Tutorial: An Introduction to Hierarchical Task Network (HTN) Planning

Pascal Bercher and Daniel Höller

Institute of Artificial Intelligence,
Ulm University, Germany

June 25th, ICAPS 2018 (Delft)

Overview Part II

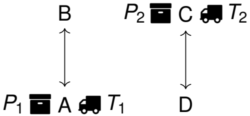
Solving HTN Planning Problems

- **Search-based Approaches**
  - Plan Space Search
  - Progression Search
- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT
- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms

Example Domain



$deliver(p, l_2)$

$get\text{-}to(v, l_1)$ $\boxed{pick\text{-}up(v, l_1, p)}$ $get\text{-}to(v, l_2)$ $\boxed{drop(v, l_2, p)}$

$m\text{-}deliver(p, l_1, l_2, v)$

$get\text{-}to(v, l_2)$

$l_1 \neq l_2$

$\boxed{drive(v, l_1, l_2)}$

$m\text{-}direct(v, l_1, l_2)$

$get\text{-}to(v, l_2)$

$l_1 \neq l_2$

$get\text{-}to(v, l_1)$ $\boxed{drive(v, l_1, l_2)}$

$m\text{-}via(v, l_1, l_2)$

$get\text{-}to(v, l)$

$\boxed{no\text{-}op()}$

$m\text{-}noop(v, l)$

Overview Part II

Solving HTN Planning Problems

- **Search-based Approaches**
  - **Plan Space Search**
  - Progression Search
- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT
- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms

Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition

Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated

Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
- Elements of the partial plan preventing it from being a solution are represented as so-called *flaws*:

Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
- Elements of the partial plan preventing it from being a solution are represented as so-called *flaws*:
  - *compound task flaw t*: the task *t* is compound, i.e., not decomposed yet

Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
- Elements of the partial plan preventing it from being a solution are represented as so-called *flaws*:
  - *compound task flaw t*: the task t is compound, i.e., not decomposed yet
  - *open precondition flaw* (t, oc): the precondition oc of the task t is still open or unprotected, i.e., no causal link protects it yet

Plan Space-based Search – Basic Characteristics

- Search bases upon Partial-Order Causal-Link (POCL) planning – extended to deal with task decomposition
- Search nodes are partially ordered partial plans, i.e., they get refined until a search node corresponding to a solution plan is generated
- Elements of the partial plan preventing it from being a solution are represented as so-called *flaws*:
  - *compound task flaw* t: the task t is compound, i.e., not decomposed yet
  - *open precondition flaw* (t, oc): the precondition oc of the task t is still open or unprotected, i.e., no causal link protects it yet
  - *causal treat flaw* $t \lightning (t', c, t'')$: there is a causal link between t' and t'' protecting the condition c and the ordering constraints allow t to be ordered between t' and t'', i.e., $t' < t < t''$ – and c is a delete effect of t.

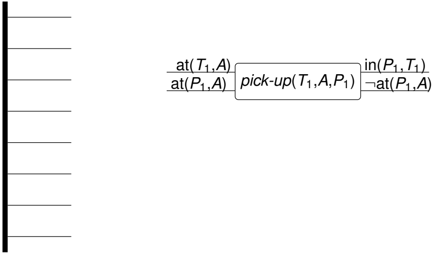
Flaws in Partial Plans

get-to(T_1,*B*)

Modifications for *compound task flaws*:

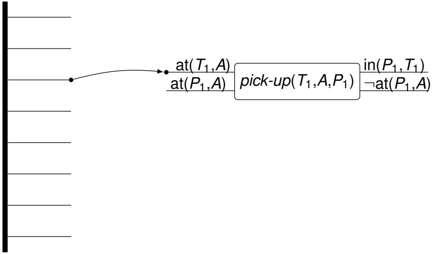
- Decompose the compound task (one modification for each method)

Flaws in Partial Plans



Modifications for *open precondition flaws*:

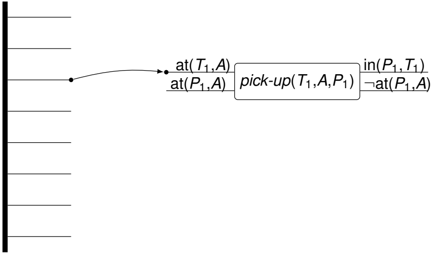
Flaws in Partial Plans



Modifications for *open precondition flaws*:

- Insert a causal link from existing plan step (one modification for each possible producer)

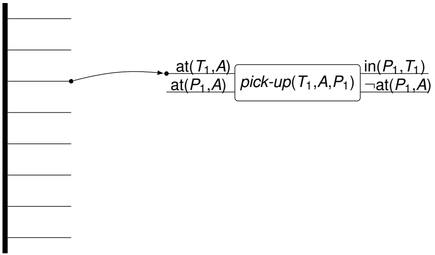
Flaws in Partial Plans



Modifications for *open precondition flaws*:

- Insert a causal link from existing plan step (one modification for each possible producer)
- Decompose a compound task if it has a sub task with a compatible effect (one modification for each method that has a compatible sub task)

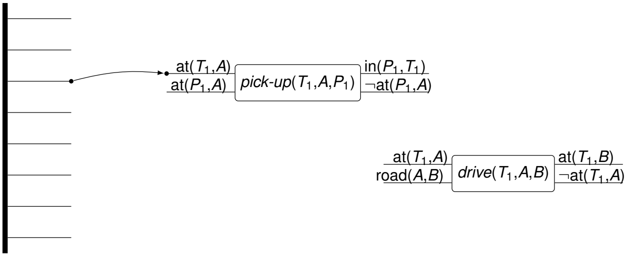
Flaws in Partial Plans



Modifications for *open precondition flaws*:

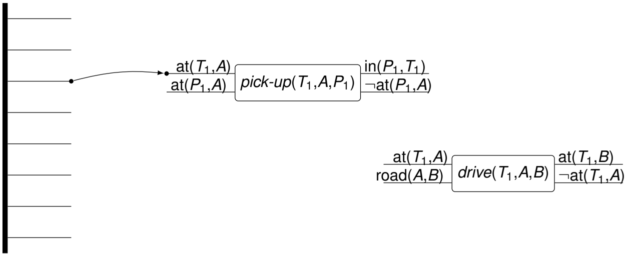
- Insert a causal link from existing plan step (one modification for each possible producer)
- Decompose a compound task if it has a sub task with a compatible effect (one modification for each method that has a compatible sub task)
- Insert a causal link from a newly inserted task (one modification for each possible producer) – only if task insertion is allowed!

Flaws in Partial Plans



Modifications for *causal threat flaws*:

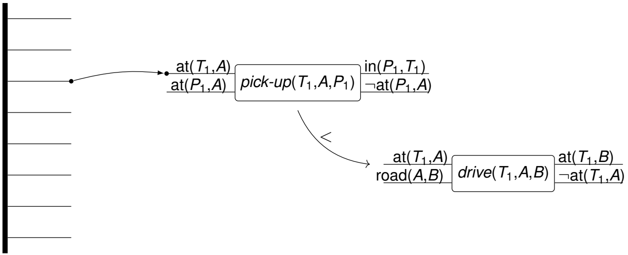
Flaws in Partial Plans



Modifications for *causal threat flaws*:

- Move the threatening task before the producer of the threatened link, called *demotion* (not possible here)

Flaws in Partial Plans



Modifications for *causal threat flaws*:

- Move the threatening task before the producer of the threatened link, called *demotion* (not possible here)
- Move the threatening task behind the consumer of the threatened link, called *promotion*

Plan Space-based Search – Basic Characteristics (Cont'd)

- Partial plans (as well as solutions) are only partially ordered, thus compactly representing many linearizations

Plan Space-based Search – Basic Characteristics (Cont'd)

- Partial plans (as well as solutions) are only partially ordered, thus compactly representing many linearizations
- Search works both top-down (decomposition of compound tasks) as well as backwards (goal-directed causal link establishment)

Plan Space-based Search – Basic Characteristics (Cont'd)

- Partial plans (as well as solutions) are only partially ordered, thus compactly representing many linearizations
- Search works both top-down (decomposition of compound tasks) as well as backwards (goal-directed causal link establishment)
- Search works in a two-step way:
  - Select a most-promising plan (via standard search strategies)

Plan Space-based Search – Basic Characteristics (Cont'd)

- Partial plans (as well as solutions) are only partially ordered, thus compactly representing many linearizations
- Search works both top-down (decomposition of compound tasks) as well as backwards (goal-directed causal link establishment)
- Search works in a two-step way:
  - Select a most-promising plan (via standard search strategies)
  - Then, select a flaw (this is *not(!)* a backtrack point) and branch over all possibilities to resolve it
- Follows the principle of least commitment

Standard Plan Space-based Algorithm

Input : $\mathit{fringe} = \{P_{\mathrm{init}}\}$
Output : A solution plan or `fail`.

```
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P})
              ∪Successors(P, f)
return fail
```

Standard Plan Space-based Algorithm

Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

```
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P})
              ∪Successors(P, f)
return fail
```

- **Initial partial plan** P_{init} equals the initial task network preceded by an artificial task encoding the initial state

Standard Plan Space-based Algorithm

Input : $\textit{fringe} = \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

```
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P})
               ∪Successors(P, f)
return fail
```

- **Initial partial plan** P_{init} equals the initial task network preceded by an artificial task encoding the initial state
- **Search nodes** contain partial plans of the form $(T, \prec, \alpha, CL)$

Standard Plan Space-based Algorithm

Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

```
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P})
              ∪ Successors(P, f)
return fail
```

- **Initial partial plan** P_{init} equals the initial task network preceded by an artificial task encoding the initial state
- **Search nodes** contain partial plans of the form $(T, \prec, \alpha, CL)$
- **Fringe** is sorted according to some heuristic

Standard Plan Space-based Algorithm

Input : $\mathit{fringe} = \{P_{\mathrm{init}}\}$
Output : A solution plan or `fail`.

while $\mathit{fringe} \neq \emptyset$ **do**
  $P :=$ **PlanSel**(fringe)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  $\mathit{fringe} := (\mathit{fringe} \setminus \{P\})$
      $\cup \mathit{Successors}(P, f)$
return `fail`

- **Initial partial plan** P_{init} equals the initial task network preceded by an artificial task encoding the initial state
- **Search nodes** contain partial plans of the form $(T, \prec, \alpha, CL)$
- **Fringe** is sorted according to some heuristic
- **F** is a the set of all flaws of the current partial plan

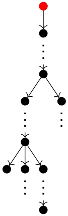
Standard Plan Space-based Algorithm

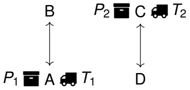
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

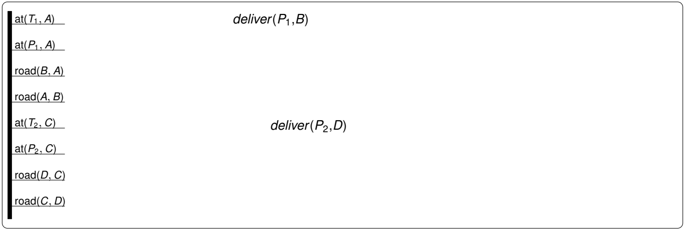
```
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P})
              ∪Successors(P, f)
return fail
```

- **Initial partial plan** P_{init} equals the initial task network preceded by an artificial task encoding the initial state
- **Search nodes** contain partial plans of the form $(T, \prec, \alpha, CL)$
- **Fringe** is sorted according to some heuristic
- **F** is a the set of all flaws of the current partial plan
- **FlawSel** selects (not a backtrack point!) a flaw according to a flaw selection strategy

Standard Plan Space-based Algorithm





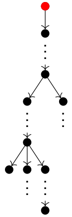


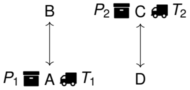
Flaws	**Modifications**

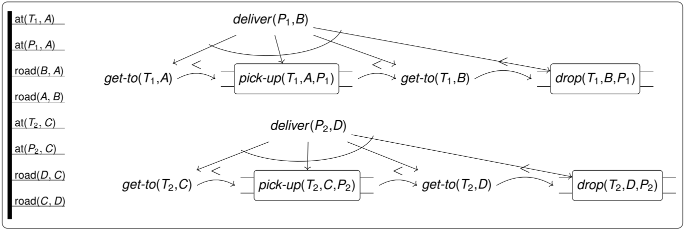
Input : $\textit{fringe} = \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while $\textit{fringe} \neq \emptyset$ **do**
  $P :=$ **PlanSel**$(\textit{fringe})$
  $F := \textit{FlawDet}(P)$
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  $\textit{fringe} := (\textit{fringe} \setminus \{P\})$
    $\cup \textit{Successors}(P, f)$
return `fail`

Standard Plan Space-based Algorithm





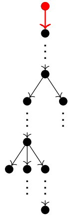


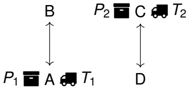
Flaws	**Modifications**
compound task: deliver(P_1, B)	decompose with *m-deliver*(P_1, A, B, T_1)
compound task: deliver(P_2, D)	decompose with *m-deliver*(P_2, C, D, T_2)

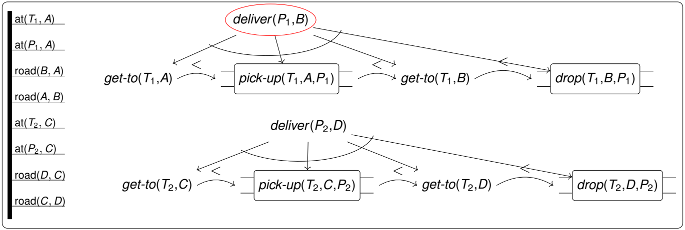
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F := FlawDet(P)$
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (fringe \setminus \{P\})$
    $\cup Successors(P, f)$
return `fail`

Standard Plan Space-based Algorithm





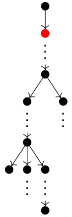


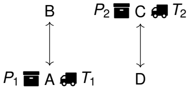
Flaws	Modifications
compound task: deliver(P_1, B***)***	**decompose with *m-deliver(*** P_1, A, B, T_1***)***
compound task: deliver(P_2, D*)*	decompose with *m-deliver(* P_2, C, D, T_2*)*

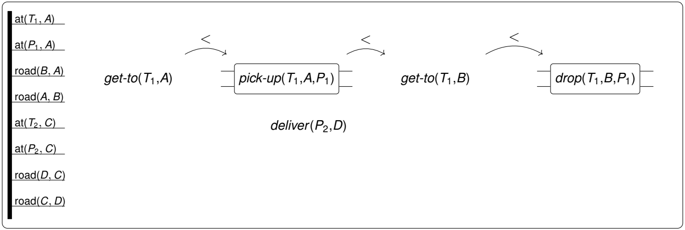
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Standard Plan Space-based Algorithm





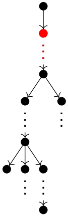


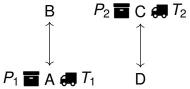
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

```
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P})
              ∪Successors(P, f)
return fail
```

Flaws	Modifications
compound task: deliver(P_2, D)	decompose with *m-deliver*(P_2, C, D, T_2)
compound task: get-to(T_1, A)	decompose with *m-direct*(T_1, B, A) decompose with *m-via*(T_1, B, A) decompose with *m-noop*(T_1, A)
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init* decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A) decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
open prec.: at(P_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init*
compound task: get-to(T_1, B)	decompose with *m-direct*(T_1, A, B) decompose with *m-via*(T_1, A, B) decompose with *m-noop*(T_1, B)
open prec.: at(T_1, B) of *drop*(T_1, B, P_1)	decompose *get-to*(T_1, B) with *m-direct*(T_1, A, B) decompose *get-to*(T_1, B) with *m-via*(T_1, A, B)
open prec.: in(P_1, T_1) of *drop*(T_1, B, P_1)	insert causal link from *pickup*($T1$, A, $P1$)

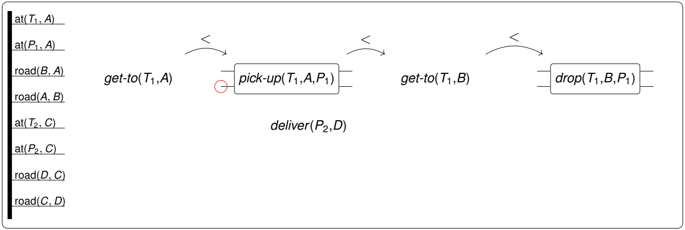
Standard Plan Space-based Algorithm





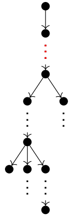
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

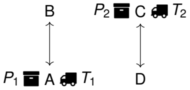
while *fringe* $\neq \emptyset$ **do**
  P := **PlanSel**(*fringe*)
  F := *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  f := **FlawSel**(F)
  fringe := (*fringe* $\setminus \{P\}$)
    $\cup$*Successors*(P, f)
return `fail`



Flaws	**Modifications**
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
open prec.:* at(P_1, A) of *pick-up(T_1, A, P_1)	**insert causal link from *init***
compound task: get-to(T_1, B)	decompose with *m-direct(T_1, A, B)* decompose with *m-via(T_1, A, B)* decompose with *m-noop(T_1, B)*
open prec.: at(T_1, B) of *drop(T_1, B, P_1)*	decompose *get-to(T_1, B)* with *m-direct(T_1, A, B)* decompose *get-to(T_1, B)* with *m-via(T_1, A, B)*
open prec.: in(P_1, T_1) of *drop(T_1, B, P_1)*	insert causal link from *pickup($T1$, A, $P1$)*

Standard Plan Space-based Algorithm

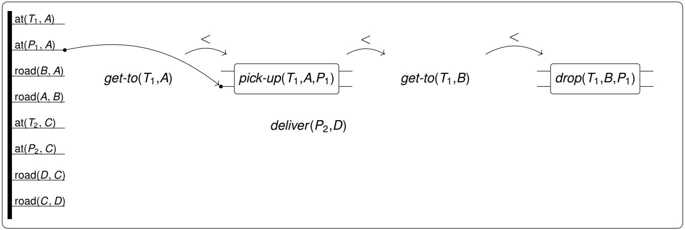




Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

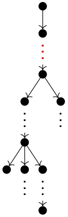
while *fringe* $\neq \emptyset$ **do**
- $P :=$ **PlanSel**(*fringe*)
- $F :=$ *FlawDet*(P)
- **if** $F = \emptyset$ **then return** P
- $f :=$ **FlawSel**(F)
- *fringe* $:= (\textit{fringe} \setminus \{P\}) \cup \textit{Successors}(P, f)$

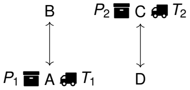
return `fail`

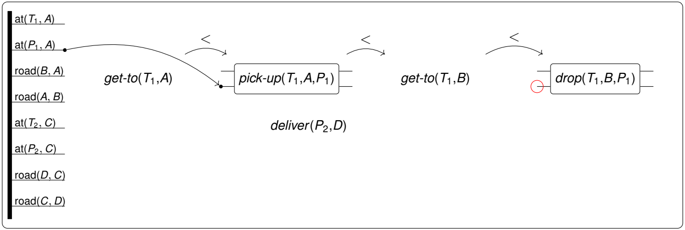


Flaws	**Modifications**
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)*
	decompose with *m-via(T_1, B, A)*
	decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init*
	decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)*
	decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
compound task: get-to(T_1, B)	decompose with *m-direct(T_1, A, B)*
	decompose with *m-via(T_1, A, B)*
	decompose with *m-noop(T_1, B)*
open prec.: at(T_1, B) of *drop(T_1, B, P_1)*	decompose *get-to(T_1, B)* with *m-direct(T_1, A, B)*
	decompose *get-to(T_1, B)* with *m-via(T_1, A, B)*
open prec.: in(P_1, T_1) of *drop(T_1, B, P_1)*	insert causal link from *pickup(T1, A, P1)*

Standard Plan Space-based Algorithm





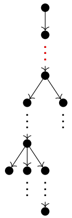


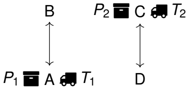
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= ($*fringe* $\setminus \{P\})$
  $\cup$*Successors*(P, f)
return `fail`

Flaws	Modifications
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
compound task: get-to(T_1, B)	decompose with *m-direct(T_1, A, B)* decompose with *m-via(T_1, A, B)* decompose with *m-noop(T_1, B)*
open prec.: at(T_1, B) of *drop(T_1, B, P_1)*	decompose *get-to(T_1, B)* with *m-direct(T_1, A, B)* decompose *get-to(T_1, B)* with *m-via(T_1, A, B)*
open prec.:* in(P_1, T_1) of *drop(T_1, B, P_1)	**insert causal link from *pickup($T1$, A, $P1$)***

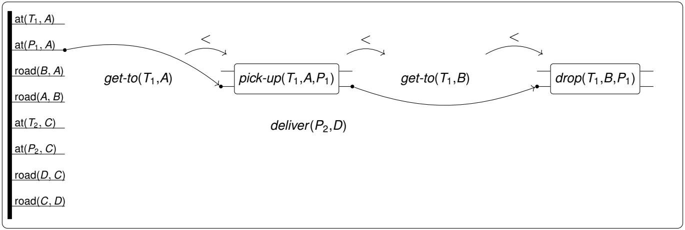
Standard Plan Space-based Algorithm





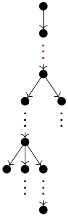
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

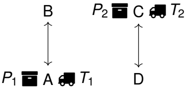
```
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P})
              ∪Successors(P, f)
return fail
```



Flaws	Modifications
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)*
	decompose with *m-via(T_1, B, A)*
	decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init*
	decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)*
	decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
compound task: get-to(T_1, B)	decompose with *m-direct(T_1, A, B)*
	decompose with *m-via(T_1, A, B)*
	decompose with *m-noop(T_1, B)*
open prec.: at(T_1, B) of *drop(T_1, B, P_1)*	decompose *get-to(T_1, B)* with *m-direct(T_1, A, B)*
	decompose *get-to(T_1, B)* with *m-via(T_1, A, B)*

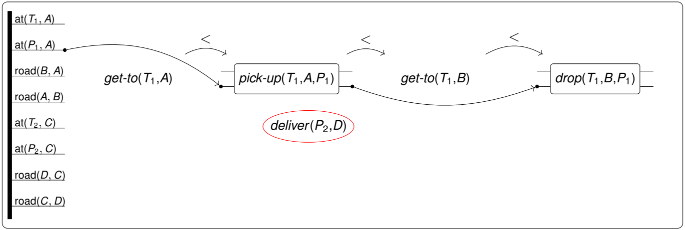
Standard Plan Space-based Algorithm





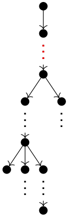
Input : $fringe = \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

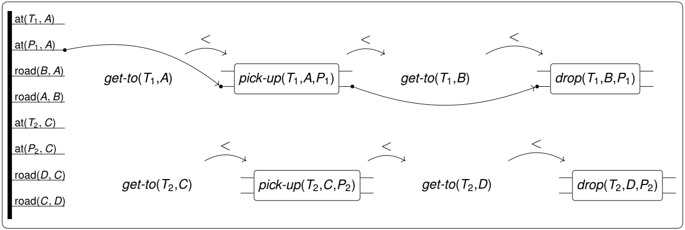
```
while fringe ≠ ∅ do
    P := PlanSel(fringe)
    F := FlawDet(P)
    if F = ∅ then return P
    f := FlawSel(F)
    fringe := (fringe \ {P})
              ∪Successors(P, f)
return fail
```

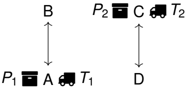


Flaws	**Modifications**
compound task: deliver(P_2, D)	**decompose with *m-deliver*(P_2, C, D, T_2)**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
compound task: get-to(T_1, B)	decompose with *m-direct(T_1, A, B)* decompose with *m-via(T_1, A, B)* decompose with *m-noop(T_1, B)*
open prec.: at(T_1, B) of *drop(T_1, B, P_1)*	decompose *get-to(T_1, B)* with *m-direct(T_1, A, B)* decompose *get-to(T_1, B)* with *m-via(T_1, A, B)*

Standard Plan Space-based Algorithm





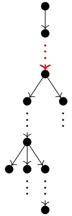


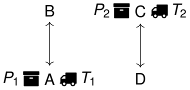
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

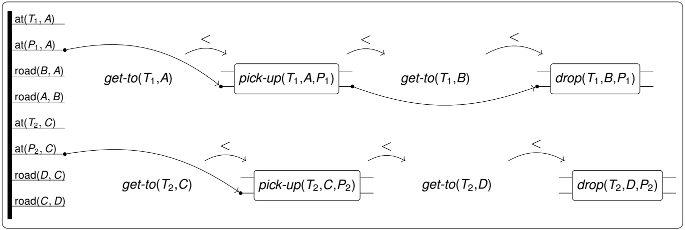
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)*
	decompose with *m-via(T_1, B, A)*
	decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init*
	decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)*
	decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
compound task: get-to(T_1, B)	decompose with *m-direct(T_1, A, B)*
	decompose with *m-via(T_1, A, B)*
	decompose with *m-noop(T_1, B)*
open prec.: at(T_1, B) of *drop(T_1, B, P_1)*	decompose *get-to(T_1, B)* with *m-direct(T_1, A, B)*
	decompose *get-to(T_1, B)* with *m-via(T_1, A, B)*
...	...

Standard Plan Space-based Algorithm





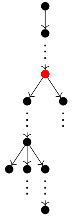


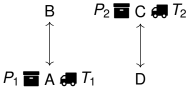
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

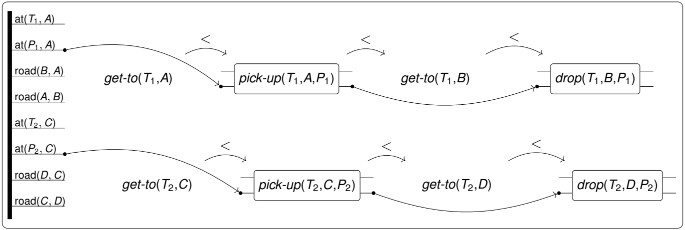
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= ($*fringe* $\setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Flaws	Modifications
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)*
	decompose with *m-via(T_1, B, A)*
	decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init*
	decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)*
	decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
compound task: get-to(T_1, B)	decompose with *m-direct(T_1, A, B)*
	decompose with *m-via(T_1, A, B)*
	decompose with *m-noop(T_1, B)*
open prec.: at(T_1, B) of *drop(T_1, B, P_1)*	decompose *get-to(T_1, B)* with *m-direct(T_1, A, B)*
	decompose *get-to(T_1, B)* with *m-via(T_1, A, B)*
...	...

Standard Plan Space-based Algorithm





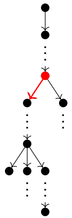


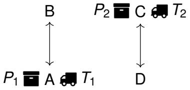
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

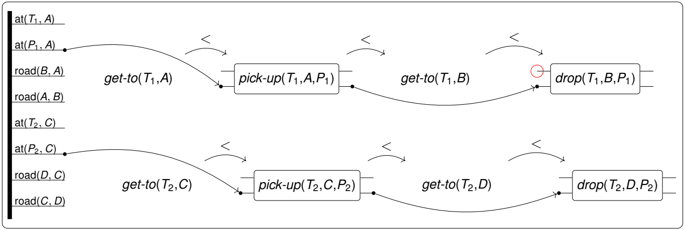
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= ($*fringe* $\setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Flaws	Modifications
compound task: get-to(T_1, A)	decompose with *m-direct*(T_1, B, A) decompose with *m-via*(T_1, B, A) decompose with *m-noop*(T_1, A)
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init* decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A) decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
compound task: get-to(T_1, B)	decompose with *m-direct*(T_1, A, B) decompose with *m-via*(T_1, A, B) decompose with *m-noop*(T_1, B)
open prec.: at(T_1, B) of *drop*(T_1, B, P_1)	decompose *get-to*(T_1, B) with *m-direct*(T_1, A, B) decompose *get-to*(T_1, B) with *m-via*(T_1, A, B)
...	...

Standard Plan Space-based Algorithm





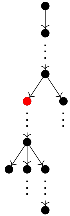


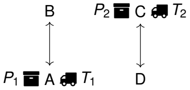
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

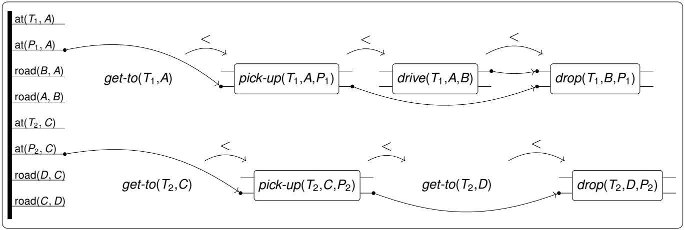
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Flaws	**Modifications**
compound task: get-to(T_1, *A*)	decompose with *m-direct*(T_1, *B*, *A*)
	decompose with *m-via*(T_1, *B*, *A*)
	decompose with *m-noop*(T_1, *A*)
open prec.: at(T_1, *A*) of *pick-up*(T_1, *A*, P_1)	insert causal link from *init*
	decompose *get-to*(T_1, *A*) with *m-direct*(T_1, *B*, *A*)
	decompose *get-to*(T_1, *A*) with *m-via*(T_1, *B*, *A*)
compound task: get-to(T_1, *B*)	decompose with *m-direct*(T_1, *A*, *B*)
	decompose with *m-via*(T_1, *A*, *B*)
	decompose with *m-noop*(T_1, *B*)
***open prec.:* at(T_1, *B*) of *drop*(T_1, *B*, P_1)**	**decompose *get-to*(T_1, *B*) with *m-direct*(T_1, *A*, *B*)**
	decompose *get-to*(T_1, *B*) with *m-via*(T_1, *A*, *B*)
...	...

Standard Plan Space-based Algorithm





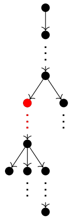


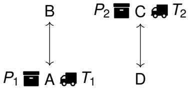
Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct*(T_1, B, A)
	decompose with *m-via*(T_1, B, A)
	decompose with *m-noop*(T_1, A)
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init*
	decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A)
	decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
open prec.: at(T_1, A) of *drive*(T_1, A, B)	insert causal link from *init*
	decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A)
	decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
open prec.: road(A, B) of *drive*(T_1, A, B)	insert causal link from *init*
...	...

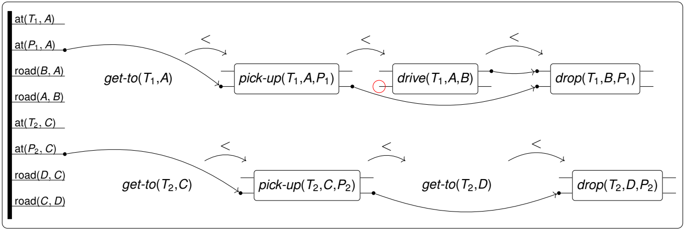
Input : $fringe = \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while $fringe \neq \emptyset$ **do**
  $P :=$ **PlanSel**($fringe$)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  $fringe := (fringe \setminus \{P\})$
    $\cup Successors(P, f)$
return `fail`

Standard Plan Space-based Algorithm





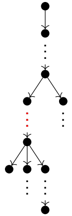


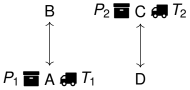
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

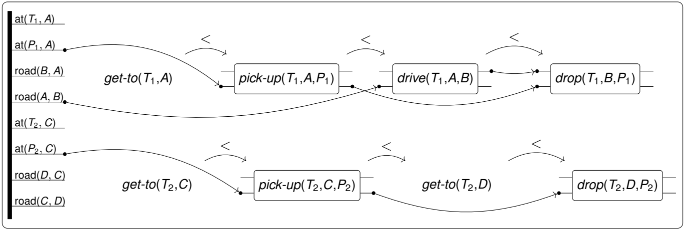
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$ *Successors*(P, f)
return `fail`

Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct*(T_1, B, A)
	decompose with *m-via*(T_1, B, A)
	decompose with *m-noop*(T_1, A)
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init*
	decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A)
	decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
open prec.: at(T_1, A) of *drive*(T_1, A, B)	insert causal link from *init*
	decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A)
	decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
***open prec.:* road(A, B) of *drive*(T_1, A, B)**	**insert causal link from *init***
...	...

Standard Plan Space-based Algorithm





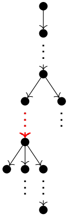


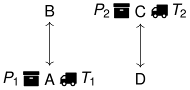
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

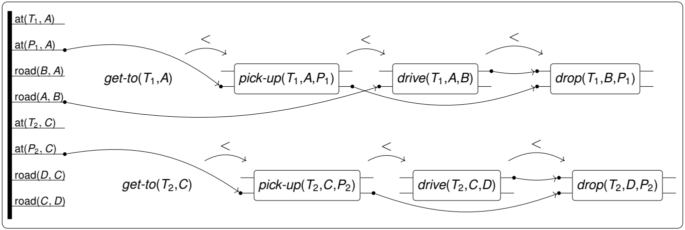
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Flaws	Modifications
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)*
	decompose with *m-via(T_1, B, A)*
	decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init*
	decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)*
	decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
open prec.: at(T_1, A) of *drive(T_1, A, B)*	insert causal link from *init*
	decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)*
	decompose *get-to(T_1, A)* with *m-via(T_1, B, A)*
...	...

Standard Plan Space-based Algorithm





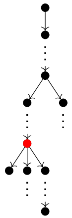


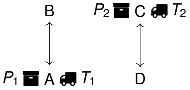
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

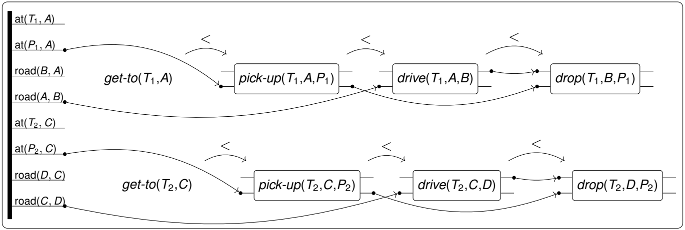
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= ($*fringe* $\setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct*(T_1, B, A)
	decompose with *m-via*(T_1, B, A)
	decompose with *m-noop*(T_1, A)
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init*
	decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A)
	decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
open prec.: at(T_1, A) of *drive*(T_1, A, B)	insert causal link from *init*
	decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A)
	decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
...	...

Standard Plan Space-based Algorithm





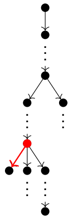


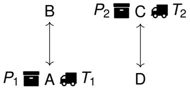
Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct*(T_1, B, A)
	decompose with *m-via*(T_1, B, A)
	decompose with *m-noop*(T_1, A)
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init*
	decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A)
	decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
open prec.: at(T_1, A) of *drive*(T_1, A, B)	insert causal link from *init*
	decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A)
	decompose *get-to*(T_1, A) with *m-via*(T_1, B, A)
...	...

Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

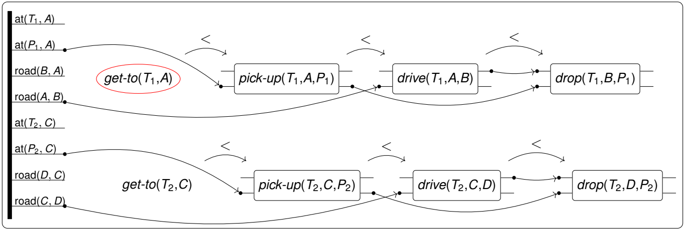
Standard Plan Space-based Algorithm





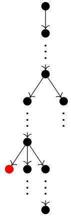
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

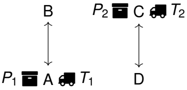
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

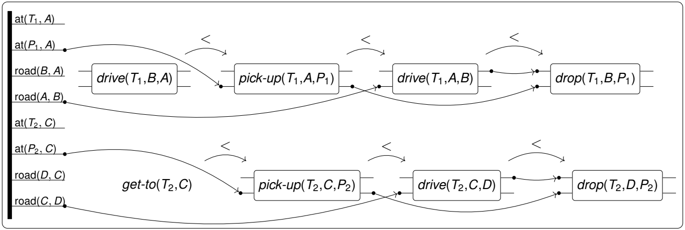


Flaws	Modifications
compound task: get-to(T_1, A***)***	**decompose with *m-direct(*** T_1, B, A***)***
	decompose with *m-via(*T_1, B, A*)*
	decompose with *m-noop(*T_1, A*)*
open prec.: at(T_1, A) of *pick-up(*T_1, A, P_1*)*	insert causal link from *init*
	decompose *get-to(*T_1, A*)* with *m-direct(*T_1, B, A*)*
	decompose *get-to(*T_1, A*)* with *m-via(*T_1, B, A*)*
open prec.: at(T_1, A) of *drive(*T_1, A, B*)*	insert causal link from *init*
	decompose *get-to(*T_1, A*)* with *m-direct(*T_1, B, A*)*
	decompose *get-to(*T_1, A*)* with *m-via(*T_1, B, A*)*
...	...

Standard Plan Space-based Algorithm





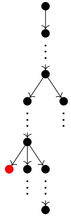


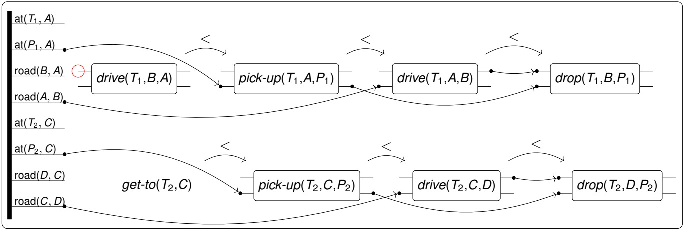
Flaws	Modifications
open prec: at(T_1, B) of *drive(T_1, B, A)*	—
open prec.: road(B, A) of *drive(T_1, B, A)*	insert causal link from *init*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* insert causal link from *drive(T_1, B, A)*
open prec.: at(T_1, A) of *drive(T_1, A, B)*	insert causal link from *init* insert causal link from *drive(T_1, B, A)*
...	...

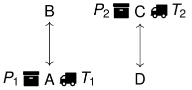
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= ($*fringe* $\setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Standard Plan Space-based Algorithm







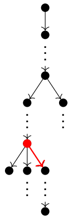
Flaws	**Modifications**
open prec:* at(T_1, B) of *drive(T_1, B, A)	**—**
open prec.: road(B, A) of *drive(T_1, B, A)*	insert causal link from *init*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* insert causal link from *drive(T_1, B, A)*
open prec.: at(T_1, A) of *drive(T_1, A, B)*	insert causal link from *init* insert causal link from *drive(T_1, B, A)*
...	...

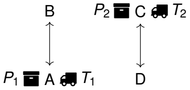
This partial plan can be discarded, because it has a flaw without modifications

Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= ($*fringe* $\setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

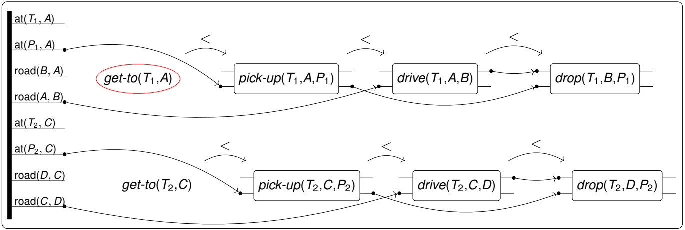
Standard Plan Space-based Algorithm





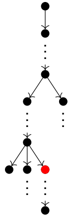
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

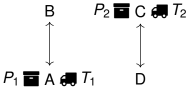
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

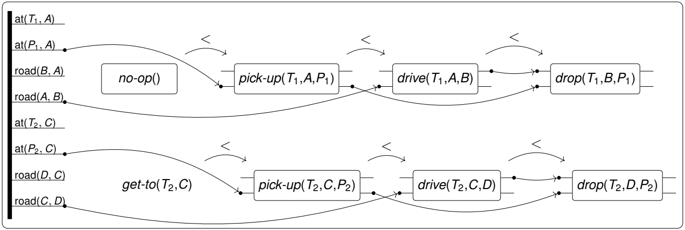


Flaws	Modifications
compound task: get-to(**T_1**, A)	decompose with *m-direct(*T_1*, B, A)*
	decompose with *m-via(*T_1*, B, A)*
	decompose with *m-noop(*T_1*, A)*
open prec.: at(T_1, A) of *pick-up(*T_1*, A, *P_1*)*	insert causal link from *init*
	decompose *get-to(*T_1*, A)* with *m-direct(*T_1*, B, A)*
	decompose *get-to(*T_1*, A)* with *m-via(*T_1*, B, A)*
open prec.: at(T_1, A) of *drive(*T_1*, A, B)*	insert causal link from *init*
	decompose *get-to(*T_1*, A)* with *m-direct(*T_1*, B, A)*
	decompose *get-to(*T_1*, A)* with *m-via(*T_1*, B, A)*
...	...

Standard Plan Space-based Algorithm





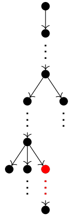


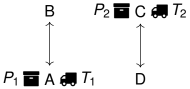
Flaws	**Modifications**
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init*
open prec.: at(T_1, A) of *drive(T_1, A, B)*	insert causal link from *init*
...	...

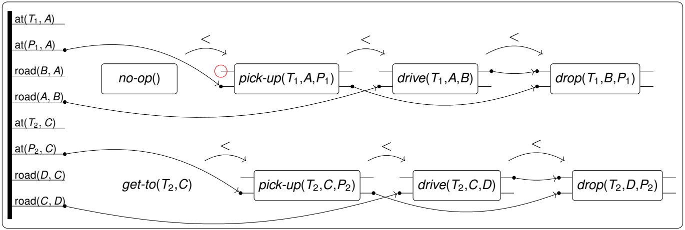
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= ($*fringe* $\setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Standard Plan Space-based Algorithm





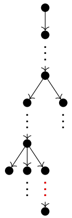


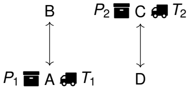
Flaws	Modifications
open prec.:* at(T_1, A) of *pick-up(T_1, A, P_1)	**insert causal link from *init***
open prec.: at(T_1, A) of *drive(T_1, A, B)*	insert causal link from *init*
...	...

Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:=$ (*fringe* $\setminus \{P\}$)
    $\cup$*Successors*(P, f)
return `fail`

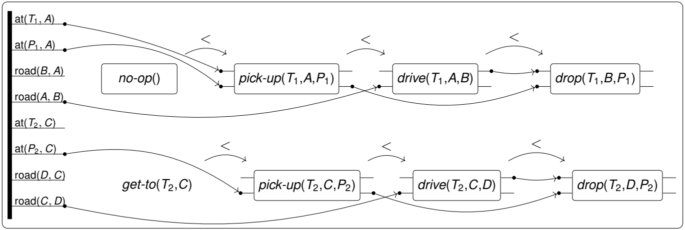
Standard Plan Space-based Algorithm





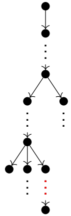
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

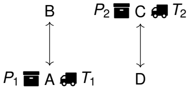
while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= ($*fringe* $\setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

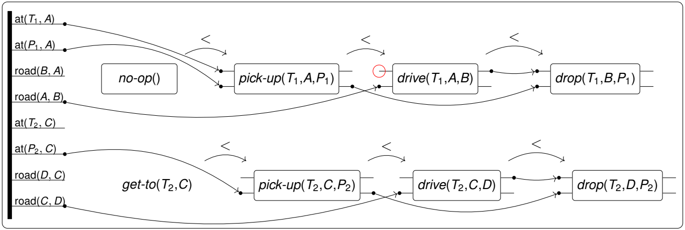


Flaws	**Modifications**
open prec.: at(T_1, A) of *drive(T_1, A, B)*	insert causal link from *init*
...	...

Standard Plan Space-based Algorithm





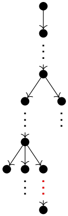


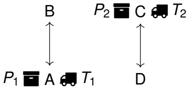
Flaws	**Modifications**
***open prec.:* at(T_1, A) of *drive*(T_1, A, B)**	**insert causal link from *init***
...	...

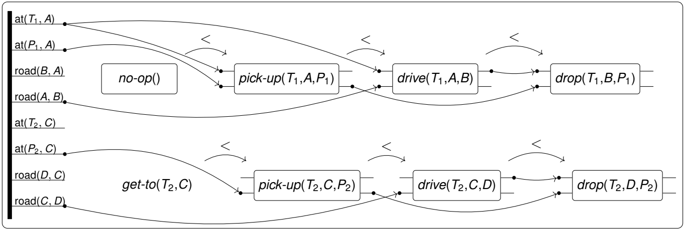
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Standard Plan Space-based Algorithm





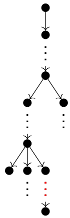


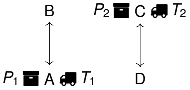
Flaws	**Modifications**
...	...

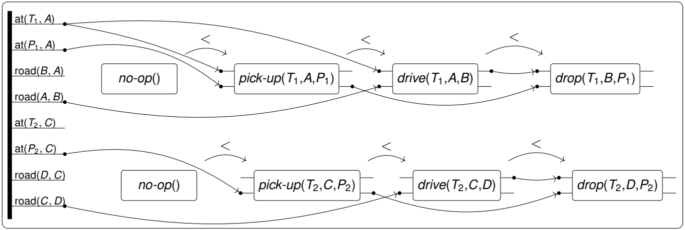
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Standard Plan Space-based Algorithm





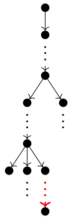


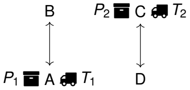
Flaws	**Modifications**
...	...

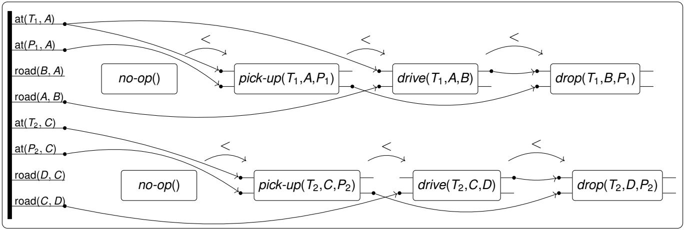
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F := \mathit{FlawDet}(P)$
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  $\mathit{fringe} := (\mathit{fringe} \setminus \{P\})$
    $\cup \mathit{Successors}(P, f)$
return `fail`

Standard Plan Space-based Algorithm





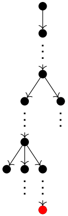


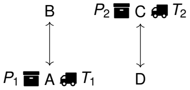
Flaws	**Modifications**
...	...

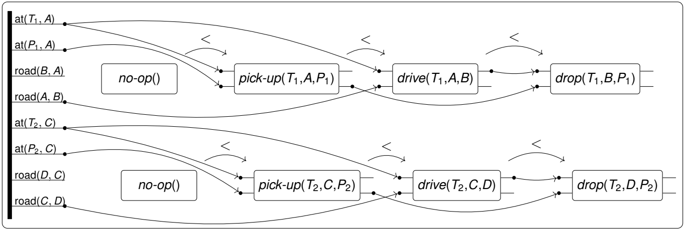
Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
  $P :=$ **PlanSel**(*fringe*)
  $F :=$ *FlawDet*(P)
  if $F = \emptyset$ **then return** P
  $f :=$ **FlawSel**(F)
  fringe $:= (\textit{fringe} \setminus \{P\})$
    $\cup$*Successors*(P, f)
return `fail`

Standard Plan Space-based Algorithm







Flaws	**Modifications**

This partial plan has no flaws, so it is a solution and returned

Input : *fringe* $= \{P_{\text{init}}\}$
Output : A solution plan or `fail`.

while *fringe* $\neq \emptyset$ **do**
 $P :=$ **PlanSel**(*fringe*)
 $F :=$ *FlawDet*(P)
 if $F = \emptyset$ **then return** P
 $f :=$ **FlawSel**(F)
 fringe $:= ($*fringe* $\setminus \{P\})$
  $\cup$*Successors*(P, f)
return `fail`

Plan Space-based Search – Properties

- Plan Space-based search is **sound**

Plan Space-based Search – Properties

- Plan Space-based search is **sound**
- ... and **complete** (completeness only depends on the plan selection function (fringe sorting), but not on the flaw selection function)

Plan Space-based Search – Properties

- Plan Space-based search is **sound**
- . . . and **complete** (completeness only depends on the plan selection function (fringe sorting), but not on the flaw selection function)
- There is **no current state** during search (the initial state is never changed)

Plan Space-based Search – Properties

- Plan Space-based search is **sound**
- . . . and **complete** (completeness only depends on the plan selection function (fringe sorting), but not on the flaw selection function)
- There is **no current state** during search (the initial state is never changed)
- Tasks are partially ordered and can be inserted anywhere in a partial plan

Overview Part II

Solving HTN Planning Problems

- **Search-based Approaches**
  - Plan Space Search
  - **Progression Search**
- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT
- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms

Progression Search – Basic Characteristics

- Only those (primitive or compound) tasks in a task network that have **no predecessor** in the ordering relations are processed
- Actions that are processed are **removed** from the network and cause **state transition**

Progression Search – Basic Characteristics

- Only those (primitive or compound) tasks in a task network that have **no predecessor** in the ordering relations are processed
- Actions that are processed are **removed** from the network and cause **state transition**

$\rightarrow$ Search nodes contain the current task network **and** state

Progression Search – Basic Characteristics

- Only those (primitive or compound) tasks in a task network that have **no predecessor** in the ordering relations are processed
- Actions that are processed are **removed** from the network and cause **state transition**

→ Search nodes contain the current task network **and** state

→ **Commitment** to the prefix of the solution during search

Progression Search – Basic Characteristics

- Only those (primitive or compound) tasks in a task network that have **no predecessor** in the ordering relations are processed
- Actions that are processed are **removed** from the network and cause **state transition**

→ Search nodes contain the current task network **and** state

→ **Commitment** to the prefix of the solution during search

→ We are searching for an **empty** task network

Standard Progression Algorithm

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```

Standard Progression Algorithm

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```

- **Search nodes** contain task network, state, and solution prefix

Standard Progression Algorithm

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```

- **Search nodes** contain task network, state, and solution prefix
- **Fringe** is sorted according to some heuristic

Standard Progression Algorithm

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```

- **Search nodes** contain task network, state, and solution prefix
- **Fringe** is sorted according to some heuristic
- **Goal test** checks for empty task network (maybe for a goal state)

Standard Progression Algorithm

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```

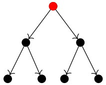
- **Search nodes** contain task network, state, and solution prefix
- **Fringe** is sorted according to some heuristic
- **Goal test** checks for empty task network (maybe for a goal state)
- **Unconstrained tasks** have no predecessor

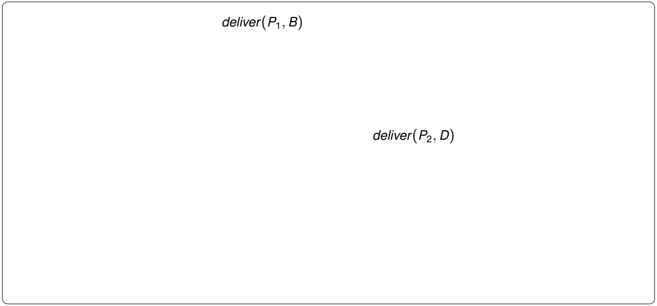
Standard Progression Algorithm

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```

- **Search nodes** contain task network, state, and solution prefix
- **Fringe** is sorted according to some heuristic
- **Goal test** checks for empty task network (maybe for a goal state)
- **Unconstrained tasks** have no predecessor
- **Action application** removes node and causes state transition

Standard Progression Algorithm

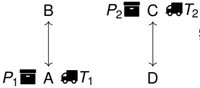




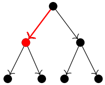
$\pi = ()$

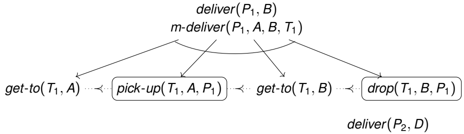
Algorithm:

fringe $\leftarrow \{(s_I, tn_I, ())\}$
while *fringe* $\neq \emptyset$ **do**
 $n \leftarrow$ *fringe.poll*()
 if *n.isgoal* **then** **return** *n*
 $U \leftarrow$ *n.unconstrainedNodes*
 for $t \in U$ **do**
  if *isPrimitive*(t) **then**
   $n' \leftarrow$ *n.apply*(t)
   fringe.add(n')
  else
   for $m \in$ *t.methods* **do**
    $n' \leftarrow$ *n.decompose*(t, m)
    fringe.add(n')



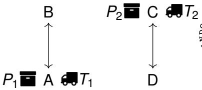
Standard Progression Algorithm



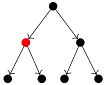


$\pi = ()$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



Standard Progression Algorithm

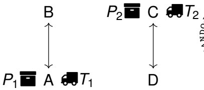


$get\text{-}to(T_1, A) \prec \boxed{pick\text{-}up(T_1, A, P_1)} \prec get\text{-}to(T_1, B) \prec \boxed{drop(T_1, B, P_1)}$

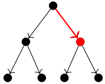
$deliver(P_2, D)$

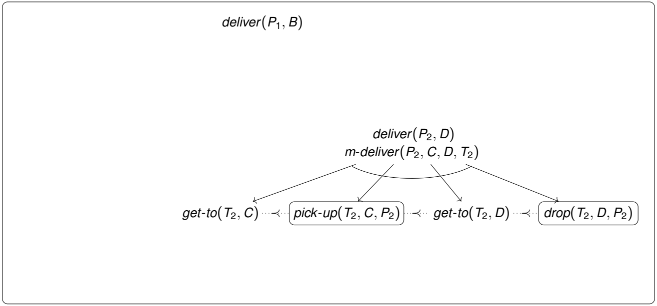
$\pi = ()$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n' ← n.apply(t)
            fringe.add(n')
        else
            for m ∈ t.methods do
                n' ← n.decompose(t, m)
                fringe.add(n')
```



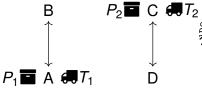
Standard Progression Algorithm



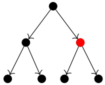


$\pi = ()$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



Standard Progression Algorithm

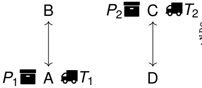


deliver(P_1, B)

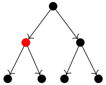
get-to$(T_2, C) \prec \boxed{\textit{pick-up}(T_2, C, P_2)} \prec$ *get-to*$(T_2, D) \prec \boxed{\textit{drop}(T_2, D, P_2)}$

$\pi = ()$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



Standard Progression Algorithm

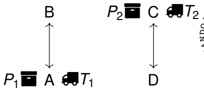


get-to(T_1, A) $\prec$ $\boxed{\textit{pick-up}(T_1, A, P_1)}$ $\prec$ *get-to*(T_1, B) $\prec$ $\boxed{\textit{drop}(T_1, B, P_1)}$

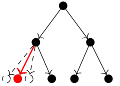
deliver(P_2, D)

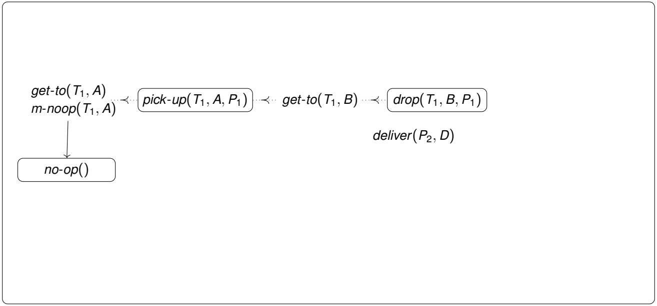
$\pi = ()$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



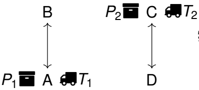
Standard Progression Algorithm



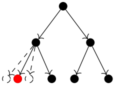


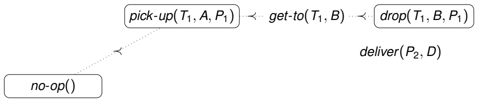
$\pi = ()$

fringe $\leftarrow \{(s_I, tn_I, ())\}$
while *fringe* $\neq \emptyset$ **do**
  $n \leftarrow$ *fringe.poll*()
  if *n.isgoal* **then return** *n*
  $U \leftarrow$ *n.unconstrainedNodes*
  for $t \in U$ **do**
    if *isPrimitive*(t) **then**
      $n' \leftarrow$ *n.apply*(t)
      fringe.add(n')
    else
      for $m \in$ *t.methods* **do**
        $n' \leftarrow$ *n.decompose*(t, m)
        fringe.add(n')



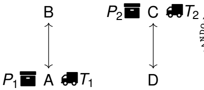
Standard Progression Algorithm



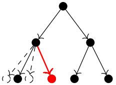


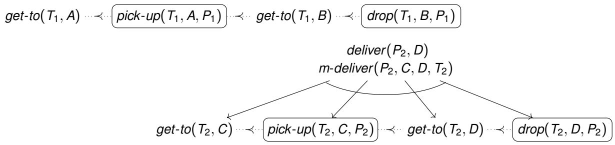
$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $U \leftarrow n.unconstrainedNodes$
  for $t \in U$ **do**
    if $isPrimitive(t)$ **then**
      $n' \leftarrow n.apply(t)$
      $fringe.add(n')$
    else
      for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$

$\pi = ()$



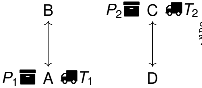
Standard Progression Algorithm



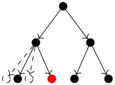


$$\pi = ()$$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n' ← n.apply(t)
            fringe.add(n')
        else
            for m ∈ t.methods do
                n' ← n.decompose(t, m)
                fringe.add(n')
```



Standard Progression Algorithm

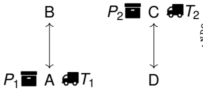


$get\text{-}to(T_1, A) \prec \boxed{pick\text{-}up(T_1, A, P_1)} \prec get\text{-}to(T_1, B) \prec \boxed{drop(T_1, B, P_1)}$

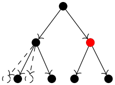
$get\text{-}to(T_2, C) \prec \boxed{pick\text{-}up(T_2, C, P_2)} \prec get\text{-}to(T_2, D) \prec \boxed{drop(T_2, D, P_2)}$

$$\pi = ()$$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



Standard Progression Algorithm

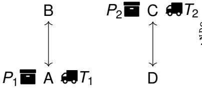


$deliver(P_1, B)$

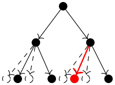
$get\text{-}to(T_2, C) \prec \boxed{pick\text{-}up(T_2, C, P_2)} \prec get\text{-}to(T_2, D) \prec \boxed{drop(T_2, D, P_2)}$

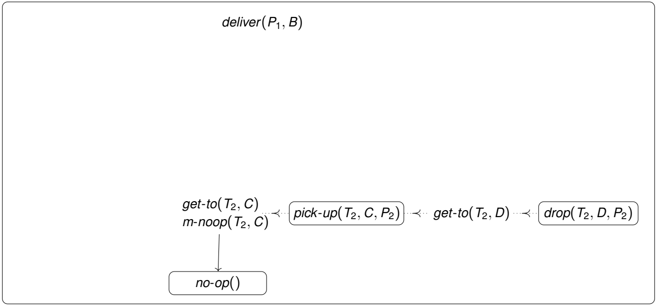
$\pi = ()$

$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
    $n \leftarrow fringe.poll()$
    if $n.isgoal$ **then return** n
    $U \leftarrow n.unconstrainedNodes$
    for $t \in U$ **do**
        if $isPrimitive(t)$ **then**
            $n' \leftarrow n.apply(t)$
            $fringe.add(n')$
        else
            for $m \in t.methods$ **do**
                $n' \leftarrow n.decompose(t, m)$
                $fringe.add(n')$



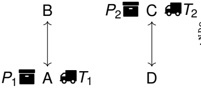
Standard Progression Algorithm



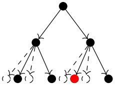


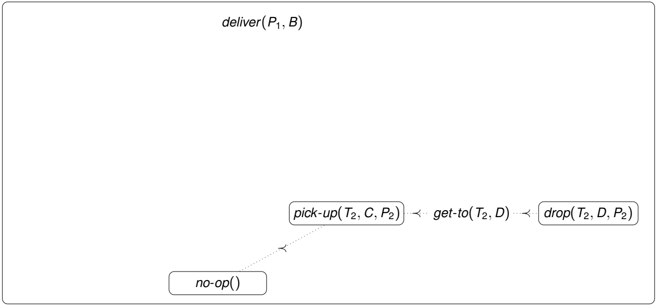
$\pi = ()$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n' ← n.apply(t)
            fringe.add(n')
        else
            for m ∈ t.methods do
                n' ← n.decompose(t, m)
                fringe.add(n')
```



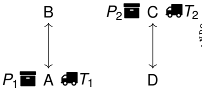
Standard Progression Algorithm



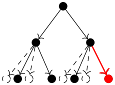


$\pi = ()$

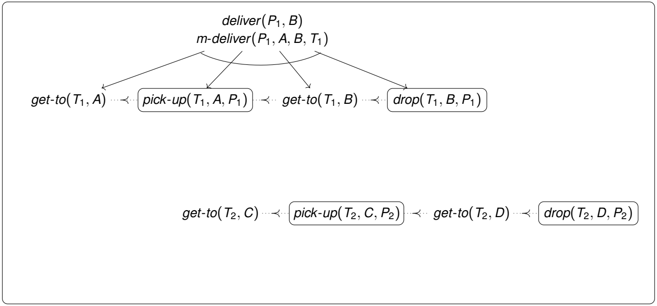
1 $fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $U \leftarrow n.unconstrainedNodes$
  for $t \in U$ **do**
    if $isPrimitive(t)$ **then**
      $n' \leftarrow n.apply(t)$
      $fringe.add(n')$
    else
      for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$



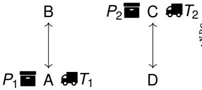
Standard Progression Algorithm



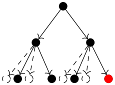
$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $U \leftarrow n.unconstrainedNodes$
  for $t \in U$ **do**
    if $isPrimitive(t)$ **then**
      $n' \leftarrow n.apply(t)$
      $fringe.add(n')$
    else
      for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$



$\pi = ()$



Standard Progression Algorithm

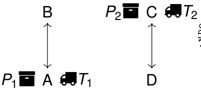


$get\text{-}to(T_1, A) \prec \boxed{pick\text{-}up(T_1, A, P_1)} \prec get\text{-}to(T_1, B) \prec \boxed{drop(T_1, B, P_1)}$

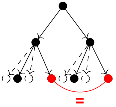
$get\text{-}to(T_2, C) \prec \boxed{pick\text{-}up(T_2, C, P_2)} \prec get\text{-}to(T_2, D) \prec \boxed{drop(T_2, D, P_2)}$

$$\pi = ()$$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n' ← n.apply(t)
            fringe.add(n')
        else
            for m ∈ t.methods do
                n' ← n.decompose(t, m)
                fringe.add(n')
```



Standard Progression Algorithm

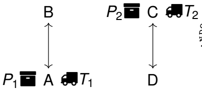


$get\text{-}to(T_1, A) \prec \boxed{pick\text{-}up(T_1, A, P_1)} \prec get\text{-}to(T_1, B) \prec \boxed{drop(T_1, B, P_1)}$

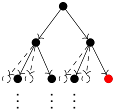
$get\text{-}to(T_2, C) \prec \boxed{pick\text{-}up(T_2, C, P_2)} \prec get\text{-}to(T_2, D) \prec \boxed{drop(T_2, D, P_2)}$

$$\pi = ()$$

```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



Standard Progression Algorithm

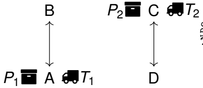


get-to$(T_1, A) \prec$ $\boxed{\textit{pick-up}(T_1, A, P_1)}$ $\prec$ *get-to*$(T_1, B) \prec$ $\boxed{\textit{drop}(T_1, B, P_1)}$

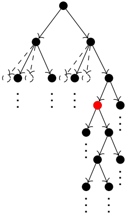
get-to$(T_2, C) \prec$ $\boxed{\textit{pick-up}(T_2, C, P_2)}$ $\prec$ *get-to*$(T_2, D) \prec$ $\boxed{\textit{drop}(T_2, D, P_2)}$

$\pi = ()$

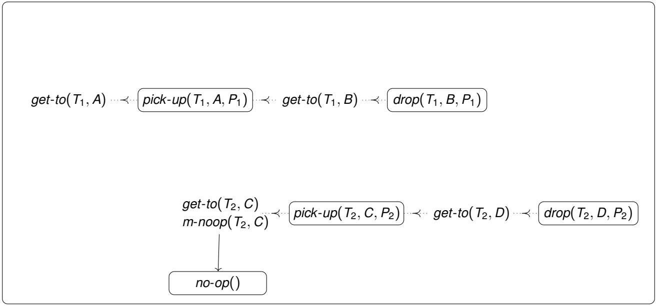
```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



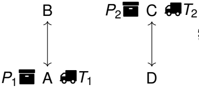
Standard Progression Algorithm



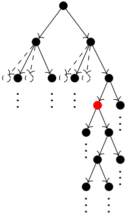
```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



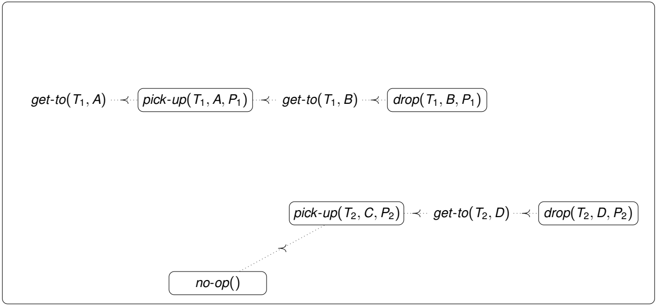
$$\pi = ()$$



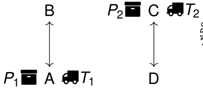
Standard Progression Algorithm



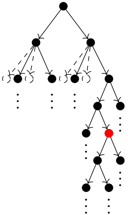
```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n' ← n.apply(t)
            fringe.add(n')
        else
            for m ∈ t.methods do
                n' ← n.decompose(t, m)
                fringe.add(n')
```



$$\pi = ()$$



Standard Progression Algorithm

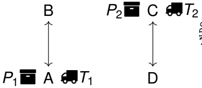


```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```

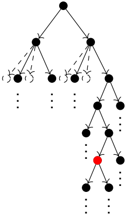
$\textit{get-to}(T_1, A) \prec \boxed{\textit{pick-up}(T_1, A, P_1)} \prec \textit{get-to}(T_1, B) \prec \boxed{\textit{drop}(T_1, B, P_1)}$

$\boxed{\textit{pick-up}(T_2, C, P_2)} \prec \textit{get-to}(T_2, D) \prec \boxed{\textit{drop}(T_2, D, P_2)}$

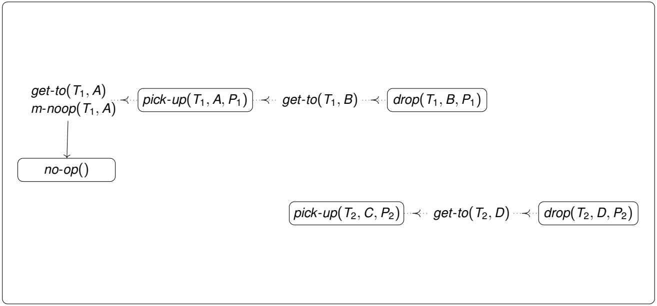
$\pi = (\textit{no-op}())$



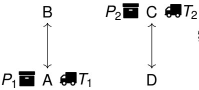
Standard Progression Algorithm



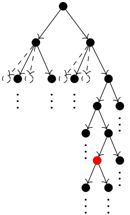
$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $U \leftarrow n.unconstrainedNodes$
  for $t \in U$ **do**
    if $isPrimitive(t)$ **then**
      $n' \leftarrow n.apply(t)$
      $fringe.add(n')$
    else
      for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$



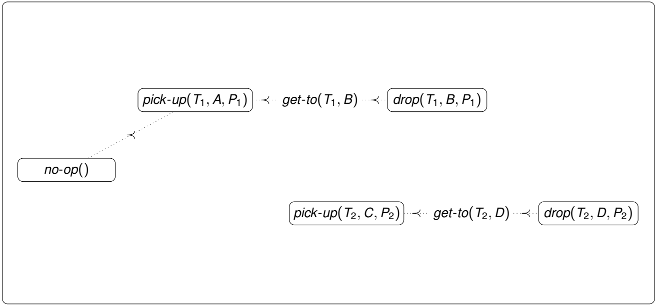
$\pi = (no\text{-}op())$



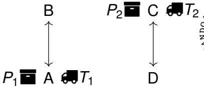
Standard Progression Algorithm



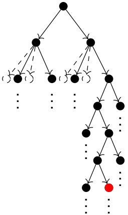
```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n' ← n.apply(t)
            fringe.add(n')
        else
            for m ∈ t.methods do
                n' ← n.decompose(t, m)
                fringe.add(n')
```



$$\pi = (\textit{no-op}())$$



Standard Progression Algorithm

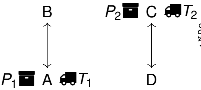


$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $U \leftarrow n.unconstrainedNodes$
  for $t \in U$ **do**
    if $isPrimitive(t)$ **then**
      $n' \leftarrow n.apply(t)$
      $fringe.add(n')$
    else
      for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$

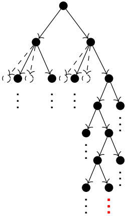
$\boxed{pick\text{-}up(T_1, A, P_1)} \prec get\text{-}to(T_1, B) \prec \boxed{drop(T_1, B, P_1)}$

$\boxed{pick\text{-}up(T_2, C, P_2)} \prec get\text{-}to(T_2, D) \prec \boxed{drop(T_2, D, P_2)}$

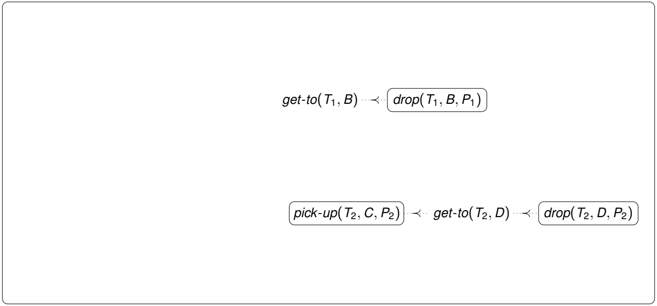
$$\pi = (no\text{-}op(), no\text{-}op())$$



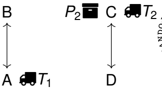
Standard Progression Algorithm



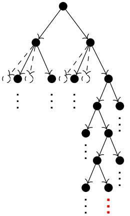
```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n' ← n.apply(t)
            fringe.add(n')
        else
            for m ∈ t.methods do
                n' ← n.decompose(t, m)
                fringe.add(n')
```



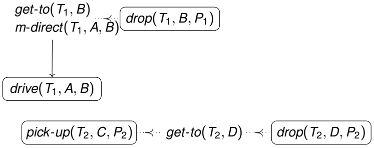
$$\pi = (\textit{no-op}(), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1))$$



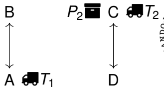
Standard Progression Algorithm



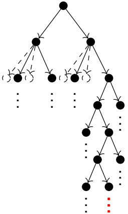
$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
    $n \leftarrow fringe.poll()$
    if $n.isgoal$ **then return** n
    $U \leftarrow n.unconstrainedNodes$
    for $t \in U$ **do**
        if $isPrimitive(t)$ **then**
            $n' \leftarrow n.apply(t)$
            $fringe.add(n')$
        else
            for $m \in t.methods$ **do**
                $n' \leftarrow n.decompose(t, m)$
                $fringe.add(n')$



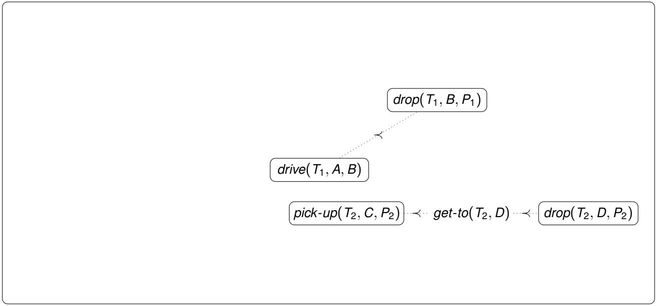
$$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1))$$



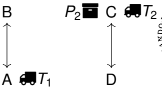
Standard Progression Algorithm



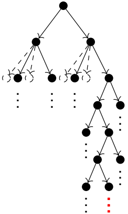
```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n' ← n.apply(t)
            fringe.add(n')
        else
            for m ∈ t.methods do
                n' ← n.decompose(t, m)
                fringe.add(n')
```



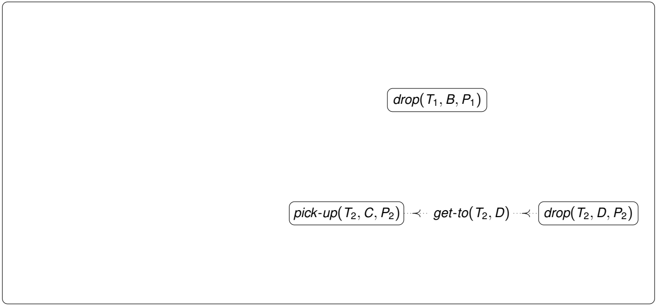
$$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1))$$



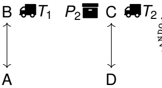
Standard Progression Algorithm



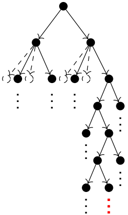
```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



$$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drive(T_1, A, B))$$

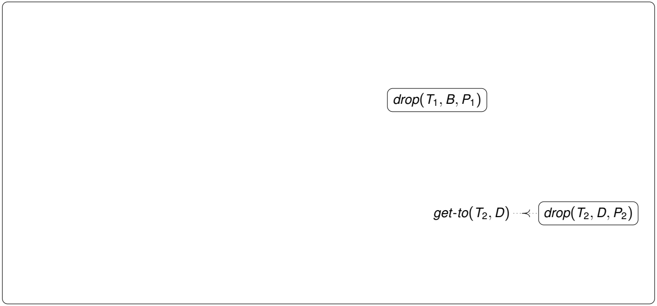


Standard Progression Algorithm

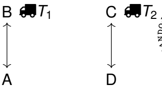


while *fringe* $\neq \emptyset$ **do**
- $n \leftarrow \mathit{fringe.poll}()$
- **if** *n.isgoal* **then return** n
- $U \leftarrow \mathit{n.unconstrainedNodes}$
- **for** $t \in U$ **do**
  - **if** $\mathit{isPrimitive}(t)$ **then**
    - $n' \leftarrow \mathit{n.apply}(t)$
    - $\mathit{fringe.add}(n')$
  - **else**
    - **for** $m \in \mathit{t.methods}$ **do**
      - $n' \leftarrow \mathit{n.decompose}(t, m)$
      - $\mathit{fringe.add}(n')$

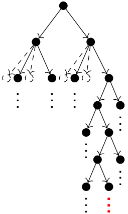
$\mathit{fringe} \leftarrow \{(s_I, tn_I, ())\}$



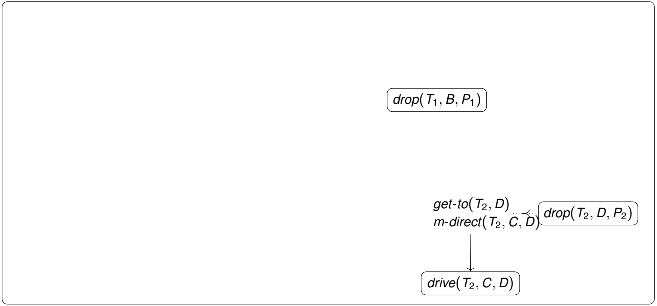
$$\pi = (\mathit{no\text{-}op}(), \mathit{no\text{-}op}(), \mathit{pick\text{-}up}(T_1, A, P_1), \mathit{drive}(T_1, A, B), \mathit{pick\text{-}up}(T_2, C, P_2))$$



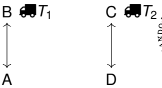
Standard Progression Algorithm



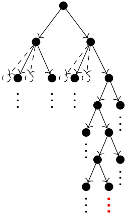
fringe $\leftarrow \{(s_I, tn_I, ())\}$
while *fringe* $\neq \emptyset$ **do**
  $n \leftarrow$ *fringe.poll*()
  if *n.isgoal* **then return** *n*
  $U \leftarrow$ *n.unconstrainedNodes*
  for $t \in U$ **do**
    if *isPrimitive*(t) **then**
      $n' \leftarrow$ *n.apply*(t)
      fringe.add(n')
    else
      for $m \in$ *t.methods* **do**
        $n' \leftarrow$ *n.decompose*(t, m)
        fringe.add(n')



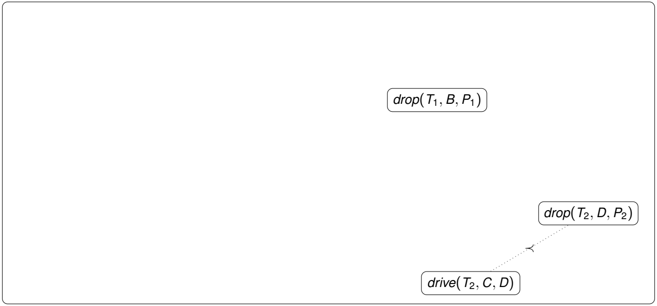
$$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drive(T_1, A, B), pick\text{-}up(T_2, C, P_2))$$



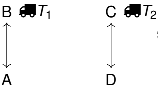
Standard Progression Algorithm



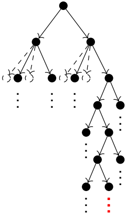
```
fringe ← {(s_I, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    U ← n.unconstrainedNodes
    for t ∈ U do
        if isPrimitive(t) then
            n′ ← n.apply(t)
            fringe.add(n′)
        else
            for m ∈ t.methods do
                n′ ← n.decompose(t, m)
                fringe.add(n′)
```



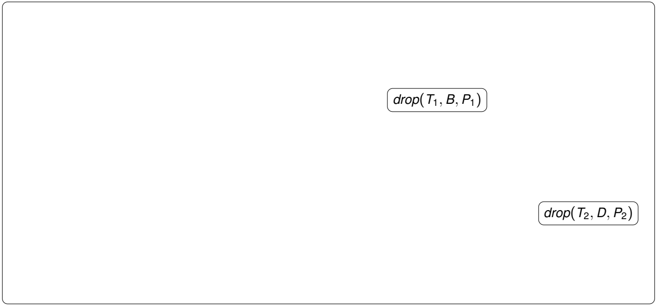
$\pi = (\textit{no-op}(), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1), \textit{drive}(T_1, A, B), \textit{pick-up}(T_2, C, P_2))$



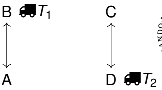
Standard Progression Algorithm



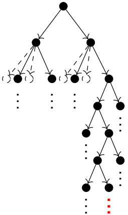
$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $U \leftarrow n.unconstrainedNodes$
  for $t \in U$ **do**
    if $isPrimitive(t)$ **then**
      $n' \leftarrow n.apply(t)$
      $fringe.add(n')$
    else
      for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$



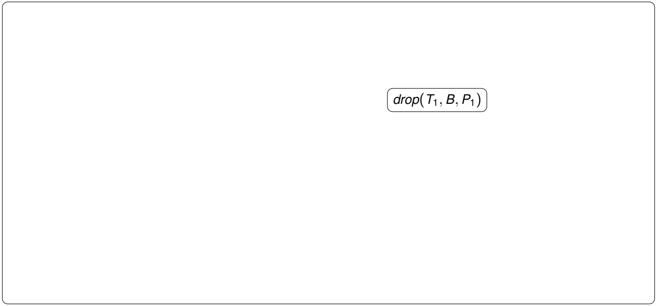
$$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drive(T_1, A, B), pick\text{-}up(T_2, C, P_2), drive(T_2, C, D))$$



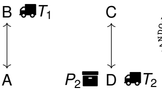
Standard Progression Algorithm



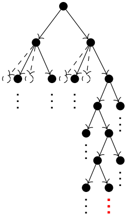
$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $U \leftarrow n.unconstrainedNodes$
  for $t \in U$ **do**
    if $isPrimitive(t)$ **then**
      $n' \leftarrow n.apply(t)$
      $fringe.add(n')$
    else
      for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$



$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drive(T_1, A, B), pick\text{-}up(T_2, C, P_2), drive(T_2, C, D), drop(T_2, D, P_2))$

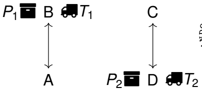


Standard Progression Algorithm



$fringe \leftarrow \{(s_I, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $U \leftarrow n.unconstrainedNodes$
  for $t \in U$ **do**
    if $isPrimitive(t)$ **then**
      $n' \leftarrow n.apply(t)$
      $fringe.add(n')$
    else
      for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$

$\pi = (\textit{no-op}(), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1), \textit{drive}(T_1, A, B), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{drop}(T_2, D, P_2), \textit{drop}(T_1, B, P_1))$



Progression Search – Properties

- Progression Search is **sound** . . .
- . . . and **complete**

Progression Search – Properties

- Progression Search is **sound** . . .
- . . . and **complete**
- It maintains the **current state** during search
- This has been used to control search via state-based **preconditions** for **methods**

Progression Search – Properties

- Progression Search is **sound** . . .
- . . . and **complete**
- It maintains the **current state** during search
- This has been used to control search via state-based **preconditions** for **methods**
- It is also useful for calculating **heuristics**

Improving Progression Search

- Observation: In partially ordered models, standard progression search searches parts of the search space more than once

Improving Progression Search

- Observation: In partially ordered models, standard progression search searches parts of the search space more than once
- This is due to branching (i.e. a non-deterministic choice) over unconstrained **compound** tasks
- When processing actions, the algorithm **commits** to an ordering in the solution
- The decision which **task is decomposed** implies **no** commitment to the solution
- The decision which **method is used** implies commitment to the solution

Improving Progression Search

- Observation: In partially ordered models, standard progression search searches parts of the search space more than once
- This is due to branching (i.e. a non-deterministic choice) over unconstrained **compound** tasks
- When processing actions, the algorithm **commits** to an ordering in the solution
- The decision which **task is decomposed** implies **no** commitment to the solution
- The decision which **method is used** implies commitment to the solution

→ For selection of the compound task, no branching is needed, we can simply "pick" one and decompose it

→ The decision which method is used must be made via branching

Improved Progression Algorithm

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```

- **Unconstrained tasks** are split into compound and primitive tasks

Improved Progression Algorithm

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```

- **Unconstrained tasks** are split into compound and primitive tasks
- **Action application** is done via branching

Improved Progression Algorithm

$\textit{fringe} \leftarrow \{(s_0, tn_I, ())\}$
while $\textit{fringe} \neq \emptyset$ **do**
    $n \leftarrow \textit{fringe.poll}()$
    if $\textit{n.isgoal}$ **then return** n
    $(U_C, U_P) \leftarrow \textit{n.unconstrainedNodes}$
    for $t \in U_P$ **do**
        $n' \leftarrow \textit{n.apply}(t)$
        $\textit{fringe.add}(n')$
    $t \leftarrow \textit{selectAbstractTask}(U_C)$
    for $m \in t.\textit{methods}$ **do**
        $n' \leftarrow \textit{n.decompose}(t, m)$
        $\textit{fringe.add}(n')$

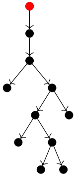
- **Unconstrained tasks** are split into compound and primitive tasks
- **Action application** is done via branching
- Only **one compound task** is processed

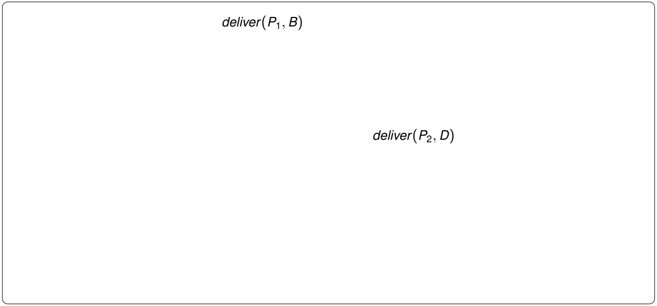
Improved Progression Algorithm

$fringe \leftarrow \{(s_0, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
    $n \leftarrow fringe.poll()$
    if $n.isgoal$ **then return** n
    $(U_C, U_P) \leftarrow n.unconstrainedNodes$
    for $t \in U_P$ **do**
        $n' \leftarrow n.apply(t)$
        $fringe.add(n')$
    $t \leftarrow selectAbstractTask(U_C)$
    for $m \in t.methods$ **do**
        $n' \leftarrow n.decompose(t, m)$
        $fringe.add(n')$

- **Unconstrained tasks** are split into compound and primitive tasks
- **Action application** is done via branching
- Only **one compound task** is processed
- **Method application** is done via branching

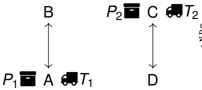
Improved Progression Algorithm



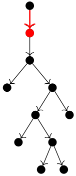


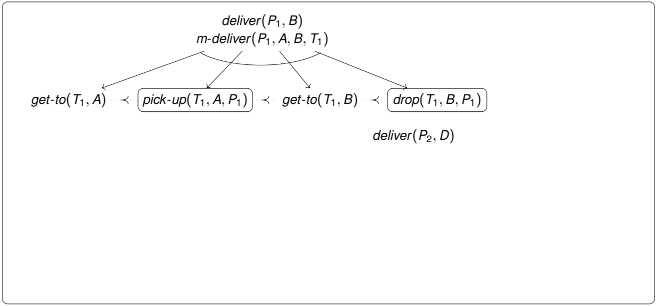
$\pi = ()$

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```



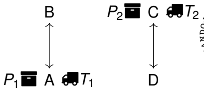
Improved Progression Algorithm



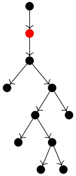


$$\pi = ()$$

$fringe \leftarrow \{(s_0, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $(U_C, U_P) \leftarrow n.unconstrainedNodes$
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    $fringe.add(n')$
  $t \leftarrow selectAbstractTask(U_C)$
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    $fringe.add(n')$



Improved Progression Algorithm

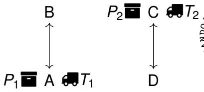


$$get\text{-}to(T_1, A) \prec \boxed{pick\text{-}up(T_1, A, P_1)} \prec get\text{-}to(T_1, B) \prec \boxed{drop(T_1, B, P_1)}$$

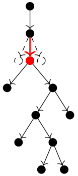
$$deliver(P_2, D)$$

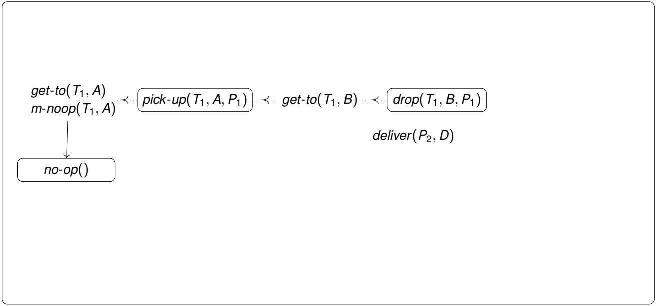
$\pi = ()$

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```



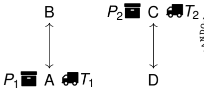
Improved Progression Algorithm



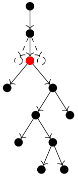


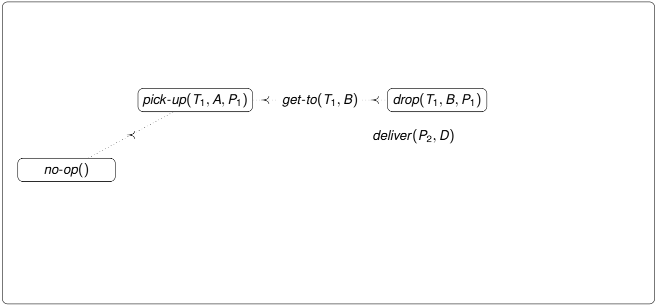
$$\pi = ()$$

$fringe \leftarrow \{(s_0, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $(U_C, U_P) \leftarrow n.unconstrainedNodes$
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    $fringe.add(n')$
  $t \leftarrow selectAbstractTask(U_C)$
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    $fringe.add(n')$



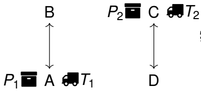
Improved Progression Algorithm



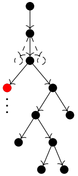


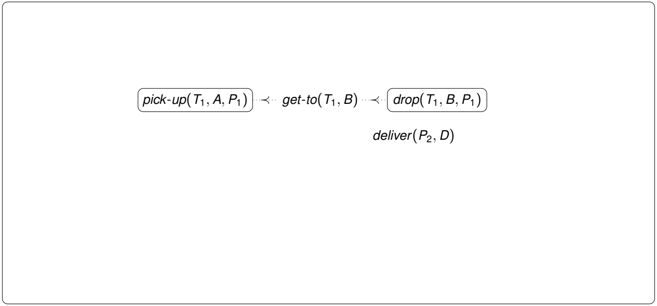
$\pi = ()$

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```



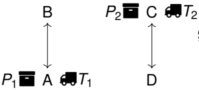
Improved Progression Algorithm



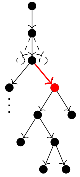


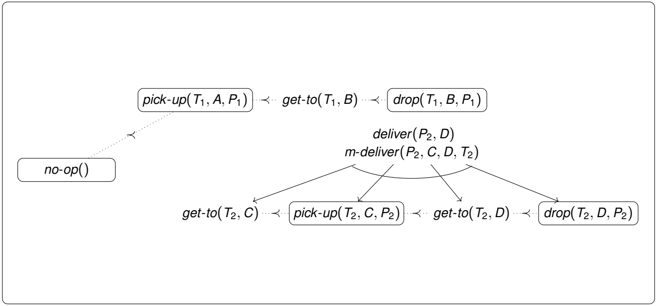
$\pi = (\textit{no-op}())$

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n′ ← n.apply(t)
        fringe.add(n′)
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n′ ← n.decompose(t, m)
        fringe.add(n′)
```



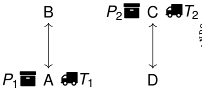
Improved Progression Algorithm



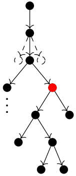


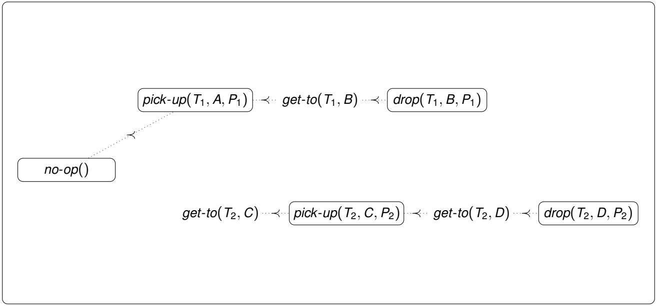
$$\pi = ()$$

fringe $\leftarrow \{(s_0, tn_I, ())\}$
while *fringe* $\neq \emptyset$ **do**
  $n \leftarrow$ *fringe.poll*()
  if *n.isgoal* **then return** n
  $(U_C, U_P) \leftarrow$ *n.unconstrainedNodes*
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    fringe.add(n')
  $t \leftarrow$ *selectAbstractTask*(U_C)
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    fringe.add(n')



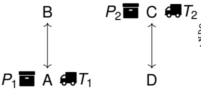
Improved Progression Algorithm



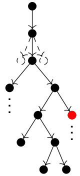


$\pi = ()$

fringe $\leftarrow \{(s_0, tn_I, ())\}$
while *fringe* $\neq \emptyset$ **do**
  $n \leftarrow$ *fringe.poll*()
  if *n.isgoal* **then return** n
  $(U_C, U_P) \leftarrow$ *n.unconstrainedNodes*
  for $t \in U_P$ **do**
    $n' \leftarrow$ *n.apply*(t)
    fringe.add(n')
  $t \leftarrow$ *selectAbstractTask*(U_C)
  for $m \in t.methods$ **do**
    $n' \leftarrow$ *n.decompose*(t, m)
    fringe.add(n')



Improved Progression Algorithm

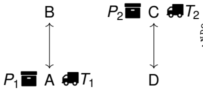


$\boxed{pick\text{-}up(T_1, A, P_1)} \prec get\text{-}to(T_1, B) \prec \boxed{drop(T_1, B, P_1)}$

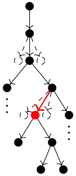
$get\text{-}to(T_2, C) \prec \boxed{pick\text{-}up(T_2, C, P_2)} \prec get\text{-}to(T_2, D) \prec \boxed{drop(T_2, D, P_2)}$

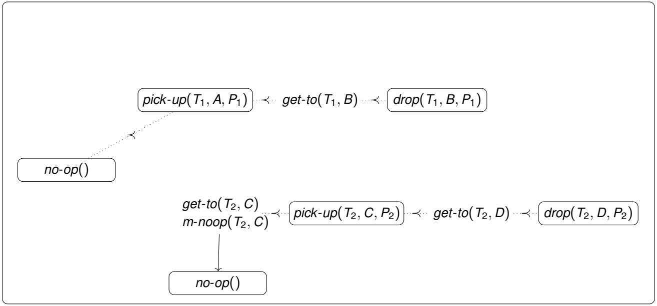
$\pi = (no\text{-}op())$

$fringe \leftarrow \{(s_0, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $(U_C, U_P) \leftarrow n.unconstrainedNodes$
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    $fringe.add(n')$
  $t \leftarrow selectAbstractTask(U_C)$
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    $fringe.add(n')$



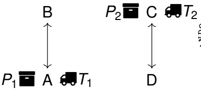
Improved Progression Algorithm



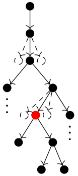


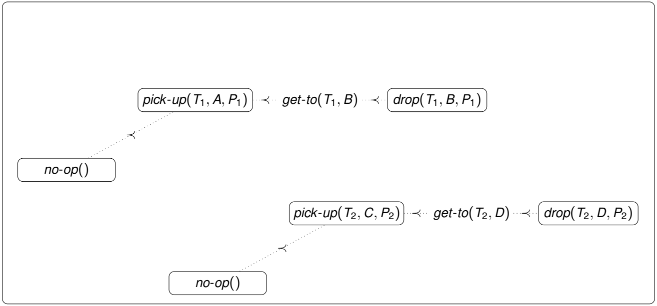
$\pi = ()$

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```



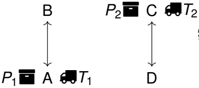
Improved Progression Algorithm



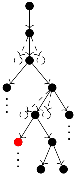


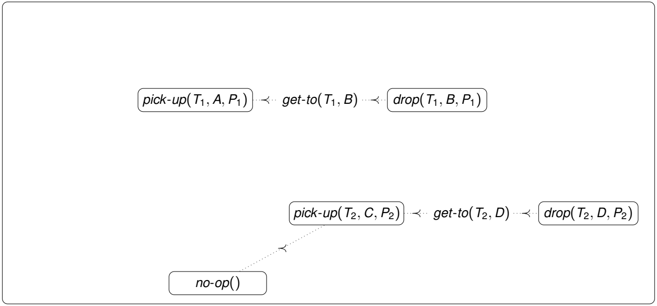
$\pi = ()$

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes

    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')

    t ← selectAbstractTask(U_C)

    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```



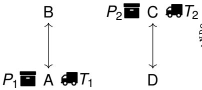
Improved Progression Algorithm



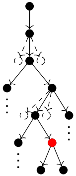


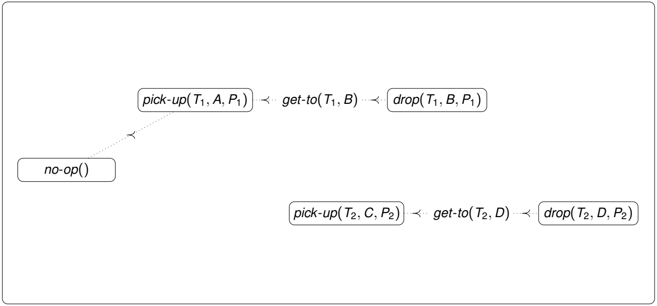
$\pi = (\mathit{no\text{-}op}())$

$\mathit{fringe} \leftarrow \{(s_0, tn_I, ())\}$
while $\mathit{fringe} \neq \emptyset$ **do**
  $n \leftarrow \mathit{fringe.poll}()$
  if $n.\mathit{isgoal}$ **then return** n
  $(U_C, U_P) \leftarrow n.\mathit{unconstrainedNodes}$
  for $t \in U_P$ **do**
    $n' \leftarrow n.\mathit{apply}(t)$
    $\mathit{fringe.add}(n')$
  $t \leftarrow \mathit{selectAbstractTask}(U_C)$
  for $m \in t.\mathit{methods}$ **do**
    $n' \leftarrow n.\mathit{decompose}(t, m)$
    $\mathit{fringe.add}(n')$



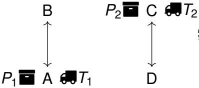
Improved Progression Algorithm



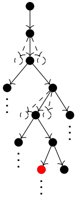


$\pi = (\textit{no-op}())$

$\textit{fringe} \leftarrow \{(s_0, tn_I, ())\}$
while $\textit{fringe} \neq \emptyset$ **do**
  $n \leftarrow \textit{fringe.poll}()$
  if $\textit{n.isgoal}$ **then return** n
  $(U_C, U_P) \leftarrow \textit{n.unconstrainedNodes}$
  for $t \in U_P$ **do**
    $n' \leftarrow \textit{n.apply}(t)$
    $\textit{fringe.add}(n')$
  $t \leftarrow \textit{selectAbstractTask}(U_C)$
  for $m \in \textit{t.methods}$ **do**
    $n' \leftarrow \textit{n.decompose}(t, m)$
    $\textit{fringe.add}(n')$



Improved Progression Algorithm

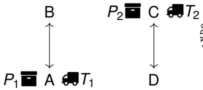


$pick\text{-}up(T_1, A, P_1) \prec get\text{-}to(T_1, B) \prec drop(T_1, B, P_1)$

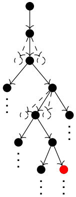
$pick\text{-}up(T_2, C, P_2) \prec get\text{-}to(T_2, D) \prec drop(T_2, D, P_2)$

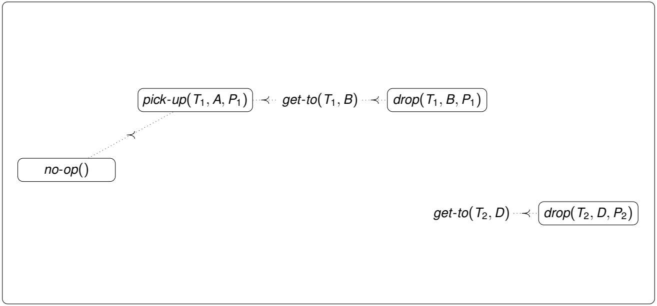
$$\pi = (no\text{-}op(), no\text{-}op())$$

$fringe \leftarrow \{(s_0, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $(U_C, U_P) \leftarrow n.unconstrainedNodes$
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    $fringe.add(n')$
  $t \leftarrow selectAbstractTask(U_C)$
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    $fringe.add(n')$



Improved Progression Algorithm

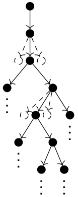


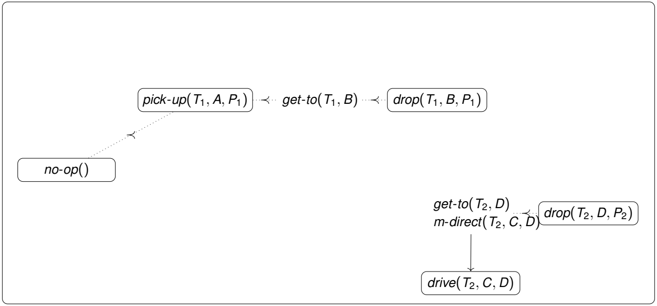


$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2))$$

$\textit{fringe} \leftarrow \{(s_0, tn_I, ())\}$
while $\textit{fringe} \neq \emptyset$ **do**
  $n \leftarrow \textit{fringe.poll}()$
  if $n.\textit{isgoal}$ **then return** n
  $(U_C, U_P) \leftarrow n.\textit{unconstrainedNodes}$
  for $t \in U_P$ **do**
    $n' \leftarrow n.\textit{apply}(t)$
    $\textit{fringe.add}(n')$
  $t \leftarrow \textit{selectAbstractTask}(U_C)$
  for $m \in t.\textit{methods}$ **do**
    $n' \leftarrow n.\textit{decompose}(t, m)$
    $\textit{fringe.add}(n')$

Improved Progression Algorithm

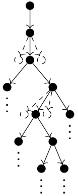


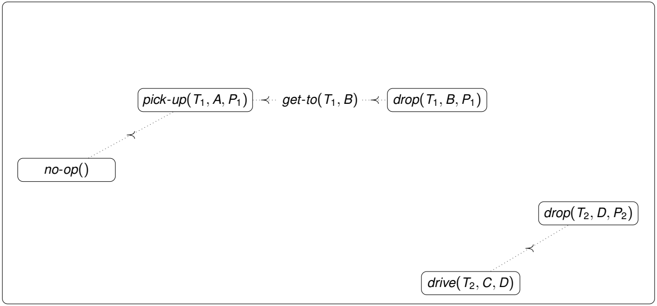



```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```

$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2))$$

Improved Progression Algorithm

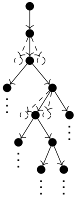


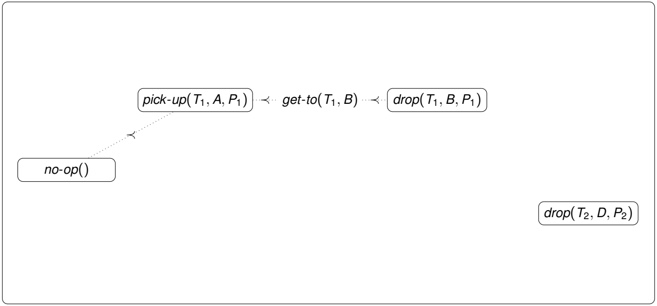


$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2))$$

$\textit{fringe} \leftarrow \{(s_0, tn_I, ())\}$
while $\textit{fringe} \neq \emptyset$ **do**
  $n \leftarrow \textit{fringe.poll}()$
  if $n.\textit{isgoal}$ **then return** n
  $(U_C, U_P) \leftarrow n.\textit{unconstrainedNodes}$
  for $t \in U_P$ **do**
    $n' \leftarrow n.\textit{apply}(t)$
    $\textit{fringe.add}(n')$
  $t \leftarrow \textit{selectAbstractTask}(U_C)$
  for $m \in t.\textit{methods}$ **do**
    $n' \leftarrow n.\textit{decompose}(t, m)$
    $\textit{fringe.add}(n')$

Improved Progression Algorithm

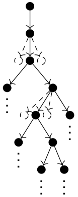


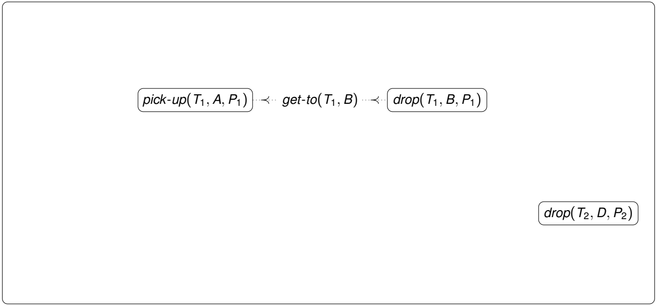


$fringe \leftarrow \{(s_0, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $(U_C, U_P) \leftarrow n.unconstrainedNodes$
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    $fringe.add(n')$
  $t \leftarrow selectAbstractTask(U_C)$
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    $fringe.add(n')$

$$\pi = (no\text{-}op(), pick\text{-}up(T_2, C, P_2), drive(T_2, C, D))$$

Improved Progression Algorithm

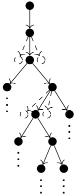




$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}())$$

$\textit{fringe} \leftarrow \{(s_0, tn_I, ())\}$
while $\textit{fringe} \neq \emptyset$ **do**
  $n \leftarrow \textit{fringe.poll}()$
  if $\textit{n.isgoal}$ **then return** n
  $(U_C, U_P) \leftarrow \textit{n.unconstrainedNodes}$
  for $t \in U_P$ **do**
    $n' \leftarrow \textit{n.apply}(t)$
    $\textit{fringe.add}(n')$
  $t \leftarrow \textit{selectAbstractTask}(U_C)$
  for $m \in t.\textit{methods}$ **do**
    $n' \leftarrow \textit{n.decompose}(t, m)$
    $\textit{fringe.add}(n')$

Improved Progression Algorithm

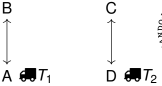


$$\text{get-to}(T_1, B) \prec \boxed{\text{drop}(T_1, B, P_1)}$$

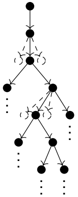
$$\boxed{\text{drop}(T_2, D, P_2)}$$

```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n′ ← n.apply(t)
        fringe.add(n′)
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n′ ← n.decompose(t, m)
        fringe.add(n′)
```

$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1))$$



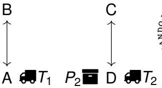
Improved Progression Algorithm



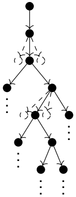
get-to$(T_1, B) \prec$ $\boxed{drop(T_1, B, P_1)}$

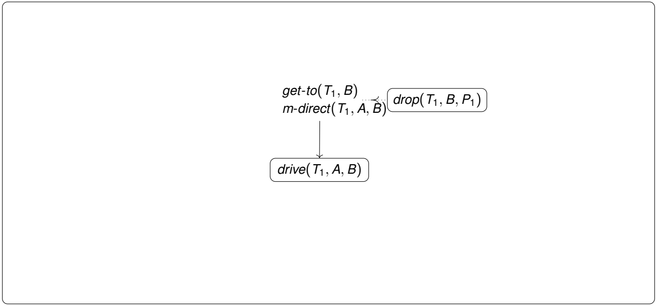
$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1), \textit{drop}(T_2, D, P_2))$$

fringe $\leftarrow \{(s_0, tn_I, ())\}$
while *fringe* $\neq \emptyset$ **do**
    $n \leftarrow$ *fringe.poll*()
    if *n.isgoal* **then return** *n*
    $(U_C, U_P) \leftarrow$ *n.unconstrainedNodes*
    for $t \in U_P$ **do**
        $n' \leftarrow$ *n.apply*(t)
        fringe.add(n')
    $t \leftarrow$ *selectAbstractTask*(U_C)
    for $m \in t.methods$ **do**
        $n' \leftarrow$ *n.decompose*(t, m)
        fringe.add(n')



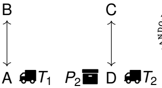
Improved Progression Algorithm



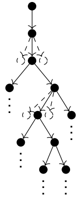


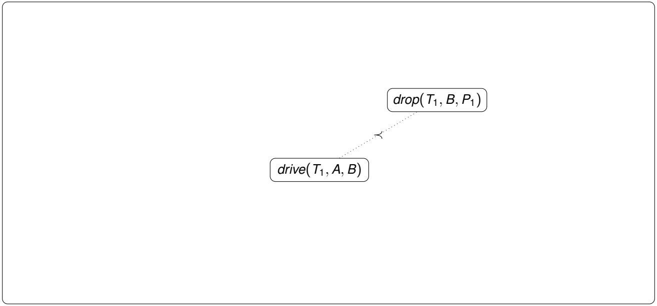
$fringe \leftarrow \{(s_0, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $(U_C, U_P) \leftarrow n.unconstrainedNodes$
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    $fringe.add(n')$
  $t \leftarrow selectAbstractTask(U_C)$
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    $fringe.add(n')$

$\pi = (no\text{-}op(), pick\text{-}up(T_2, C, P_2), drive(T_2, C, D), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drop(T_2, D, P_2))$



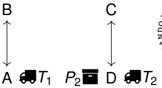
Improved Progression Algorithm



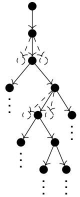


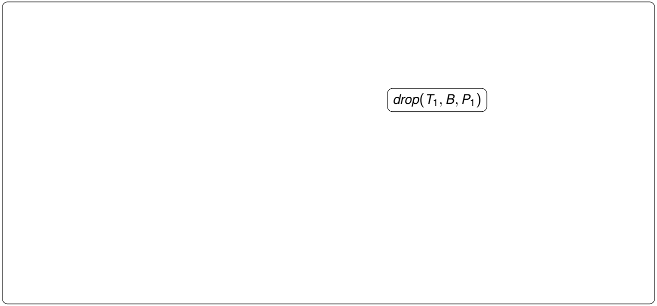
fringe $\leftarrow \{(s_0, tn_I, ())\}$
while *fringe* $\neq \emptyset$ **do**
  $n \leftarrow$ *fringe.poll*()
  if *n.isgoal* **then return** n
  $(U_C, U_P) \leftarrow$ *n.unconstrainedNodes*
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    fringe.add(n')
  $t \leftarrow$ *selectAbstractTask*(U_C)
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    fringe.add(n')

$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1), \textit{drop}(T_2, D, P_2))$$



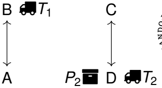
Improved Progression Algorithm



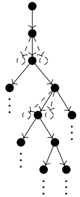



```
fringe ← {(s_0, tn_I, ())}
while fringe ≠ ∅ do
    n ← fringe.poll()
    if n.isgoal then return n
    (U_C, U_P) ← n.unconstrainedNodes
    for t ∈ U_P do
        n' ← n.apply(t)
        fringe.add(n')
    t ← selectAbstractTask(U_C)
    for m ∈ t.methods do
        n' ← n.decompose(t, m)
        fringe.add(n')
```

$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1), \textit{drop}(T_2, D, P_2), \textit{drive}(T_1, A, B))$$

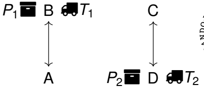


Improved Progression Algorithm



$fringe \leftarrow \{(s_0, tn_I, ())\}$
while $fringe \neq \emptyset$ **do**
  $n \leftarrow fringe.poll()$
  if $n.isgoal$ **then return** n
  $(U_C, U_P) \leftarrow n.unconstrainedNodes$
  for $t \in U_P$ **do**
    $n' \leftarrow n.apply(t)$
    $fringe.add(n')$
  $t \leftarrow selectAbstractTask(U_C)$
  for $m \in t.methods$ **do**
    $n' \leftarrow n.decompose(t, m)$
    $fringe.add(n')$

$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1), \textit{drop}(T_2, D, P_2), \textit{drive}(T_1, A, B), \textit{drop}(T_1, B, P_1)))$$



Improved Progression Search – Properties

- Improved version of progression search is still **sound** and **complete**

Improved Progression Search – Properties

- Improved version of progression search is still **sound** and **complete**
- Searching the search space more than once is **avoided** (to a certain extend) but still **possible**

Improved Progression Search – Properties

- Improved version of progression search is still **sound** and **complete**
- Searching the search space more than once is **avoided** (to a certain extend) but still **possible**
- It may **increase the progression bound** necessary to solve the problem (problematic for some planning systems)

Overview Part II

Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search
- **Compilation-based Approaches**
  - **Compilations to STRIPS/ADL**
  - Compilations to SAT
- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms

Translating HTN Problems to STRIPS/ADL

The basic idea is quite simple:

- **Translate** the input (HTN) problem in to a classical planning problem
- Use a **classical planning system** to solve it
- Compile classical solution back to one for the HTN problem

Translating HTN Problems to STRIPS/ADL

The basic idea is quite simple:

- **Translate** the input (HTN) problem in to a classical planning problem
- Use a **classical planning system** to solve it
- Compile classical solution back to one for the HTN problem

Approach:

- Add a **new part to the state** that represents the current task network

Translating HTN Problems to STRIPS/ADL

The basic idea is quite simple:

- **Translate** the input (HTN) problem in to a classical planning problem
- Use a **classical planning system** to solve it
- Compile classical solution back to one for the HTN problem

Approach:

- Add a **new part to the state** that represents the current task network
- **Simulate a progression search** on this part of the state
  - Adapt original actions with respect to applicability and to maintain the new state features
  - Add actions that simulate decomposition methods

HTN to STRIPS/ADL – Example

$\boxed{\textit{pick-up}(T_1, A, P_1)} \prec \textit{get-to}(T_1, B) \prec \boxed{\textit{drop}(T_1, B, P_1)}$

$\textit{deliver}(P_2, D)$

- Introduce id variables: $t_0, t_1, \ldots, t_b$
- Introduce new predicate for every task

HTN to STRIPS/ADL – Example

$\boxed{\textit{pick-up}(T_1, A, P_1)} \prec \textit{get-to}(T_1, B) \prec \boxed{\textit{drop}(T_1, B, P_1)}$

$\textit{deliver}(P_2, D)$

tasks: $\textit{ppick-up}(T_1, A, P_1, t_2)$,
$\textit{pget-to}(T_1, B, t_3)$,
$\textit{pdrop}(T_1, B, P_1, t_5)$,
$\textit{pdeliver}(P_2, D, t_4)$,
orderings: $\textit{before}(t_2, t_3), \textit{before}(t_3, t_5)$

- Introduce id variables: $t_0, t_1, \ldots, t_b$
- Introduce new predicate for every task, represent current *tn* in the state

HTN to STRIPS/ADL – Example

$\boxed{\textit{pick-up}(T_1, A, P_1)} \prec \textit{get-to}(T_1, B) \prec \boxed{\textit{drop}(T_1, B, P_1)}$

$\textit{deliver}(P_2, D)$

$\textit{ppick-up}(T_1, A, P_1, t_2)$,
$\textit{pget-to}(T_1, B, t_3)$,
$\textit{pdrop}(T_1, B, P_1, t_5)$,
$\textit{pdeliver}(P_2, D, t_4)$,
$\textit{before}(t_2, t_3), \textit{before}(t_3, t_5)$

$\textit{pick-up}(T_1, A, P_1, t_2)$

pre :precs from domain,
$\textit{ppick-up}(T_1, A, P_1, t_2)$,
$\forall t_i \in \{t_0 \ldots t_b\} : \neg \textit{before}(t_i, t_2)$

eff :effects from domain,
$\neg \textit{ppick-up}(T_1, A, P_1, t_2), \textit{free}(t_2)$
$\forall t_i \in \{t_0 \ldots t_b\} : \neg \textit{before}(t_2, t_i)$

- Introduce id variables: $t_0, t_1, \ldots, t_b$
- Introduce new predicate for every task, represent current *tn* in the state
- Modify existing actions

HTN to STRIPS/ADL – Example

$\boxed{\textit{pick-up}(T_1, A, P_1)} \prec \textit{get-to}(T_1, B) \prec \boxed{\textit{drop}(T_1, B, P_1)}$

$\textit{deliver}(P_2, D)$

$\textit{pick-up}(T_1, A, P_1, t_2)$

pre :precs from domain,

$\textit{ppick-up}(T_1, A, P_1, t_2)$,

$\forall t_i \in \{t_0 \ldots t_b\} : \neg \textit{before}(t_i, t_2)$

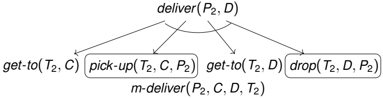
eff :effects from domain,

$\neg \textit{ppick-up}(T_1, A, P_1, t_2), \textit{free}(t_2)$

$\forall t_i \in \{t_0 \ldots t_b\} : \neg \textit{before}(t_2, t_i)$

- Introduce id variables: $t_0, t_1, \ldots, t_b$
- Introduce new predicate for every task, represent current *tn* in the state
- Modify existing actions
- Add new actions simulating methods

$\textit{ppick-up}(T_1, A, P_1, t_2)$,
$\textit{pget-to}(T_1, B, t_3)$,
$\textit{pdrop}(T_1, B, P_1, t_5)$,
$\textit{pdeliver}(P_2, D, t_4)$,
$\textit{before}(t_2, t_3), \textit{before}(t_3, t_5)$



$\textit{m-deliver}(P_2, C, D, T_2, t_4, t_1, t_6, t_7)$

pre :$\textit{pdeliver}(P_2, D, t_4)$

$\forall t_i \in \{t_0 \ldots t_b\} : \neg \textit{before}(t_i, t_4)$

$\textit{free}(t_1), \textit{free}(t_6), \textit{free}(t_7)$

eff :$\neg \textit{pdeliver}(P_2, D, t_4)$

$\textit{pget-to}(T_2, C, t_1)$

$\textit{ppick-up}(T_2, C, P_2, t_6)$

$\textit{pget-to}(T_2, D, t_7)$

$\textit{pdrop}(T_2, D, P_2, t_4)$

$\textit{before}(t_1, t_6), \ldots \neg \textit{free}(t_1), \ldots$

Translating HTN Problems to STRIPS/ADL

Benefits:

- Sophisticated planning system(s) available
- Large portfolio of heuristics available

Translating HTN Problems to STRIPS/ADL

Benefits:

- Sophisticated planning system(s) available
- Large portfolio of heuristics available

Challenges:

- How to represent the task network? (example was simplified)
  - To get a compact state
  - To get a small set of actions
  - To break symmetry
  - To preserve information when using available classical heuristics (e.g. delete-relaxation)

Translating HTN Problems to STRIPS/ADL

Benefits:

- Sophisticated planning system(s) available
- Large portfolio of heuristics available

Challenges:

- How to represent the task network? (example was simplified)
  - To get a compact state
  - To get a small set of actions
  - To break symmetry
  - To preserve information when using available classical heuristics (e.g. delete-relaxation)
- How many ids are sufficient?
  - Only computable for subclasses of HTN planning problems

Translating HTN Problems to STRIPS/ADL

Benefits:

- Sophisticated planning system(s) available
- Large portfolio of heuristics available

Challenges:

- How to represent the task network? (example was simplified)
  - To get a compact state
  - To get a small set of actions
  - To break symmetry
  - To preserve information when using available classical heuristics (e.g. delete-relaxation)
- How many ids are sufficient?
  - Only computable for subclasses of HTN planning problems
  - Approach for general HTN planning problems:
    - Incrementally increase it like in SAT-based classical planning
    - But there is no upper bound, so only stop when a plan was found

Overview Part II

Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search
- **Compilation-based Approaches**
  - Compilations to STRIPS/ADL
  - **Compilations to SAT**
- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms

HTN to SAT Compilations

Basic idea:

- Translate HTN planning problem to a **propositional formula**
- Solve it with a standard **SAT solver**
- Formula represents solution to the HTN

HTN to SAT Compilations

Basic idea:

- Translate HTN planning problem to a **propositional formula**
- Solve it with a standard **SAT solver**
- Formula represents solution to the HTN

Similar to approach in classical planning:

- Encodings of **state transition** can be re-used
- Translation to **a series** of **increasing** problems (instead of a single one)

HTN to SAT Compilations

Basic idea:

- Translate HTN planning problem to a **propositional formula**
- Solve it with a standard **SAT solver**
- Formula represents solution to the HTN

Similar to approach in classical planning:

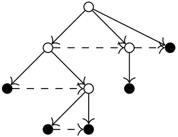
- Encodings of **state transition** can be re-used
- Translation to **a series** of **increasing** problems (instead of a single one)

Challenges:

- How to represent **decomposition**?
- What is the best way to **bound** the problem?

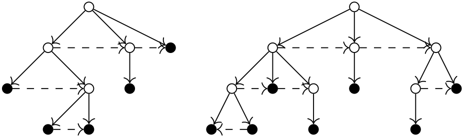
Representing Decomposition

- We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)



Representing Decomposition

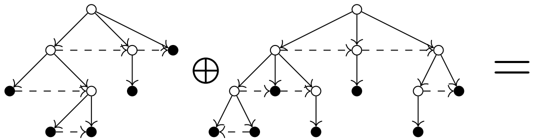
- We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)
- But: There are (double-exponentially) **many trees** for a single planning problem



Representing Decomposition

- We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)
- But: There are (double-exponentially) **many trees** for a single planning problem

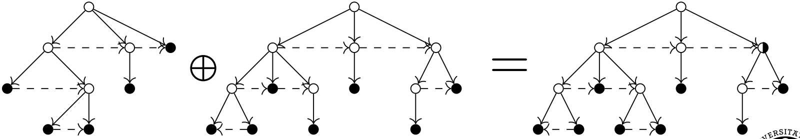
→ Compact representation of all possible decompositions of the initial task network: **Path Decomposition Trees** (PDTs)



Representing Decomposition

- We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)
- But: There are (double-exponentially) **many trees** for a single planning problem

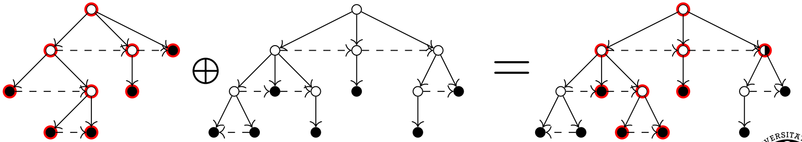
→ Compact representation of all possible decompositions of the initial task network: **Path Decomposition Trees** (PDTs)



Representing Decomposition

- We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)
- But: There are (double-exponentially) **many trees** for a single planning problem

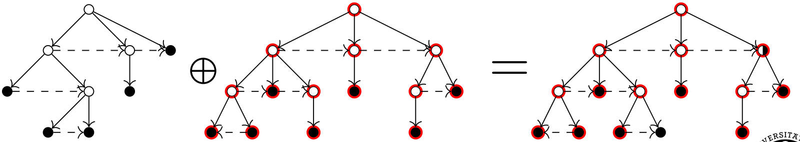
→ Compact representation of all possible decompositions of the initial task network: **Path Decomposition Trees** (PDTs)



Representing Decomposition

- We have already seen a structure to represent decomposition: **Decomposition Trees** (in the proof for TIHTN problems)
- But: There are (double-exponentially) **many trees** for a single planning problem

→ Compact representation of all possible decompositions of the initial task network: **Path Decomposition Trees** (PDTs)



Path Decomposition Trees

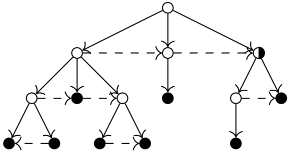
All Decomposition Trees can not be represented

Path Decomposition Trees

All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees

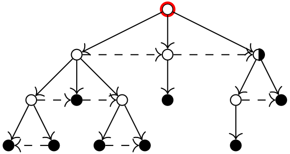
Path Decomposition Trees

All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees



Path Decomposition Trees

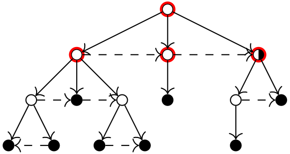
All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees



$c_l \rightarrow ABC$ and $c_l \rightarrow ACp$ and $c_l \rightarrow Ar$

Path Decomposition Trees

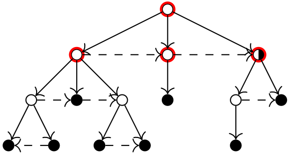
All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees



$$c_l \rightarrow ABC \text{ and } c_l \rightarrow ACp \text{ and } c_l \rightarrow Ar$$
$$\{A\} \quad \{B, C\} \quad \{C, p, r\}$$

Path Decomposition Trees

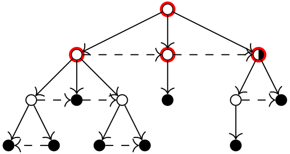
All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees



$c_l \rightarrow ABC$ and $c_l \rightarrow ACp$ and $c_l \rightarrow Ar$
$\{A\}$ $\{B, C\}$ $\{C, p, r\}$

Path Decomposition Trees

All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees

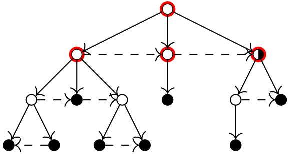


$c_l \rightarrow ABC$ and $c_l \rightarrow ACp$ and $c_l \rightarrow Ar$

$\{A\} \quad \{B, C\} \quad \{C, p, r\}$

Path Decomposition Trees

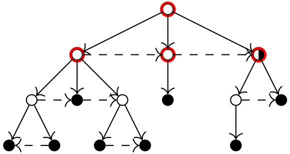
All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees



$c_l \rightarrow ABC$ and $c_l \rightarrow ACp$ and $c_l \rightarrow Ar$
$\{A\}$ $\{B, C\}$ $\{C, p, r\}$

Path Decomposition Trees

All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees



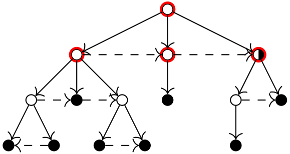
$c_l \rightarrow ABC$ and $c_l \rightarrow ACp$ and $c_l \rightarrow Ar$

$\{A\} \quad \{B, C\} \quad \{C, p, r\}$

Also possible: $\{A\} \quad \{A, B, C\} \quad \{C, p, r\}$

Path Decomposition Trees

All Decomposition Trees can not be represented
$\Rightarrow$ bound the height of represented trees



$c_l \rightarrow ABC$ and $c_l \rightarrow ACp$ and $c_l \rightarrow Ar$

$\{A\} \quad \{B, C\} \quad \{C, p, r\}$

Also possible: $\{A\} \quad \{A, B, C\} \quad \{C, p, r\}$

Also possible: $\{A\} \quad \{B\} \quad \{C\} \quad \{p, r\}$

Generating PDTs

- PDTs *can* be generated by locally deciding on how to assign sub-tasks to children

Generating PDTs

- PDTs *can* be generated by locally deciding on how to assign sub-tasks to children
- Difficult question: How does an optimal PDT look like?

Generating PDTs

- PDTs *can* be generated by locally deciding on how to assign sub-tasks to children
- Difficult question: How does an optimal PDT look like?
  - Least amount of leafs?

Generating PDTs

- PDTs *can* be generated by locally deciding on how to assign sub-tasks to children
- Difficult question: How does an optimal PDT look like?
  - Least amount of leafs?
  - Fewer tasks per leaf?

Generating PDTs

- PDTs *can* be generated by locally deciding on how to assign sub-tasks to children
- Difficult question: How does an optimal PDT look like?
  - Least amount of leafs?
  - Fewer tasks per leaf?
  - Fewer tasks per inner node?

Generating PDTs

- PDTs *can* be generated by locally deciding on how to assign sub-tasks to children
- Difficult question: How does an optimal PDT look like?
  - Least amount of leafs?
  - Fewer tasks per leaf?
  - Fewer tasks per inner node?

$\Rightarrow$ Locally optimizing #children does not lead to global minimum!

Generating PDTs

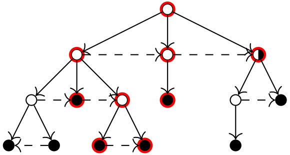
- PDTs *can* be generated by locally deciding on how to assign sub-tasks to children
- Difficult question: How does an optimal PDT look like?
  - Least amount of leafs?
  - Fewer tasks per leaf?
  - Fewer tasks per inner node?

$\Rightarrow$ Locally optimizing #children does not lead to global minimum!

- Current work tries greedily to put as few tasks as possible to each child

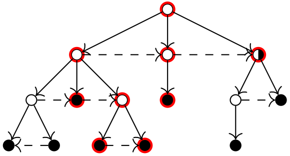
What are PDTs good for?

- A PDT contains **every** Decomposition Tree of height $\leq K$ as a sub-**graph**



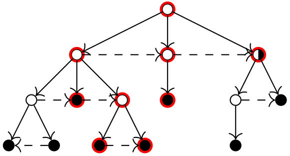
What are PDTs good for?

- A PDT contains **every** Decomposition Tree of height $\leq K$ as a sub-**graph**
- Let the valuation of a SAT formula describe such a tree



What are PDTs good for?

- A PDT contains **every** Decomposition Tree of height $\leq K$ as a sub-**graph**
- Let the valuation of a SAT formula describe such a tree
- The formula then asserts that it is a valid Decomposition Tree



Overview Part II

Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search
- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT
- **Heuristics for Heuristic Search**
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms

Possible Heuristic Estimates

What do we want to estimate?

- Number of missing actions (or their costs, resp.) or

Possible Heuristic Estimates

What do we want to estimate?

- Number of missing actions (or their costs, resp.) or
- Number of missing modifications, i.e.,
  - decompositions,
  - task insertions (if allowed),
  - causal link and ordering insertions (in plan space-based search), and
  - action applications (in progression-based search)

Possible Heuristic Estimates

What do we want to estimate?

- Number of missing actions (or their costs, resp.) or
- Number of missing modifications, i.e.,
  - decompositions,
  - task insertions (if allowed),
  - causal link and ordering insertions (in plan space-based search), and
  - action applications (in progression-based search)

→ To be used for the selection of a search node (task network/partial plan) out of the fringe

Overview Part II

Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search
- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT
- **Heuristics for Heuristic Search**
  - **TDG-based Heuristics**
  - Relaxed Composition Heuristics

Excursion

- Further Hierarchical Planning Formalisms

Problem Relaxations for Heuristic Calculation

How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

Problem Relaxations for Heuristic Calculation

How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

- Perform task insertion

Problem Relaxations for Heuristic Calculation

How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

- Perform task insertion
- Perform delete relaxation

Problem Relaxations for Heuristic Calculation

How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

- Perform task insertion
- Perform delete relaxation

$\rightarrow$ This makes the (TI)HTN plan existence problem decidable in **P**

Problem Relaxations for Heuristic Calculation

How to calculate such an estimate, given that the HTN plan existence problem is in general undecidable?

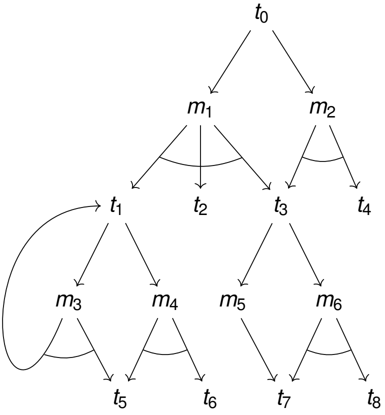
- Perform task insertion
- Perform delete relaxation

$\rightarrow$ This makes the (TI)HTN plan existence problem decidable in **P**

We introduce the Task Decomposition Graph (TDG) – which bases upon task insertion and delete relaxation – as a means to represent the task hierarchy.

TDG-based Heuristics

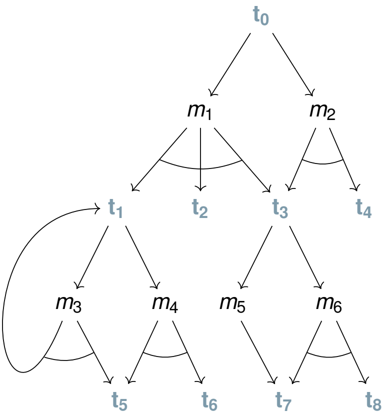
A TDG represents the decomposition structure:



A TDG is a (possibly cyclic) bipartite graph $\mathcal{G} = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

TDG-based Heuristics

A TDG represents the decomposition structure:

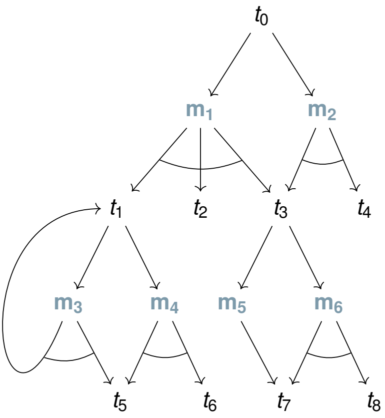


A TDG is a (possibly cyclic) bipartite graph $\mathcal{G} = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

- N_T, the task nodes,

TDG-based Heuristics

A TDG represents the decomposition structure:

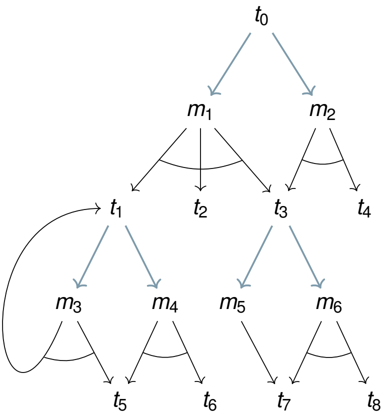


A TDG is a (possibly cyclic) bipartite graph $\mathcal{G} = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

- N_T, the task nodes,
- N_M, the method nodes,

TDG-based Heuristics

A TDG represents the decomposition structure:

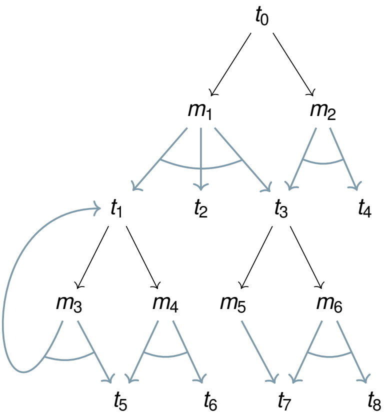


A TDG is a (possibly cyclic) bipartite graph $\mathcal{G} = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

- N_T, the task nodes,
- N_M, the method nodes,
- $E_{(T,M)}$, the task edges,

TDG-based Heuristics

A TDG represents the decomposition structure:

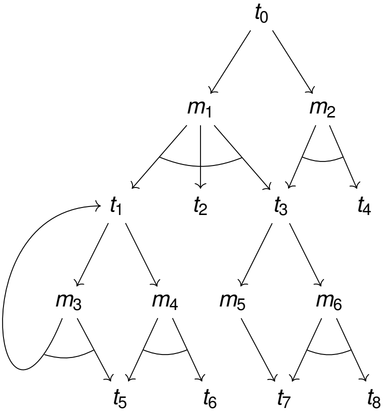


A TDG is a (possibly cyclic) bipartite graph $\mathcal{G} = \langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

- N_T, the task nodes,
- N_M, the method nodes,
- $E_{(T,M)}$, the task edges,
- $E_{(M,T)}$, the method edges.

TDG-based Heuristics

A TDG represents the decomposition structure:



How to use the TDG to calculate an heuristic estimate?

Step 1:
Calculate the TDG in a preprocessing step.

Step 2:
Calculate heuristic $h(t)$ for each task t in TDG (still via preprocessing).

Step 3:
For a search node (partial plan) P and its task identifiers T, calculate $h(P) := \sum_{t \in T} h(t)$.

Cost-aware heuristic TDG-c

Let $\langle N_T, N_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} cost(n_t) & \text{if } n_t \text{ primitive} \\ \min\limits_{(n_t,n_m)\in E_{T\to M}} h_M(n_m) & \text{else} \end{cases}$$

Cost-aware heuristic TDG-c

Let $\langle N_T, N_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} cost(n_t) & \text{if } n_t \text{ primitive} \\ \min\limits_{(n_t,n_m)\in E_{T\to M}} h_M(n_m) & \text{else} \end{cases}$$

For method nodes $n_m = \langle T, \prec, \alpha\rangle$:

$$h_M(n_m) := \sum_{(n_m,n_t)\in E_{M\to T}} h_T(n_t)$$

Cost-aware heuristic TDG-c

Let $\langle N_T, N_M, E_{T\rightarrow M}, E_{M\rightarrow T}\rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} cost(n_t) & \text{if } n_t \text{ primitive} \\ \min\limits_{(n_t,n_m)\in E_{T\rightarrow M}} h_M(n_m) & \text{else} \end{cases}$$

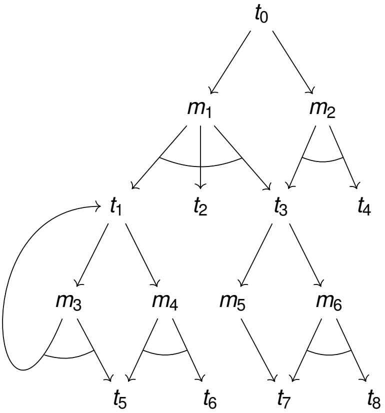
For method nodes $n_m = \langle T, \prec, \alpha\rangle$:

$$h_M(n_m) := \sum_{(n_m,n_t)\in E_{M\rightarrow T}} h_T(n_t)$$

For a given partial plan $P = (T, \prec, \alpha, CL)$, i.e, a search node, its heuristic is $h(P) := \sum_{t\in T} h(t)$ to estimate the cost of the cheapest reachable plan.

Cost-aware heuristic TDG-c (Example)

A TDG represents the decomposition structure:



How to use the TDG to calculate an heuristic estimate?

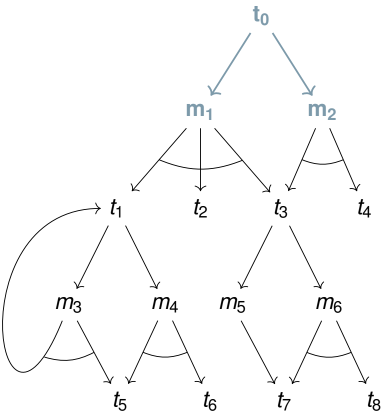
Step 1:
Calculate the TDG in a preprocessing step.

Step 2:
Calculate heuristic $h(t)$ for each task t in TDG (still via preprocessing).

Step 3:
For a search node (partial plan) P and its task identifiers T, calculate $h(P) := \sum_{t \in T} h(t)$.

Cost-aware heuristic TDG-c (Example)

A TDG represents the decomposition structure:

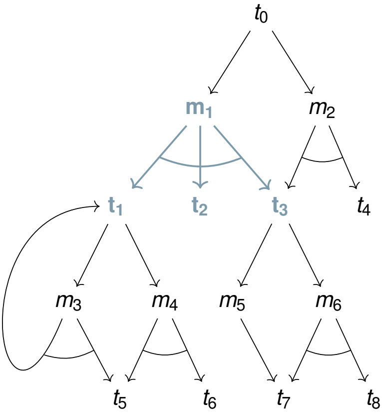


Example:

$$h_T(t_0) = min\{h_M(m_1), h_M(m_2)\}$$

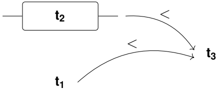
Cost-aware heuristic TDG-c (Example)

A TDG represents the decomposition structure:



Example:

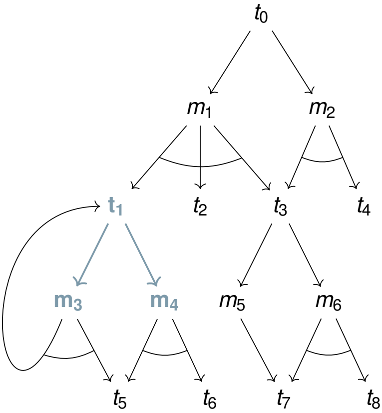
Method $m_1 = (t_0, tn)$ with task network tn:



$$h_M(m_1) = \sum_{t_i \in \{t_1, t_2, t_3\}} h_T(t_i)$$
$$= h_T(t_1) + cost(t_2) + h_T(t_3)$$

Cost-aware heuristic TDG-c (Example)

A TDG represents the decomposition structure:

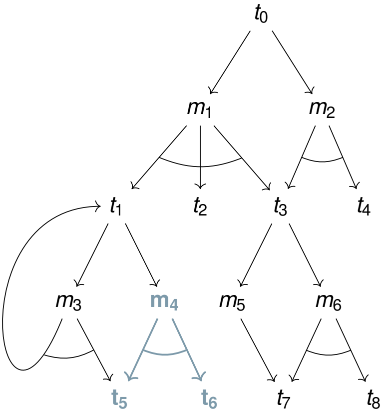


Example:

$$h_T(t_1) = min\{h_M(m_3), h_M(m_4)\}$$

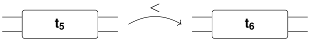
Cost-aware heuristic TDG-c (Example)

A TDG represents the decomposition structure:



Example:

Method $m_4 = (t_1, tn)$ with task network tn:



$$
\begin{aligned}
h_M(m_4) &= \sum_{t_i \in \{t_5, t_6\}} h_T(t_i) \\
&= h_T(t_5) + h_T(t_6) \\
&= cost(t_5) + cost(t_6)
\end{aligned}
$$

Modification-aware heuristic TDG-m

Let $\langle N_T, N_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} |pre(n_t)| & \text{if } n_t \text{ primitive} \\ 1 + \min\limits_{(n_t, n_m) \in E_{T\to M}} h_M(n_m) & \text{else} \end{cases}$$

Modification-aware heuristic TDG-m

Let $\langle N_T, N_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} |pre(n_t)| & \text{if } n_t \text{ primitive} \\ 1 + \min\limits_{(n_t,n_m)\in E_{T\to M}} h_M(n_m) & \text{else} \end{cases}$$

For method nodes $n_m = \langle T, \prec, \alpha\rangle$:

$$h_M(n_m) := \sum_{(n_m,n_t)\in E_{M\to T}} h_T(n_t)$$

Modification-aware heuristic TDG-m

Let $\langle N_T, N_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.

The estimates of the TDG are defined as follows:

$$h_T(n_t) := \begin{cases} |pre(n_t)| & \text{if } n_t \text{ primitive} \\ 1 + \min_{(n_t,n_m)\in E_{T\to M}} h_M(n_m) & \text{else} \end{cases}$$

For method nodes $n_m = \langle T, \prec, \alpha\rangle$:

$$h_M(n_m) := \sum_{(n_m,n_t)\in E_{M\to T}} h_T(n_t)$$

For a given partial plan $P = (T, \prec, \alpha, CL)$, i.e, a search node, its heuristic is $h(P) := \sum_{t\in T} h(t) - |CL|$ to estimate the least number of required modifications to turn P into a plan.

Summary

TDG-c and TDG-m are admissible estimates of:

- The costs of still missing actions – or
- The number of still missing decompositions and causal link insertions (the latter is specific for plan space-based planners)

Summary

TDG-c and TDG-m are admissible estimates of:

- The costs of still missing actions – or
- The number of still missing decompositions and causal link insertions (the latter is specific for plan space-based planners)

Further properties:

- Both can be calculated in polynomial time (also for the general, undecidable case)

Summary

TDG-c and TDG-m are admissible estimates of:

- The costs of still missing actions – or
- The number of still missing decompositions and causal link insertions (the latter is specific for plan space-based planners)

Further properties:

- Both can be calculated in polynomial time (also for the general, undecidable case)
- Both rely on task insertion and delete relaxation (for the construction process of the TDG)

Summary

TDG-c and TDG-m are admissible estimates of:

- The costs of still missing actions – or
- The number of still missing decompositions and causal link insertions (the latter is specific for plan space-based planners)

Further properties:

- Both can be calculated in polynomial time (also for the general, undecidable case)
- Both rely on task insertion and delete relaxation (for the construction process of the TDG)
- Only tasks within the the TDG account for the heuristic estimate, so task insertion is not reflected within the estimates ($\rightarrow$ room for improvement; but this guarantees admissibility)

Overview Part II

Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search
- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT
- **Heuristics for Heuristic Search**
  - TDG-based Heuristics
  - **Relaxed Composition Heuristics**

Excursion

- Further Hierarchical Planning Formalisms

Classical Heuristics in HTN Planning

- We have seen two search-based approaches that can be instantiated as **heuristic search**
- We need to **sort** the fringe (according to what?)
- In a first step, estimate **goal distance** ($\rightarrow$ Satisficing Planning)

Classical Heuristics in HTN Planning

- We have seen two search-based approaches that can be instantiated as **heuristic search**
- We need to **sort** the fringe (according to what?)
- In a first step, estimate **goal distance** ($\rightarrow$ Satisficing Planning)

Using Techniques from Classical Planning – Challenges:

- More expressive formalism $\rightarrow$ techniques not applicable directly
- Hierarchy has huge impact on valid solutions

Classical Heuristics in HTN Planning

- We have seen two search-based approaches that can be instantiated as **heuristic search**
- We need to **sort** the fringe (according to what?)
- In a first step, estimate **goal distance** ($\rightarrow$ Satisficing Planning)

Using Techniques from Classical Planning – Challenges:

- More expressive formalism $\rightarrow$ techniques not applicable directly
- Hierarchy has huge impact on valid solutions
  - Which actions are reachable?

Classical Heuristics in HTN Planning

- We have seen two search-based approaches that can be instantiated as **heuristic search**
- We need to **sort** the fringe (according to what?)
- In a first step, estimate **goal distance** ($\rightarrow$ Satisficing Planning)

Using Techniques from Classical Planning – Challenges:

- More expressive formalism $\rightarrow$ techniques not applicable directly
- Hierarchy has huge impact on valid solutions
  - Which actions are reachable?
  - What is the objective, the “goal”?
    $\rightarrow$ usually no state-based goal given

Classical Heuristics in HTN Planning

Approach:

1. Relax HTN to a classical planning problem
   - Search is done in an HTN planning system on the original model
   - This model is only used for heuristic calculation

Classical Heuristics in HTN Planning

Approach:

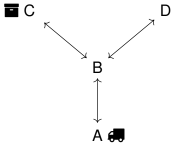
1. Relax HTN to a classical planning problem
   - Search is done in an HTN planning system on the original model
   - This model is only used for heuristic calculation
2. Apply classical heuristics to that problem
   - For some search node, the “heuristic model” is adapted
   - Goal distance is estimated

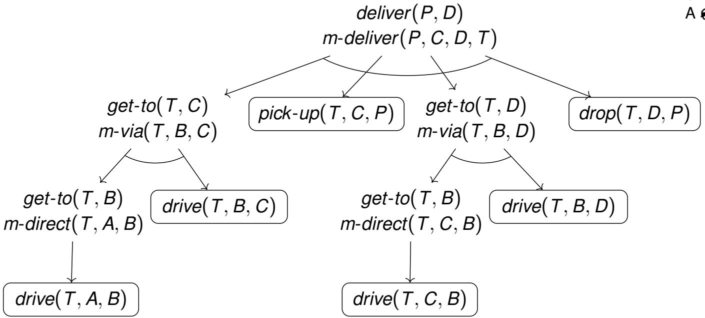
Classical Heuristics in HTN Planning

Approach:

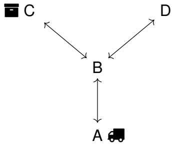
1. Relax HTN to a classical planning problem
   - Search is done in an HTN planning system on the original model
   - This model is only used for heuristic calculation
2. Apply classical heuristics to that problem
   - For some search node, the "heuristic model" is adapted
   - Goal distance is estimated
3. Use heuristic value in HTN planning
   - The fringe of the HTN planning system is sorted according to the heuristic value

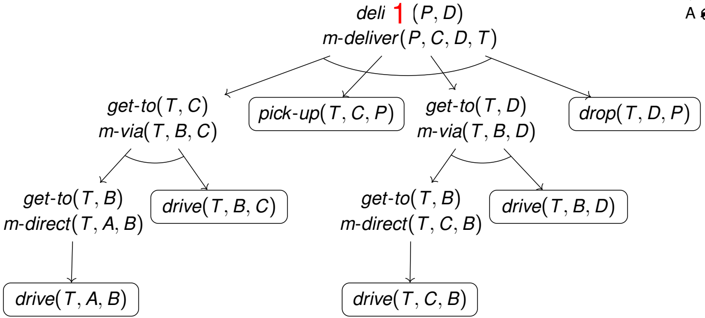
Simulating Composition



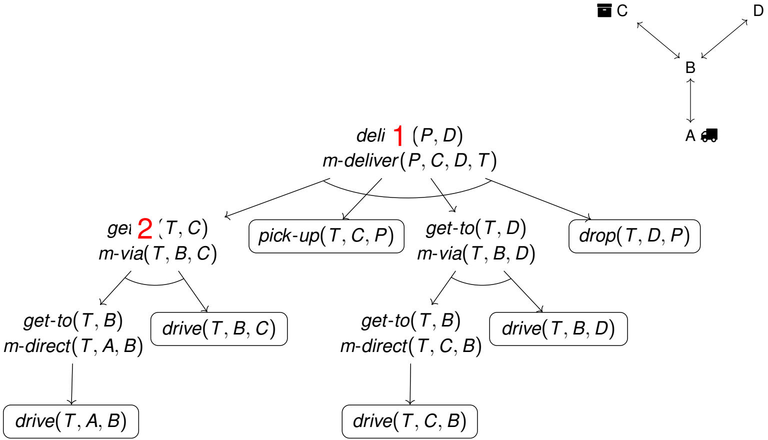


Simulating Composition

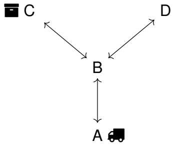


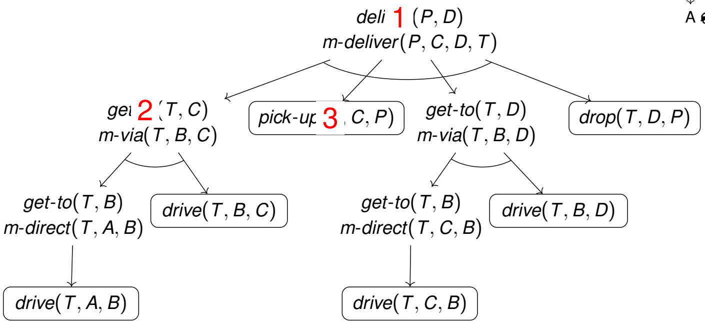


Simulating Composition

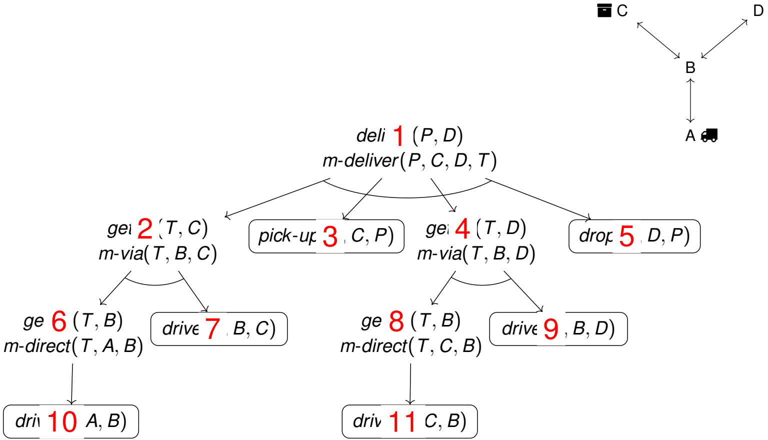


Simulating Composition

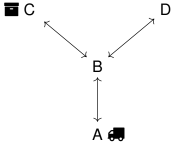


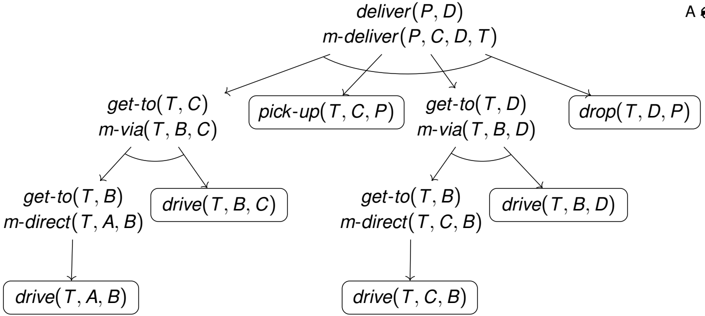


Simulating Composition



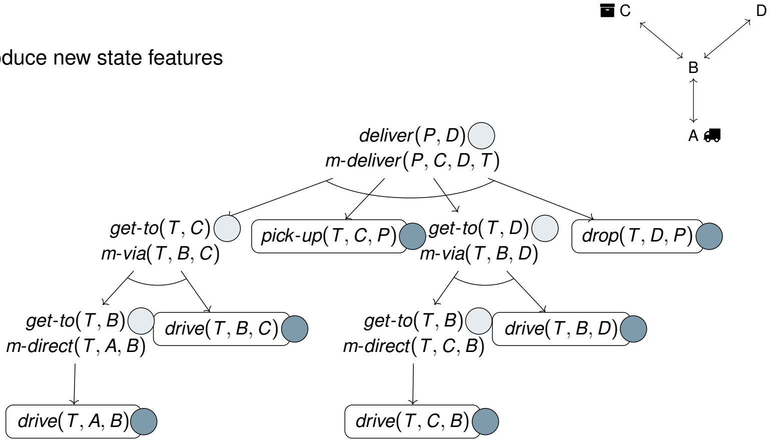
Simulating Composition



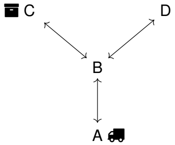


Simulating Composition

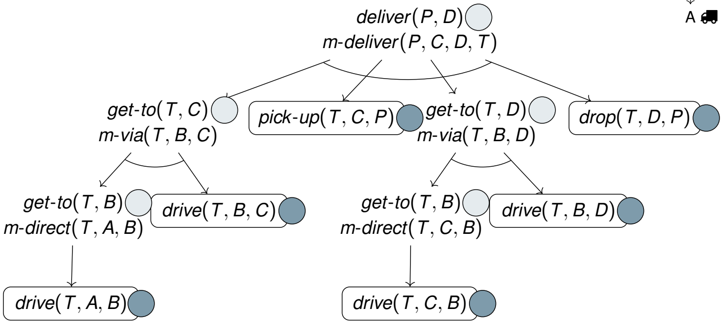
- Introduce new state features

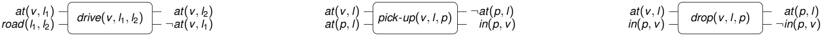


Simulating Composition

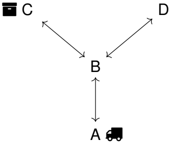


- Introduce new state features

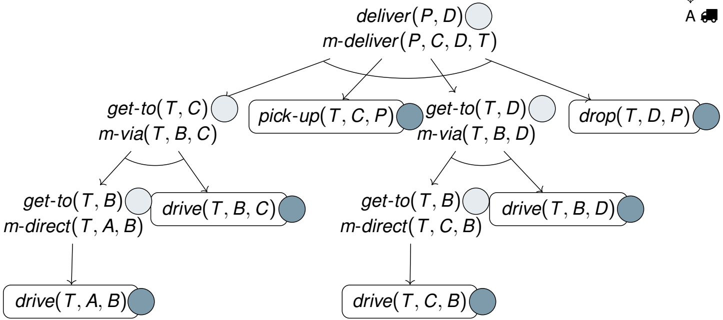


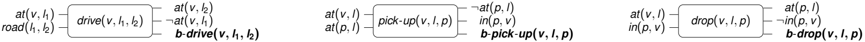


Simulating Composition

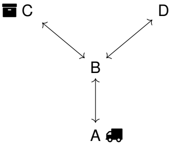


- Introduce new state features
- Modify actions

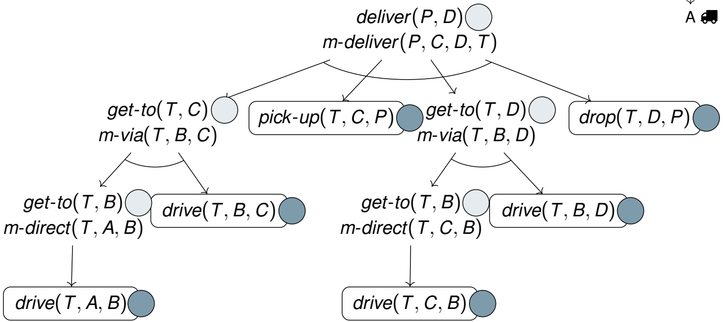


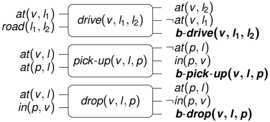


Simulating Composition

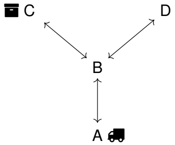


- Introduce new state features
- Modify actions

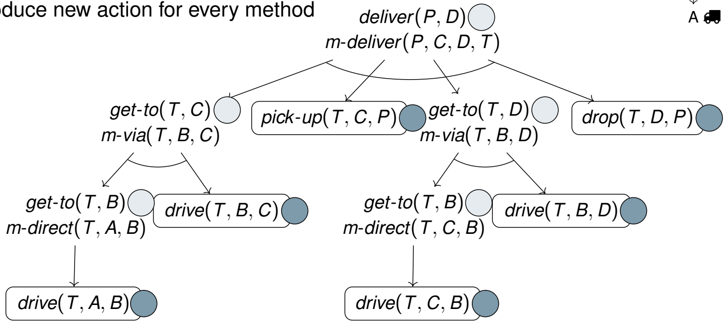


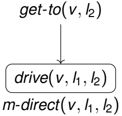


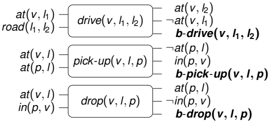
Simulating Composition



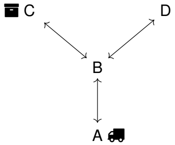
- Introduce new state features
- Modify actions
- Introduce new action for every method



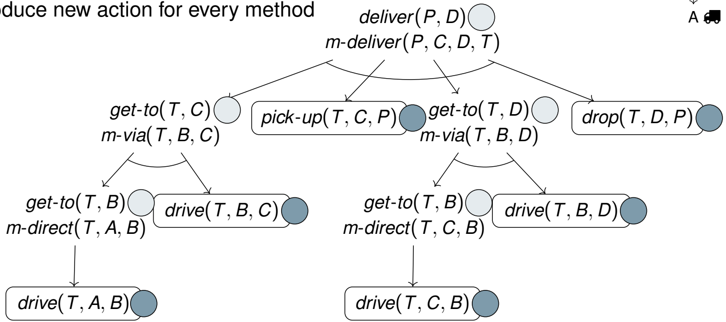


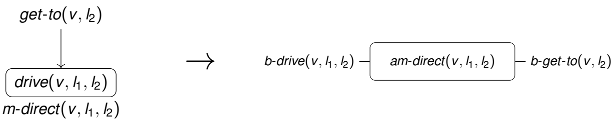


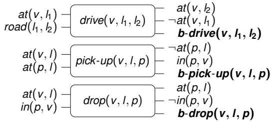
Simulating Composition



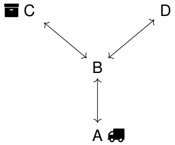
- Introduce new state features
- Modify actions
- Introduce new action for every method



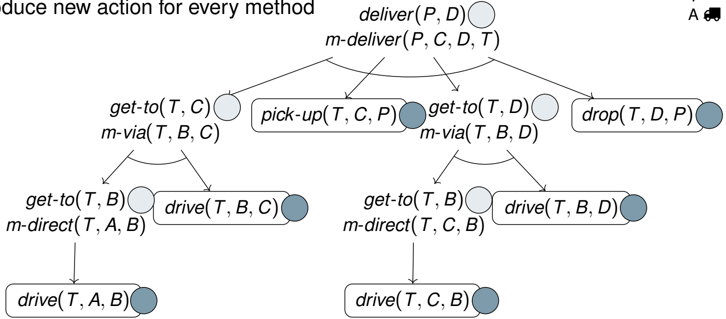


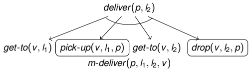


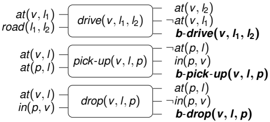
Simulating Composition



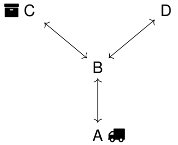
- Introduce new state features
- Modify actions
- Introduce new action for every method



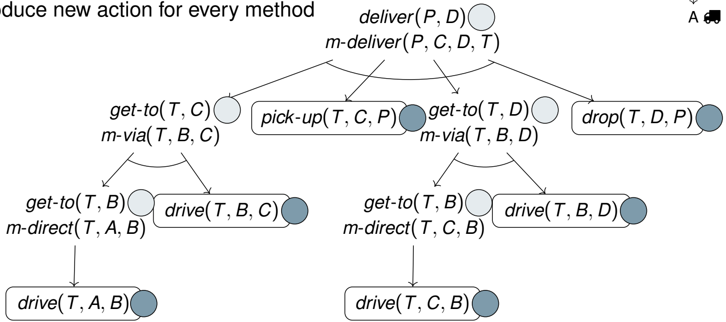


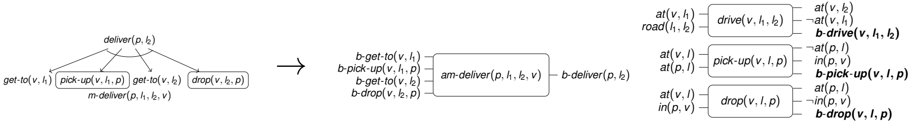


Simulating Composition

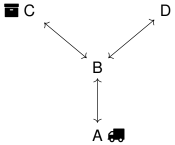


- Introduce new state features
- Modify actions
- Introduce new action for every method

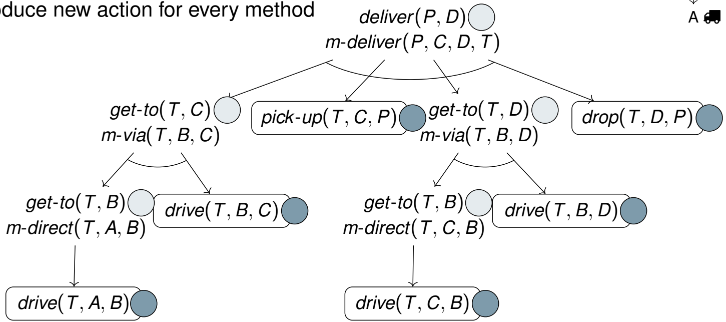




Simulating Composition

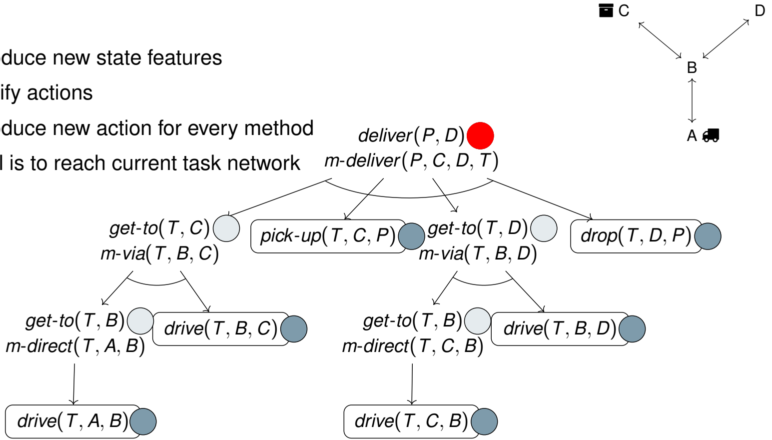


- Introduce new state features
- Modify actions
- Introduce new action for every method

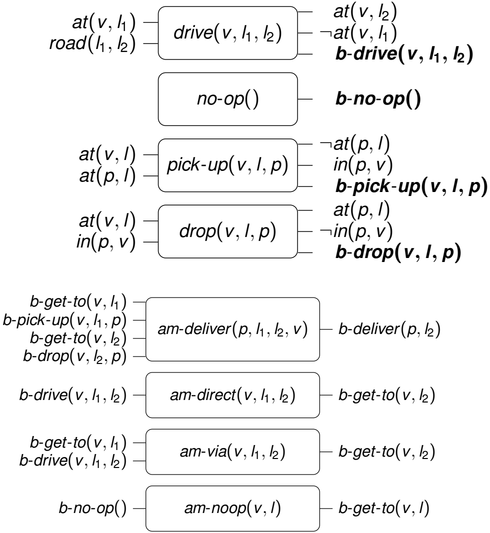


Simulating Composition

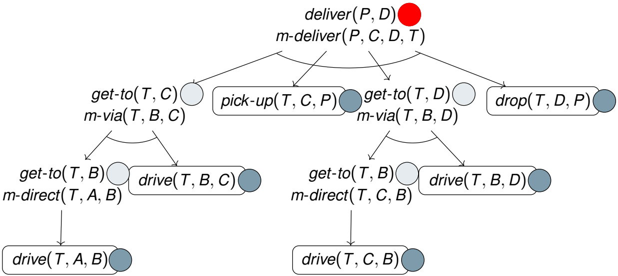
- Introduce new state features
- Modify actions
- Introduce new action for every method
- Goal is to reach current task network

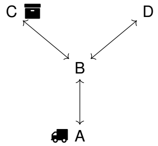


Simulating Composition – Resulting Model



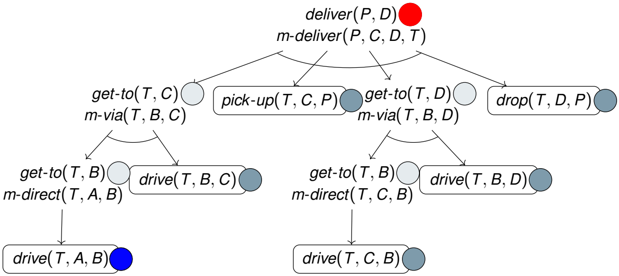
Planning in the Transformed Model

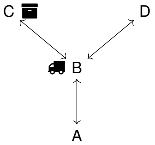




$\{at(T, A),$
$at(P, C)\}$

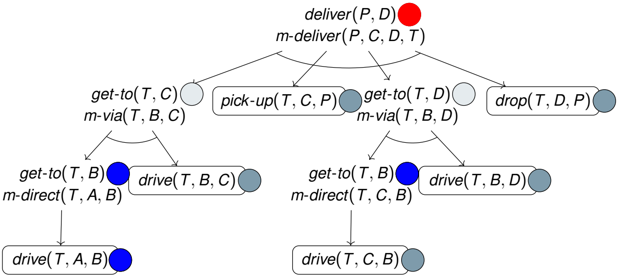
Planning in the Transformed Model

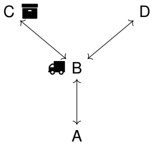


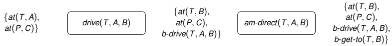


$\{at(T, A), at(P, C)\}$ *drive*(T, A, B) $\{at(T, B), at(P, C), b\text{-}drive(T, A, B)\}$

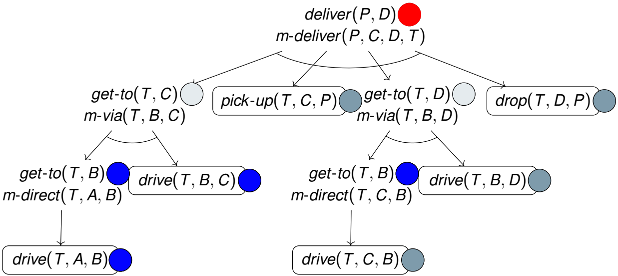
Planning in the Transformed Model

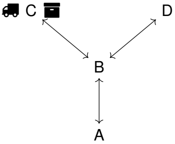


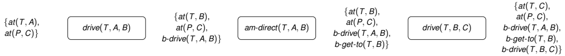




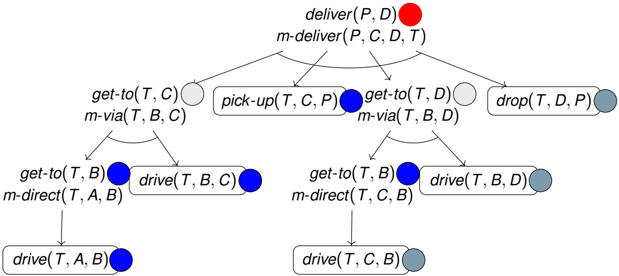
Planning in the Transformed Model

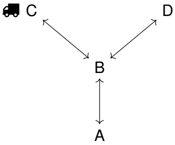


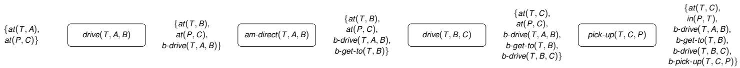




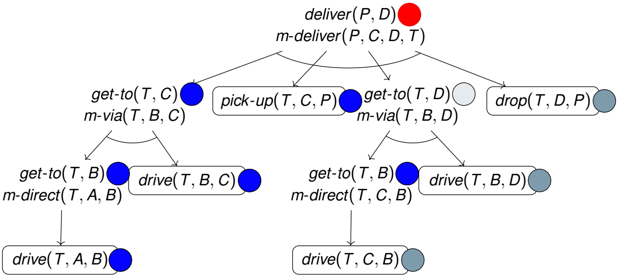
Planning in the Transformed Model

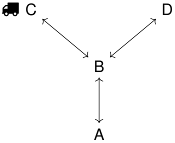


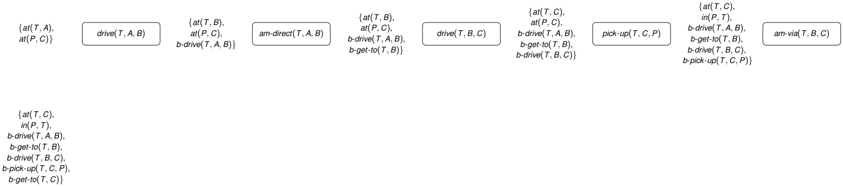




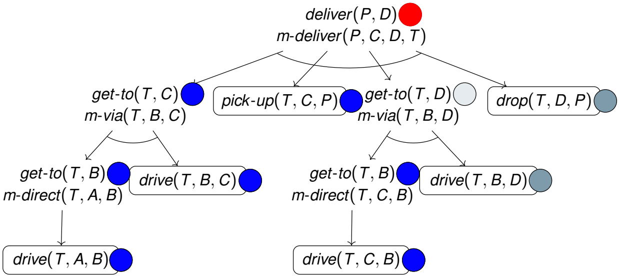
Planning in the Transformed Model

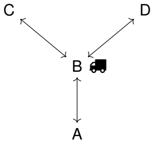


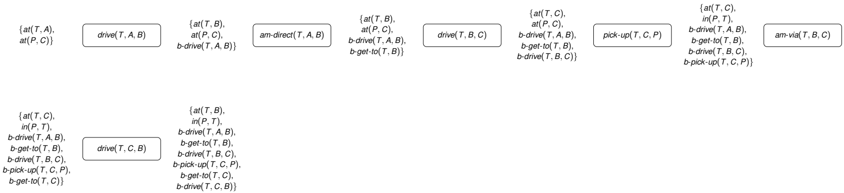




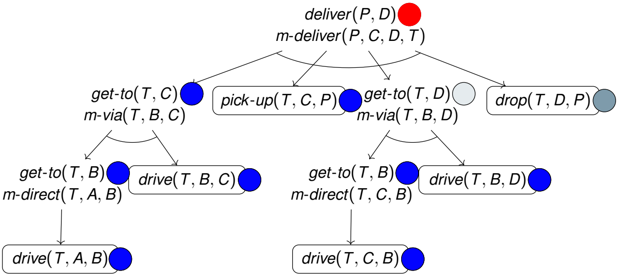
Planning in the Transformed Model

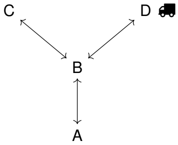


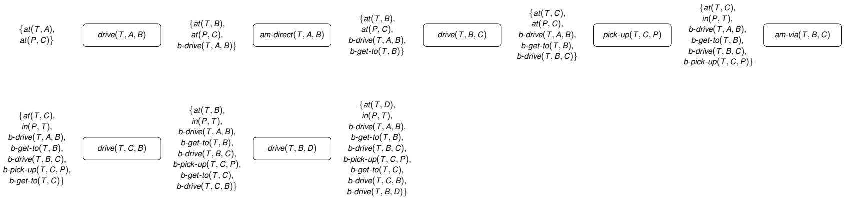




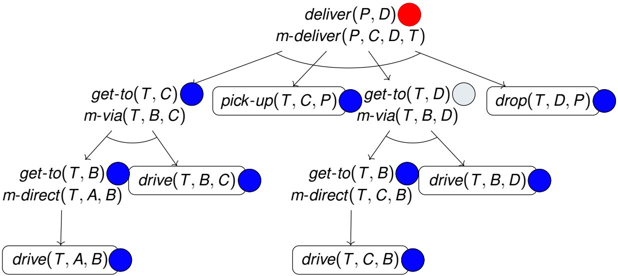
Planning in the Transformed Model

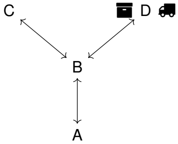


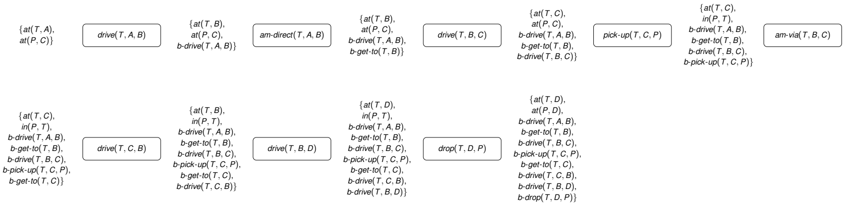




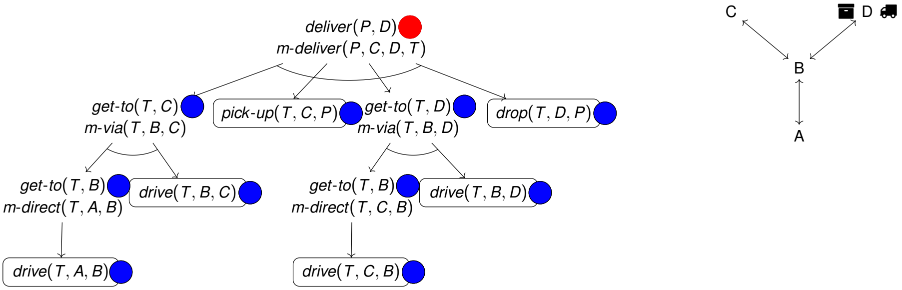
Planning in the Transformed Model







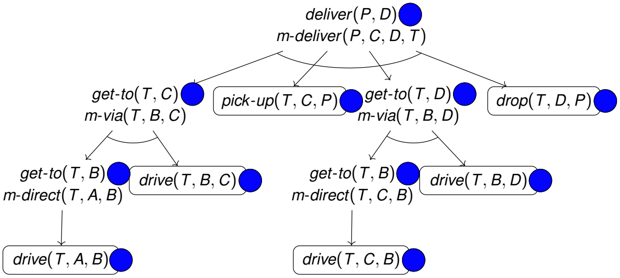
Planning in the Transformed Model

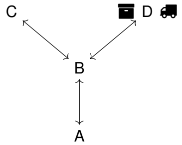


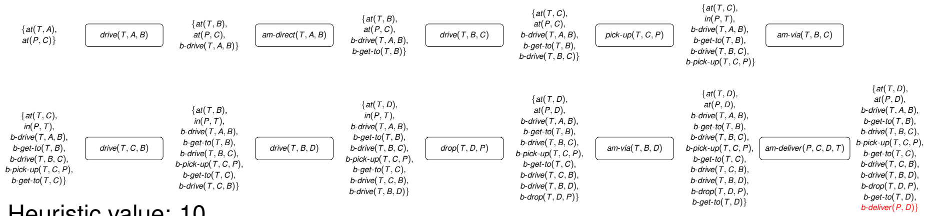
$\{at(T,A), at(P,C)\}$ | $drive(T,A,B)$ | $\{at(T,B), at(P,C), b\text{-}drive(T,A,B)\}$ | $am\text{-}direct(T,A,B)$ | $\{at(T,B), at(P,C), b\text{-}drive(T,A,B), b\text{-}get\text{-}to(T,B)\}$ | $drive(T,B,C)$ | $\{at(T,C), at(P,C), b\text{-}drive(T,A,B), b\text{-}get\text{-}to(T,B), b\text{-}drive(T,B,C)\}$ | $pick\text{-}up(T,C,P)$ | $\{at(T,C), in(P,T), b\text{-}drive(T,A,B), b\text{-}get\text{-}to(T,B), b\text{-}drive(T,B,C), b\text{-}pick\text{-}up(T,C,P)\}$ | $am\text{-}via(T,B,C)$

$\{at(T,C), in(P,T), b\text{-}drive(T,A,B), b\text{-}get\text{-}to(T,B), b\text{-}drive(T,B,C), b\text{-}pick\text{-}up(T,C,P), b\text{-}get\text{-}to(T,C)\}$ | $drive(T,C,B)$ | $\{at(T,B), in(P,T), b\text{-}drive(T,A,B), b\text{-}get\text{-}to(T,B), b\text{-}drive(T,B,C), b\text{-}pick\text{-}up(T,C,P), b\text{-}get\text{-}to(T,C), b\text{-}drive(T,C,B)\}$ | $drive(T,B,D)$ | $\{at(T,D), in(P,T), b\text{-}drive(T,A,B), b\text{-}get\text{-}to(T,B), b\text{-}drive(T,B,C), b\text{-}pick\text{-}up(T,C,P), b\text{-}get\text{-}to(T,C), b\text{-}drive(T,C,B), b\text{-}drive(T,B,D)\}$ | $drop(T,D,P)$ | $\{at(T,D), at(P,D), b\text{-}drive(T,A,B), b\text{-}get\text{-}to(T,B), b\text{-}drive(T,B,C), b\text{-}pick\text{-}up(T,C,P), b\text{-}get\text{-}to(T,C), b\text{-}drive(T,C,B), b\text{-}drive(T,B,D), b\text{-}drop(T,D,P)\}$ | $am\text{-}via(T,B,D)$ | $\{at(T,D), at(P,D), b\text{-}drive(T,A,B), b\text{-}get\text{-}to(T,B), b\text{-}drive(T,B,C), b\text{-}pick\text{-}up(T,C,P), b\text{-}get\text{-}to(T,C), b\text{-}drive(T,C,B), b\text{-}drive(T,B,D), b\text{-}drop(T,D,P), b\text{-}get\text{-}to(T,D)\}$

Planning in the Transformed Model

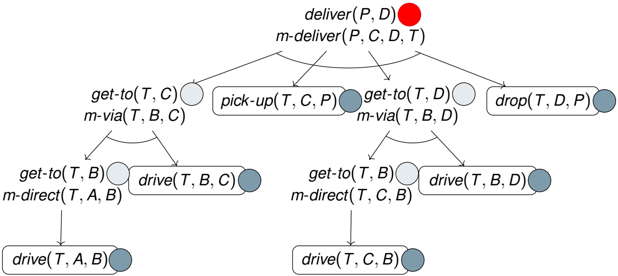


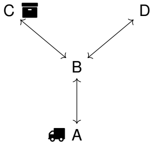




Heuristic value: 10

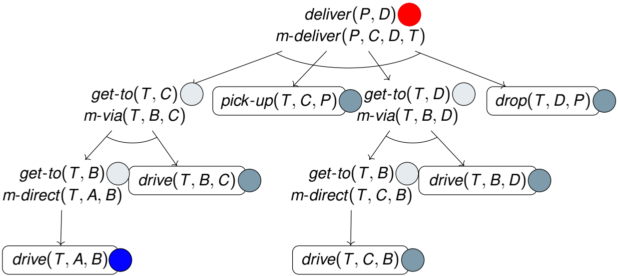
Heuristic Calculation (Delete Relaxed)

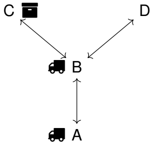


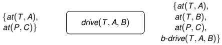


$\{at(T, A),$
$at(P, C)\}$

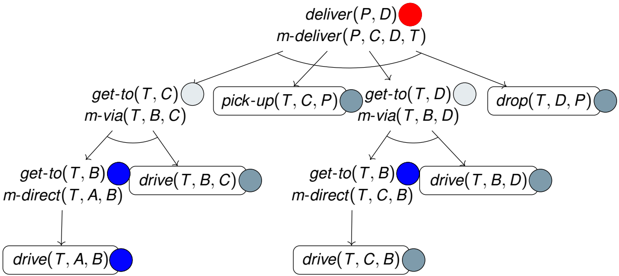
Heuristic Calculation (Delete Relaxed)

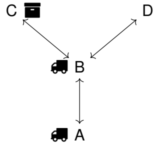






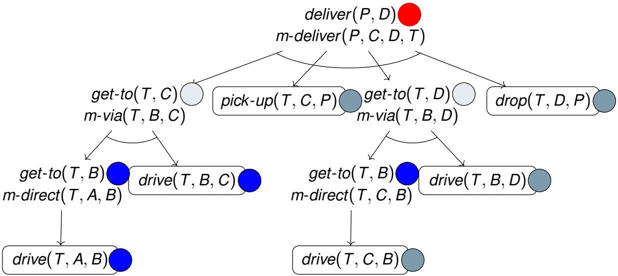
Heuristic Calculation (Delete Relaxed)

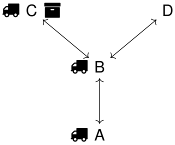


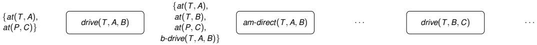


{*at*(*T*, *A*), *at*(*P*, *C*)} | *drive*(*T*, *A*, *B*) | {*at*(*T*, *A*), *at*(*T*, *B*), *at*(*P*, *C*), *b-drive*(*T*, *A*, *B*)} | *am-direct*(*T*, *A*, *B*) | ...

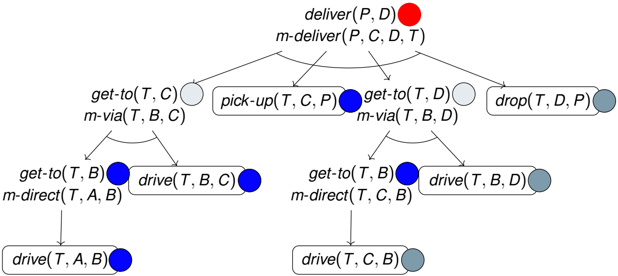
Heuristic Calculation (Delete Relaxed)

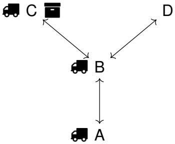




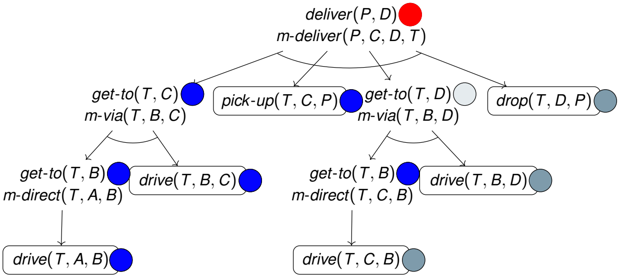


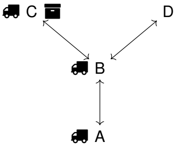
Heuristic Calculation (Delete Relaxed)





Heuristic Calculation (Delete Relaxed)



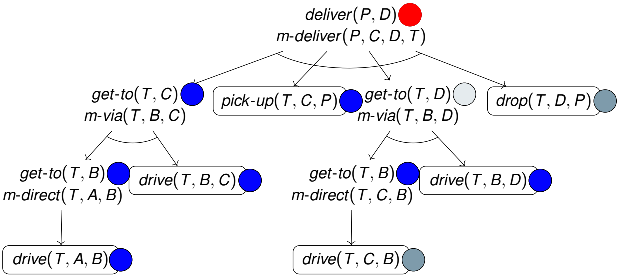


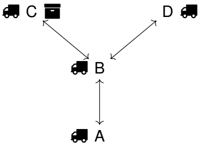
$\{at(T, A), at(P, C)\}$ | *drive*(T, A, B) | $\{at(T, A), at(T, B), at(P, C), b\text{-}drive(T, A, B)\}$ | *am-direct*(T, A, B) | … | *drive*(T, B, C) | … | *pick-up*(T, C, P) | … | *am-via*(T, B, C)

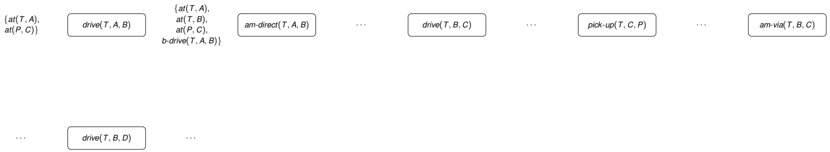
…

Using Classical Heuristics to Guide HTN Search

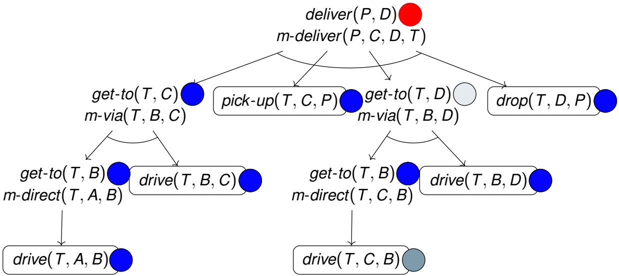
Heuristic Calculation (Delete Relaxed)

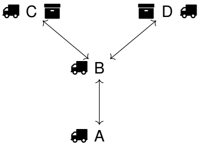




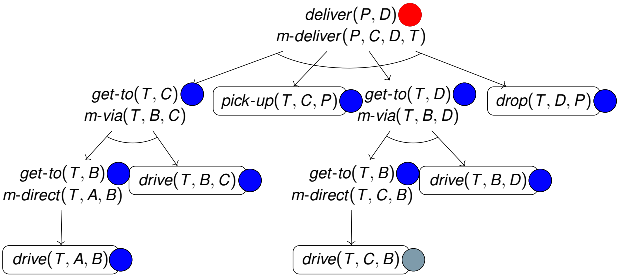


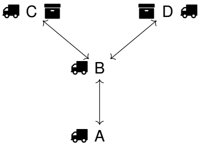
Heuristic Calculation (Delete Relaxed)





Heuristic Calculation (Delete Relaxed)



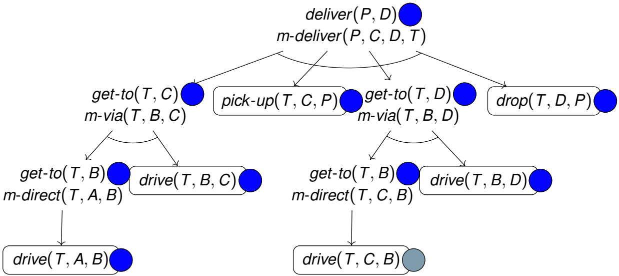


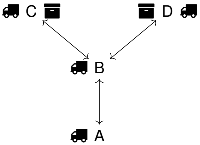
{*at*(*T*, *A*), *at*(*P*, *C*)} | *drive*(*T*, *A*, *B*) | {*at*(*T*, *A*), *at*(*T*, *B*), *at*(*P*, *C*), *b-drive*(*T*, *A*, *B*)} | *am-direct*(*T*, *A*, *B*) | ... | *drive*(*T*, *B*, *C*) | ... | *pick-up*(*T*, *C*, *P*) | ... | *am-via*(*T*, *B*, *C*)

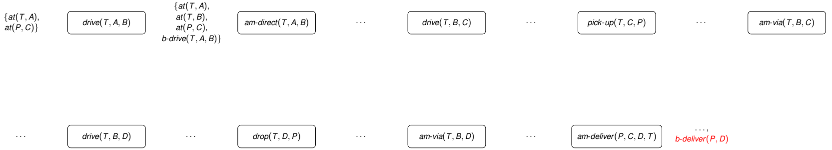
... | *drive*(*T*, *B*, *D*) | ... | *drop*(*T*, *D*, *P*) | ... | *am-via*(*T*, *B*, *D*) | ...

Using Classical Heuristics to Guide HTN Search

Heuristic Calculation (Delete Relaxed)







Using delete-relaxed classical heuristic: 9

General Characteristics

- Simulates task composition

General Characteristics

- Simulates task composition
- ✓ Incorporates hierarchical reachability information
- ✓ Combines it with information on state-based executability
- ✓ Solves the problem of a missing state-based goal

General Characteristics

- Simulates task composition
- ✓ Incorporates hierarchical reachability information
- ✓ Combines it with information on state-based executability
- ✓ Solves the problem of a missing state-based goal

- The transformation from HTN to classical problem is a relaxation
- → The set of valid solutions increases

General Characteristics

- Simulates task composition
- ✓ Incorporates hierarchical reachability information
- ✓ Combines it with information on state-based executability
- ✓ Solves the problem of a missing state-based goal

- The transformation from HTN to classical problem is a relaxation
- → The set of valid solutions increases

- Heuristic function is allowed to do
  - Task sharing (every task must be proceeded only once)
  - Task insertion (e.g. to fulfill preconditions)
  - HTN ordering relations are relaxed

General Characteristics

- Simulates task composition
- ✓ Incorporates hierarchical reachability information
- ✓ Combines it with information on state-based executability
- ✓ Solves the problem of a missing state-based goal

- The transformation from HTN to classical problem is a relaxation
- → The set of valid solutions increases

- Heuristic function is allowed to do
  - Task sharing (every task must be proceeded only once)
  - Task insertion (e.g. to fulfill preconditions)
  - HTN ordering relations are relaxed
- Heuristic function may only insert tasks that lie within the decomposition hierarchy (not given here)

Computational Aspects

- Size is **linear** in the input HTN domain, but the model is **large**
- **State** and **action** set are **extended**

Computational Aspects

- Size is **linear** in the input HTN domain, but the model is **large**
- **State** and **action** set are **extended**
- Most parts of the **model** are **static** during search, one needs to update
  - Initial state
  - Goal

$\rightarrow$ **Efficient update** of the “heuristic model” possible

Computational Aspects

- Size is **linear** in the input HTN domain, but the model is **large**
- **State** and **action** set are **extended**
- Most parts of the **model** are **static** during search, one needs to update
  - Initial state
  - Goal

→ **Efficient update** of the “heuristic model” possible

→ Classical heuristic combined with the encoding should be able to **deal with changed goal efficiently**

Resulting Heuristic Values

- Perfect HTN solution (in terms of modifications) corresponds to a classical plan in the transformation with equal costs
- Perfect classical heuristic on the transformation has **less or equal costs**

Resulting Heuristic Values

- Perfect HTN solution (in terms of modifications) corresponds to a classical plan in the transformation with equal costs
- Perfect classical heuristic on the transformation has **less or equal costs**
- When the used classical heuristic has one of the following properties, the resulting HTN heuristic has it too:
  - Safety
  - Goal-awareness
  - Admissibility

Discussion

- Can be combined with many classical heuristics

Discussion

- Can be combined with many classical heuristics
- In principle applicable in both – plan space or progression search
  - $\rightarrow$ Progression search provides more precise state information

Discussion

- Can be combined with many classical heuristics
- In principle applicable in both – plan space or progression search
  - $\rightarrow$ Progression search provides more precise state information
- Comparison to "HTN to STRIPS/ADL translation"
  - This transformation is a relaxation (set of solutions changes)
  - It is smaller
  - It is easier to compute

Overview Part II

Solving HTN Planning Problems

- Search-based Approaches
  - Plan Space Search
  - Progression Search
- Compilation-based Approaches
  - Compilations to STRIPS/ADL
  - Compilations to SAT
- Heuristics for Heuristic Search
  - TDG-based Heuristics
  - Relaxed Composition Heuristics

Excursion

- **Further Hierarchical Planning Formalisms**

Overview of Hierarchical Planning Variants

Which variants of HTN planning and further hierarchical planning problem classes exist?

Overview of Hierarchical Planning Variants

Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)

Overview of Hierarchical Planning Variants

Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
- Task sharing

Overview of Hierarchical Planning Variants

Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
- Task sharing
- Hybrid planning (i.e., HTN + POCL Planning)

Overview of Hierarchical Planning Variants

Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
- Task sharing
- Hybrid planning (i.e., HTN + POCL Planning)
- Decompositional planning (i.e., hybrid without initial plan)

Overview of Hierarchical Planning Variants

Which variants of HTN planning and further hierarchical planning problem classes exist?

- HTN planning with *task insertion* (TIHTN planning)
- Task sharing
- Hybrid planning (i.e., HTN + POCL Planning)
- Decompositional planning (i.e., hybrid without initial plan)
- GTN planning (decompose goals, not tasks)

Problem Definition

In *HTN planning with task insertion*, *TIHTN planning*, tasks may be added arbitrarily to task networks (not just via decomposition):

Let $\mathcal{P}^{\star} = (V, P, \delta, C, M, s_I, c_I)$ be a **TIHTN planning problem**.

In *HTN planning with task insertion*, *TIHTN planning*, tasks may be added arbitrarily to task networks (not just via decomposition):

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, c_I)$ be a **TIHTN planning problem**.

Then, a task network *tn* is a solution if and only if:

- There is a sequence of decomposition methods $\overline{m}$ **and task insertions** that transforms c_I into *tn*,
- *tn* contains only primitive tasks, and
- the (still partially ordered) task network *tn* admits an executable linearization $\overline{t}$ of its tasks.

Problem Definition

In *HTN planning with task insertion*, *TIHTN planning*, tasks may be added arbitrarily to task networks (not just via decomposition):

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, c_I)$ be a **TIHTN planning problem**.

Then, a task network *tn* is a solution if and only if:

- There is a sequence of decomposition methods $\overline{m}$ that transforms c_I into $\boldsymbol{tn'}$,
- $\boldsymbol{tn \supseteq tn'}$ **contains all tasks and orderings of** $\boldsymbol{tn'}$,
- *tn* contains only primitive tasks, and
- the (still partially ordered) task network *tn* admits an executable linearization $\overline{t}$ of its tasks.

Motivation

Benefits of allowing task insertion:

- Task insertion plus goal description fully subsumes classical planning (while allowing task hierarchies as well)

Motivation

Benefits of allowing task insertion:

- Task insertion plus goal description fully subsumes classical planning (while allowing task hierarchies as well)
- Task insertion makes the modeling process easier: certain parts can be left to the planner

Motivation

Benefits of allowing task insertion:

- Task insertion plus goal description fully subsumes classical planning (while allowing task hierarchies as well)
- Task insertion makes the modeling process easier: certain parts can be left to the planner
- Task insertion makes the problem computationally easier (can be exploited for heuristics)

Task sharing allows unconstrained tasks to be merged:

Let $\mathcal{P}^{\star} = (V, P, \delta, C, M, s_I, c_I)$ be an **HTN problem with task sharing**.

Task sharing allows unconstrained tasks to be merged:

Let $\mathcal{P}^\star = (V, P, \delta, C, M, s_I, c_I)$ be an **HTN problem with task sharing**.

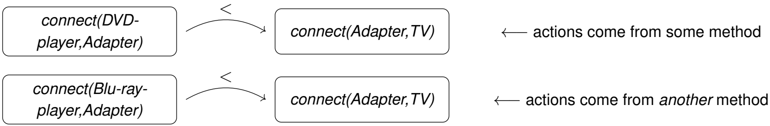
Then, a task network *tn* is a solution if and only if:

- There is a sequence of decomposition methods $\overline{m}$ and **task mergings** that transform c_I into *tn* **(two tasks can be merged if they are identical and not ordered with respect to another)**,
- *tn* contains only primitive tasks, and
- the (still partially ordered) task network *tn* admits an executable linearization $\overline{t}$ of its tasks.

Motivation

Benefits of allowing task sharing:

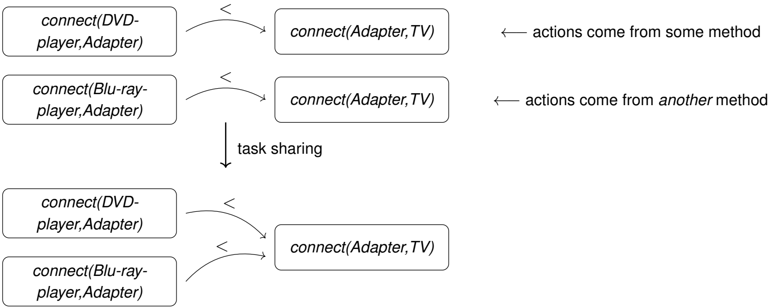
- Allows to eliminate duplicates that might just be modeling artifacts



Motivation

Benefits of allowing task sharing:

- Allows to eliminate duplicates that might just be modeling artifacts



Problem Definition

Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Problem Definition

Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Core differences to standard HTN planning:

- Compound tasks can have preconditions and effects as well

Problem Definition

Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Core differences to standard HTN planning:

- Compound tasks can have preconditions and effects as well
- Decomposition methods must adhere certain criteria (so that they are implementations of their compound tasks)

Problem Definition

Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Core differences to standard HTN planning:

- Compound tasks can have preconditions and effects as well
- Decomposition methods must adhere certain criteria (so that they are implementations of their compound tasks)
- Rather than task networks, we have partial plans that may contain causal links

Problem Definition

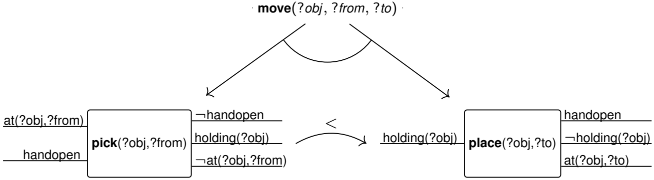
Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.

Core differences to standard HTN planning:

- Compound tasks can have preconditions and effects as well
- Decomposition methods must adhere certain criteria (so that they are implementations of their compound tasks)
- Rather than task networks, we have partial plans that may contain causal links
- In solution plans, *all* linearizations must be executable

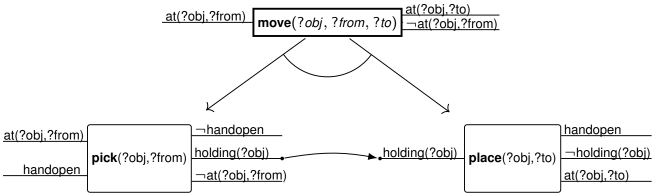
Problem Definition

Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.



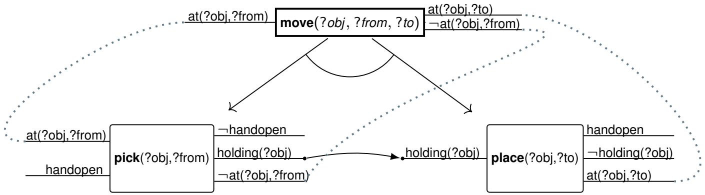
Problem Definition

Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.



Problem Definition

Hybrid planning fuses HTN planning with Partial-Order Causal-Link (POCL) Planning.



Motivation

Benefits of hybrid planning:

- Modeling support due to preconditions and effects of compound tasks and legality criteria for their methods

Motivation

Benefits of hybrid planning:

- Modeling support due to preconditions and effects of compound tasks and legality criteria for their methods
- In combination with task insertion: compound tasks can be inserted easier due to their preconditions and effects

Motivation

Benefits of hybrid planning:

- Modeling support due to preconditions and effects of compound tasks and legality criteria for their methods
- In combination with task insertion: compound tasks can be inserted easier due to their preconditions and effects
- Solution criteria (*all* linearizations are executable) is more practical than the classical one (there *exist* an executable linearization)

Motivation

Benefits of hybrid planning:

- Modeling support due to preconditions and effects of compound tasks and legality criteria for their methods
- In combination with task insertion: compound tasks can be inserted easier due to their preconditions and effects
- Solution criteria (*all* linearizations are executable) is more practical than the classical one (there *exist* an executable linearization)
- Plan explanation and visualization becomes more natural

Problem Definition

Decompositional planning is defined just as hybrid planning with task insertion – with the exception that there is no initial partial plan.

Motivation

Benefits of decompositional planning:

- Everything like in hybrid planning, except:

  lower expressivity (identical to non-hierarchical, classical planning), because the hierarchy does not induce constraints

Problem Definition

Hierarchical Goal Network (HGN) planning is concerned with the decomposition of *goals* instead of tasks.

Problem Definition

Hierarchical Goal Network (HGN) planning is concerned with the decomposition of *goals* instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions

Problem Definition

Hierarchical Goal Network (HGN) planning is concerned with the decomposition of *goals* instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
- Instead of task networks, HGN planning uses *goal networks*: partially ordered sets of goals (each being a formula over state variables)

Problem Definition

Hierarchical Goal Network (HGN) planning is concerned with the decomposition of *goals* instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
- Instead of task networks, HGN planning uses *goal networks*: partially ordered sets of goals (each being a formula over state variables)
- Decomposition methods refine/substitute *goals* rather than *tasks*

Problem Definition

Hierarchical Goal Network (HGN) planning is concerned with the decomposition of *goals* instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
- Instead of task networks, HGN planning uses *goal networks*: partially ordered sets of goals (each being a formula over state variables)
- Decomposition methods refine/substitute *goals* rather than *tasks*
- The hierarchy induced on goals does not partition them into primitive and non-primitive goals

Problem Definition

Hierarchical Goal Network (HGN) planning is concerned with the decomposition of *goals* instead of tasks.

Core differences to HTN planning:

- There is only one kind of tasks, i.e., (primitive) actions
- Instead of task networks, HGN planning uses *goal networks*: partially ordered sets of goals (each being a formula over state variables)
- Decomposition methods refine/substitute *goals* rather than *tasks*
- The hierarchy induced on goals does not partition them into primitive and non-primitive goals
- All actions can be applied to the current state, as long as they achieve a possibly first goal

Motivation

Benefits of HGN planning?

- The application of state-based heuristics is more directly applicable than in HTN planning

Motivation

Benefits of HGN planning?

- The application of state-based heuristics is more directly applicable than in HTN planning
- In some domains, defining a hierarchy on state features might be easier than defining a hierarchy on tasks

Thank You for Your Attention!

Thank you for your attention!

Are there questions?