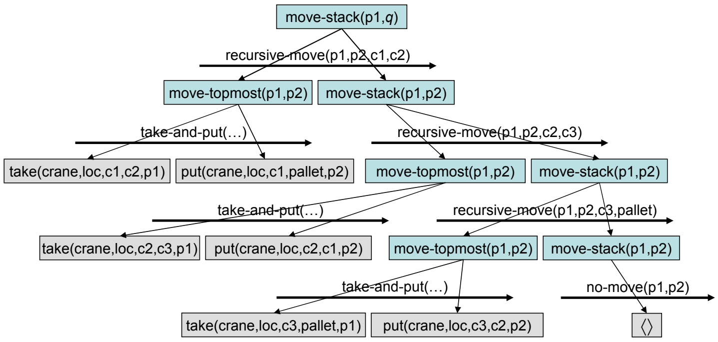
Artificial Intelligence Planning

Hierarchical Planning

Artificial Intelligence Planning

•Hierarchical Planning

Example: Decomposition Tree



Overview

- Tasks and Task Networks
- Methods (Refinements)
- Decomposition of Tasks
- Domains, Problems and Solutions
- Planning with Task Networks
- General HTN Planning

Overview

➢Tasks and Task Networks

- now: a different view of planning: “tasks to do” vs. “goals to achieve”

•Methods (Refinements)

•Decomposition of Tasks

•Domains, Problems and Solutions

•Planning with Task Networks

•General HTN Planning

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

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:**

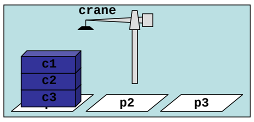
•**tasks to be performed**

•**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 that 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

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

- <u>task symbols</u>: $T_S = \{t_1,\ldots,t_n\}$
  - operator names $\subsetneq T_S$: primitive tasks
  - non-primitive task symbols: T_S - operator names
- <u>task</u>: $t_i(r_1,\ldots,r_k)$
  - t_i: task symbol (primitive or non-primitive)
  - $r_1,\ldots,r_k$: terms, objects manipulated by the task
  - ground task: are ground
- action $a=op(c_1,\ldots,c_k)$ <u>accomplishes</u> ground primitive task $t_i(r_1,\ldots,r_k)$ in state s iff
  - name(a) = t_i and $c_1 = r_1$ and … and $c_k = r_k$ and
  - a is applicable in s

Tasks

•**<u>task symbols</u>: $T_S = \{t_1,\ldots,t_n\}$**

•used for giving unique names to tasks

•**operator names $\subsetneq T_S$: primitive tasks**

•**non-primitive task symbols: T_S - operator names**

•**<u>task</u>: $t_i(r_1,\ldots,r_k)$**

•**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* <u>accomplishes</u> ground primitive task $t_i(r_1,\ldots,r_k)$ in state *s* iff**

•action *a* = (name(*a*), precond(*a*), effects(*a*))

•**name(a) = t_i and**

•***a* is applicable in *s***

•applicability: *s* satisfies precond(*a*)

•note: unique operator names, hence primitive tasks can only be performed in one way – no search!

Simple Task Networks

- A <u>simple task network</u> *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 <u>ground/primitive</u> if all tasks $t_u \in U$ are ground/primitive, otherwise it is unground/non-primitive.

Simple Task Networks

•A <u>simple task network</u> *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 <u>ground/primitive</u> 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: $\langle t_1,\ldots,t_n \rangle$
- 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 $a_i = t_i$; $1 \le i \le n$

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: $\langle t_1,\ldots,t_n \rangle$**

    •sequence is special case of acyclic directed graph

    •t_1: first task in U; t_2 :second task in U; …; t_n: last task in U

•**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 $a_i = t_i$; $1 \le i \le n$

STNs: DWR Example

- tasks:
  - t_1 = take(crane,loc,c1,c2,p1): primitive, ground
  - t_2 = take(crane,loc,c2,c3,p1): primitive, ground
  - t_3 = 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
    - $\pi(w_2)$ = ⟨take(crane,loc,c1,c2,p1),take(crane,loc,c2,c3,p1)⟩

STNs: DWR Example

•**tasks:**

•**t_1 = take(crane,loc,c1,c2,p1): primitive, ground**

•carne “crane” at location “loc” takes container “c1” of container “c2” in pile “p1”

•**t_2 = take(crane,loc,c2,c3,p1): primitive, ground**

•**t_3 = 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

•**$\pi(w_2)$ = ⟨take(crane,loc,c1,c2,p1),take(crane,loc,c2,c3,p1)⟩**

Overview

- Tasks and Task Networks
- Methods (Refinements)
- Decomposition of Tasks
- Domains, Problems and Solutions
- Planning with Task Networks
- General HTN Planning

Overview

•Tasks and Task Networks

- just done: a different view of planning: “tasks to do” vs. “goals to achieve”

➢Methods (Refinements)

- now: methods that describe how to break down tasks into simpler sub-tasks

•Decomposition of Tasks

•Domains, Problems and Solutions

•Planning with Task Networks

•General HTN Planning

STN Methods

- Let M_S be a set of method symbols. An <u>STN method</u> is a 4-tuple *m*=(name(*m*),task(*m*),precond(*m*),network(*m*)) where:
  - 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;
  - 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 <u>subtasks</u> *U* of *m*.

STN Methods

•Let M_S be a set of method symbols. An <u>STN method</u> is a 4-tuple *m*=(name(*m*),task(*m*),precond(*m*),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
- **•network(*m*): task network (*U*,*E*) containing the set of <u>subtasks</u> *U* of *m*.**
  - •describes one way of performing the task task(*m*); other methods may describe different way of performing same task: search!
    - •method is totally ordered iff network is totally ordered

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: ⟨take(k,l,c,x_o,p_o),put(k,l,c,x_d,p_d)⟩

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:

•**top(c,p_o), on(c,x_o)**: pile must be empty with container c on top

•**attached(p_o,l), belong(k,l), attached(p_d,l)**: piles and crane must be at same location

•**top(x_d,p_d)**: destination object must be top of its pile

•subtasks: ⟨take(k,l,c,x_o,p_o),put(take(k,l,c,x_d,p_d))⟩

•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 satck 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: ⟨move-stack(p_o,p_i),move-stack(p_i,p_d)⟩

STN Methods: DWR Example (3)

•move via intermediate: move stack to intermediate pallet (reversing order) and then to final destination (reversing order again)

•move-stack-twice(p_o,p_i,p_d)

•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: ⟨move-stack(p_o,p_i),move-stack(p_i,p_d)⟩

•the two stack moves

Applicability and Relevance

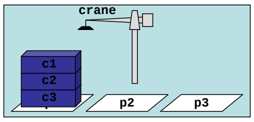
- A method instance m is applicable in a state s if
  - precond$^+(m) \subseteq s$ and
  - precond$^-(m) \cap s = \{\ \}$.
- A method instance m is relevant for a task t if
  - there is a substitution σ such that $\sigma(t)$ = task(m).
- The decomposition of a task t by a relevant method m under σ is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

Applicability and Relevance

- **A method instance m is applicable in a state s if**
  - **precond$^+(m) \subseteq s$ and**
  - **precond$^-(m) \cap s = \{\ \}$.**
- **A method instance m is relevant for a task t if**
  - **there is a substitution σ such that $\sigma(t)$ = task(m).**
- **The decomposition of a task t by a relevant method m under σ 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 = move-stack(p1,q)
- state s (as shown)



  - method instance m_i = recursive-move(p1,p2,c1,c2)
    - m_i is applicable in s
    - m_i is relevant for t under σ = {q←p2}

Method Applicability and Relevance: DWR Example

•task *t* = move-stack(p1,*q*)

•state *s* (as shown)

•method instance m_i = recursive-move(p1,p2,c1,c2)

•m_i is applicable in *s*

•m_i is relevant for *t* under σ = {*q*←p2}

Overview

- Tasks and Task Networks
- Methods (Refinements)
- Decomposition of Tasks
- Domains, Problems and Solutions
- Planning with Task Networks
- General HTN Planning

Overview

•Tasks and Task Networks

•Methods (Refinements)

- just done: methods that describe how to break down tasks into simpler sub-tasks

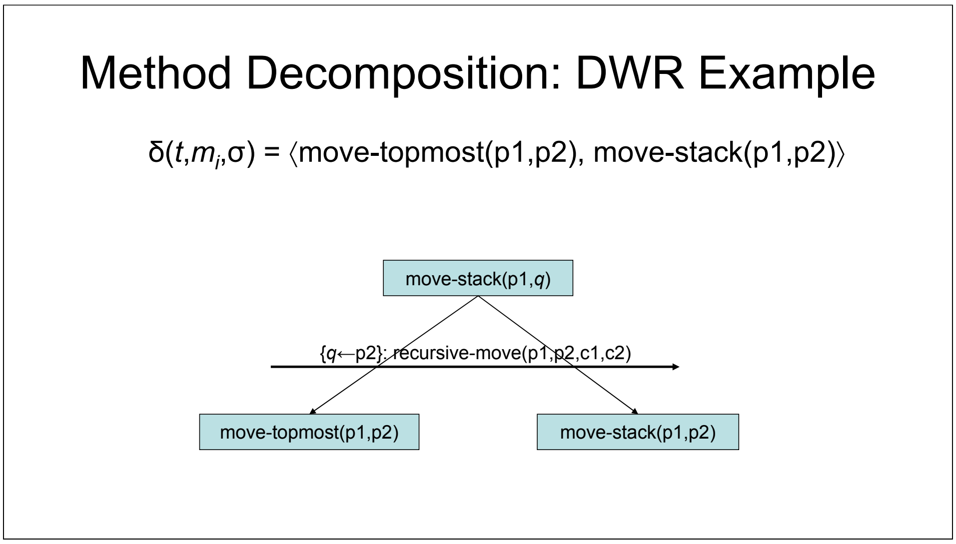
➢Decomposition of Tasks

- now: using methods to refine task networks (state-transitions)

•Domains, Problems and Solutions

•Planning with Task Networks

•General HTN Planning



Method Decomposition: DWR Example

•**$\delta(t,m_i,\sigma)$ = ⟨move-topmost(p1,p2), move-stack(p1,p2)⟩**

•**[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 σ with network(m) = (U_m,E_m).
- The <u>decomposition of *t* in *w* by *m* under σ</u> is the STN $\delta(w,t,m,\sigma)$ where:
  - t is replaced in U by $\sigma(U_m)$ and
  - edges in E involving t are replaced by edges to appropriate nodes in $\sigma(U_m)$.

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 σ with network(*m*) = (U_m,E_m).

•The <u>decomposition of *t* in *w* by *m* under σ</u> is the STN $\delta(w,u,m,\sigma)$ where:

•*t* is replaced in *U* by $\sigma(U_m)$ and

•replacement with copy (method maybe used more than once)

•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 precond(*m*) remains true even after subsequent decompositions

Overview

- Tasks and Task Networks
- Methods (Refinements)
- Decomposition of Tasks
- Domains, Problems and Solutions
- Planning with Task Networks
- General HTN Planning

Overview

•Tasks and Task Networks

•Methods (Refinements)

•Decomposition of Tasks

- just done: using methods to refine task networks (state-transitions)

➢Domains, Problems and Solutions

- now: defining the semantics of STN planning problems and solutions

•Planning with Task Networks

•General HTN Planning

STN Planning Domains

- An <u>STN planning domain</u> 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 <u>total-order STN planning domain</u> if every $m \in M$ is totally ordered.

STN Planning Domains

•An <u>STN planning domain</u> 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 <u>total-order STN planning domain</u> if every $m \in M$ is totally ordered.

STN Planning Problems

- An <u>STN planning problem</u> 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 <u>initial task network</u> and
  - $\mathcal{D}=(O,M)$ is an STN planning domain.
- $\mathcal{P}$ is a <u>total-order STN planning problem</u> if w_i and $\mathcal{D}$ are both totally ordered.

STN Planning Problems

•An <u>STN planning problem</u> 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 <u>initial task network</u> and

•$\mathcal{D}=(O,M)$ is an STN planning domain.

•$\mathcal{P}$ is a <u>total-order STN planning domain</u> 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 π 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 $\mathcal{P}'=(\gamma(s_i,a_1), w_i-\{t\}, O, M)$
  - 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. $\sigma(t)$ = task(m) and applicable in s_i and
    - π is a solution for $\mathcal{P}'=(s_i, \delta(w_i,t,m,\sigma), O, M)$.

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:**

•if π is a solution for $\mathcal{P}$, then we say that <u>π accomplishes P</u>

•intuition: there is a way to decompose w_i into π such that:

•π is executable in s_i and

•each decomposition is applicable in an appropriate state of the world

•**w_i is empty and π 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 $\mathcal{P}'=(\gamma(s_i,a_1), w_i-\{t\}, O, M)$**

•**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. $\sigma(t)$ = task(m) and applicable in s_i and**

•**π is a solution for $\mathcal{P}'=(s_i, \delta(w_i,t,m,\sigma), O, M)$.**

•2^{nd} and 3^{rd} case: recursive definition

•if w_i is not totally ordered more than one node may have no predecessors and both cases may apply

Overview

- Tasks and Task Networks
- Methods (Refinements)
- Decomposition of Tasks
- Domains, Problems and Solutions
- Planning with Task Networks
- General HTN Planning

Overview

•Tasks and Task Networks

•Methods (Refinements)

•Decomposition of Tasks

•Domains, Problems and Solutions

- just done: defining the semantics of STN planning problems and solutions

➢Planning with Task Networks

- now: two algorithms for solving STN planning problems

•General HTN Planning

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)
```

Ground-TFD: Pseudo Code

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

•**function Ground-TFD(s,⟨t_1,…,t_k⟩,O,M)**

•**if k=0 return ⟨⟩**

•if **t_1.isPrimitive() then**

•***actions* = {(a,σ) | a=σ(t_1) and a applicable in s}**

•**if *actions*.isEmpty() then return failure**

•**(a,σ) = *actions*.chooseOne()**

•***plan* ← Ground-TFD(γ(s,a),σ(⟨t_2,…,t_k⟩),O,M)**

•**if *plan* = failure then return failure**

•**else return ⟨a⟩ • *plan***

•**else** t_1 is non-primitive

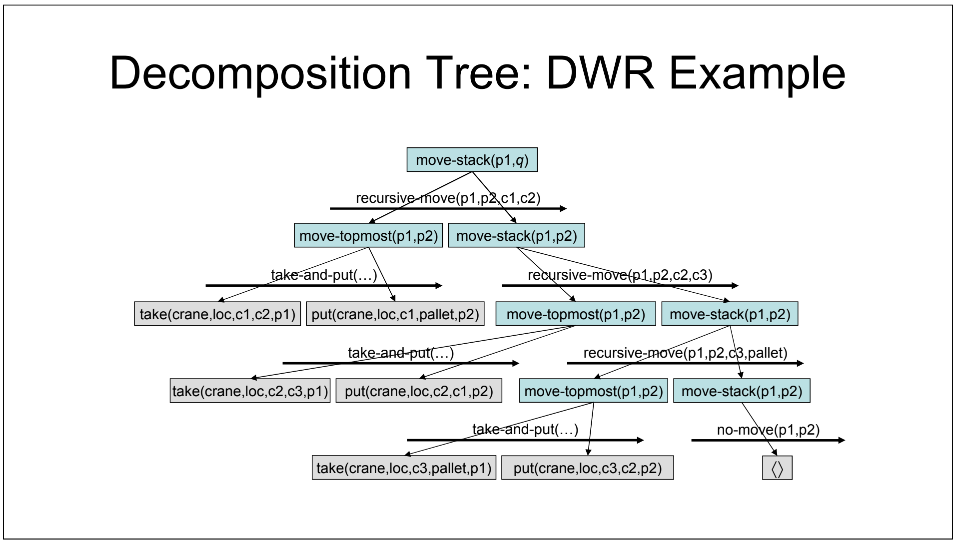
•***methods* = {(m,σ) | m is relevant for σ(t_1) and m is applicable in s}**

•**if *methods*.isEmpty() then return failure**

•**(m,σ) = *methods*.chooseOne()**

•***plan* ← subtasks(m) • σ(⟨t_2,…,t_k⟩)**

•**return Ground-TFD(s,*plan*,O,M)**



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 δ
- •no search required in this example

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

```
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(t1) 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(t1) 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)
```

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 ⟨⟩

•*task* ← {*t*∈*U* | *t* has no predecessors in *w*.*E*}.chooseOne()

•if *task*.isPrimitive() then

•*actions* = {(*a*,σ) | *a*=σ(t_1) 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_1) 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

- Tasks and Task Networks
- Methods (Refinements)
- Decomposition of Tasks
- Domains, Problems and Solutions
- Planning with Task Networks
- General HTN Planning

Overview

•Tasks and Task Networks

•Methods (Refinements)

•Decomposition of Tasks

•Domains, Problems and Solutions

•Planning with Task Networks

- just done : two algorithms for solving STN planning problems

➢General HTN Planning

- now: generalizing the STN planning problem and approach

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

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**

HTN Methods

- Let M_S be a set of method symbols. An <u>HTN method</u> is a 4-tuple m=(name(m),task(m),subtasks(m),constr(m)) where:
  - 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
  - task(m): a non-primitive task
  - (subtasks(m),constr(m)): a hierarchical task network (HTN).

HTN Methods

•extension of the definition of an STN method

•Let M_S be a set of method symbols. An <u>HTN method</u> is a 4-tuple m=(name(m),task(m),subtasks(m),constr(m)) where:

- **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;**
- **task(m): a non-primitive task;**
- **(subtasks(m),constr(m)): a** hierarchical **task network (HTN).**

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_d,p_d)}
    - constraints: {$t_1 \prec t_2$, before({t_1}, top(c,p_o)), before({t_1}, on(c,x_o)), before({t_1}, attached(p_o,l)), before({t_1}, belong(k,l)), before({t_2}, attached(p_d,l)), before({t_2}, top(x_d,p_d))}

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_d,p_d)}

•constraints: {$t_1 \prec t_2$, before({t_1}, top(c,p_o)), before({t_1}, on(c,x_o)), before({t_1}, attached(p_o,l)), before({t_1}, belong(k,l)), before({t_2}, attached(p_d,l)), before({t_2}, 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: {$t_1 \prec t_2$, 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))}

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: {$t_1 \prec t_2$, 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))}

    •note: problem with no-move: cannot add before-constraint when there are no tasks

•move-stack-twice(p_o,p_i,p_d) trivial; not shown again

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

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

- Tasks and Task Networks
- Methods (Refinements)
- Decomposition of Tasks
- Domains, Problems and Solutions
- Planning with Task Networks
- General HTN Planning

Overview

•Tasks and Task Networks

•Methods (Refinements)

•Decomposition of Tasks

•Domains, Problems and Solutions

•Planning with Task Networks

•General HTN Planning