) \) exactly as for propositional case

Relevance and Regression Sets

Let \( P=(\Sigma, s, g) \) be a state-variable planning problem. An action \( a \in A \) is relevant for \( g \) if

1. \( g \cap \text{effects}(a) \neq \emptyset \) and
2. \( a \) has an effect that contributes to \( g \)
3. for every \( x_s=c \in g \), there is no \( x_s\leftarrow d \in \text{effects}(a) \) such that \( c\neq d \).

- effects of \( a \) do not change any of the state variables in \( g \)

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
- \( \theta(a) = \{ x_s=c \mid x_s\leftarrow c \in \text{effects}(a) \} \)

- necessary to change syntax: replace left arrow with equals sign
- otherwise definition is as before

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 \( \mathcal{P} = (\mathcal{O}, \mathcal{s}_i, \mathcal{g}) \) where:

- \( \mathcal{O} \) is a set of planning operators in an appropriate state-variable planning domain \( \Sigma = (\mathcal{S}, \mathcal{A}, \mathcal{g}) \) on \( \mathcal{X} \)
- \( \mathcal{s}_i \) is the initial state in an appropriate state-variable planning problem \( \mathcal{P} = (\Sigma, \mathcal{s}_i, \mathcal{g}) \)
- \( \mathcal{g} \) is a goal in the same state-variable planning problem \( \mathcal{P} \)
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.

result is a statement of a state-variable planning problem
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.

result is a statement of a STRIPS planning problem
Overview

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

- Simple Task Networks
  - HTN Planning
  - Extensions
  - State-Variable Representation
    - just done: different style of representation (used in O-Plan/I-Plan)