Lecture *Hierarchical Planning*

Chapter:
Solving Hierarchical Problems via Search

Dr. Pascal Bercher

Institute of Artificial Intelligence,
Ulm University, Germany

Winter Term 2018/2019

(Compiled on: February 19, 2019)

ulm university universität ulm

Overview:

How to Solve Hierarchical Planning Problems?

- Via reduction, i.e., compilation to other problems like

How to Solve Hierarchical Planning Problems?

- Via reduction, i.e., compilation to other problems like
  - SAT, i.e., Satisfiability (later in this lecture).

How to Solve Hierarchical Planning Problems?

- Via reduction, i.e., compilation to other problems like
  - SAT, i.e., Satisfiability (later in this lecture).
  - ASP, i.e., Answer Set Programming (not covered).

How to Solve Hierarchical Planning Problems?

- Via reduction, i.e., compilation to other problems like
  - SAT, i.e., Satisfiability (later in this lecture).
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).

How to Solve Hierarchical Planning Problems?

- Via reduction, i.e., compilation to other problems like
  - SAT, i.e., Satisfiability (later in this lecture).
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).
- Search:

How to Solve Hierarchical Planning Problems?

- Via reduction, i.e., compilation to other problems like
  - SAT, i.e., Satisfiability (later in this lecture).
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).
- Search:
  - Forward progression search in the space of world state – plus the remaining task network to go thereby extending classical planning.

How to Solve Hierarchical Planning Problems?

- Via reduction, i.e., compilation to other problems like
  - SAT, i.e., Satisfiability (later in this lecture).
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).
- Search:
  - Forward progression search in the space of world state – plus the remaining task network to go thereby extending classical planning.
  - (Regression-like) search in the space of partial plans – extends POCL planning to deal with abstract tasks.

How to Solve Hierarchical Planning Problems?

- Via reduction, i.e., compilation to other problems like
  - SAT, i.e., Satisfiability (later in this lecture).
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).
- Search:
  - Forward progression search in the space of world state – plus the remaining task network to go thereby extending classical planning.
  - (Regression-like) search in the space of partial plans – extends POCL planning to deal with abstract tasks.
  - Local search (not covered).

High-Level Description of Example Domain

- We have a delivery domain consisting of four locations, $A, \ldots, D$.

High-Level Description of Example Domain

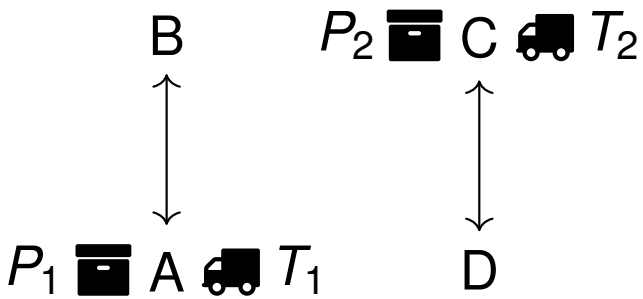
- We have a delivery domain consisting of four locations, $A, \ldots, D$.
- A can be reached from B and vice versa. Similar for C and D.

High-Level Description of Example Domain

- We have a delivery domain consisting of four locations, $A, \ldots, D$.
- A can be reached from B and vice versa. Similar for C and D.
- There are two trucks and two packages.

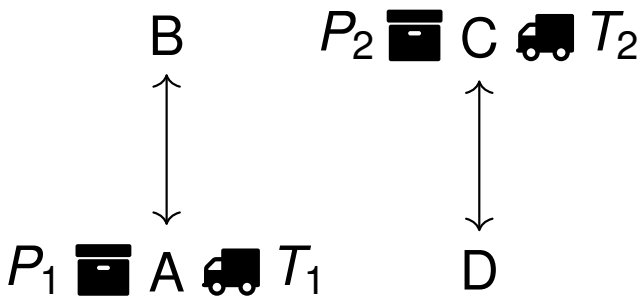
High-Level Description of Example Domain

- We have a delivery domain consisting of four locations, $A, \ldots, D$.
- A can be reached from B and vice versa. Similar for C and D.
- There are two trucks and two packages.
- Trucks can load and unload packages.



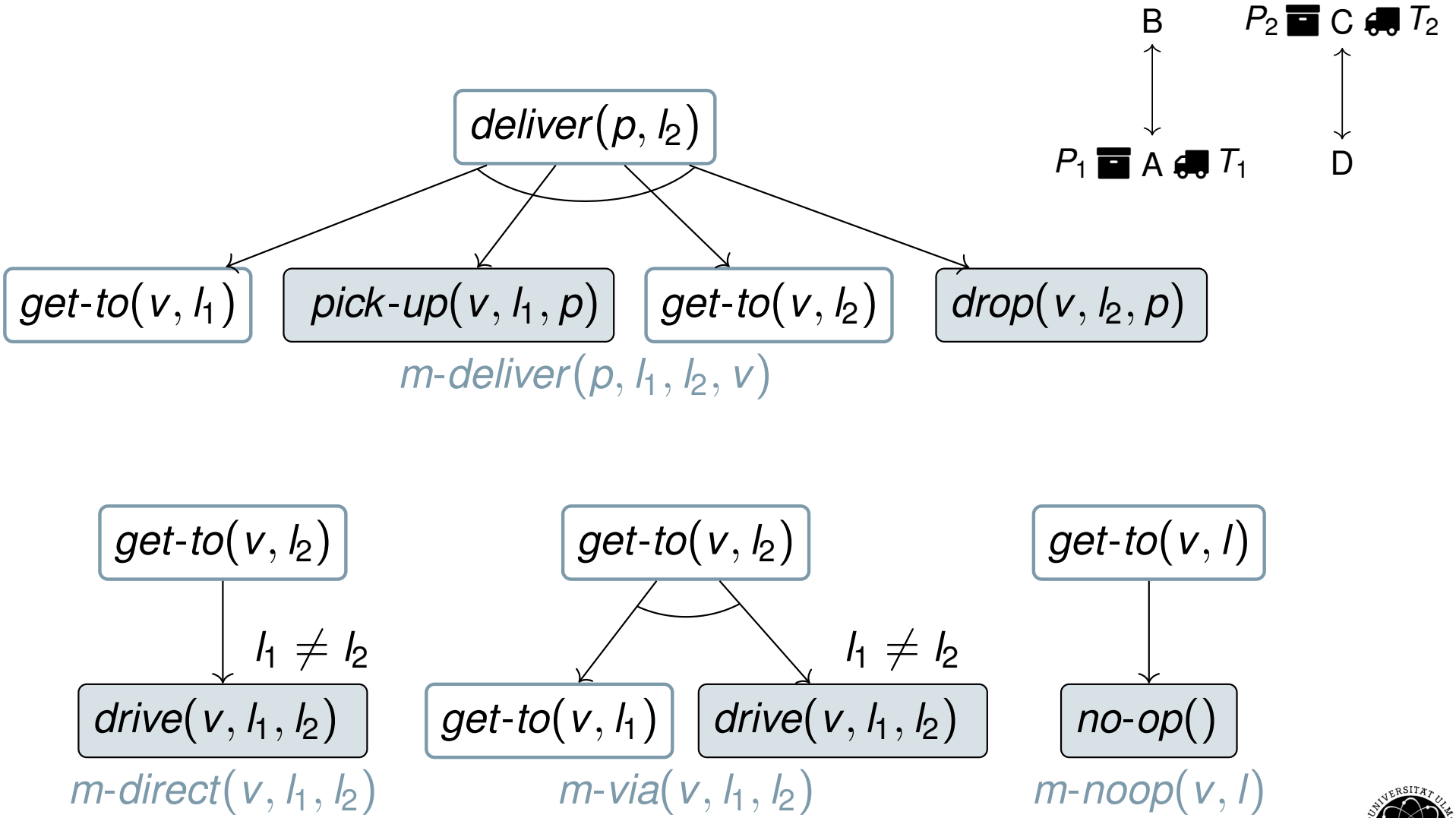
High-Level Description of Example Domain

- We have a delivery domain consisting of four locations, $A, \ldots, D$.
- A can be reached from B and vice versa. Similar for C and D.
- There are two trucks and two packages.
- Trucks can load and unload packages.

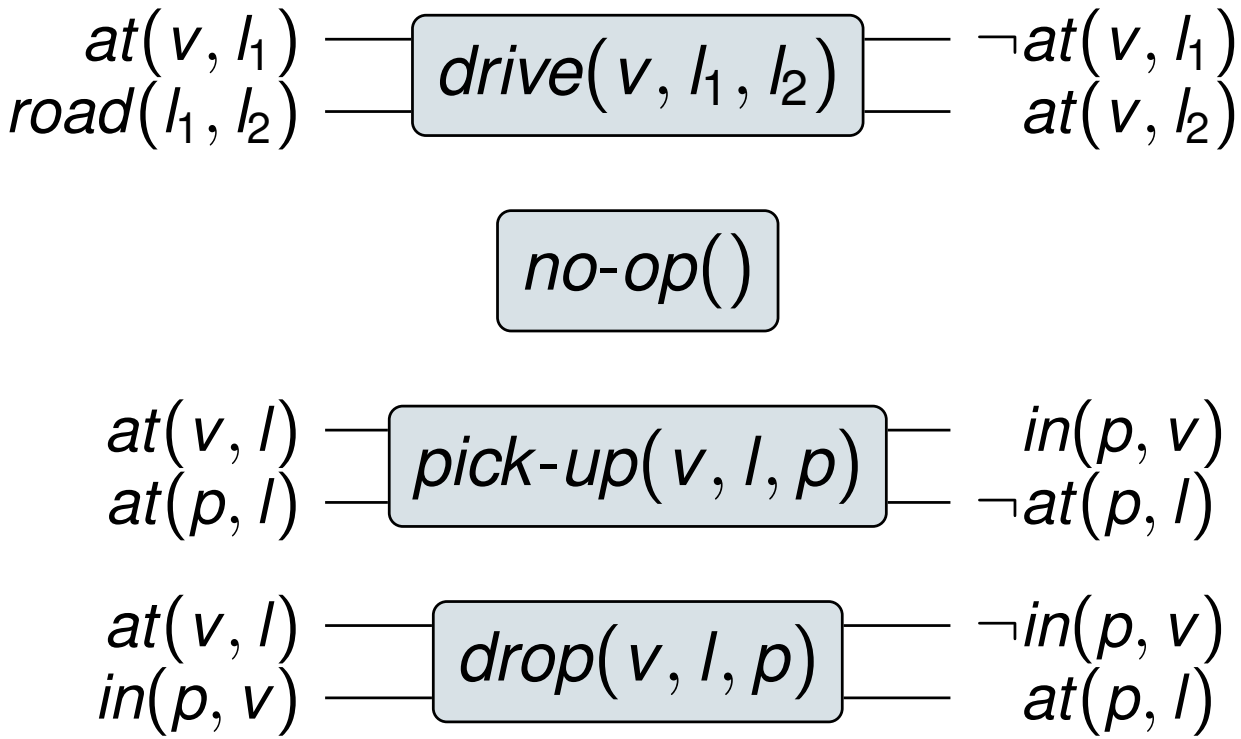


- We model the respective domain and problem as an HTN problem.

Graphical Illustration of Domain Model



Formal Action Model



Assume the following sorts/types: v – *vehicle*, l, l_1, l_2 – *location*, and p – *package*. Further assume that constants of the respective sorts/types are provided.

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.
- To perform progression, we identify the set of tasks without predecessors. Only those can get applied:

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.
- To perform progression, we identify the set of tasks without predecessors. Only those can get applied:
  - A primitive task gets applied to the current state as usual.

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.
- To perform progression, we identify the set of tasks without predecessors. Only those can get applied:
  - A primitive task gets applied to the current state as usual.
  - A compound task gets “applied” by decomposing it.

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.
- To perform progression, we identify the set of tasks without predecessors. Only those can get applied:
  - A primitive task gets applied to the current state as usual.
  - A compound task gets “applied” by decomposing it.
- When are we done? What are the termination criteria?

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.
- To perform progression, we identify the set of tasks without predecessors. Only those can get applied:
  - A primitive task gets applied to the current state as usual.
  - A compound task gets "applied" by decomposing it.
- When are we done? What are the termination criteria?
  $\rightarrow$ The current task network is empty!

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.
- To perform progression, we identify the set of tasks without predecessors. Only those can get applied:
  - A primitive task gets applied to the current state as usual.
  - A compound task gets "applied" by decomposing it.
- When are we done? What are the termination criteria?
  $\rightarrow$ The current task network is empty!
- Thus, progression HTN planning produces totally ordered solutions!
  Reminder: Technically they are not even solutions. Why?

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.
- To perform progression, we identify the set of tasks without predecessors. Only those can get applied:
  - A primitive task gets applied to the current state as usual.
  - A compound task gets "applied" by decomposing it.
- When are we done? What are the termination criteria?
  $\rightarrow$ The current task network is empty!
- Thus, progression HTN planning produces totally ordered solutions!
  Reminder: Technically they are not even solutions. Why?
  $\rightarrow$ In the general case, these totally ordered action sequences can not be obtained via decomposition. They are *witnesses* of solutions, though.

Introduction

HTN progression search behaves similar to classical planning, but performs both search in the space of states *and* in the space of *task networks*:

- We maintain a *current state*, starting with the initial state.
- In addition, maintain a *current task network*, starting with the initial one.
- To perform progression, we identify the set of tasks without predecessors. Only those can get applied:
  - A primitive task gets applied to the current state as usual.
  - A compound task gets "applied" by decomposing it.
- When are we done? What are the termination criteria?
  $\rightarrow$ The current task network is empty!
- Thus, progression HTN planning produces totally ordered solutions!
  Reminder: Technically they are not even solutions. Why?
  $\rightarrow$ In the general case, these totally ordered action sequences can not be obtained via decomposition. They are *witnesses* of solutions, though.
- Note: The standard progression algorithm, SHOP2, relies on *preconditions of methods*. (We only discuss this briefly here.)

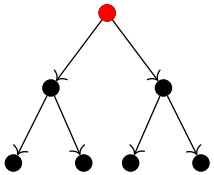
HTN Progression, Pseudo Code

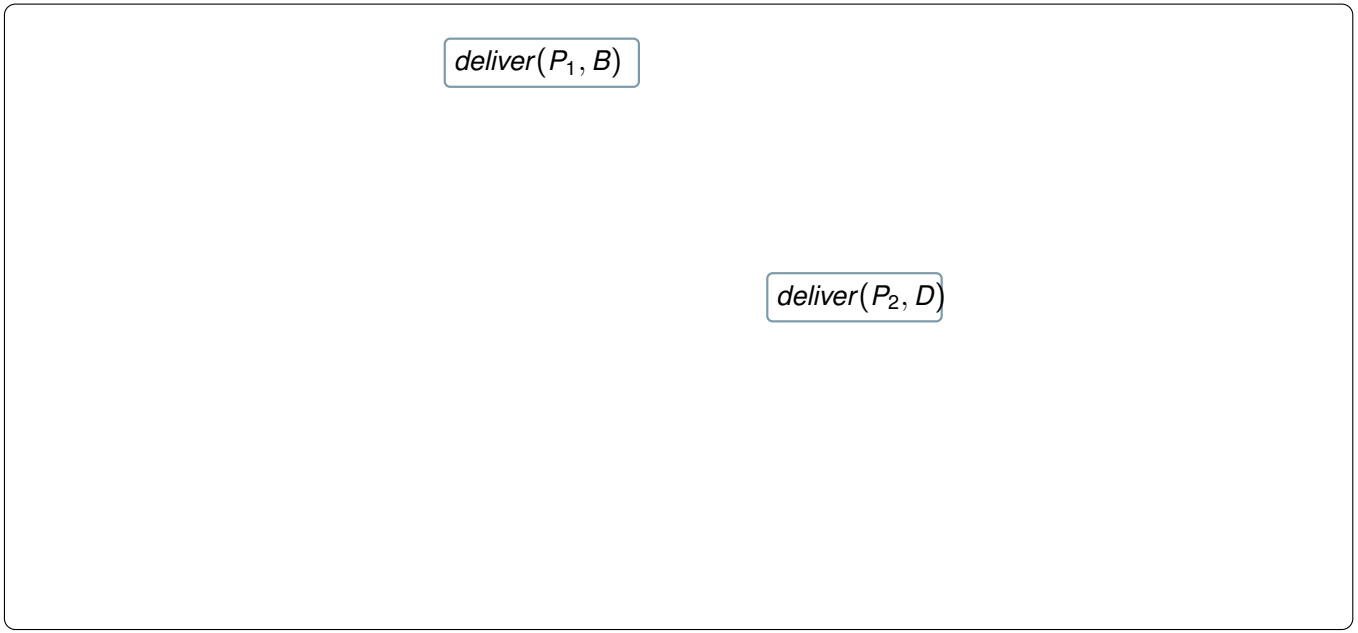
Algorithm: HTN Progression Search

Input: An HTN problem $\mathcal{P} = (V, P, \delta, C, M, s_I, tn_I)$
Output: A solution $\bar{a}$ or ***fail*** if none exists

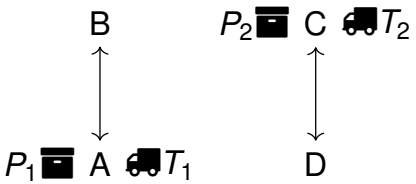
```
1  fringe ← {(s_I, tn_I, ε)}
2  while fringe ≠ ∅ do
3      n = (s, tn, ā) ← nodeSelectAndRemove(fringe)
4      if tn is empty then
5          return ā
6      else
7          U ← detectUnconstrainedSteps(tn)
8          for t ∈ U do
9              if isPrimitive(t) and pre(t) ⊆ s then
10                 fringe ← fringe ∪ {n.apply(t)}
11             else if isCompound(t) then
12                 fringe ← fringe ∪ {n.decompose(t, m) |
                                      m ∈ M with m = (α(t), tn_m)}
13 return fail
```

HTN Progression, Example

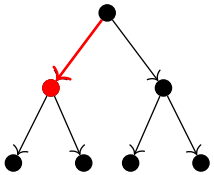


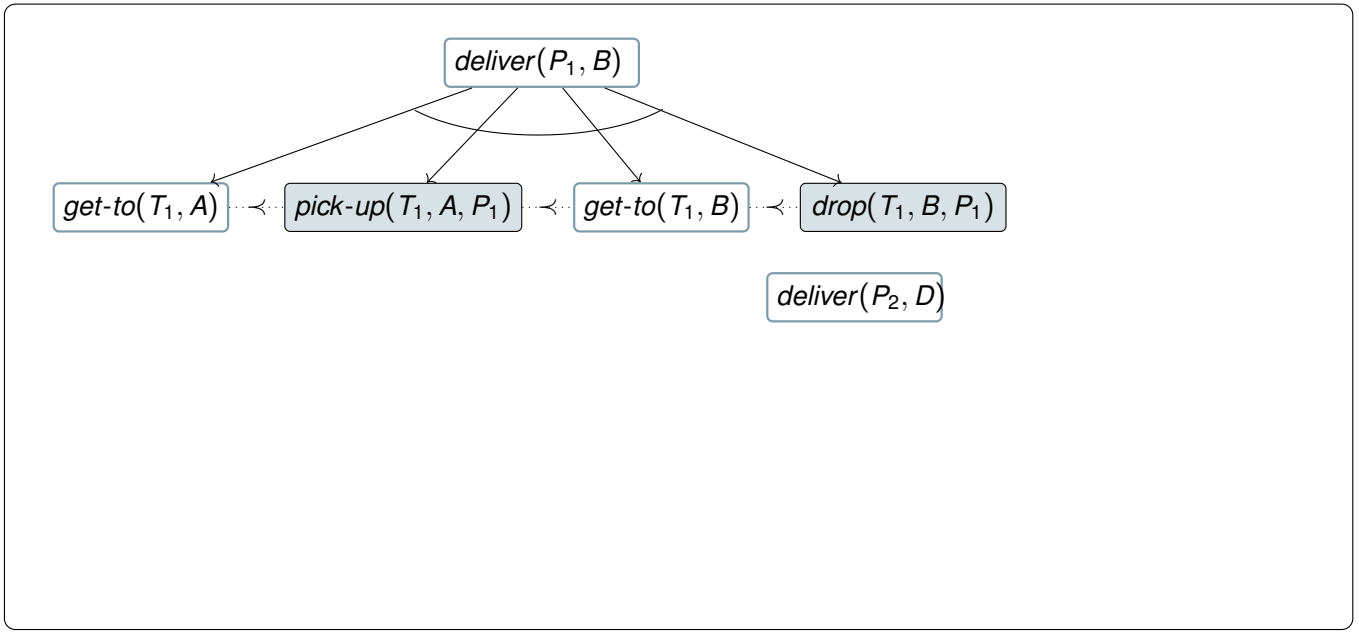


$\pi = ()$

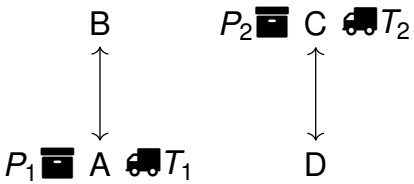


HTN Progression, Example

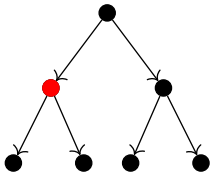


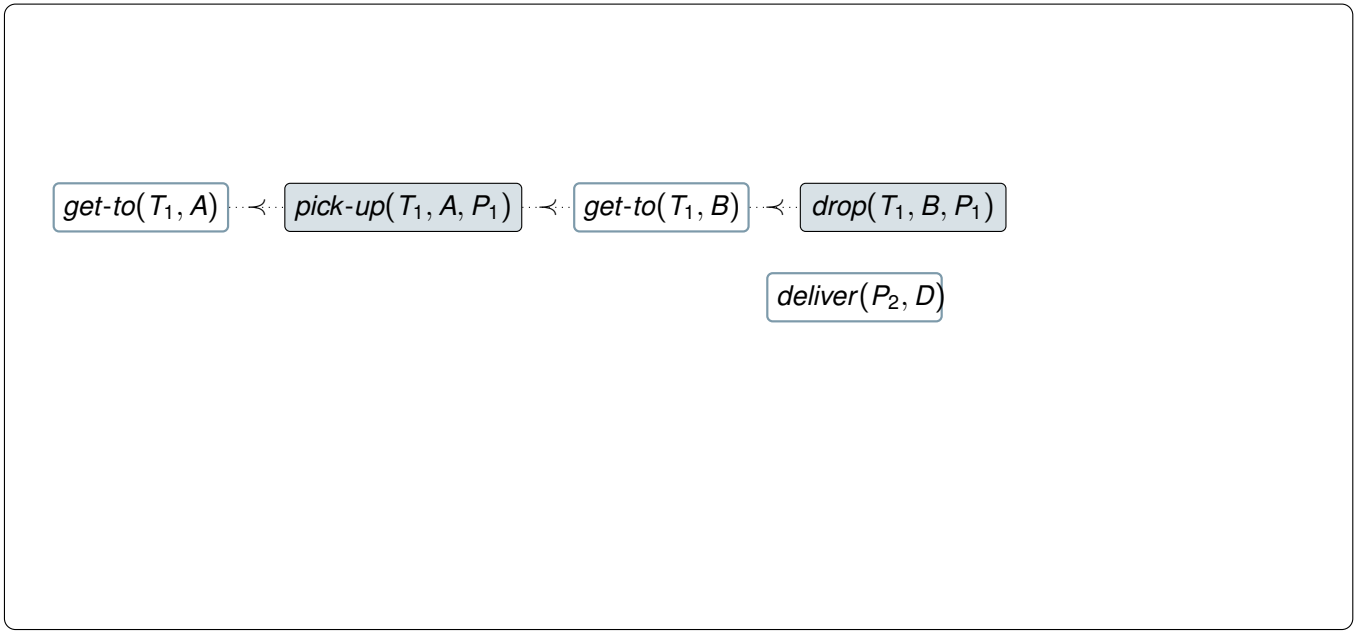


$\pi = ()$

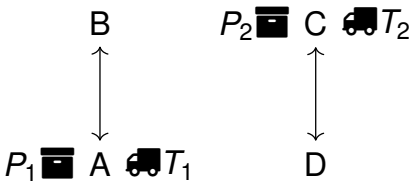


HTN Progression, Example

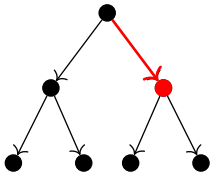


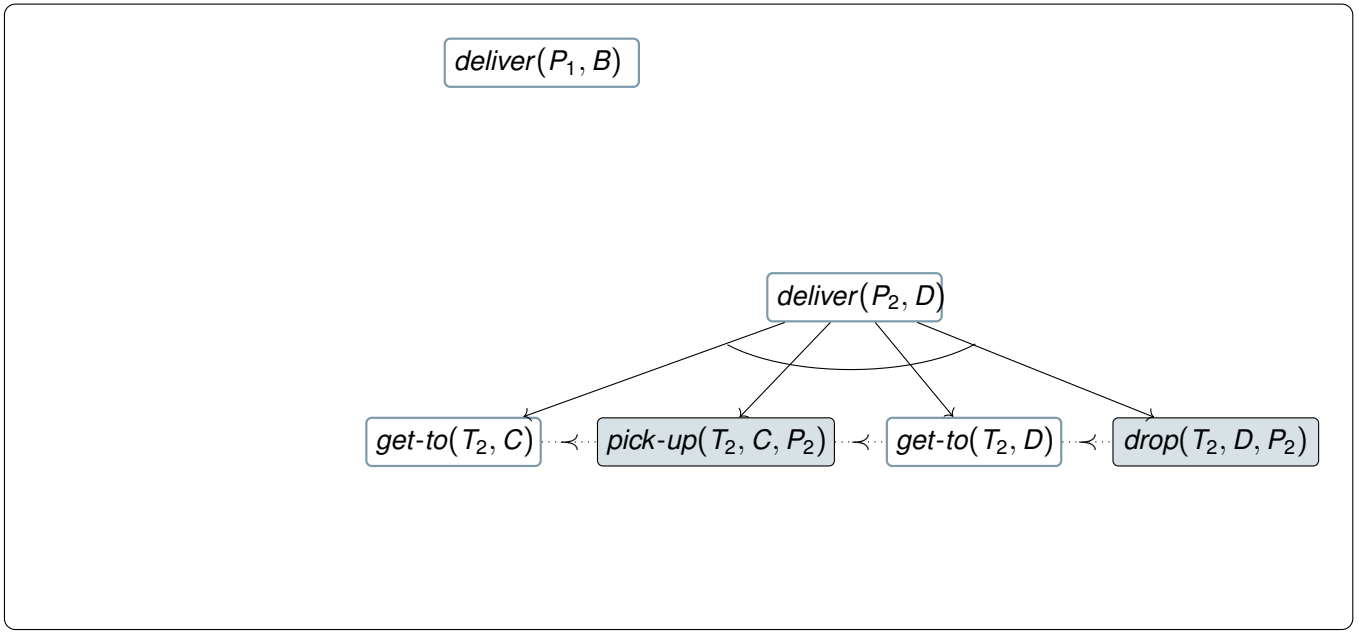


$\pi = ()$

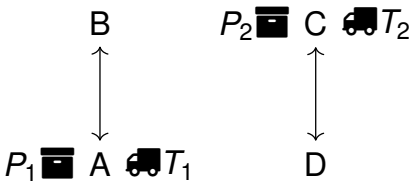


HTN Progression, Example

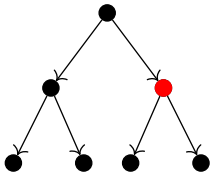


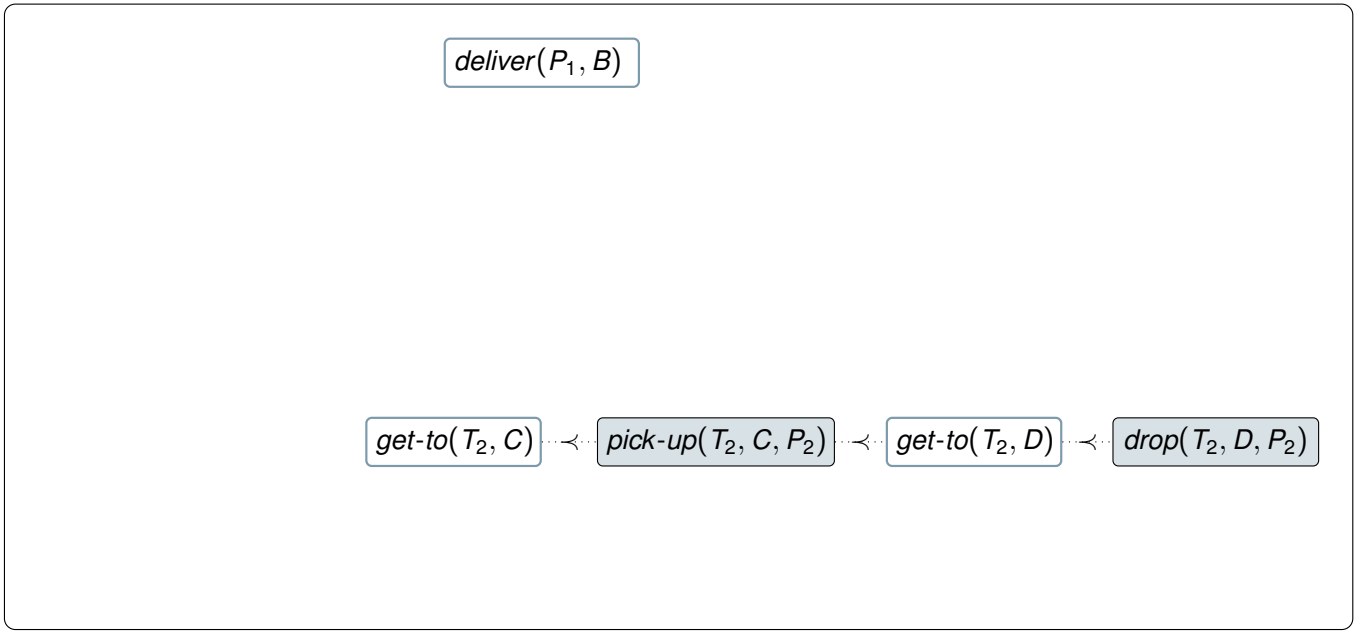


$\pi = ()$

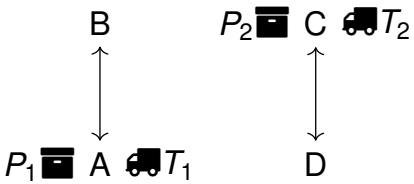


HTN Progression, Example

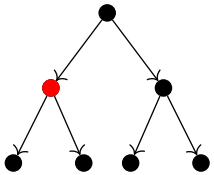


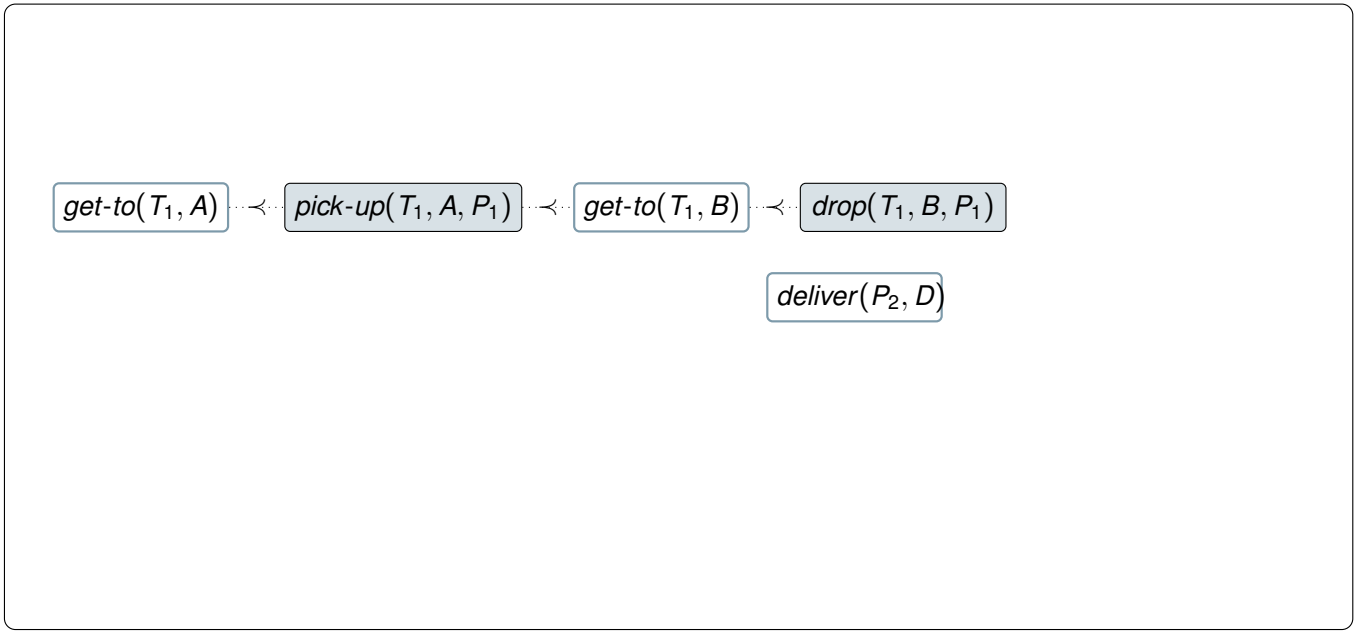


$\pi = ()$

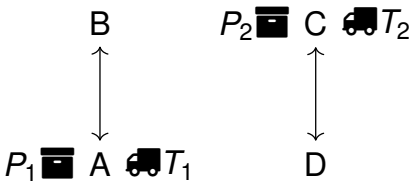


HTN Progression, Example

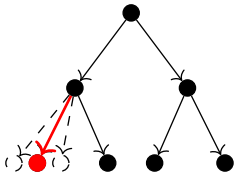


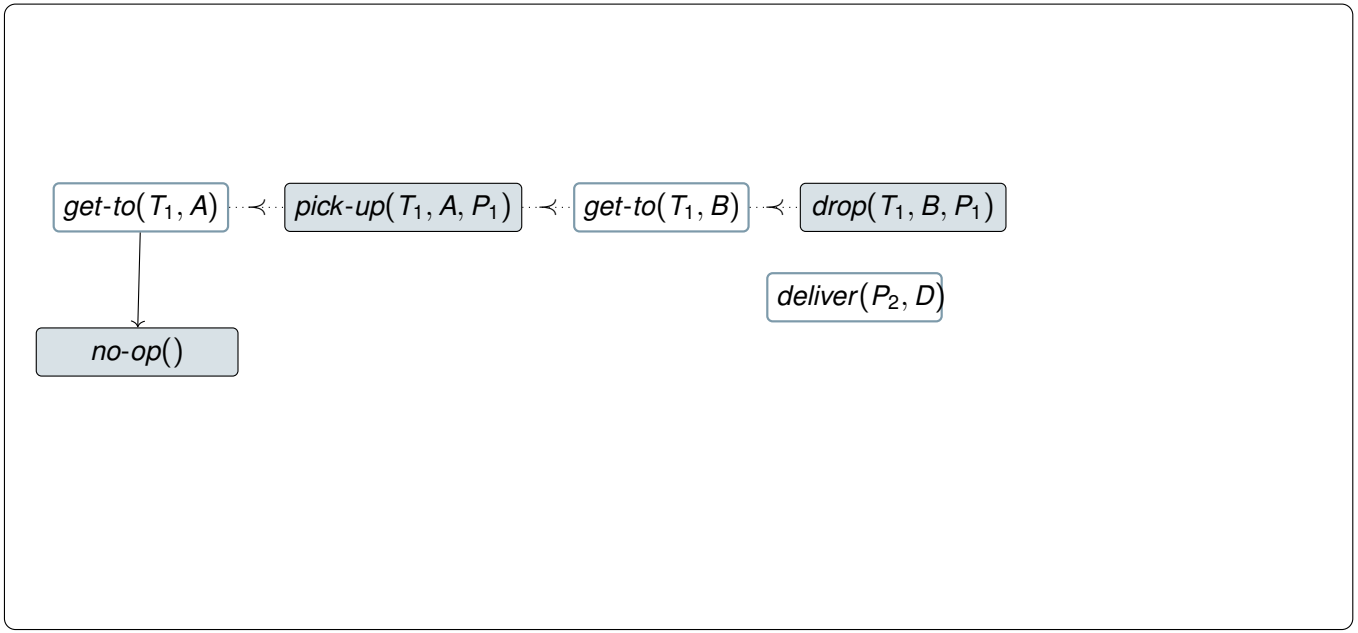


$\pi = ()$

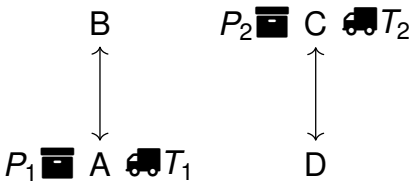


HTN Progression, Example

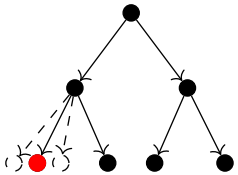


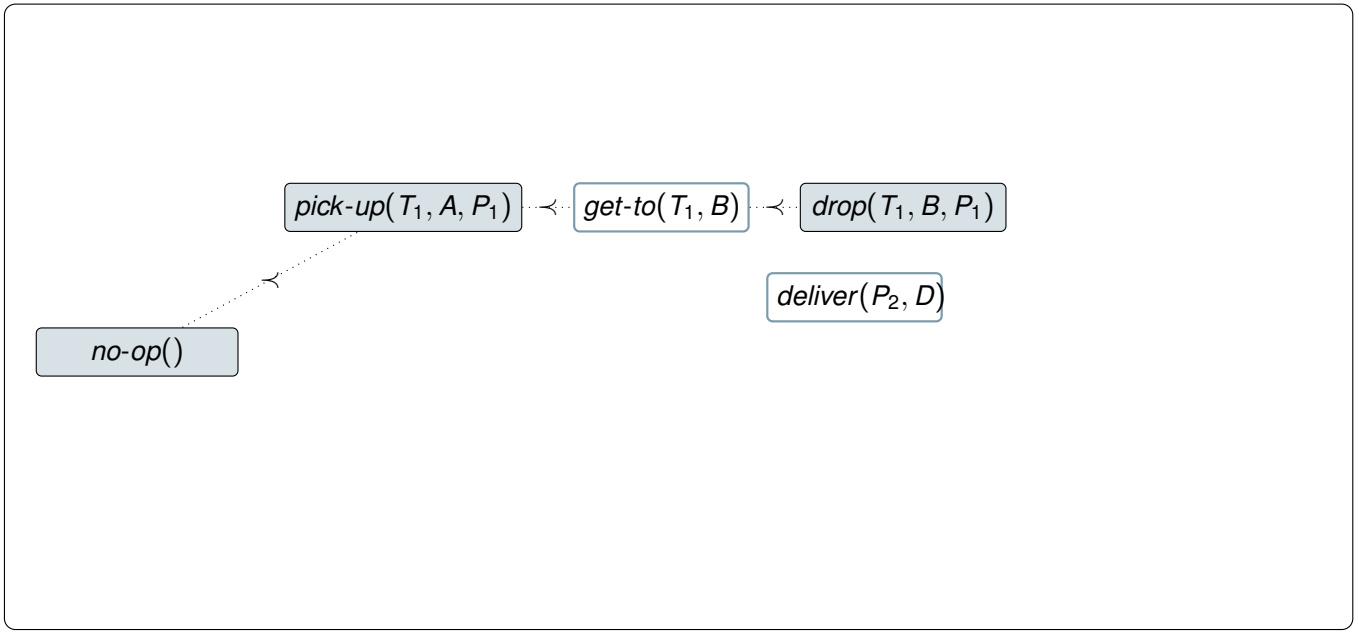


$\pi = ()$

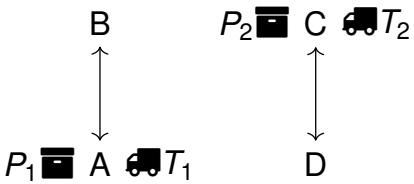


HTN Progression, Example

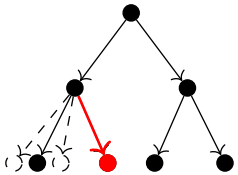


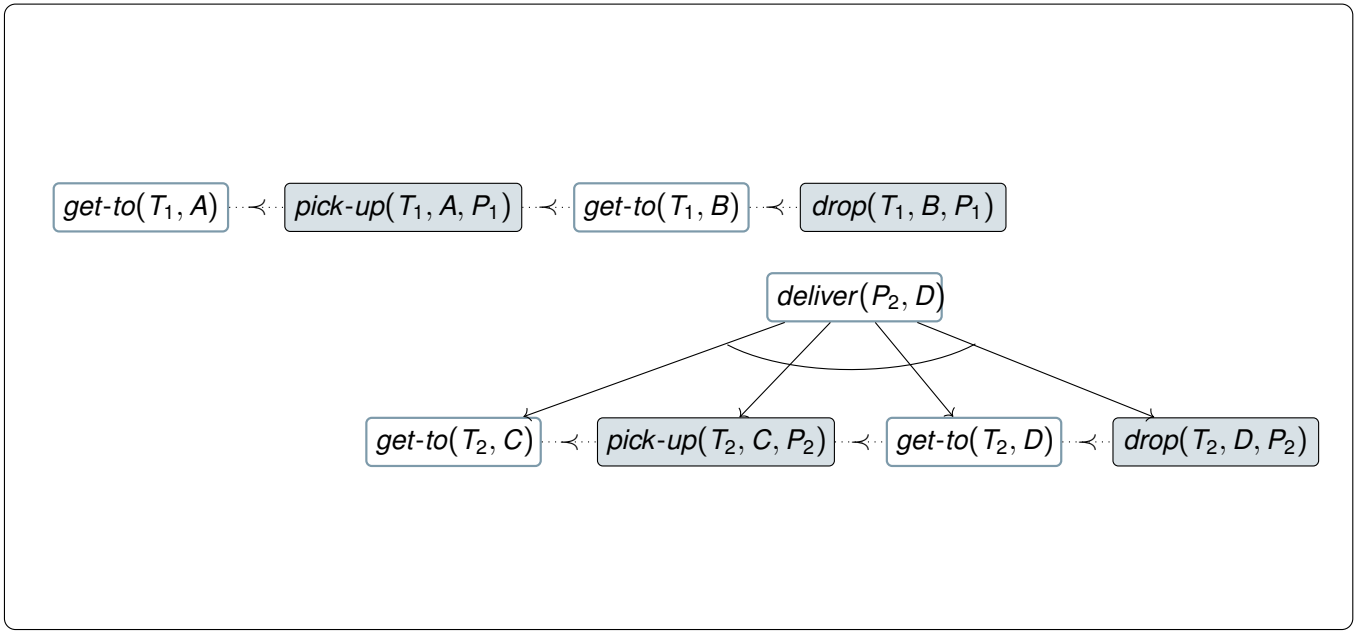


$\pi = ()$

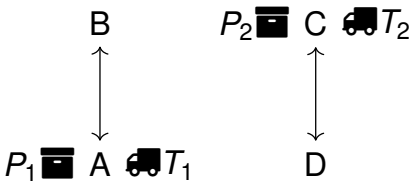


HTN Progression, Example

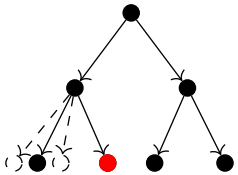


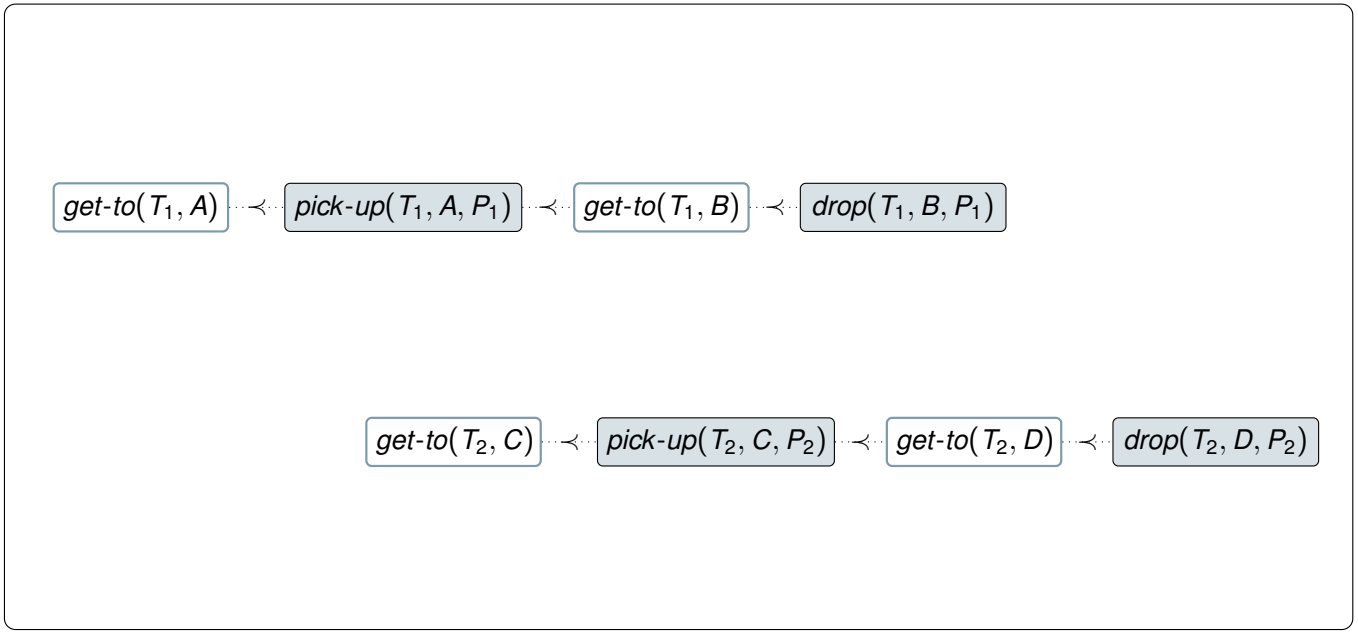


$\pi = ()$

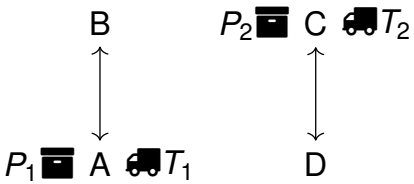


HTN Progression, Example

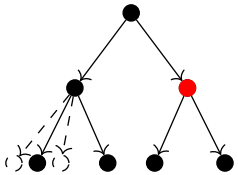


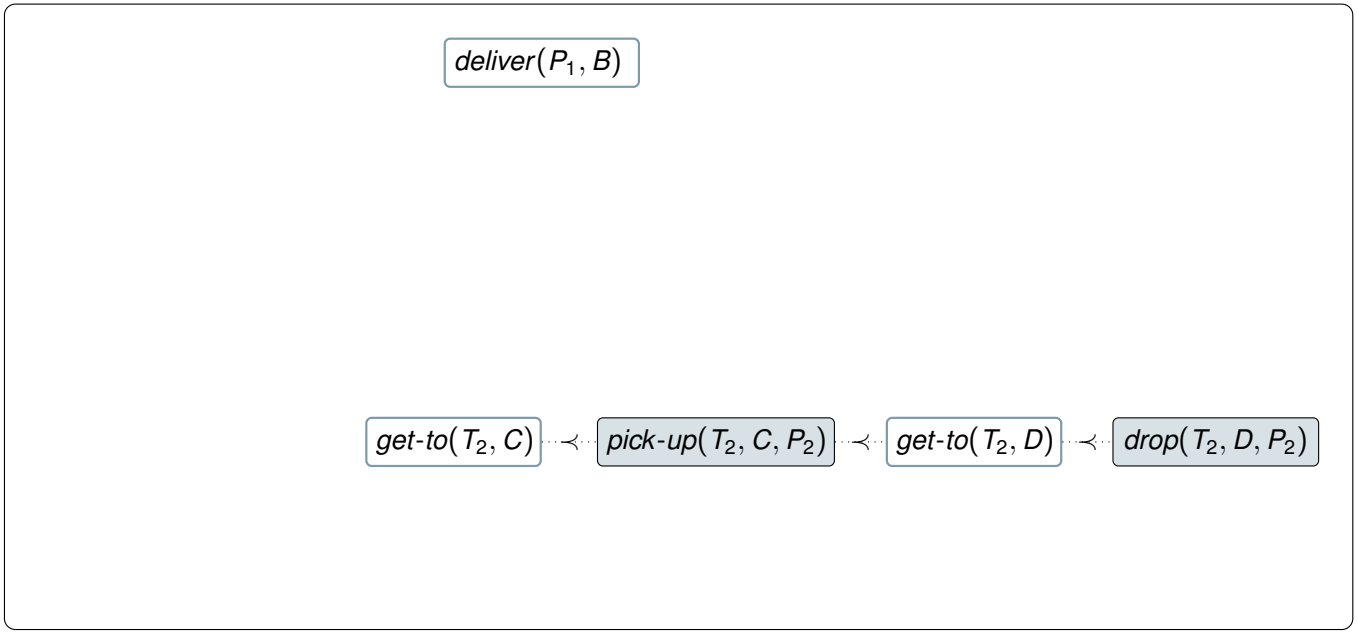


$\pi = ()$

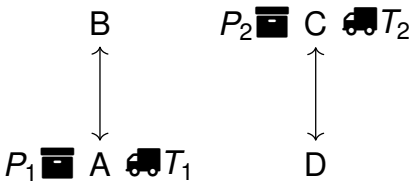


HTN Progression, Example

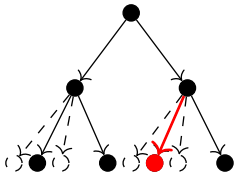


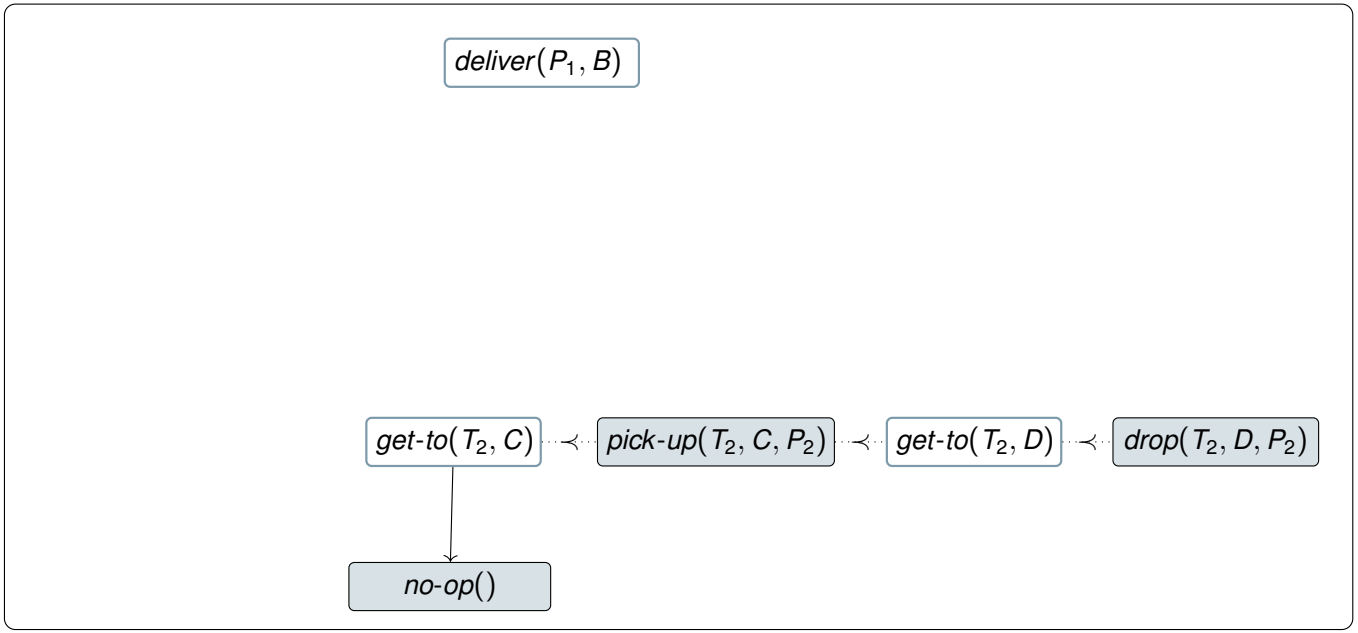


$\pi = ()$

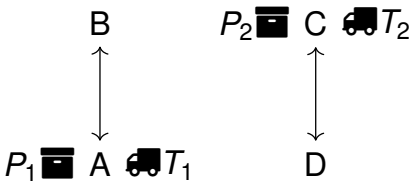


HTN Progression, Example

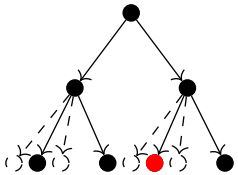


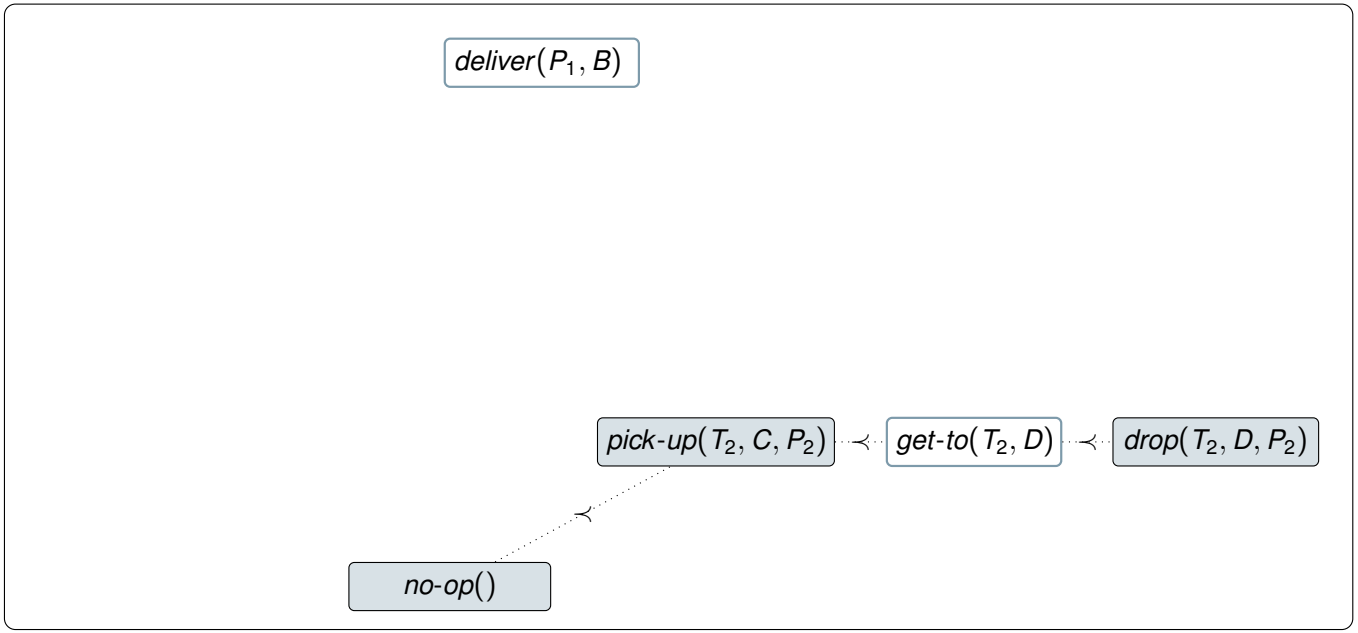


$\pi = ()$

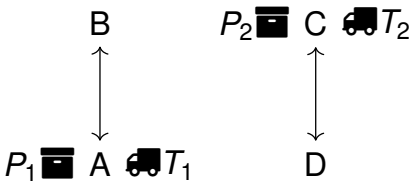


HTN Progression, Example

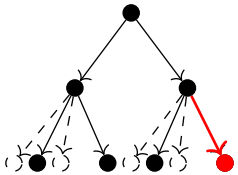


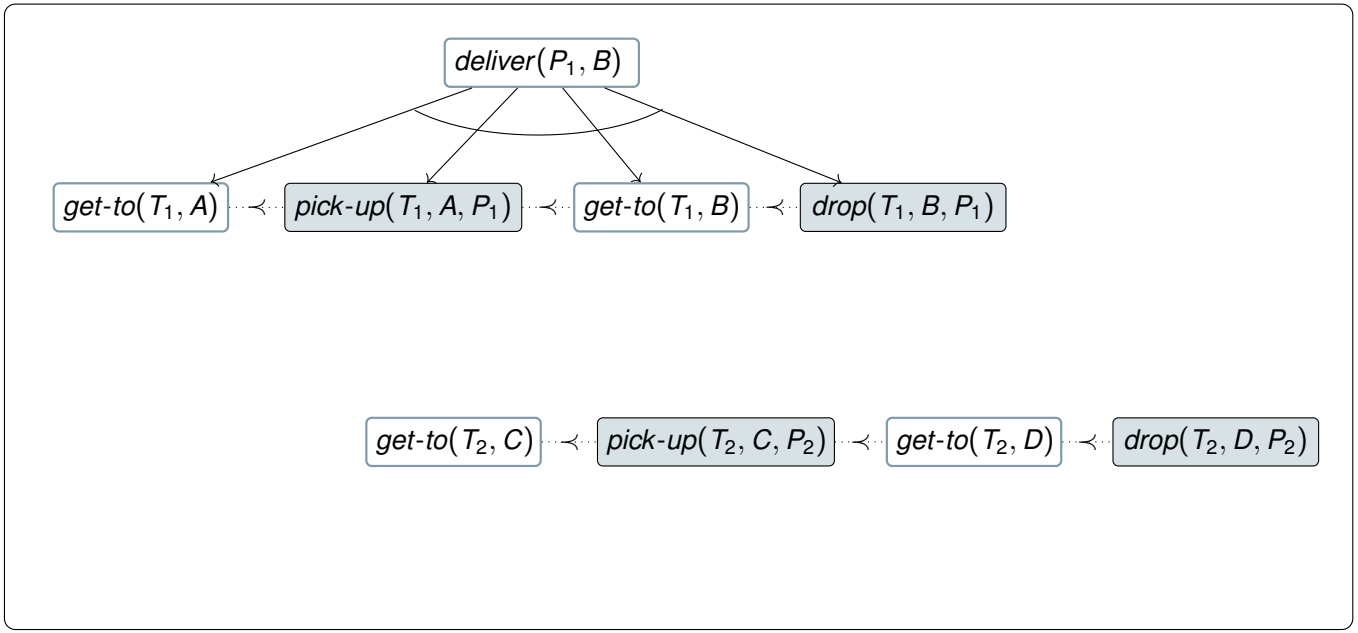


$\pi = ()$

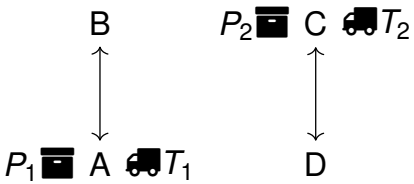


HTN Progression, Example

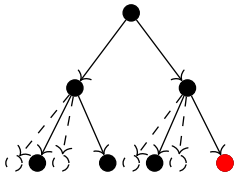


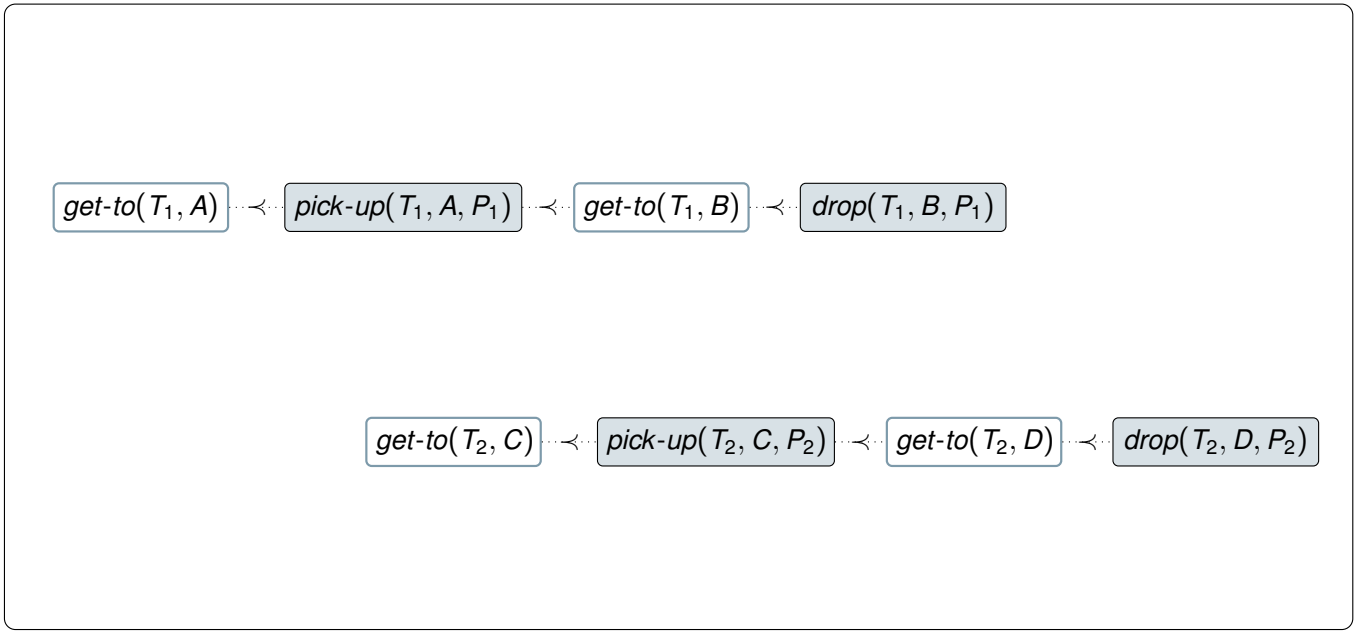


$\pi = ()$

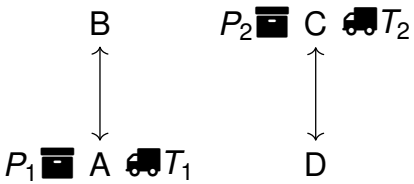


HTN Progression, Example

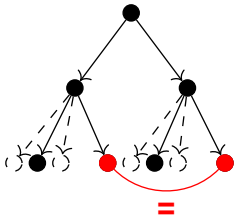


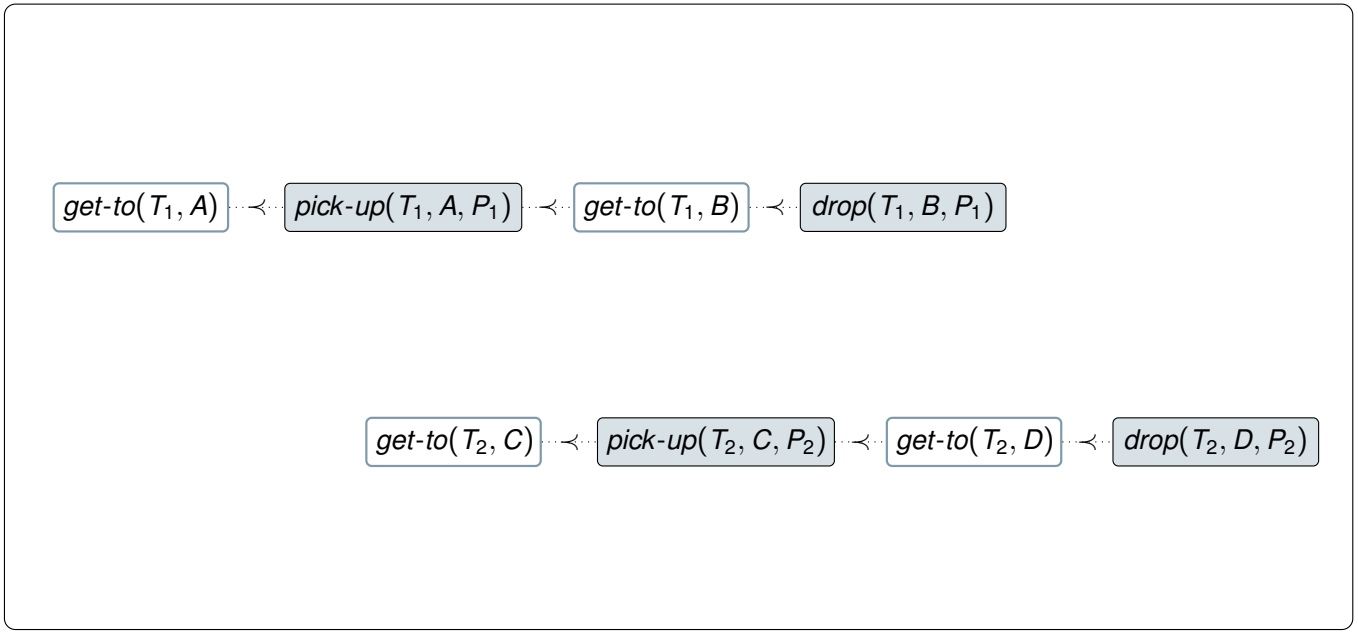


$\pi = ()$

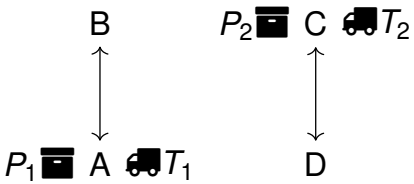


HTN Progression, Example

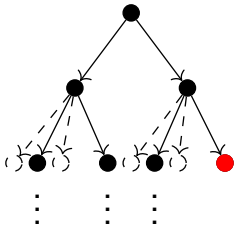


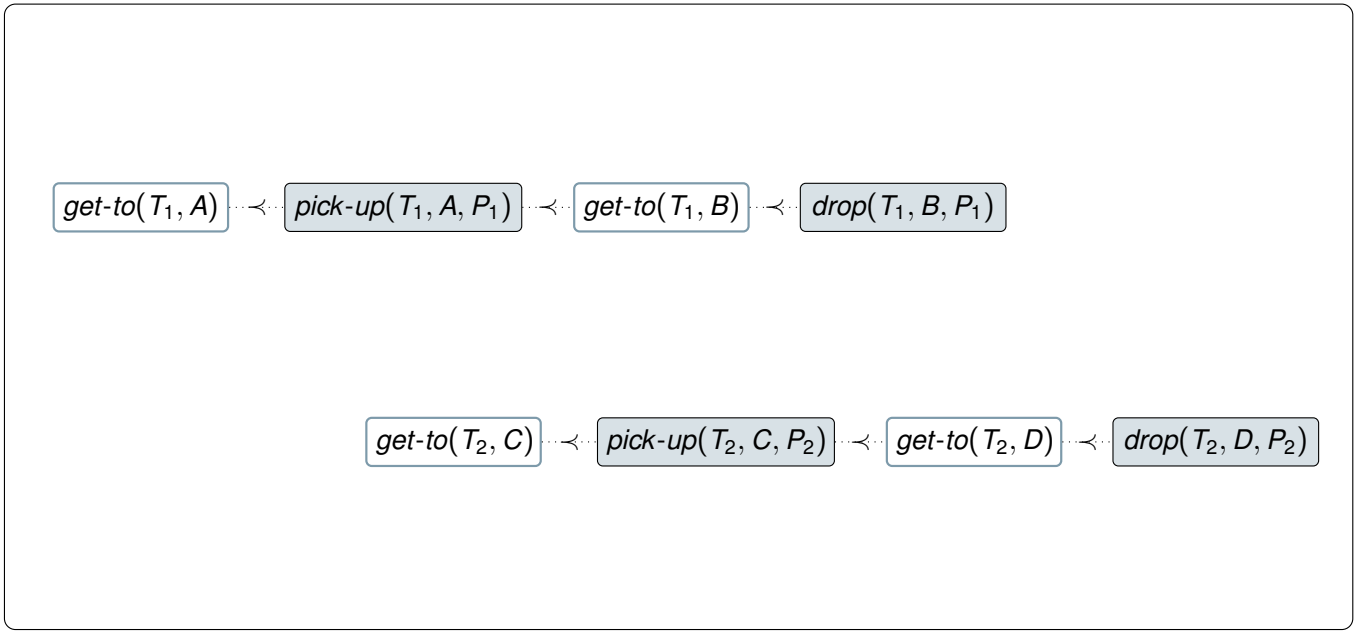


$\pi = ()$

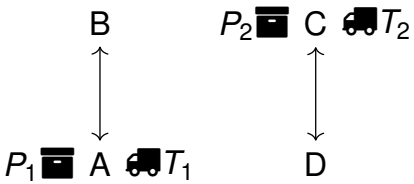


HTN Progression, Example

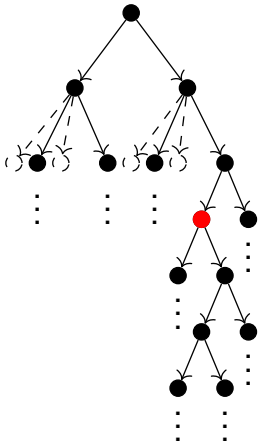


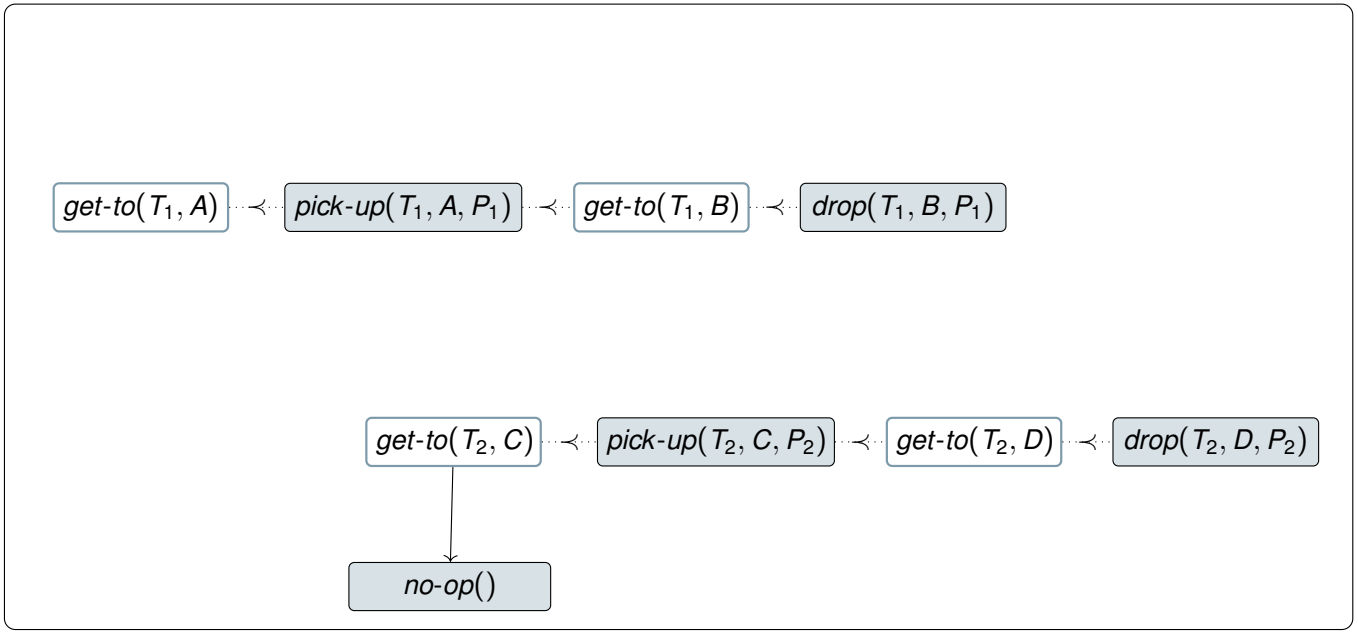


$\pi = ()$

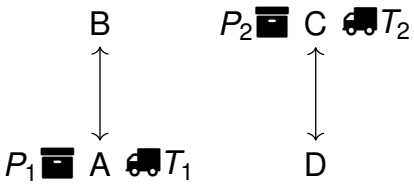


HTN Progression, Example

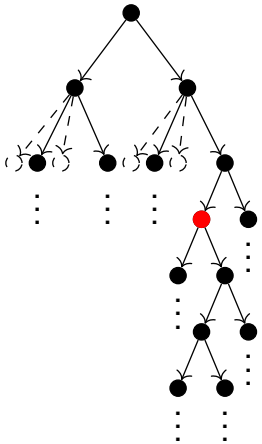


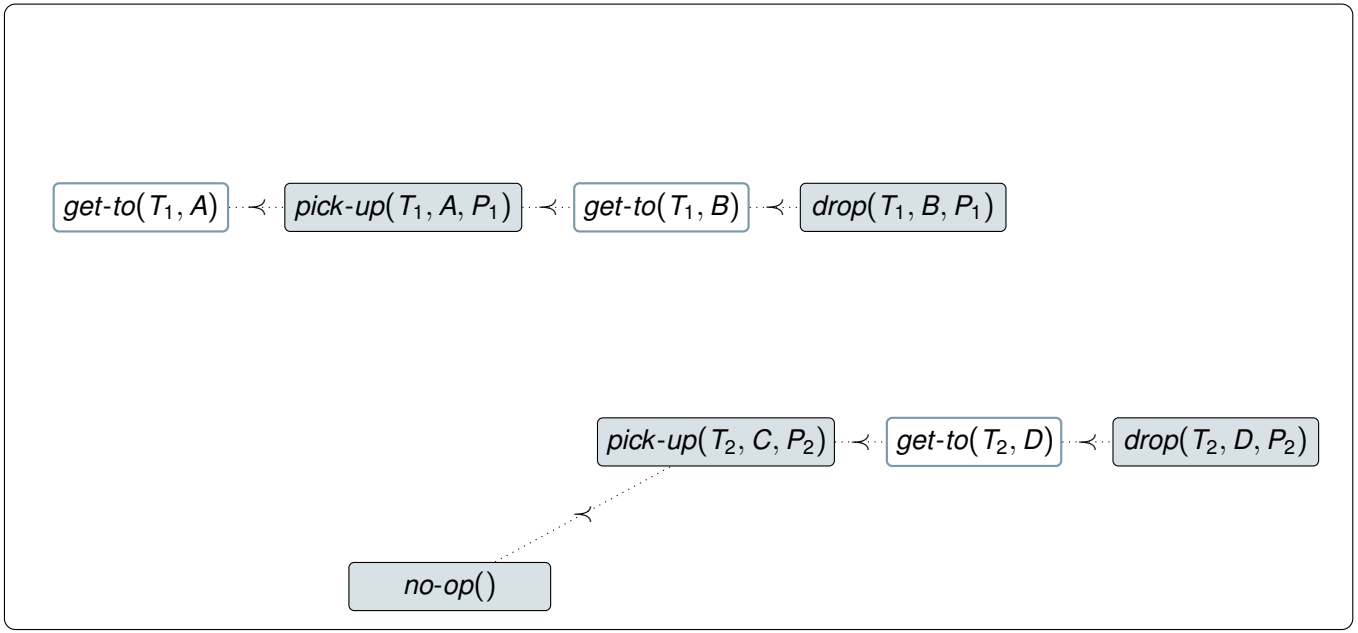


$\pi = ()$

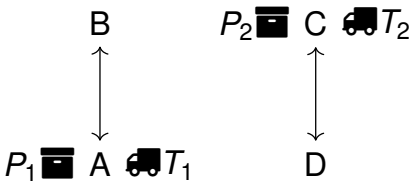


HTN Progression, Example

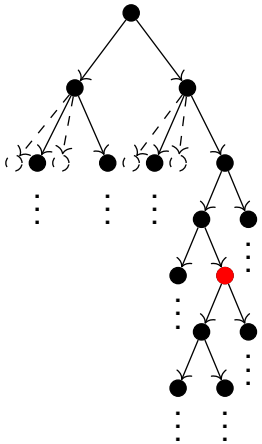


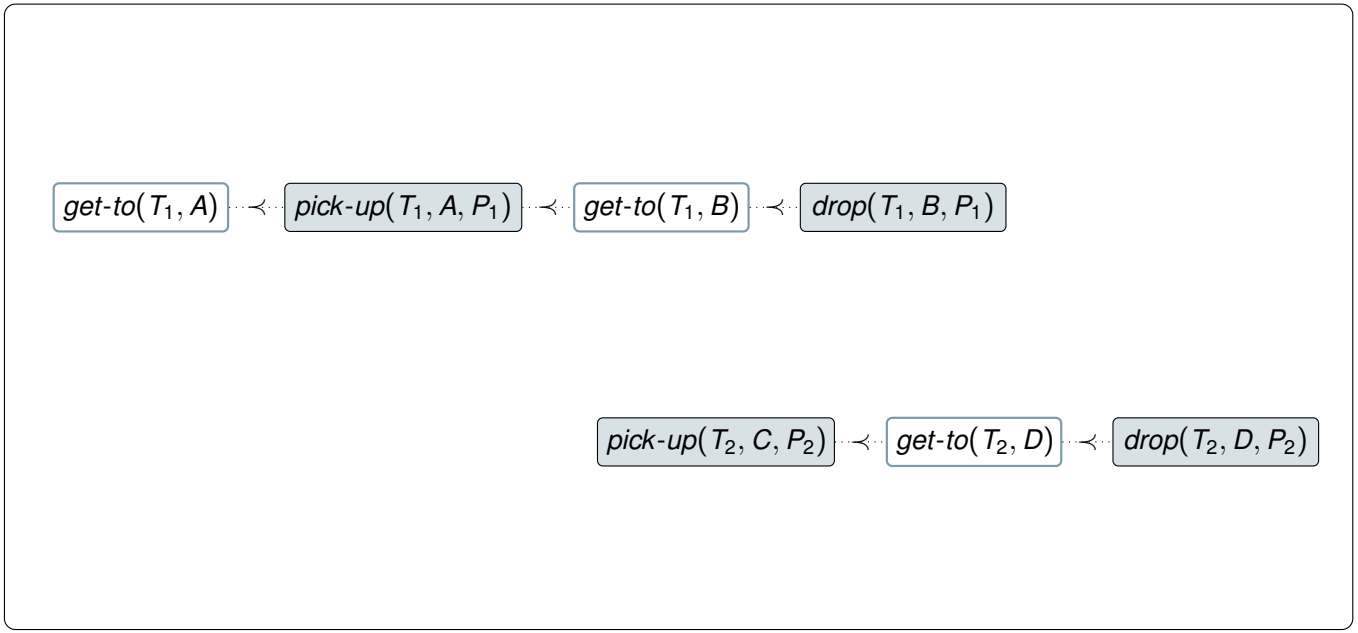


$\pi = ()$

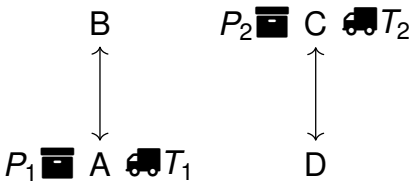


HTN Progression, Example

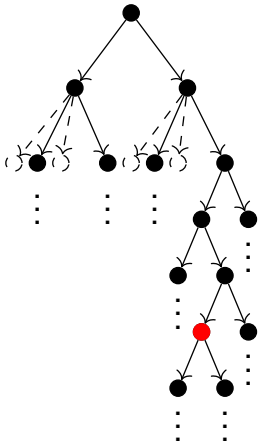


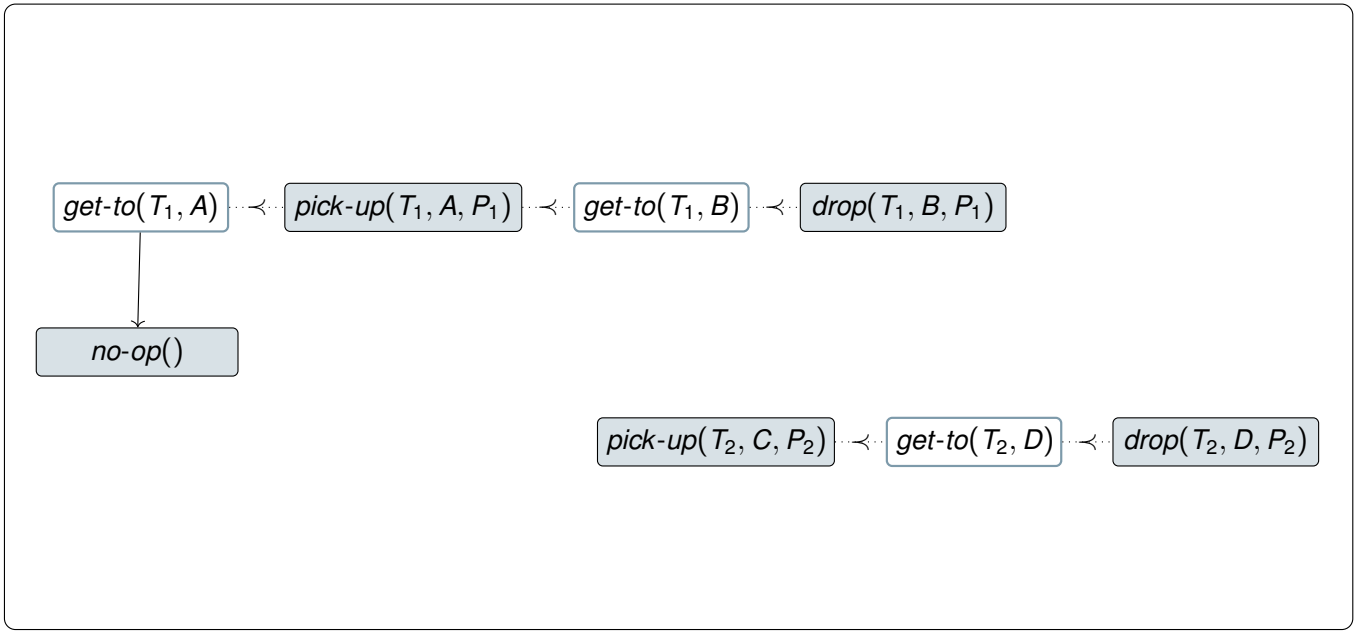


$\pi = (\textit{no-op}())$

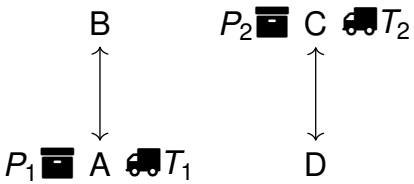


HTN Progression, Example

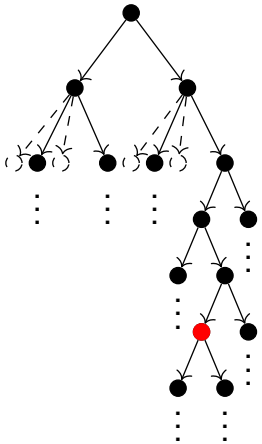


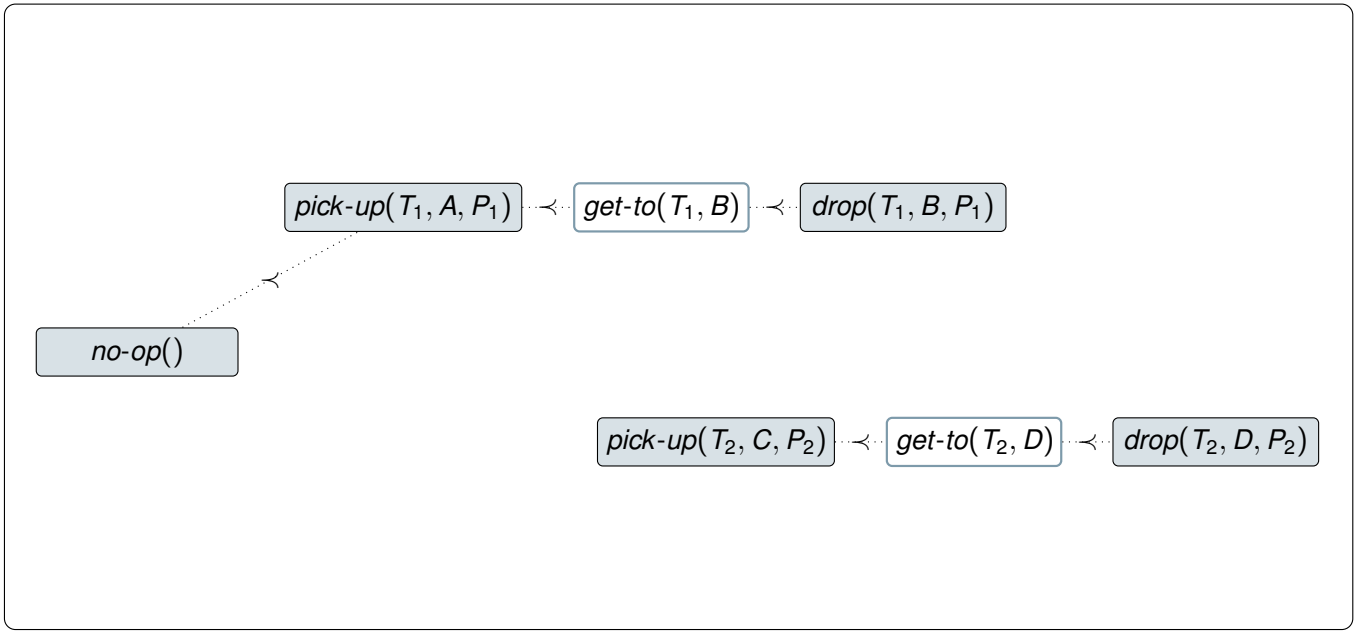


$\pi = (\textit{no-op}())$

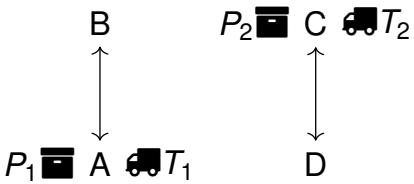


HTN Progression, Example

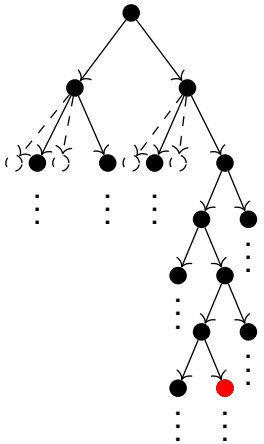


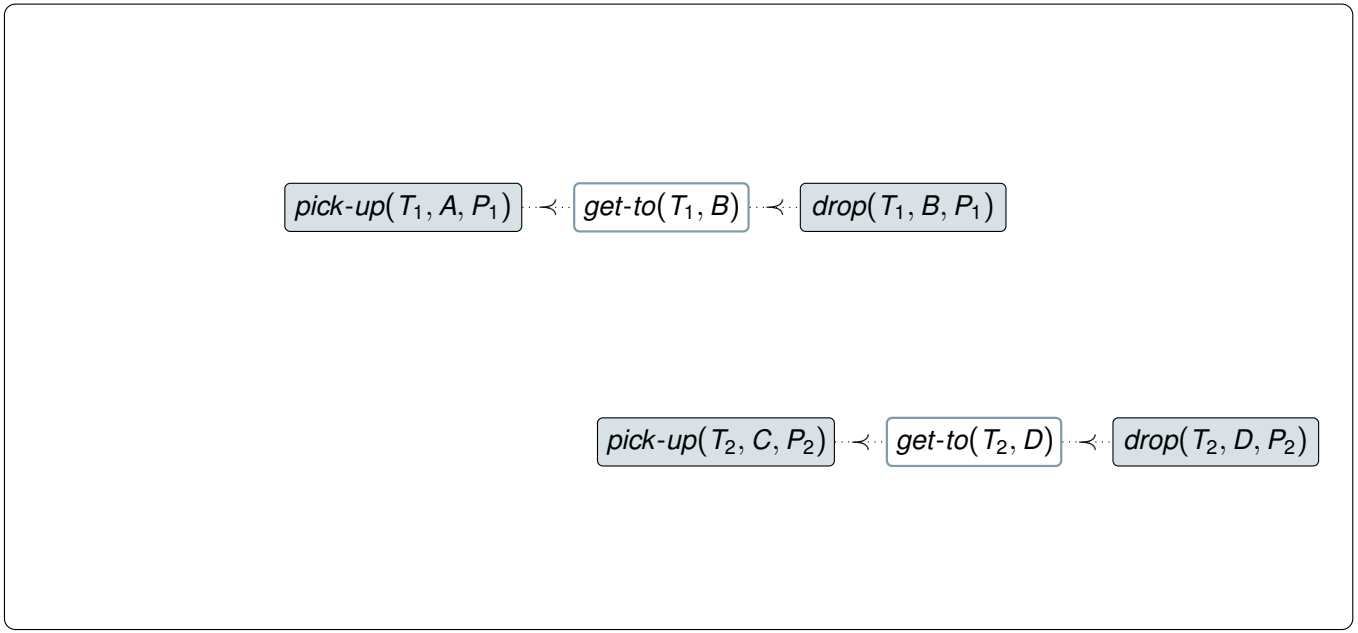


$\pi = (\textit{no-op}())$

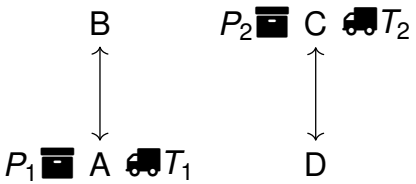


HTN Progression, Example

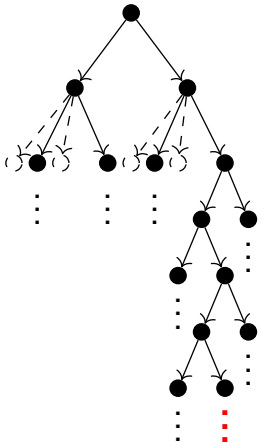


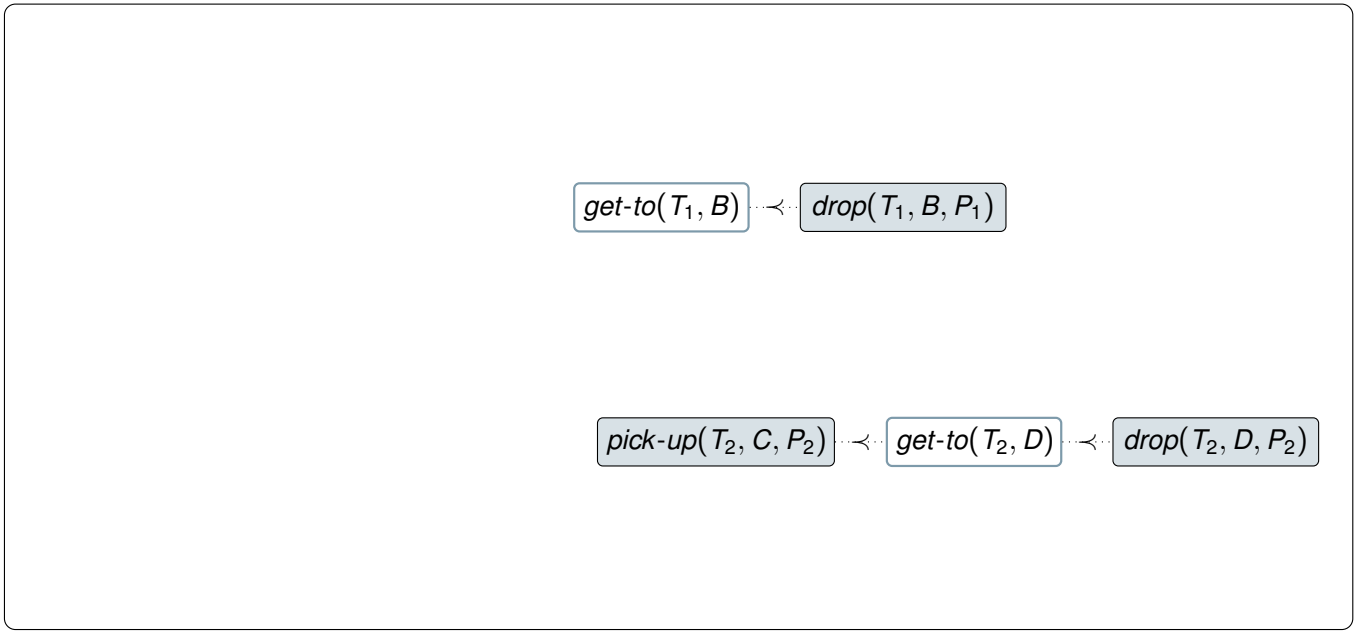


$$\pi = (\textit{no-op}(), \textit{no-op}())$$

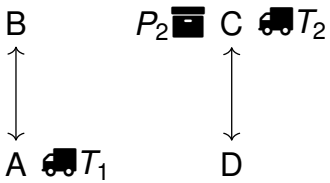


HTN Progression, Example

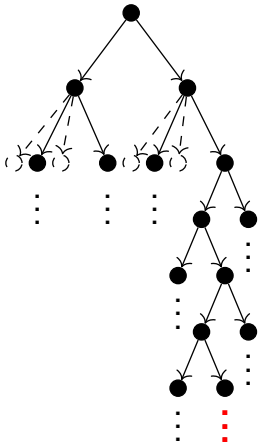


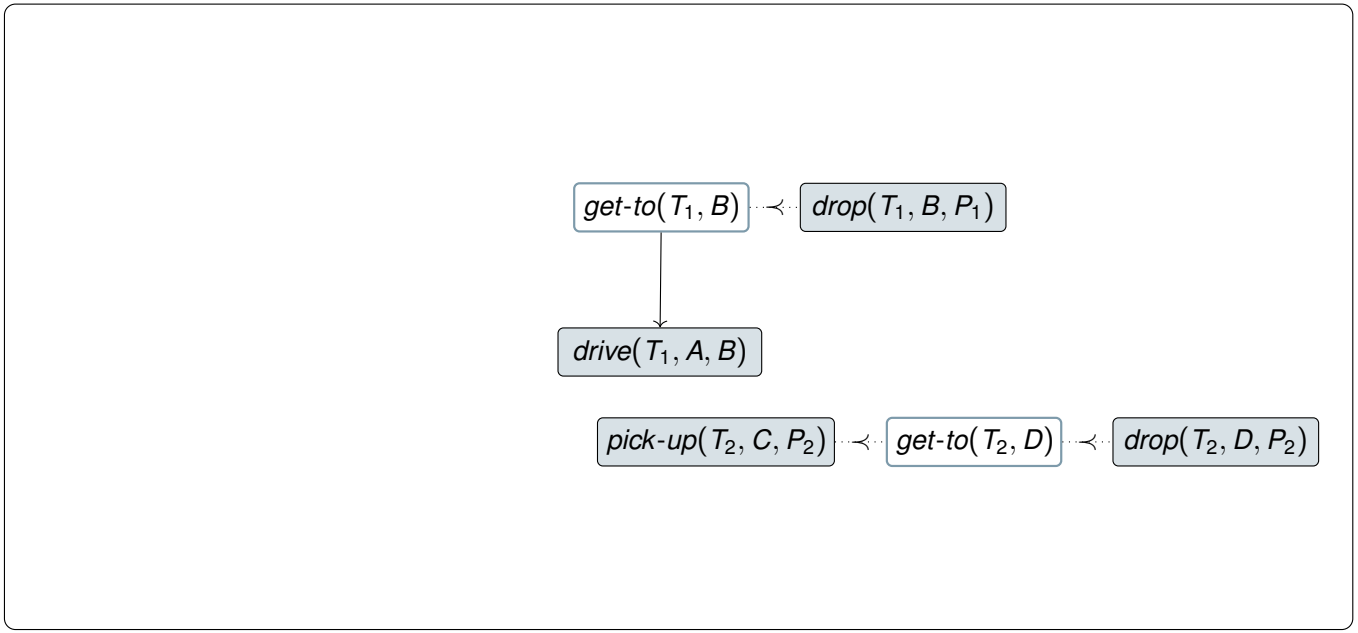


$$\pi = (\textit{no-op}(), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1))$$

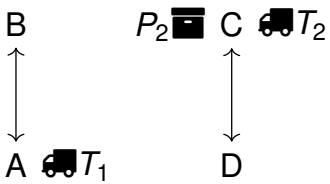


HTN Progression, Example

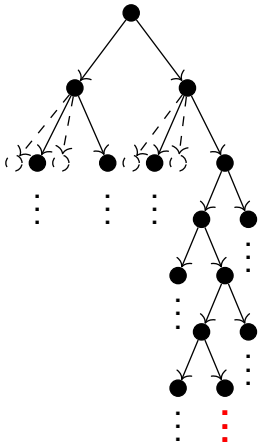


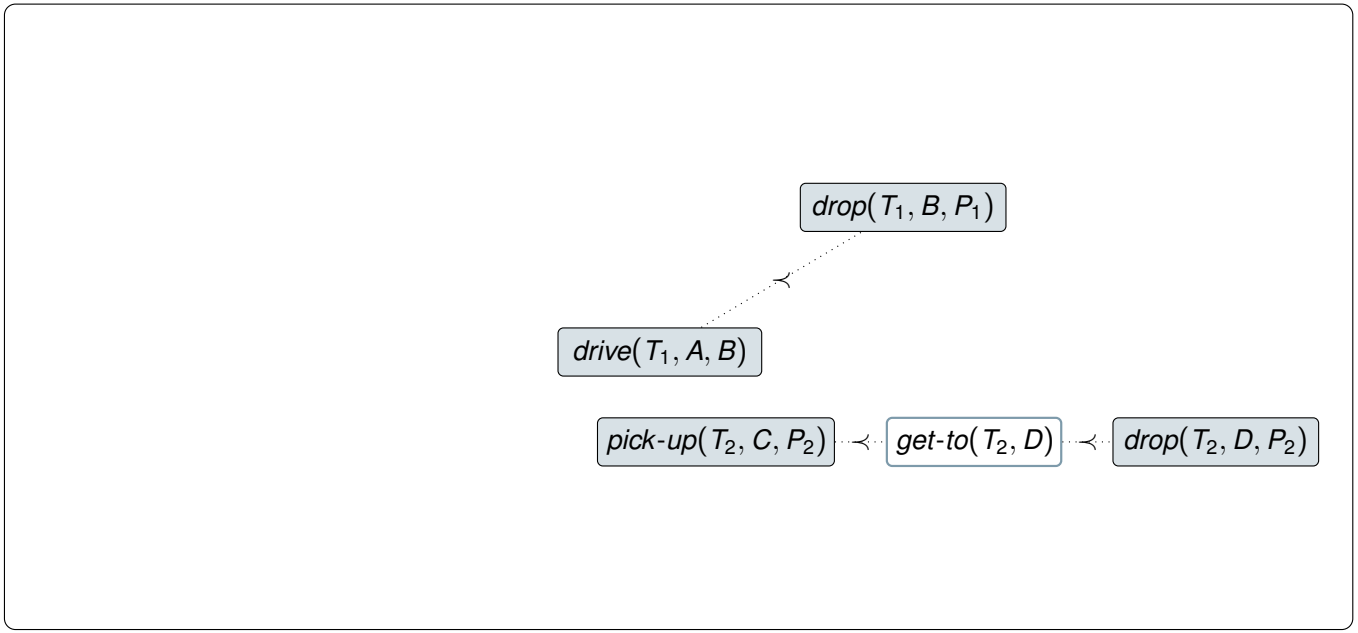


$$\pi = (\textit{no-op}(), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1))$$

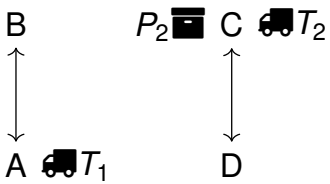


HTN Progression, Example

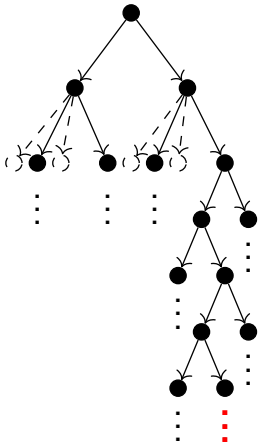


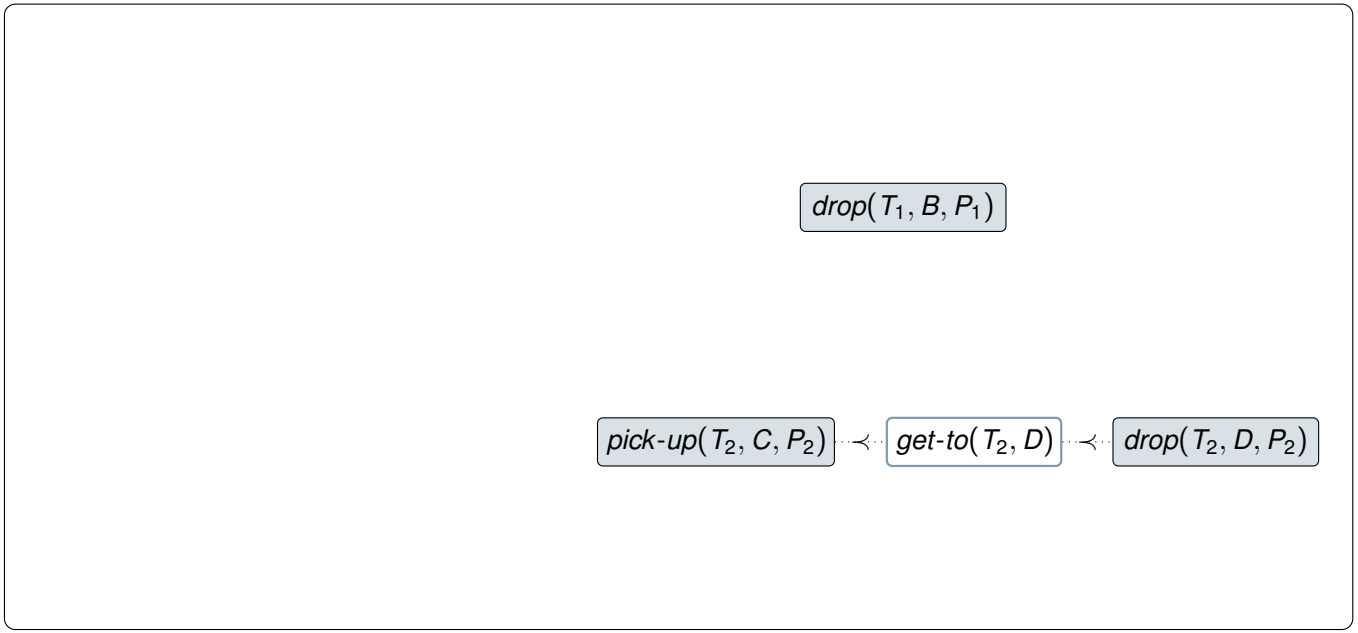


$$\pi = (\textit{no-op}(), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1))$$

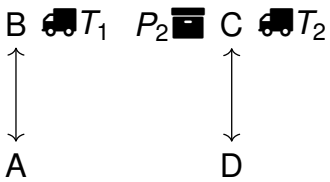


HTN Progression, Example

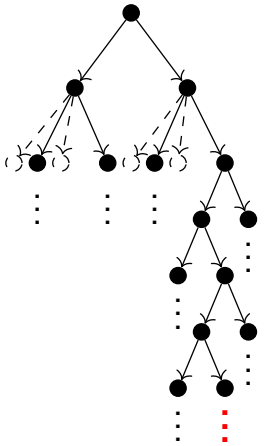


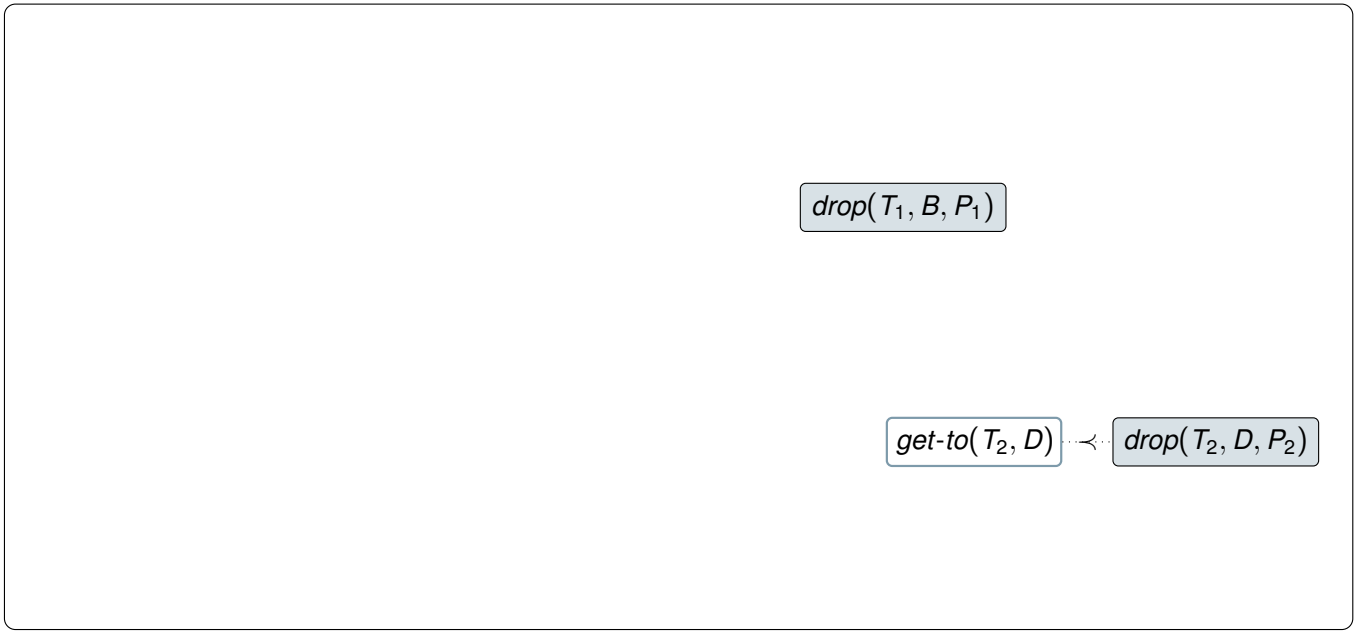


$$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drive(T_1, A, B))$$

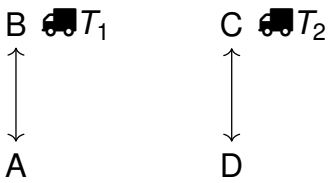


HTN Progression, Example

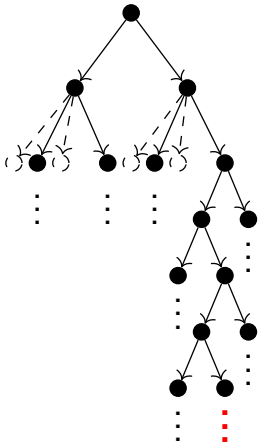


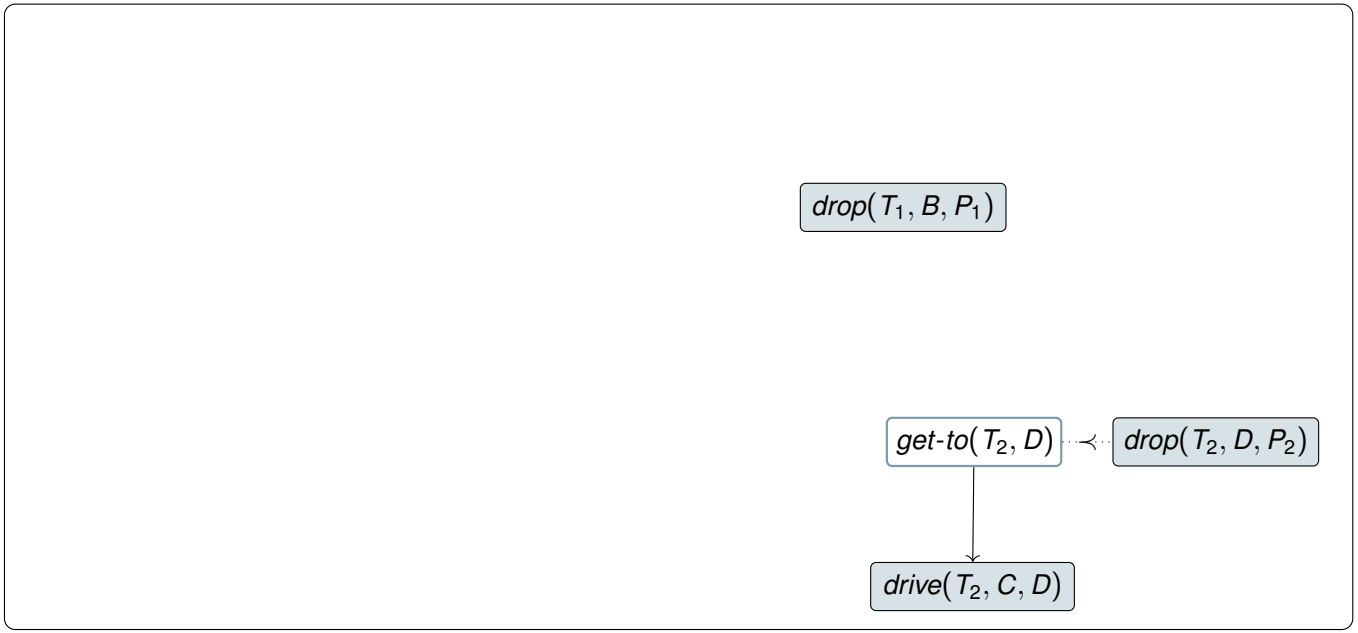


$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drive(T_1, A, B), pick\text{-}up(T_2, C, P_2))$

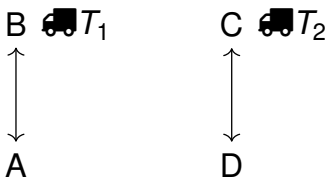


HTN Progression, Example

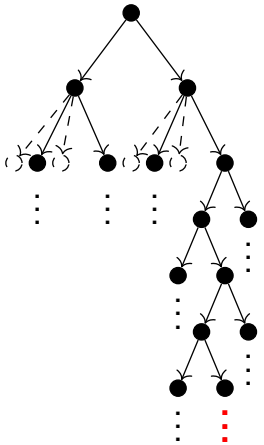


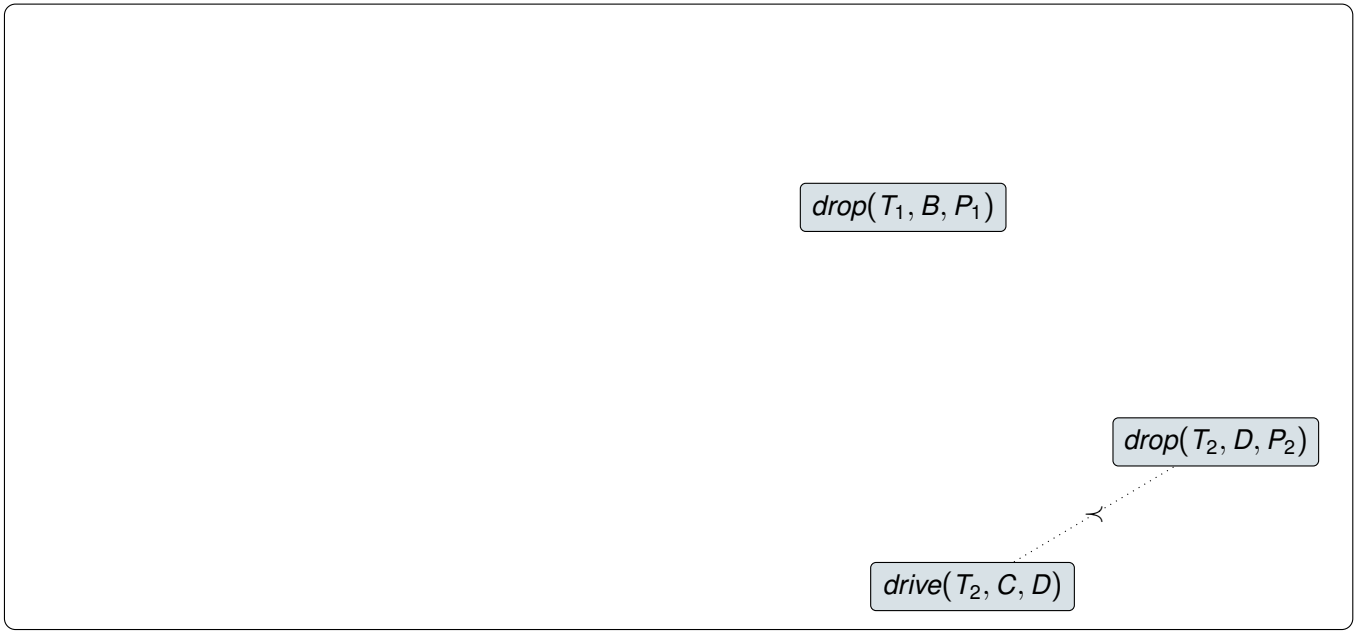


$\pi = (\textit{no-op}(), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1), \textit{drive}(T_1, A, B), \textit{pick-up}(T_2, C, P_2))$

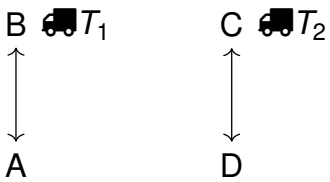


HTN Progression, Example

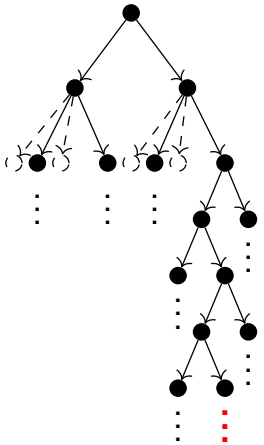




$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drive(T_1, A, B), pick\text{-}up(T_2, C, P_2))$



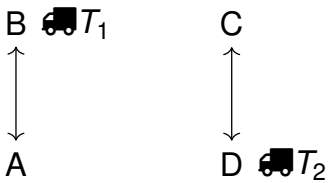
HTN Progression, Example



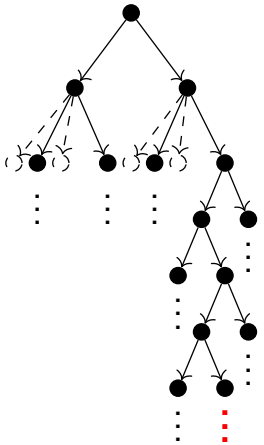
$drop(T_1, B, P_1)$

$drop(T_2, D, P_2)$

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

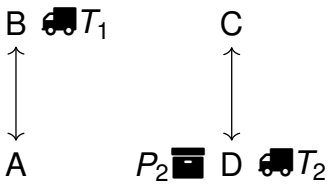


HTN Progression, Example

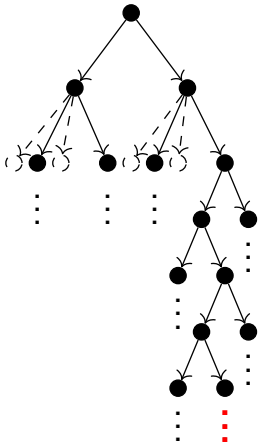


$drop(T_1, B, P_1)$

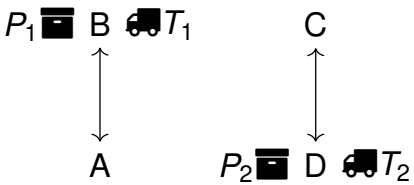
$\pi = (no\text{-}op(), no\text{-}op(), pick\text{-}up(T_1, A, P_1), drive(T_1, A, B),$
$pick\text{-}up(T_2, C, P_2), drive(T_2, C, D), drop(T_2, D, P_2))$



HTN Progression, Example



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



Pseudo Code of Standard HTN Progression Search – Can We Do Better?

Algorithm: HTN Progression Search

Input: An HTN problem $\mathcal{P} = (V, P, \delta, C, M, s_I, tn_I)$
Output: A solution $\bar{a}$ or ***fail*** if none exists

```
1  fringe ← {(s_I, tn_I, ε)}
2  while fringe ≠ ∅ do
3      n = (s, tn, ā) ← nodeSelectAndRemove(fringe)
4      if tn is empty then
5          return ā
6      else
7          U ← detectUnconstrainedSteps(tn)
8          for t ∈ U do
9              if isPrimitive(t) and pre(t) ⊆ s then
10                 fringe ← fringe ∪ {n.apply(t)}
11             else if isCompound(t) then
12                 fringe ← fringe ∪ {n.decompose(t, m) |
                                      m ∈ M with m = (α(t), tn_m)}
13 return fail
```

Eliminating Redundancy in Progression Search

- The previous algorithm branches over:

Eliminating Redundancy in Progression Search

- The previous algorithm branches over:
  - All applicable primitive tasks.

Eliminating Redundancy in Progression Search

- The previous algorithm branches over:
  - All applicable primitive tasks.
  - All decomposition methods for all compound tasks.

Eliminating Redundancy in Progression Search

- The previous algorithm branches over:
  - All applicable primitive tasks.
  - All decomposition methods for all compound tasks.
- We have to decompose *all* compound tasks and – in contrast to action application – the order in which they are handled has no influence on the resulting solutions.

Eliminating Redundancy in Progression Search

- The previous algorithm branches over:
  - All applicable primitive tasks.
  - All decomposition methods for all compound tasks.
- We have to decompose *all* compound tasks and – in contrast to action application – the order in which they are handled has no influence on the resulting solutions.

→ It's also correct to *pick* an abstract task!

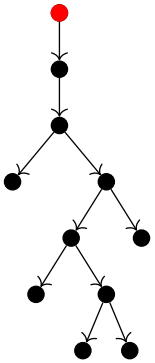
Improved HTN Progression, Pseudo Code

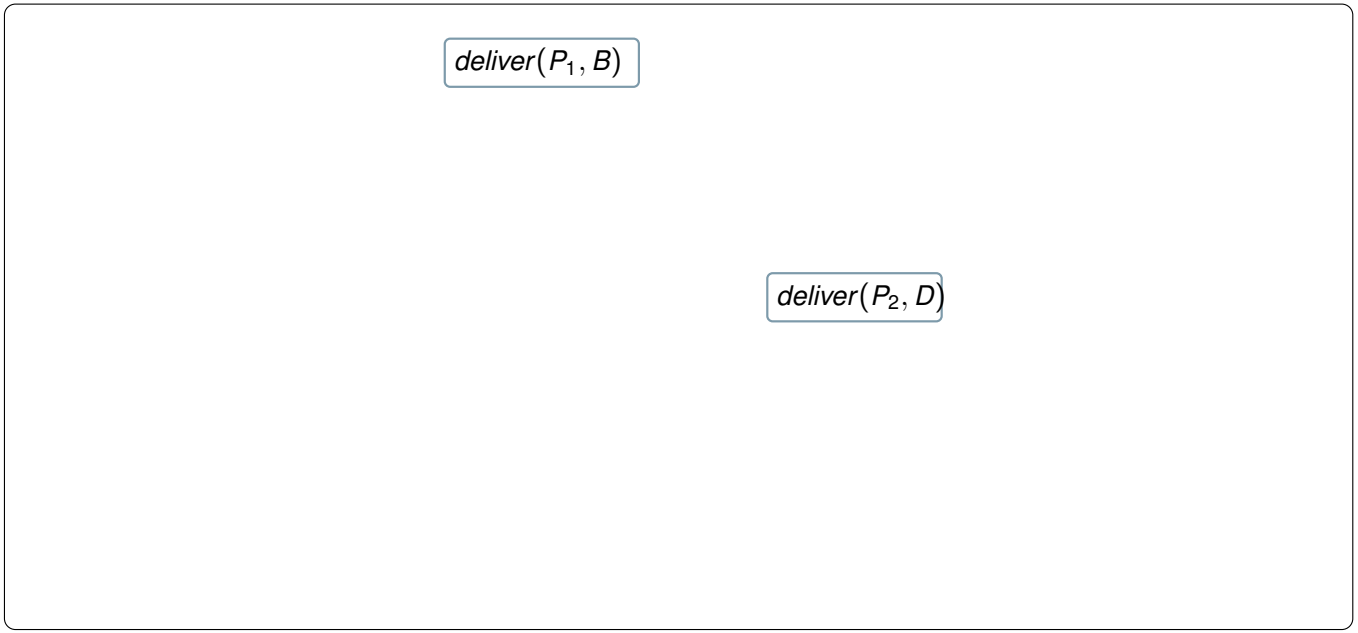
Algorithm: HTN Progression Search

Input: An HTN problem $\mathcal{P} = (V, P, \delta, C, M, s_I, tn_I)$
Output: A solution $\bar{a}$ or ***fail*** if none exists

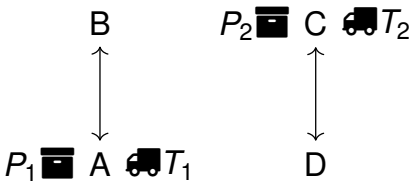
```
1  fringe ← {(s_I, tn_I, ε)}
2  while fringe ≠ ∅ do
3      n = (s, tn, ā) ← nodeSelectAndRemove(fringe)
4      if tn is empty then
5          return ā
6      else
7          (U_P, U_C) ← detectUnconstrainedSteps(tn)
8          for t ∈ U_P do
9              if pre(t) ⊆ s then
10                 fringe ← fringe ∪ {n.apply(t)}
11         t ← compoundTaskSelect(U_C)
12         fringe ← fringe ∪ {n.decompose(t, m) |
                       m ∈ M with m = (α(t), tn_m)}
13 return fail
```

Improved HTN Progression, Example

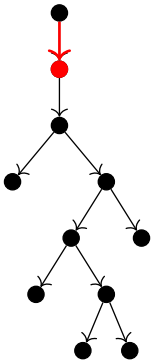


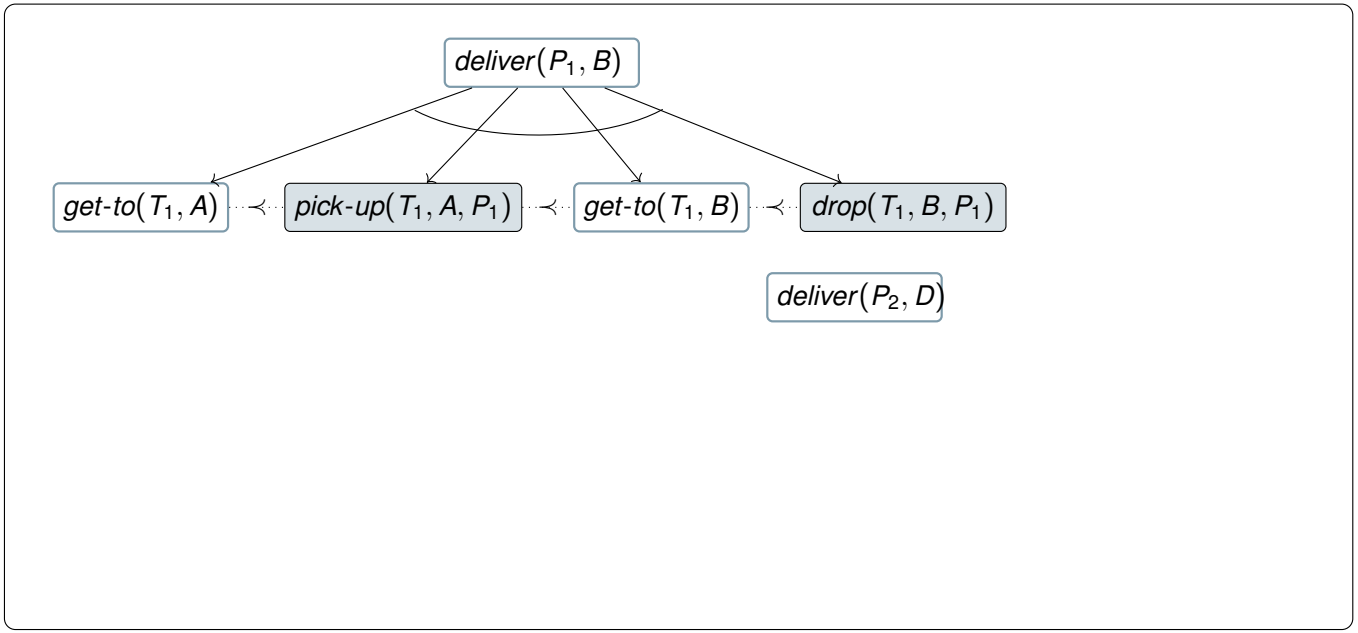


$\pi = ()$

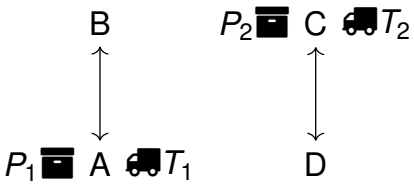


Improved HTN Progression, Example

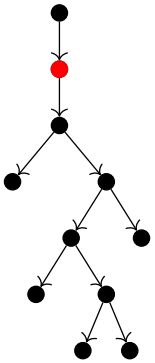


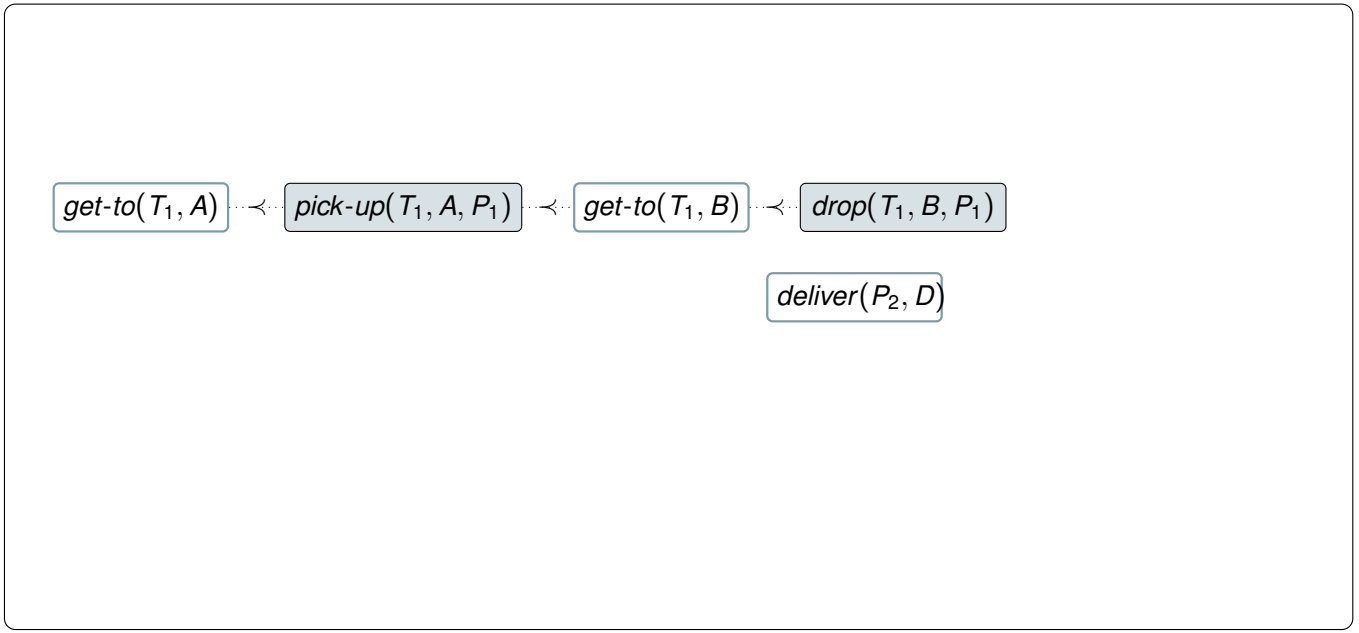


$\pi = ()$

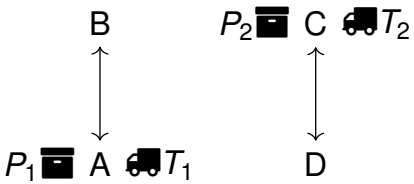


Improved HTN Progression, Example

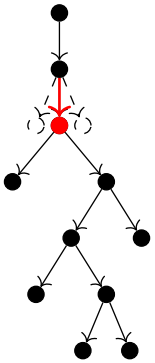


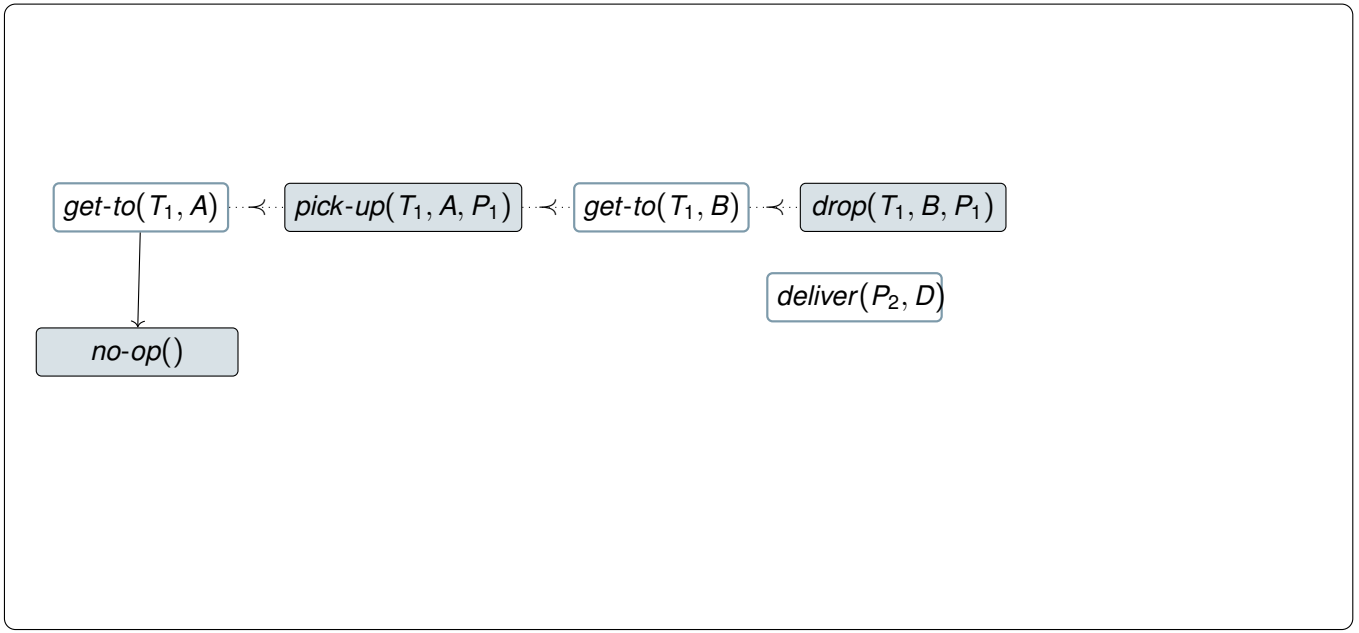


$\pi = ()$

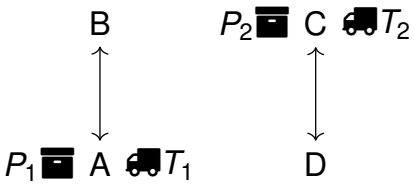


Improved HTN Progression, Example

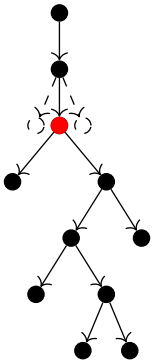


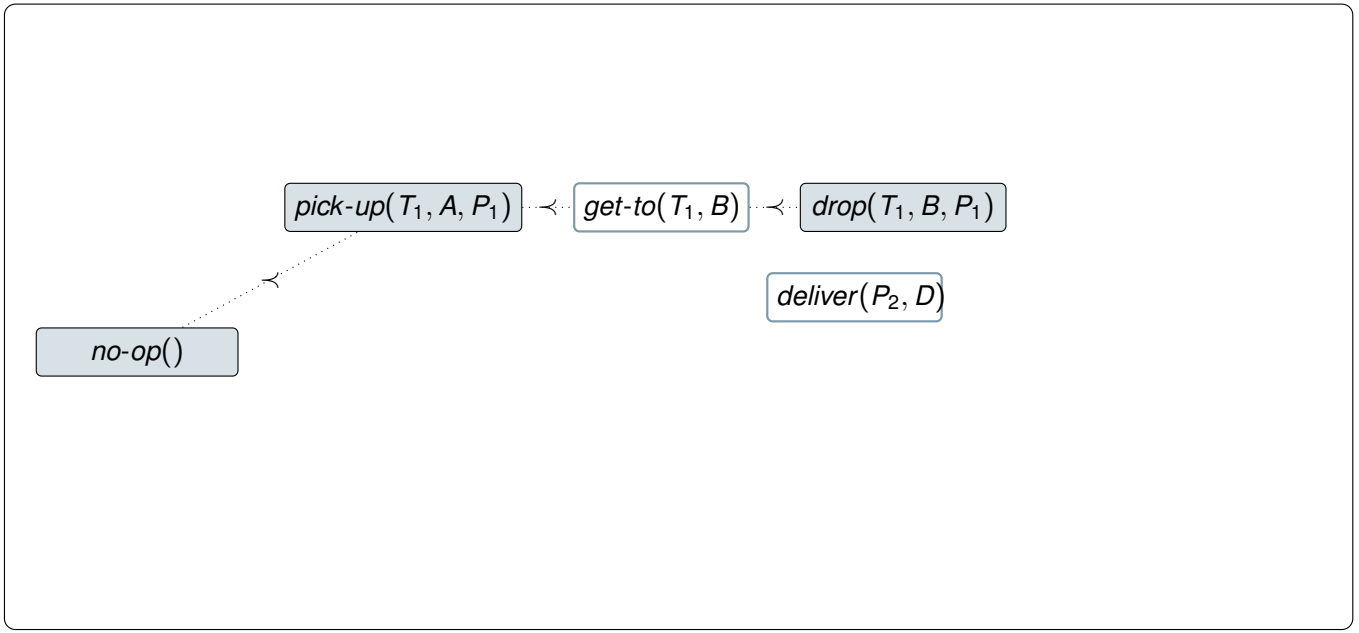


$\pi = ()$

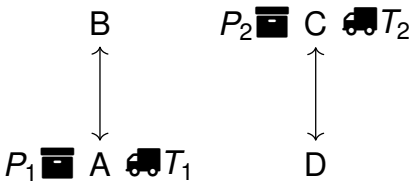


Improved HTN Progression, Example

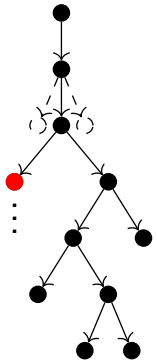


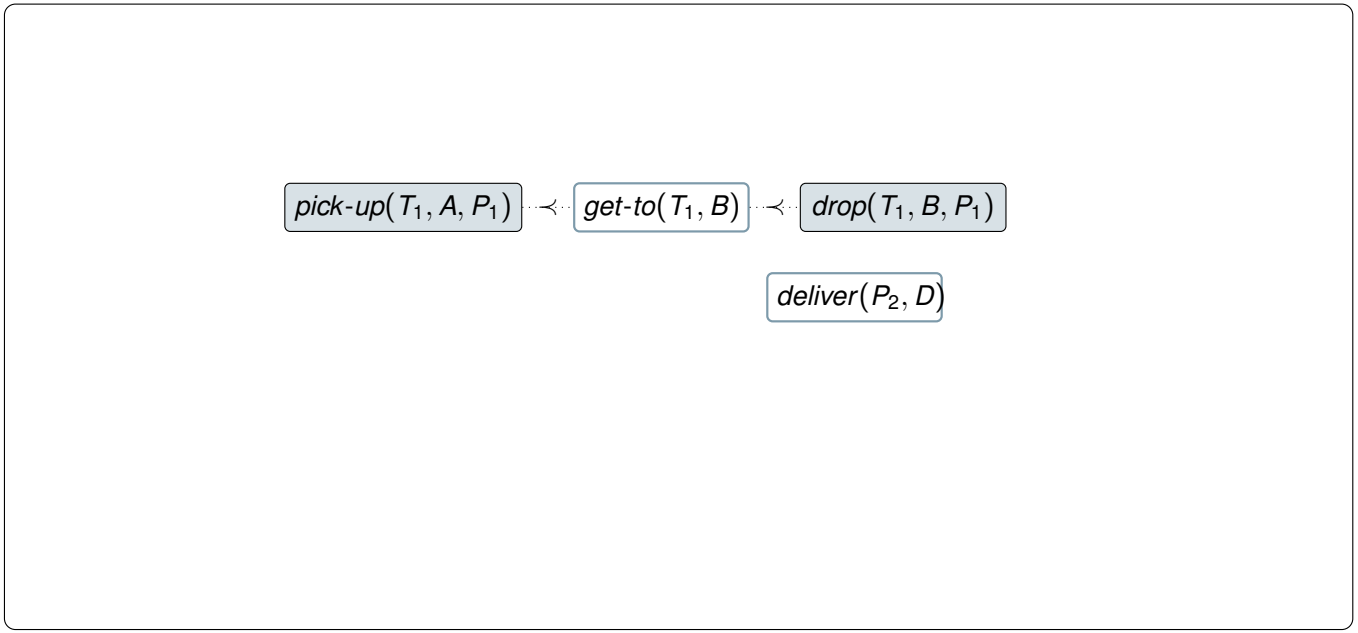


$\pi = ()$

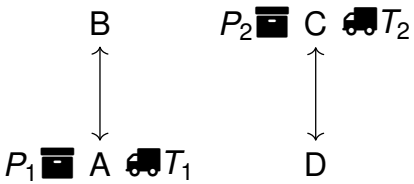


Improved HTN Progression, Example

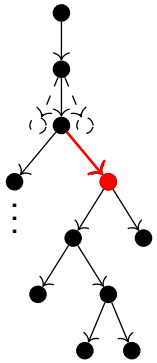


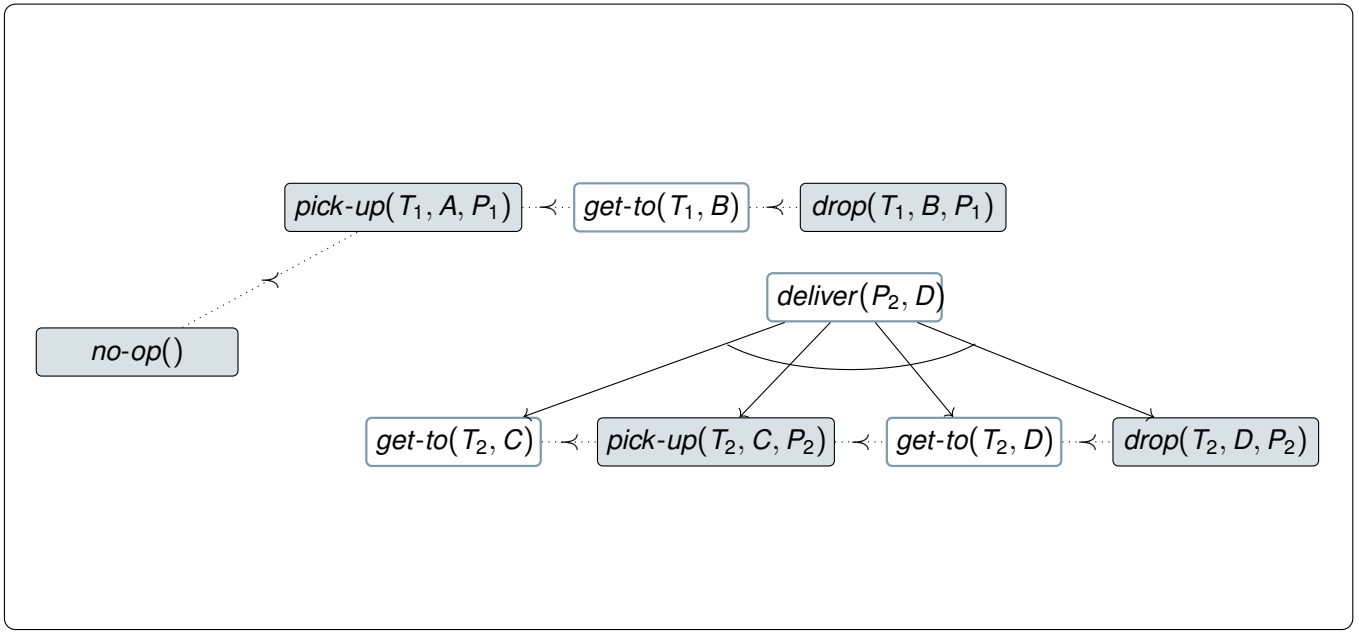


$\pi = (\textit{no-op}())$

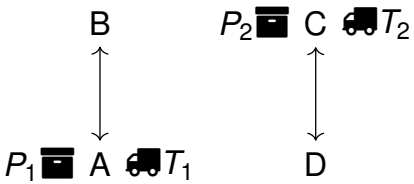


Improved HTN Progression, Example

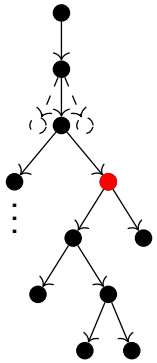


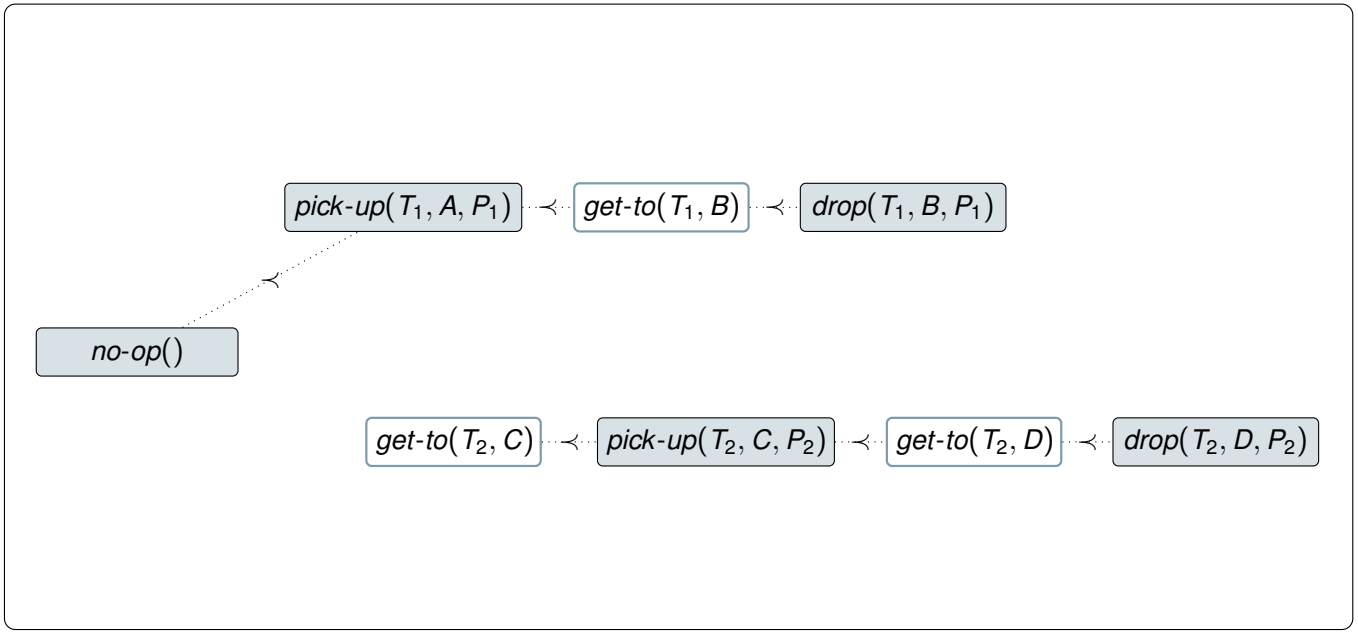


$\pi = ()$

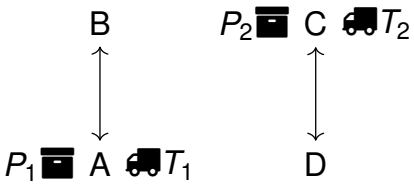


Improved HTN Progression, Example

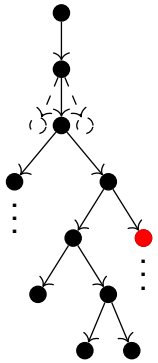


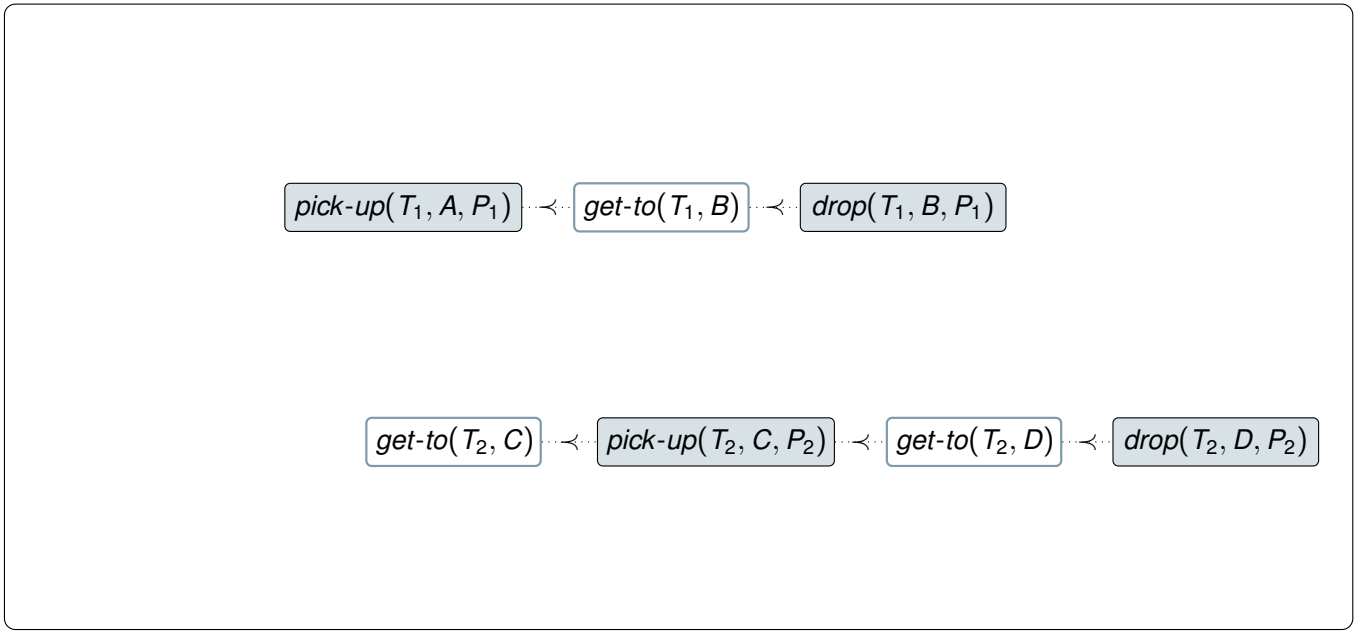


$\pi = ()$

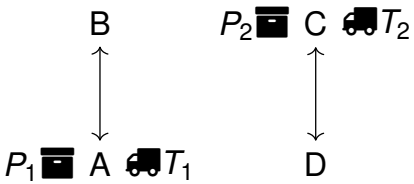


Improved HTN Progression, Example

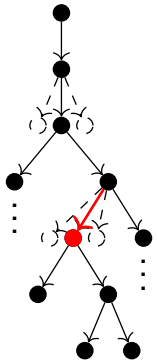


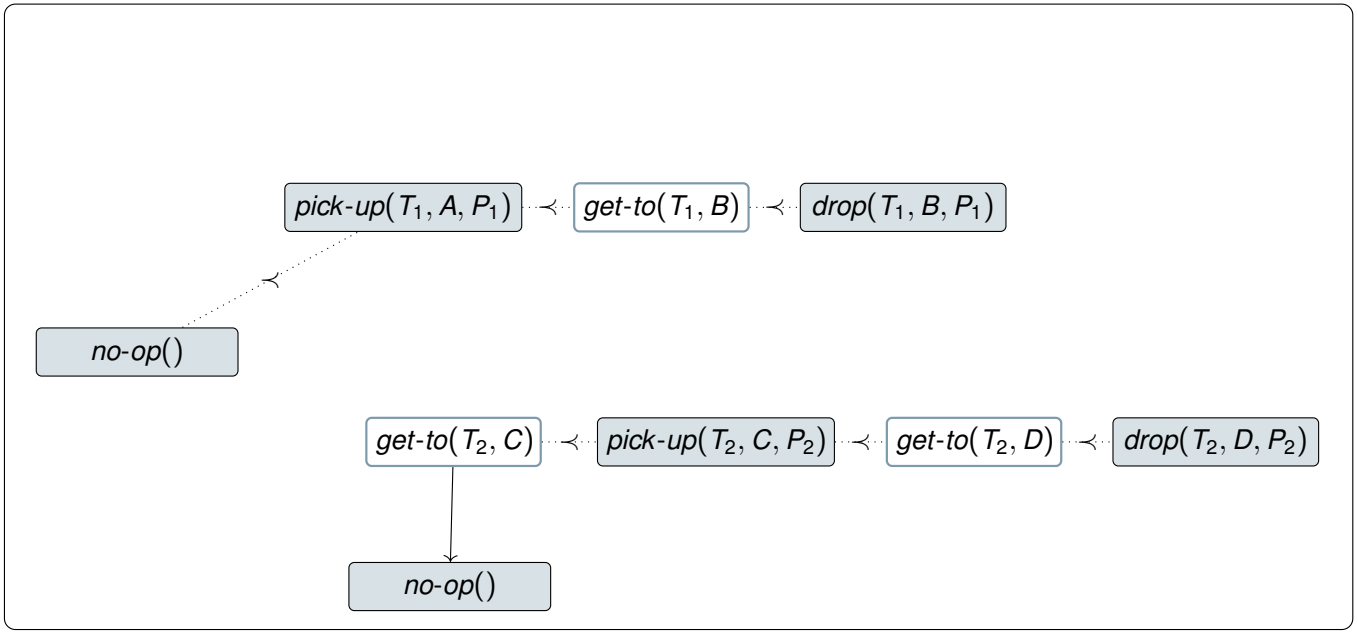


$\pi = (\textit{no-op}())$

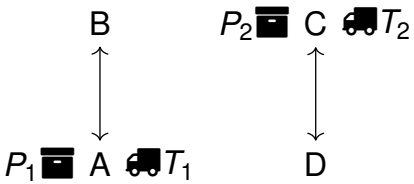


Improved HTN Progression, Example

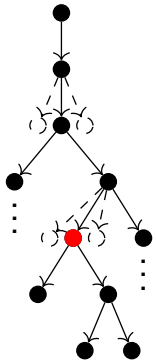


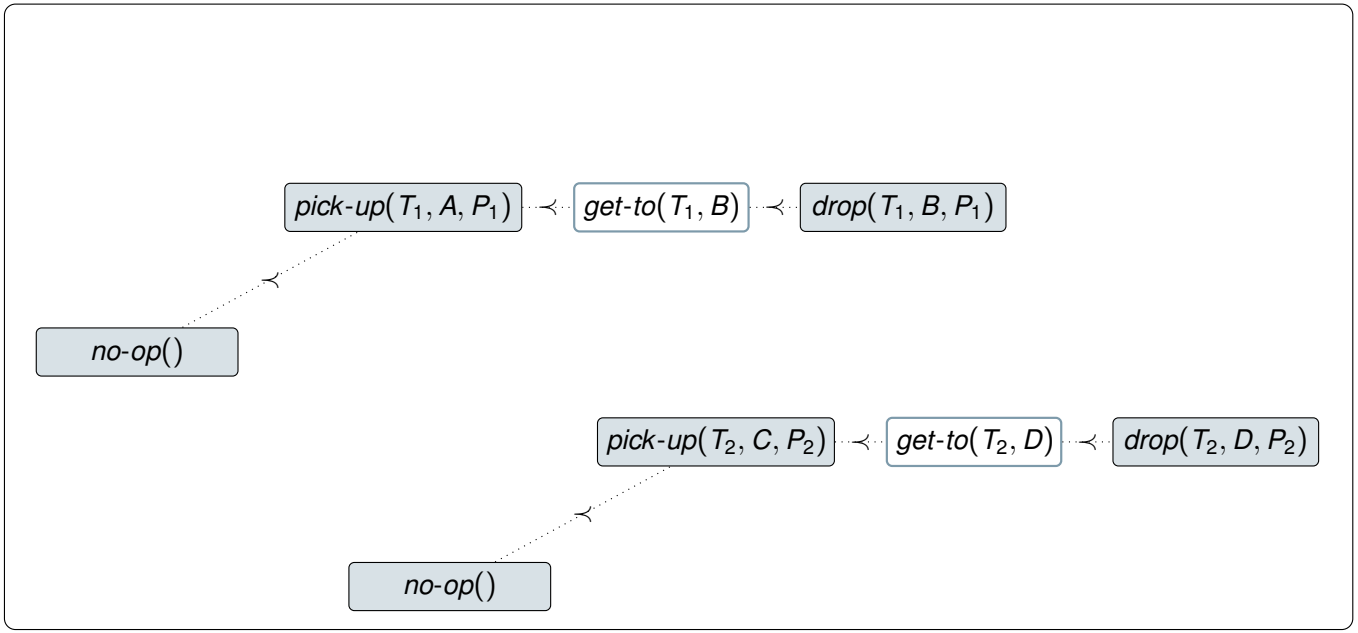


$\pi = ()$

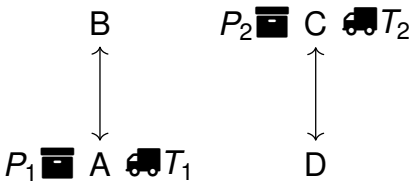


Improved HTN Progression, Example

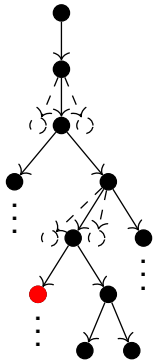


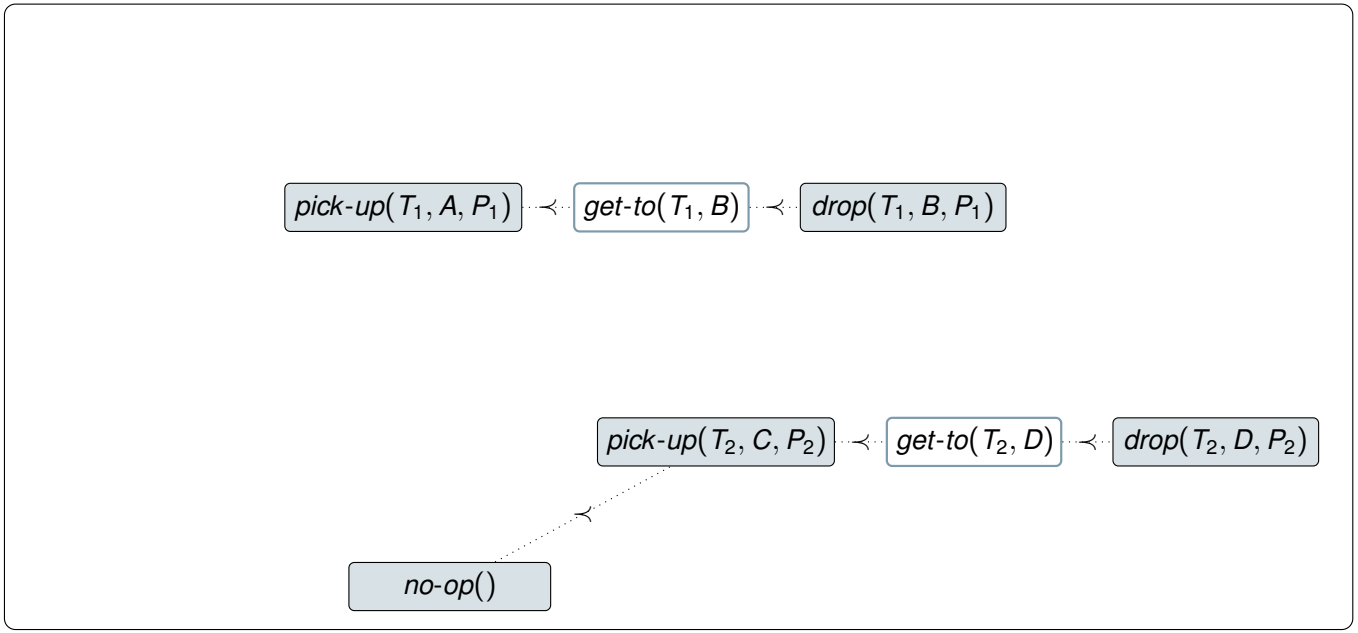


$\pi = ()$

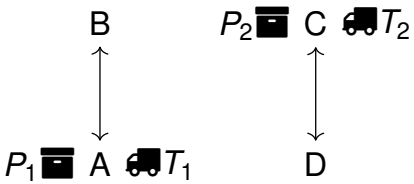


Improved HTN Progression, Example

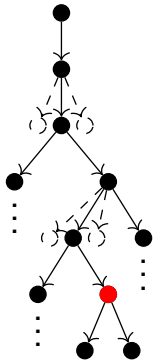


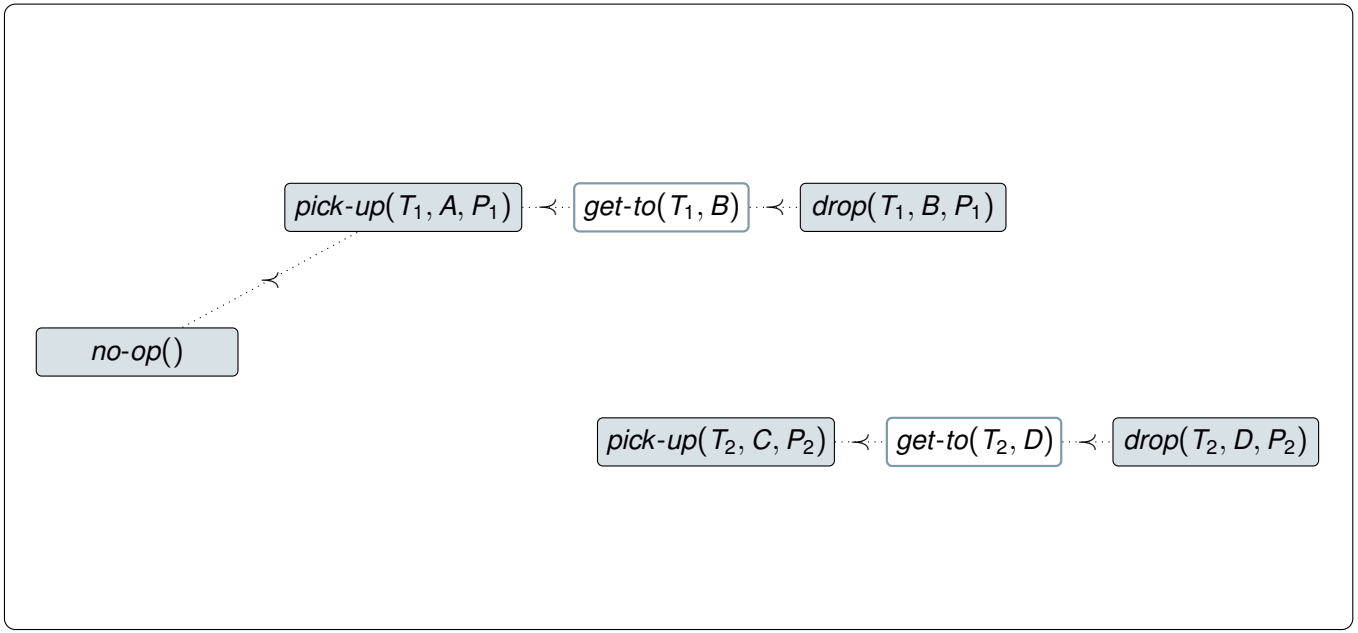


$\pi = (\textit{no-op}())$

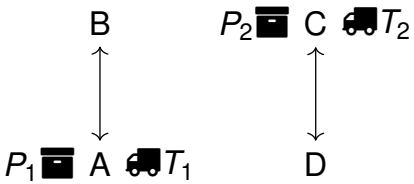


Improved HTN Progression, Example

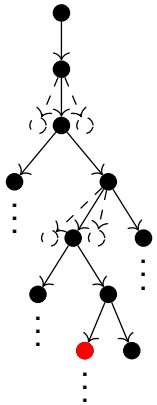


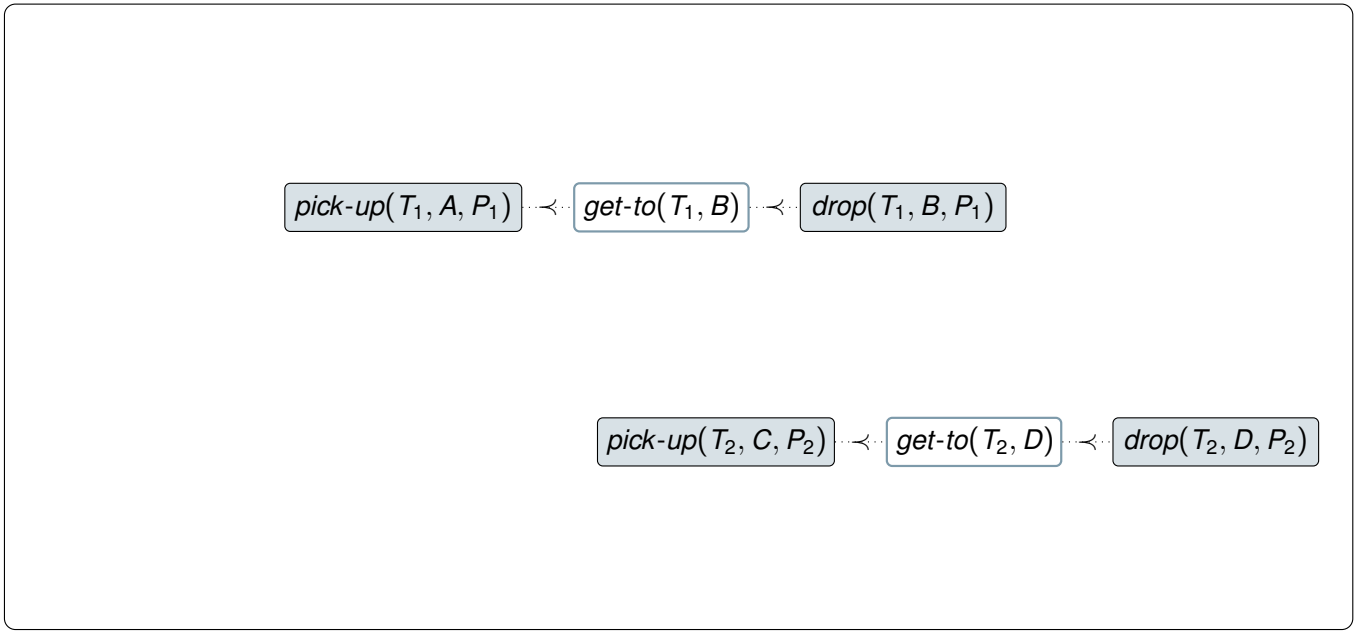


$\pi = (\textit{no-op}())$

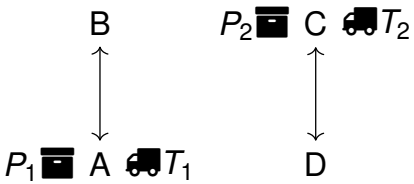


Improved HTN Progression, Example

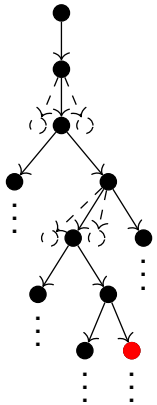


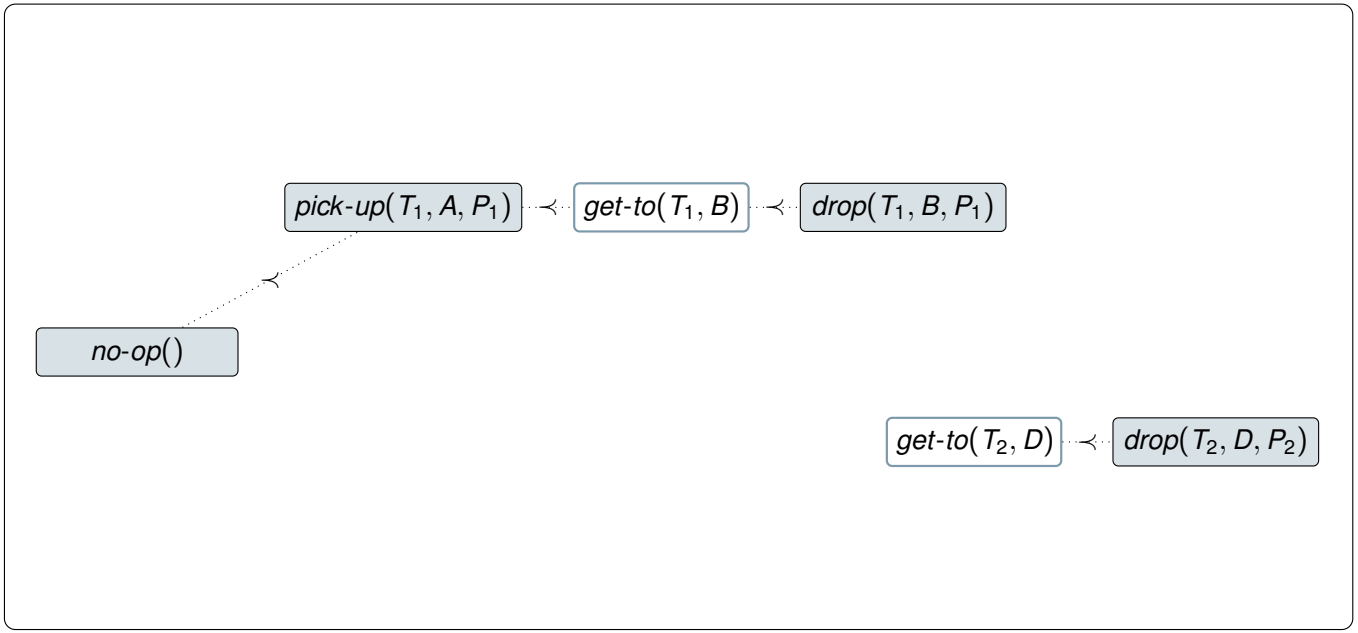


$\pi = (\textit{no-op}(), \textit{no-op}())$



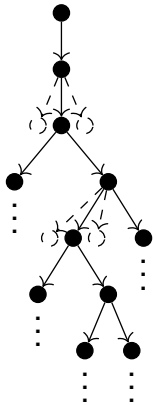
Improved HTN Progression, Example

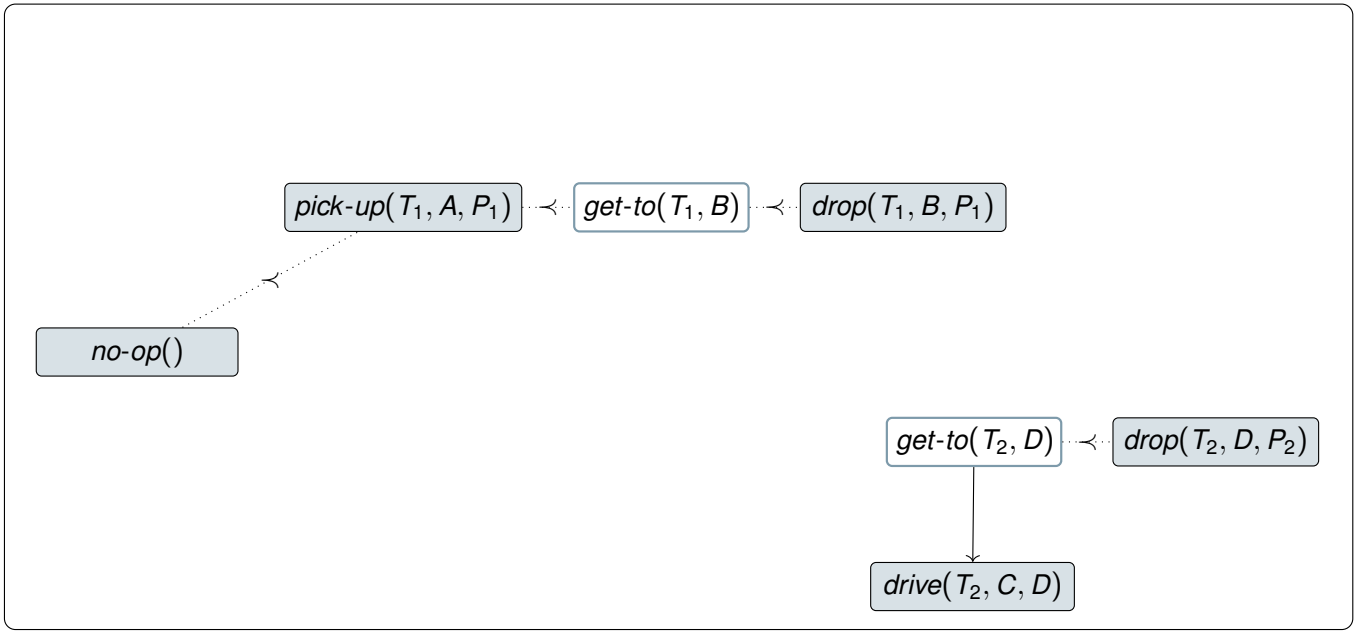




$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2))$$

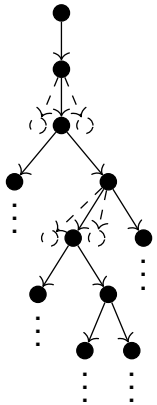
Improved HTN Progression, Example

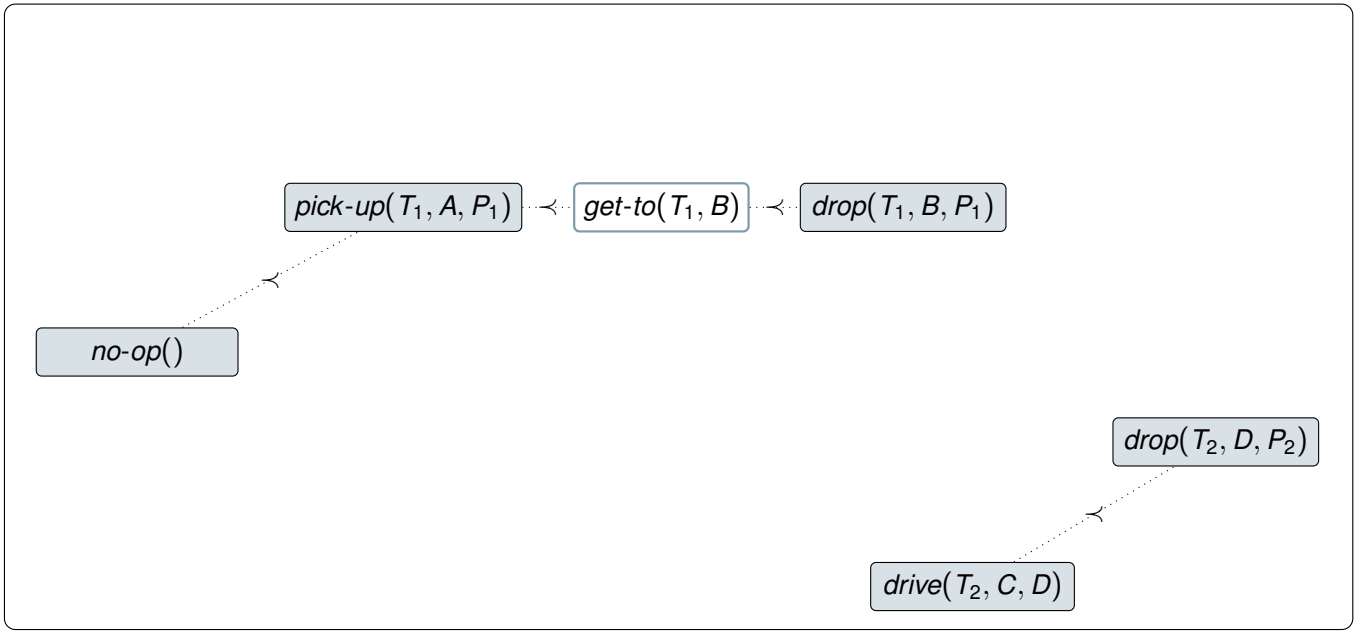




$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2))$$

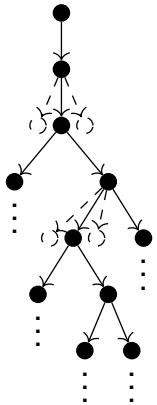
Improved HTN Progression, Example

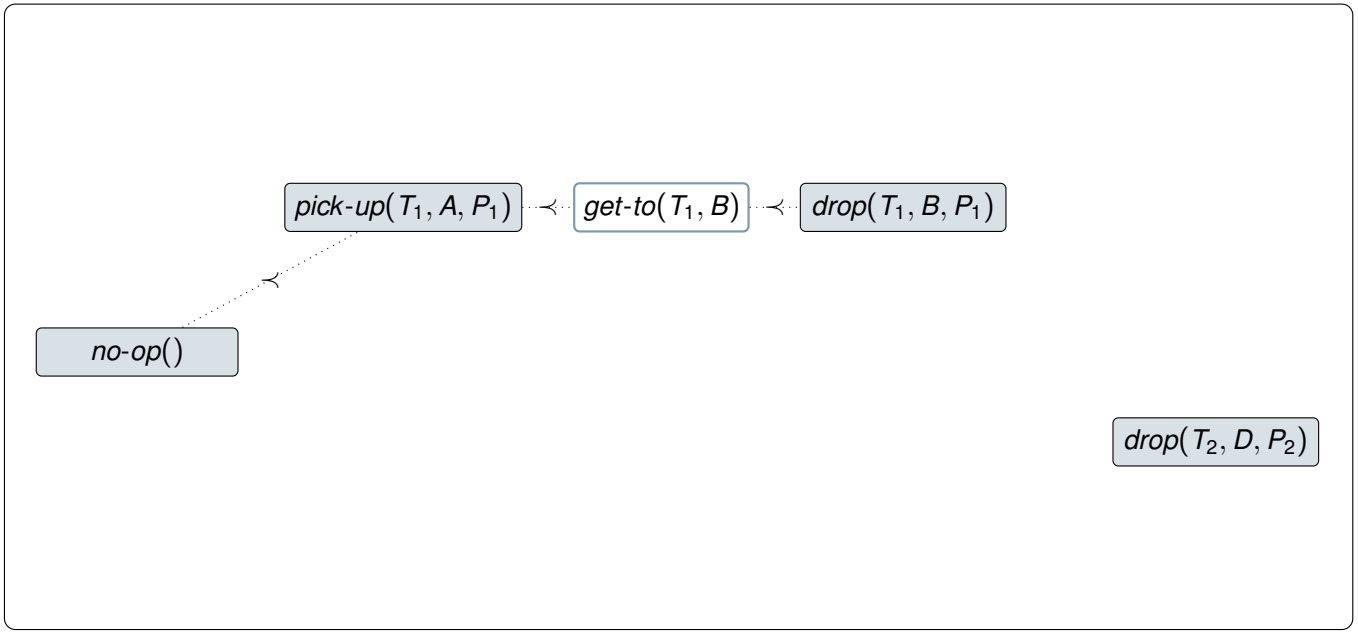




$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2))$$

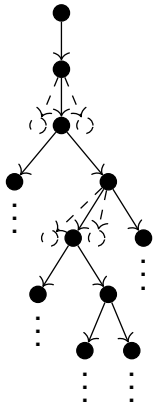
Improved HTN Progression, Example

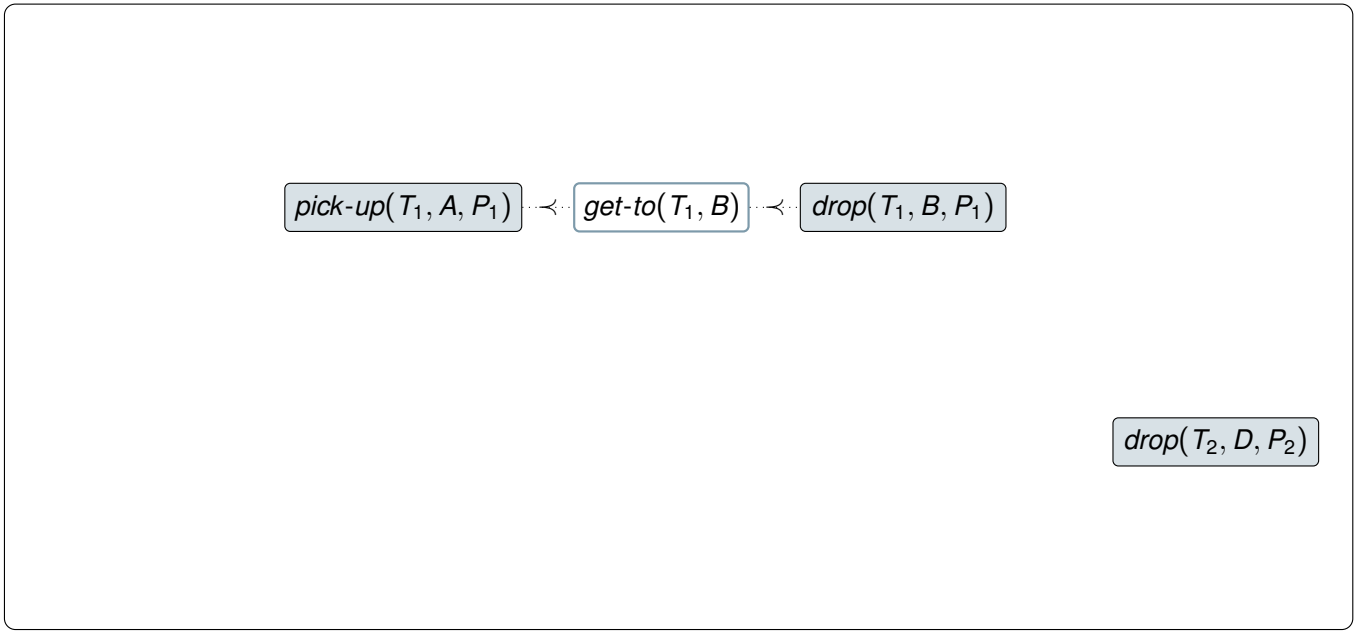




$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D))$$

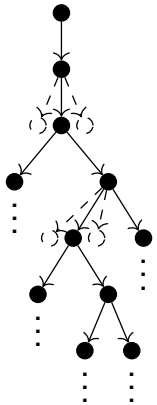
Improved HTN Progression, Example

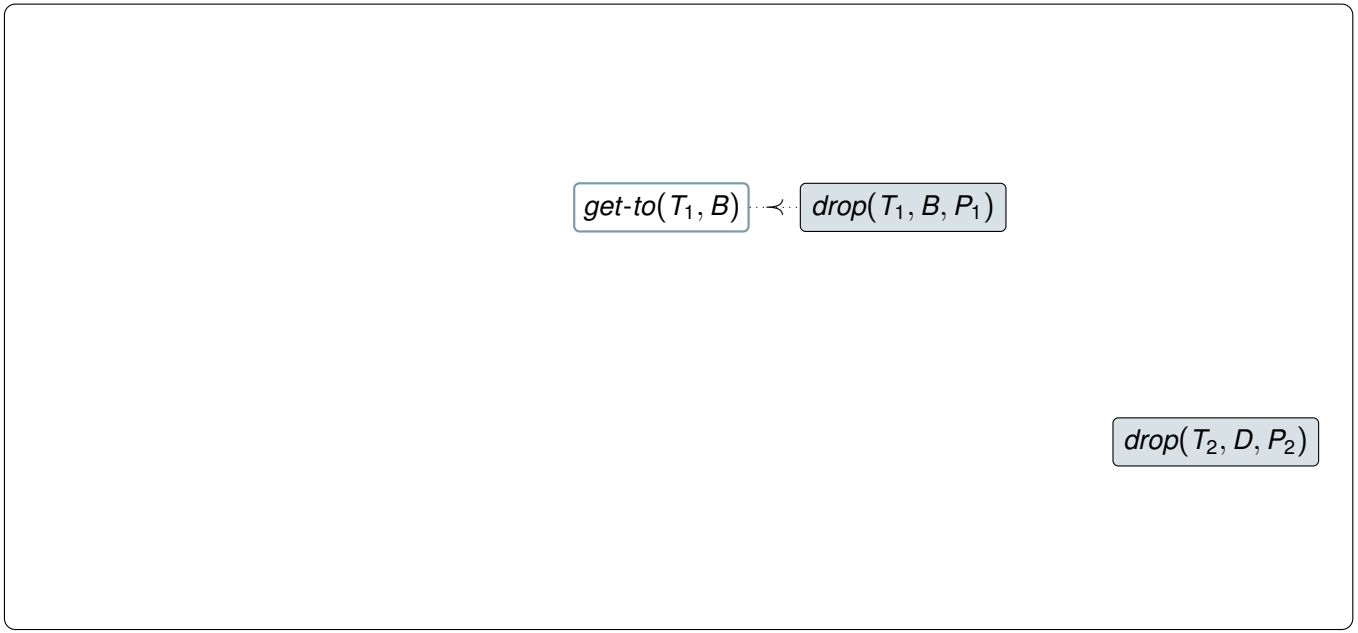




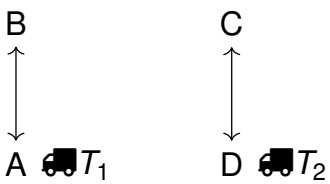
$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}())$

Improved HTN Progression, Example

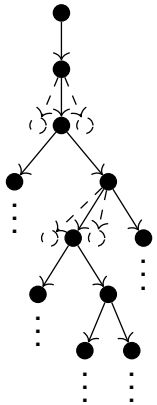


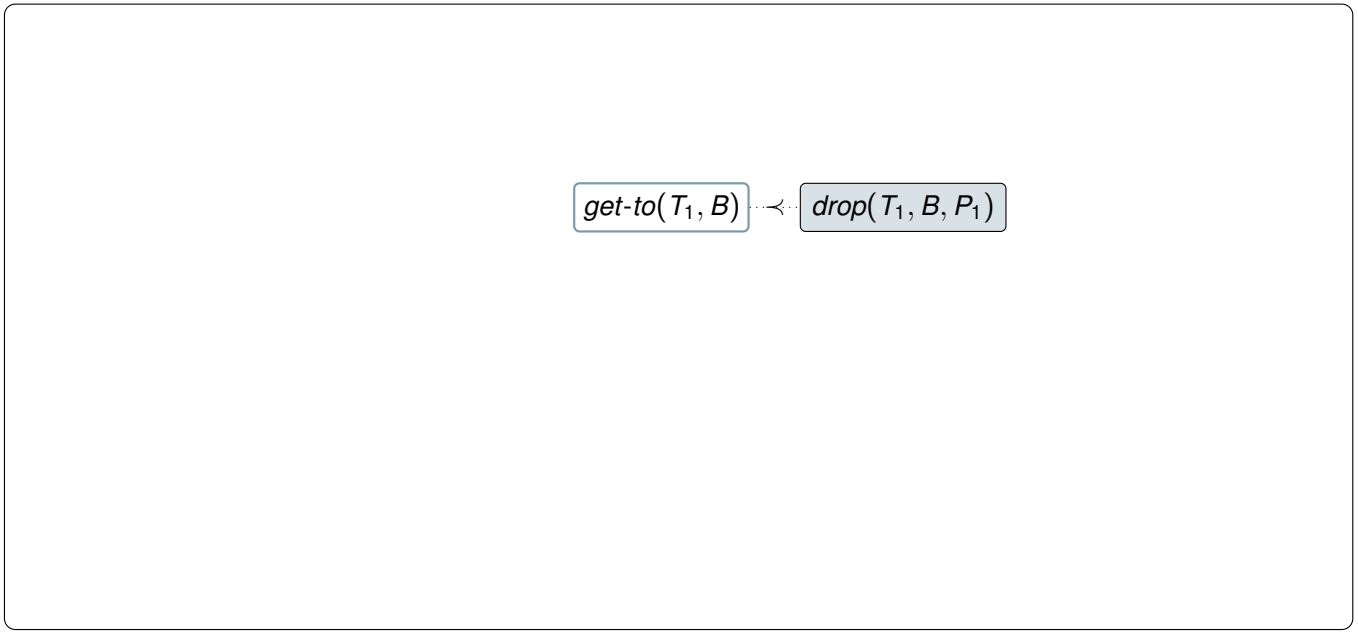


$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1))$

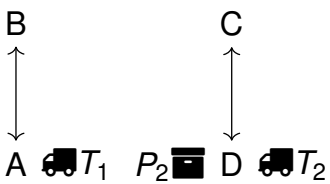


Improved HTN Progression, Example

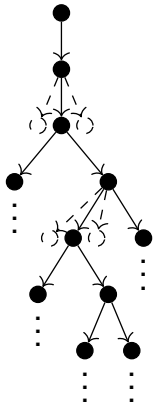


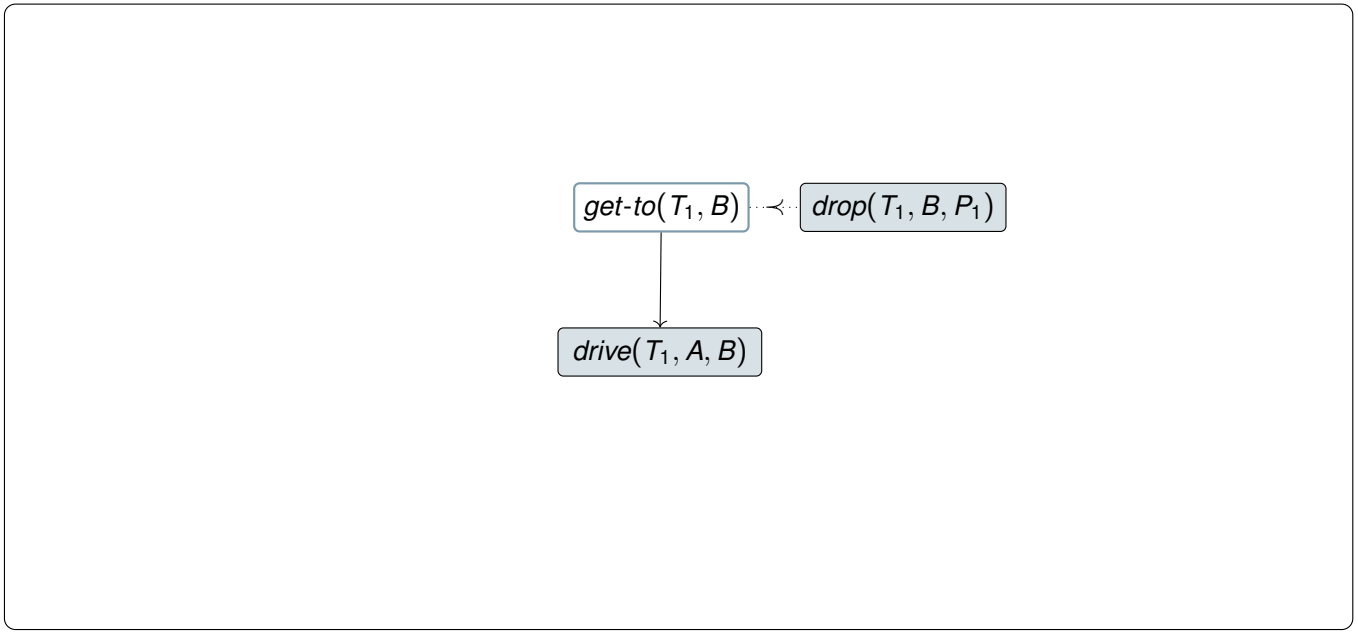


$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(),$
$\textit{pick-up}(T_1, A, P_1), \textit{drop}(T_2, D, P_2))$

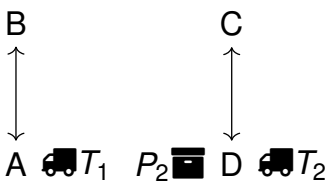


Improved HTN Progression, Example

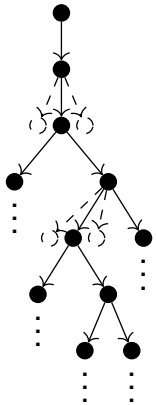


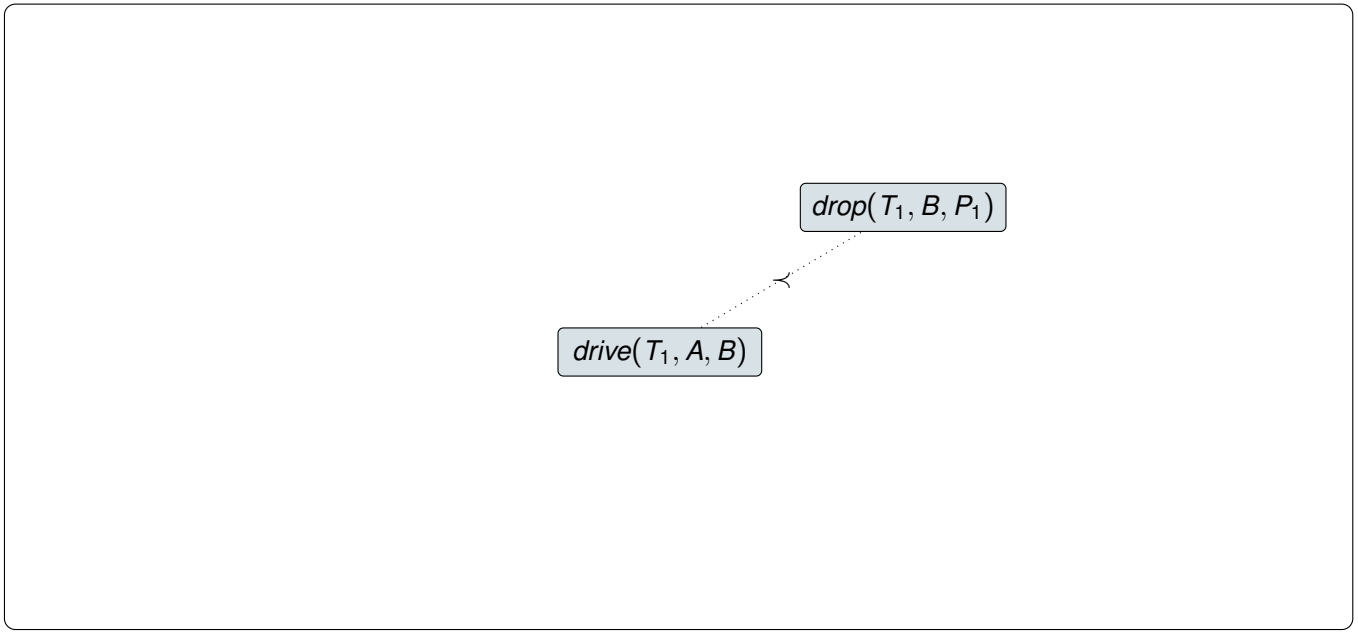


$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(),$
$\textit{pick-up}(T_1, A, P_1), \textit{drop}(T_2, D, P_2))$

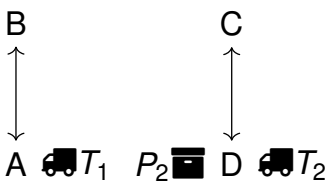


Improved HTN Progression, Example

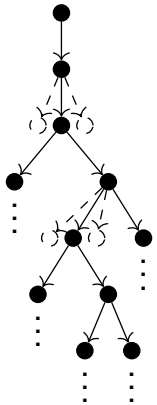




$$\pi = (\textit{no-op}(), \textit{pick-up}(T_2, C, P_2), \textit{drive}(T_2, C, D), \textit{no-op}(), \textit{pick-up}(T_1, A, P_1), \textit{drop}(T_2, D, P_2))$$

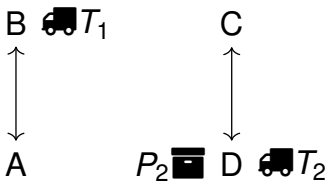


Improved HTN Progression, Example

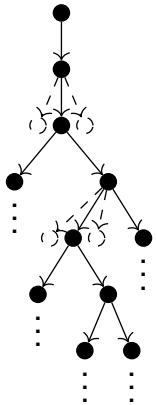


$drop(T_1, B, P_1)$

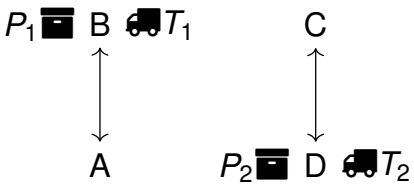
$\pi = (no\text{-}op(), pick\text{-}up(T_2, C, P_2), drive(T_2, C, D), no\text{-}op(),$
$pick\text{-}up(T_1, A, P_1), drop(T_2, D, P_2), drive(T_1, A, B))$



Improved HTN Progression, Example



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



Properties

Theorem

HTN progression search is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*().

Properties

Theorem

HTN progression search is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*().

Proof:
Follows from the properties of the underlying search algorithm.
However:

- Be aware that the transition system is not finite!
- We need to argue why the restricted algorithm is still complete although not branching over all choices.

SHOP/SHOP2 and Method Preconditions

- One of the best-known (and still in use) HTN planners is SHOP2, which performs progression search.

SHOP/SHOP2 and Method Preconditions

- One of the best-known (and still in use) HTN planners is SHOP2, which performs progression search.
- SHOP, the predecessor of SHOP2, can only cope with totally ordered methods (and a totally ordered initial task network).

SHOP/SHOP2 and Method Preconditions

- One of the best-known (and still in use) HTN planners is SHOP2, which performs progression search.
- SHOP, the predecessor of SHOP2, can only cope with totally ordered methods (and a totally ordered initial task network).
- Both SHOP and SHOP2 perform by default depth-first search and specify in which order decomposition methods should be applied. This order relies on additional preconditions, e.g.:

SHOP/SHOP2 and Method Preconditions

- One of the best-known (and still in use) HTN planners is SHOP2, which performs progression search.
- SHOP, the predecessor of SHOP2, can only cope with totally ordered methods (and a totally ordered initial task network).
- Both SHOP and SHOP2 perform by default depth-first search and specify in which order decomposition methods should be applied. This order relies on additional preconditions, e.g.:
  - If φ holds in s, use method m_i for task t, otherwise

SHOP/SHOP2 and Method Preconditions

- One of the best-known (and still in use) HTN planners is SHOP2, which performs progression search.
- SHOP, the predecessor of SHOP2, can only cope with totally ordered methods (and a totally ordered initial task network).
- Both SHOP and SHOP2 perform by default depth-first search and specify in which order decomposition methods should be applied. This order relies on additional preconditions, e.g.:
  - If φ holds in s, use method m_i for task t, otherwise
  - if ψ holds in s, use method m_j for task t, else

SHOP/SHOP2 and Method Preconditions

- One of the best-known (and still in use) HTN planners is SHOP2, which performs progression search.
- SHOP, the predecessor of SHOP2, can only cope with totally ordered methods (and a totally ordered initial task network).
- Both SHOP and SHOP2 perform by default depth-first search and specify in which order decomposition methods should be applied. This order relies on additional preconditions, e.g.:
  - If φ holds in s, use method m_i for task t, otherwise
  - if ψ holds in s, use method m_j for task t, else
  - use method m_k for task t.

SHOP/SHOP2 and Method Preconditions

- One of the best-known (and still in use) HTN planners is SHOP2, which performs progression search.
- SHOP, the predecessor of SHOP2, can only cope with totally ordered methods (and a totally ordered initial task network).
- Both SHOP and SHOP2 perform by default depth-first search and specify in which order decomposition methods should be applied. This order relies on additional preconditions, e.g.:
  - If φ holds in s, use method m_i for task t, otherwise
  - if ψ holds in s, use method m_j for task t, else
  - use method m_k for task t.
  - Note: these valuations can be arbitrary program calls.

SHOP/SHOP2 and Method Preconditions

- One of the best-known (and still in use) HTN planners is SHOP2, which performs progression search.
- SHOP, the predecessor of SHOP2, can only cope with totally ordered methods (and a totally ordered initial task network).
- Both SHOP and SHOP2 perform by default depth-first search and specify in which order decomposition methods should be applied. This order relies on additional preconditions, e.g.:
  - If φ holds in s, use method m_i for task t, otherwise
  - if ψ holds in s, use method m_j for task t, else
  - use method m_k for task t.
  - Note: these valuations can be arbitrary program calls.
- Note the semantical difference of method preconditions in total-order HTN probelms (i.e., SHOP) versus partial-order HTN problems (i.e, SHOP2).

Further Extensions

- TIHTN problems:

Further Extensions

- TIHTN problems:
  - Progression search is also applicable for TIHTN problems.

Further Extensions

- TIHTN problems:
  - Progression search is also applicable for TIHTN problems.
  - The only required extension is that in addition to progressing compound or primitive tasks in the task network we can also apply primitive tasks from the model.

Further Extensions

- TIHTN problems:
  - Progression search is also applicable for TIHTN problems.
  - The only required extension is that in addition to progressing compound or primitive tasks in the task network we can also apply primitive tasks from the model.
- Goal description:

Further Extensions

- TIHTN problems:
  - Progression search is also applicable for TIHTN problems.
  - The only required extension is that in addition to progressing compound or primitive tasks in the task network we can also apply primitive tasks from the model.
- Goal description: Add the criterion that the current state needs to be a goal state (in addition to the current task network being empty).

Further Extensions

- TIHTN problems:
  - Progression search is also applicable for TIHTN problems.
  - The only required extension is that in addition to progressing compound or primitive tasks in the task network we can also apply primitive tasks from the model.
- Goal description: Add the criterion that the current state needs to be a goal state (in addition to the current task network being empty).
- State constraints:

Further Extensions

- TIHTN problems:
  - Progression search is also applicable for TIHTN problems.
  - The only required extension is that in addition to progressing compound or primitive tasks in the task network we can also apply primitive tasks from the model.
- Goal description: Add the criterion that the current state needs to be a goal state (in addition to the current task network being empty).
- State constraints: They can simply be tracked as well (and removed as soon as satisfied) in accordance to the definition given in the lecture.

Introduction

- Progression search commits to executable linearizations, similar to classical planning.

Introduction

- Progression search commits to executable linearizations, similar to classical planning.
- In particular if a problem admits solutions with many linearizations, this approach might suffer from large search spaces.

Introduction

- Progression search commits to executable linearizations, similar to classical planning.
- In particular if a problem admits solutions with many linearizations, this approach might suffer from large search spaces.
- An alternative is *decomposition-based HTN planning* (also: *plan space-based planning* or *hybrid planning*), which extends POCL planning by the necessary concepts from hierarchical planning.

Introduction

- Progression search commits to executable linearizations, similar to classical planning.
- In particular if a problem admits solutions with many linearizations, this approach might suffer from large search spaces.
- An alternative is *decomposition-based HTN planning* (also: *plan space-based planning* or *hybrid planning*), which extends POCL planning by the necessary concepts from hierarchical planning.

Terminology:

- In the remainder, we will fuse the terminologies from POCL planning with those from HTN planning.

Introduction

- Progression search commits to executable linearizations, similar to classical planning.
- In particular if a problem admits solutions with many linearizations, this approach might suffer from large search spaces.
- An alternative is *decomposition-based HTN planning* (also: *plan space-based planning* or *hybrid planning*), which extends POCL planning by the necessary concepts from hierarchical planning.

Terminology:

- In the remainder, we will fuse the terminologies from POCL planning with those from HTN planning.
- Rather than talking about *task networks*, we refer to them as *partial plans*.

Extensions to POCL Planning, New Flaws

New flaws:

- Compound task flaw:

Extensions to POCL Planning, New Flaws

New flaws:

- Compound task flaw:
  - Each compound task needs to be refined, thus raises an flaw.

Extensions to POCL Planning, New Flaws

New flaws:

- Compound task flaw:
  - Each compound task needs to be refined, thus raises an flaw.
  - For each abstract task flaw, the set of modifications equals the set of methods for that task.

Extensions to POCL Planning, New Flaws

New flaws:

- Compound task flaw:
  - Each compound task needs to be refined, thus raises an flaw.
  - For each abstract task flaw, the set of modifications equals the set of methods for that task.
- Any further flaws?

Extensions to POCL Planning, New Flaws

New flaws:

- Compound task flaw:
  - Each compound task needs to be refined, thus raises an flaw.
  - For each abstract task flaw, the set of modifications equals the set of methods for that task.
- Any further flaws? No, but we need to alter the remaining flaws and modifications.

Alterations to POCL Planning, Open Preconditions

Open precondition flaw:

- As in POCL planning, each precondition without causal link raises an open precondition flaw.

Alterations to POCL Planning, Open Preconditions

Open precondition flaw:

- As in POCL planning, each precondition without causal link raises an open precondition flaw.
- In POCL planning, we provided one modification for each possible producer:

Alterations to POCL Planning, Open Preconditions

Open precondition flaw:

- As in POCL planning, each precondition without causal link raises an open precondition flaw.
- In POCL planning, we provided one modification for each possible producer:
  - In the current partial plan: only add causal link.

Alterations to POCL Planning, Open Preconditions

Open precondition flaw:

- As in POCL planning, each precondition without causal link raises an open precondition flaw.
- In POCL planning, we provided one modification for each possible producer:
  - In the current partial plan: only add causal link.
  - In the model: add action plus link.

Alterations to POCL Planning, Open Preconditions

Open precondition flaw:

- As in POCL planning, each precondition without causal link raises an open precondition flaw.
- In POCL planning, we provided one modification for each possible producer:
  - In the current partial plan: only add causal link.
  - In the model: add action plus link.
- In hybrid planning, we also provide one modification for each possible producer:

Alterations to POCL Planning, Open Preconditions

Open precondition flaw:

- As in POCL planning, each precondition without causal link raises an open precondition flaw.
- In POCL planning, we provided one modification for each possible producer:
  - In the current partial plan: only add causal link.
  - In the model: add action plus link.
- In hybrid planning, we also provide one modification for each possible producer:
  - Producer *is already* in the current partial plan: only add causal link.

Alterations to POCL Planning, Open Preconditions

Open precondition flaw:

- As in POCL planning, each precondition without causal link raises an open precondition flaw.
- In POCL planning, we provided one modification for each possible producer:
  - In the current partial plan: only add causal link.
  - In the model: add action plus link.
- In hybrid planning, we also provide one modification for each possible producer:
  - Producer *is already* in the current partial plan: only add causal link.
  - Producer *could be added* via decomposing a compound task: decompose with the respective methods (more details later).

Alterations to POCL Planning, Open Preconditions, cont'd I

Let $(PS, \prec, CL, \alpha)$ be a partial plan (where a plan step $ps \in PS$ can also contain compound tasks, $\alpha(ps) \in P \cup C$).

- Let $ps \in PS$ a primitive plan step with open condition (v, ps) an open precondition flaw.

Alterations to POCL Planning, Open Preconditions, cont'd I

Let $(PS, \prec, CL, \alpha)$ be a partial plan (where a plan step $ps \in PS$ can also contain compound tasks, $\alpha(ps) \in P \cup C$).

- Let $ps \in PS$ a primitive plan step with open condition (v, ps) an open precondition flaw.
- Let $ps' \in PS$ be compound (i.e., $\alpha(ps') \in C$) and *possibly* be ordered before ps (i.e., $(ps, ps') \notin \prec$).

Alterations to POCL Planning, Open Preconditions, cont'd I

Let $(PS, \prec, CL, \alpha)$ be a partial plan (where a plan step $ps \in PS$ can also contain compound tasks, $\alpha(ps) \in P \cup C$).

- Let $ps \in PS$ a primitive plan step with open condition (v, ps) an open precondition flaw.
- Let $ps' \in PS$ be compound (i.e., $\alpha(ps') \in C$) and *possibly* be ordered before ps (i.e., $(ps, ps') \notin \prec$).

What to do *exactly* to offer modifications that address/resolve (v, ps)?

Alterations to POCL Planning, Open Preconditions, cont'd I

Let $(PS, \prec, CL, \alpha)$ be a partial plan (where a plan step $ps \in PS$ can also contain compound tasks, $\alpha(ps) \in P \cup C$).

- Let $ps \in PS$ a primitive plan step with open condition (v, ps) an open precondition flaw.
- Let $ps' \in PS$ be compound (i.e., $\alpha(ps') \in C$) and *possibly* be ordered before ps (i.e., $(ps, ps') \notin \prec$).

What to do *exactly* to offer modifications that address/resolve (v, ps)?

- Only checking the very next level of $\alpha(ps')$ (i.e., the tasks in the methods of $\alpha(ps')$) is *not* sufficient and might lead to an incomplete algorithm.

Alterations to POCL Planning, Open Preconditions, cont'd I

Let $(PS, \prec, CL, \alpha)$ be a partial plan (where a plan step $ps \in PS$ can also contain compound tasks, $\alpha(ps) \in P \cup C$).

- Let $ps \in PS$ a primitive plan step with open condition (v, ps) an open precondition flaw.
- Let $ps' \in PS$ be compound (i.e., $\alpha(ps') \in C$) and *possibly* be ordered before ps (i.e., $(ps, ps') \notin \prec$).

What to do *exactly* to offer modifications that address/resolve (v, ps)?

- Only checking the very next level of $\alpha(ps')$ (i.e., the tasks in the methods of $\alpha(ps')$) is *not* sufficient and might lead to an incomplete algorithm.
- We need a mapping from each compound task to each reachable state variable. For efficiency reasons, this has to be done *once* in a preprocessing step.

Alterations to POCL Planning, Open Preconditions, cont'd I

Let $(PS, \prec, CL, \alpha)$ be a partial plan (where a plan step $ps \in PS$ can also contain compound tasks, $\alpha(ps) \in P \cup C$).

- Let $ps \in PS$ a primitive plan step with open condition (v, ps) an open precondition flaw.
- Let $ps' \in PS$ be compound (i.e., $\alpha(ps') \in C$) and *possibly* be ordered before ps (i.e., $(ps, ps') \notin \prec$).

What to do *exactly* to offer modifications that address/resolve (v, ps)?

- Only checking the very next level of $\alpha(ps')$ (i.e., the tasks in the methods of $\alpha(ps')$) is *not* sufficient and might lead to an incomplete algorithm.
- We need a mapping from each compound task to each reachable state variable. For efficiency reasons, this has to be done *once* in a preprocessing step.
- How to deal with cylces?

Alterations to POCL Planning, Open Preconditions, cont'd II

Let $ps \in PS$ be (primitive), $ps' \in PS$ (compound), and (v, ps) (open condition) as before.

- Let the planning problem be acyclic. Then, we can offer *one modification* for each producer for (v, ps). Note that this might include applying methods over several levels of abstraction at once.

Alterations to POCL Planning, Open Preconditions, cont'd II

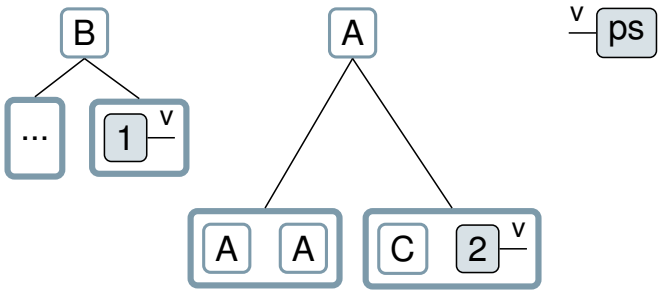
Let $ps \in PS$ be (primitive), $ps' \in PS$ (compound), and (v, ps) (open condition) as before.

- Let the planning problem be cyclic. With the previous strategy, there might be *infinitely many* modifications. Otherwise, we might become incomplete:

Alterations to POCL Planning, Open Preconditions, cont'd II

Let $ps \in PS$ be (primitive), $ps' \in PS$ (compound), and (v, ps) (open condition) as before.

- Let the planning problem be cyclic. With the previous strategy, there might be *infinitely many* modifications. Otherwise, we might become incomplete:

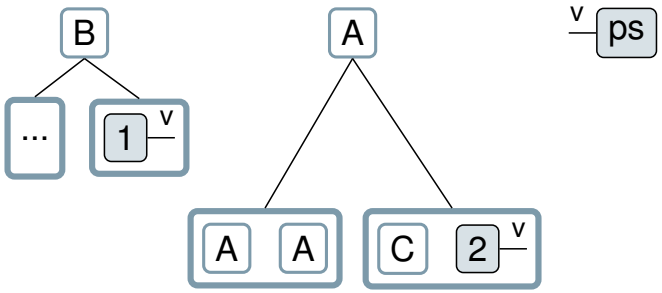


$\rightarrow$ Only offering two modifications (one for A and one for B) will wrongly prevent the planner from inserting C arbitrarily often.

Alterations to POCL Planning, Open Preconditions, cont'd II

Let $ps \in PS$ be (primitive), $ps' \in PS$ (compound), and (v, ps) (open condition) as before.

- Let the planning problem be cyclic. With the previous strategy, there might be *infinitely many* modifications. Otherwise, we might become incomplete:



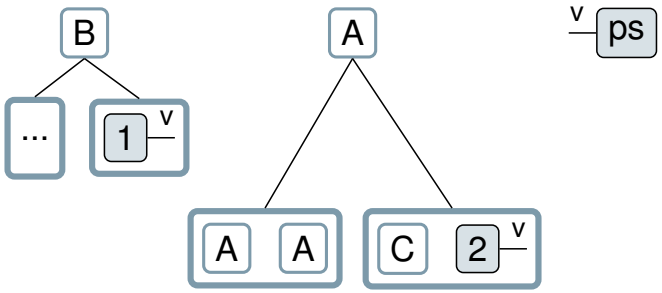
$\rightarrow$ Only offering two modifications (one for A and one for B) will wrongly prevent the planner from inserting C arbitrarily often.

- Solution (in such cyclic cases): We just decompose A, but without resolving the open precondition flaw. So, how many modifications do we get here?

Alterations to POCL Planning, Open Preconditions, cont'd II

Let $ps \in PS$ be (primitive), $ps' \in PS$ (compound), and (v, ps) (open condition) as before.

- Let the planning problem be cyclic. With the previous strategy, there might be *infinitely many* modifications. Otherwise, we might become incomplete:



$\rightarrow$ Only offering two modifications (one for A and one for B) will wrongly prevent the planner from inserting C arbitrarily often.

- Solution (in such cyclic cases): We just decompose A, but without resolving the open precondition flaw. So, how many modifications do we get here?
- Three! Two of them do insert a link and hence resolve the flaw.

Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

- If ps'' is primitive: Just as in POCL planning.

Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

- If ps'' is primitive: Just as in POCL planning.
- If ps'' is compound: If there is some primitive task reachable with v in its delete list (and the ordering restrictions as usual).

Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

- If ps'' is primitive: Just as in POCL planning.
- If ps'' is compound: If there is some primitive task reachable with v in its delete list (and the ordering restrictions as usual).

Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

- If ps'' is primitive: Just as in POCL planning.
- If ps'' is compound: If there is some primitive task reachable with v in its delete list (and the ordering restrictions as usual).

Modifications if ps'' is compound:

- Promotion and Demotion:

Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

- If ps'' is primitive: Just as in POCL planning.
- If ps'' is compound: If there is some primitive task reachable with v in its delete list (and the ordering restrictions as usual).

Modifications if ps'' is compound:

- Promotion and Demotion: Doing this is correct and resolves the flaw, but introduces non-systematicity and violates least commitment. Why?

Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

- If ps'' is primitive: Just as in POCL planning.
- If ps'' is compound: If there is some primitive task reachable with v in its delete list (and the ordering restrictions as usual).

Modifications if ps'' is compound:

- Promotion and Demotion: Doing this is correct and resolves the flaw, but introduces non-systematicity and violates least commitment. Why? Because it orders all sub tasks rather than just those required for eliminating the treatening step.

Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

- If ps'' is primitive: Just as in POCL planning.
- If ps'' is compound: If there is some primitive task reachable with v in its delete list (and the ordering restrictions as usual).

Modifications if ps'' is compound:

- Promotion and Demotion: Doing this is correct and resolves the flaw, but introduces non-systematicity and violates least commitment. Why? Because it orders all sub tasks rather than just those required for eliminating the treatening step.
- Decomposition: Could we just choose decompositions that prevent deleting v?

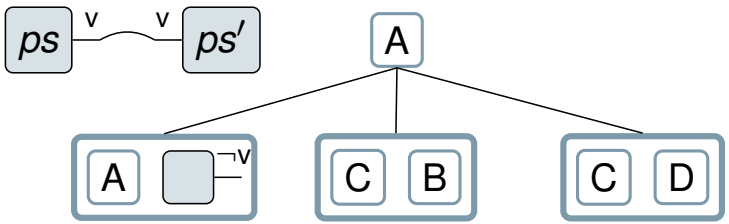
Alterations to POCL Planning, Causal Threats

Let $ps, ps' \in PS$ be primitive tasks sharing a causal link (ps, v, ps').
When does a further step $ps'' \in PS$ threaten that causal link?

- If ps'' is primitive: Just as in POCL planning.
- If ps'' is compound: If there is some primitive task reachable with v in its delete list (and the ordering restrictions as usual).

Modifications if ps'' is compound:

- Promotion and Demotion: Doing this is correct and resolves the flaw, but introduces non-systematicity and violates least commitment. Why? Because it orders all sub tasks rather than just those required for eliminating the treatening step.
- Decomposition: Could we just choose decompositions that prevent deleting v? No!



Plan Space-based HTN Planning, Pseudo Code

Algorithm: Plan space-based HTN Search

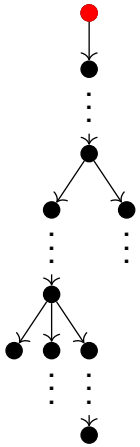
Input: An HTN problem $\mathcal{P} = (V, P, \delta, C, M, s_I, tn_I)$

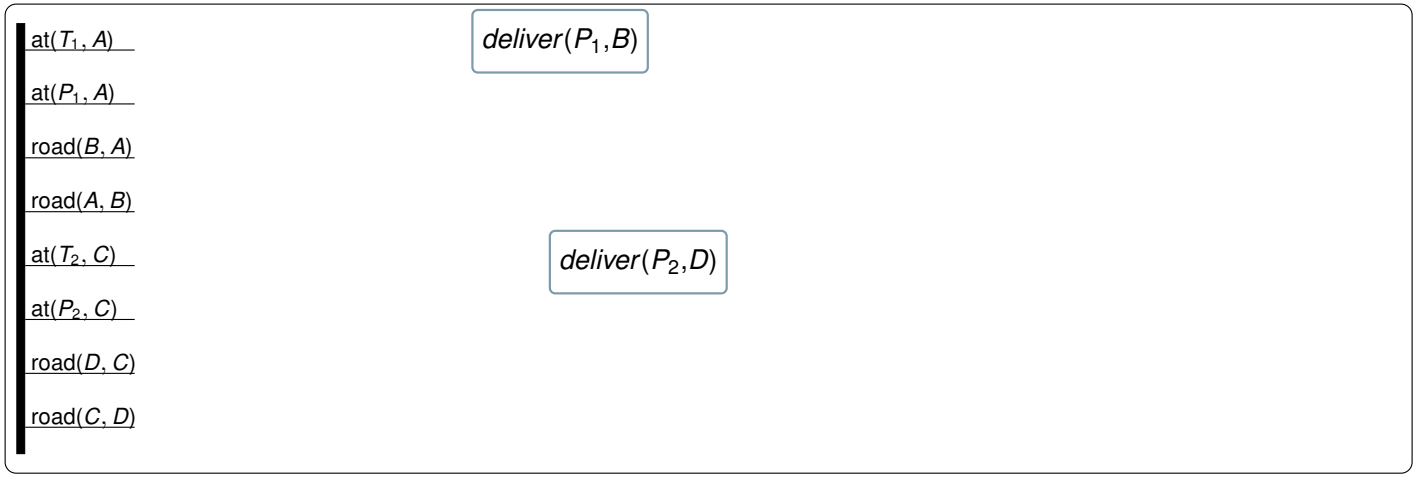
Output: A solution plan or ***fail***.

```
1 fringe = {P_I} // Created from tn_I as seen in first lecture.
2 while fringe ≠ ∅ do
3     P := nodeSelectAndRemove(()fringe)
4     F := flawDetection(P)
5     if F = ∅ then return P
6     f := flawSelection(F)
7     fringe := {applyModification(m, f) | m is a modification for f}
8 return fail
```

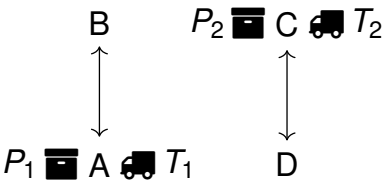
Note: Syntactically, this algorithm looks *exactly* like the POCL algorithm, but with flaws/modifications altered accordingly.

Plan Space-based HTN Planning, Example

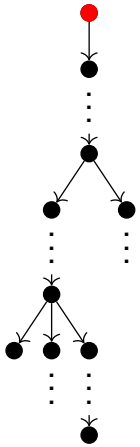


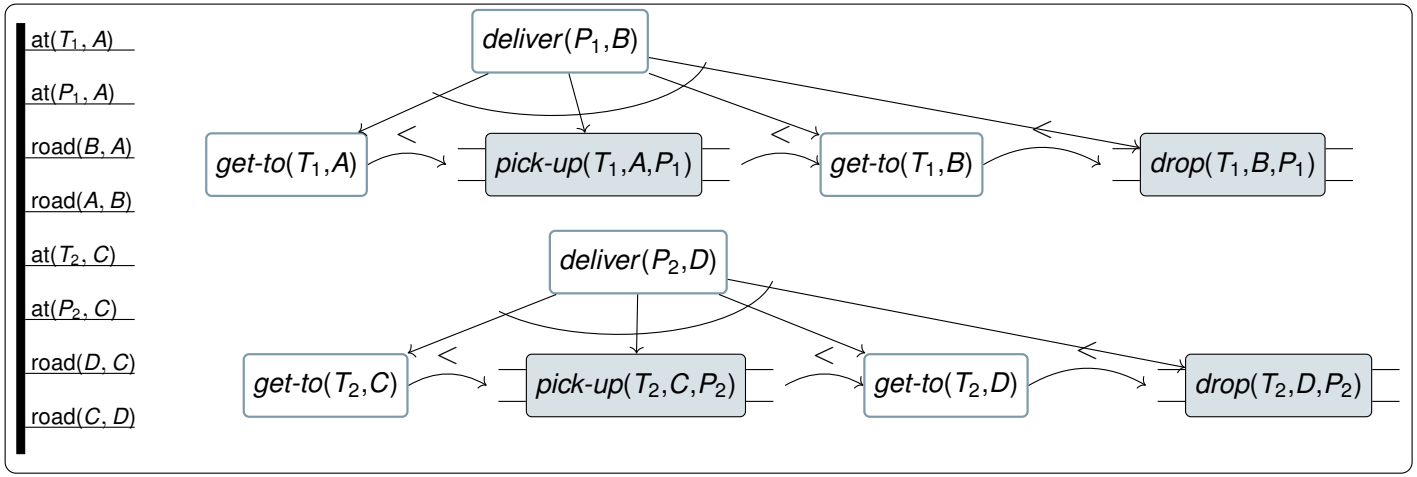


Flaws **Modifications**

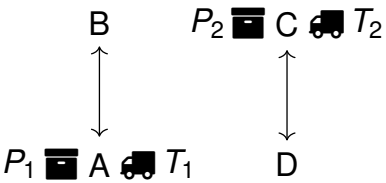


Plan Space-based HTN Planning, Example

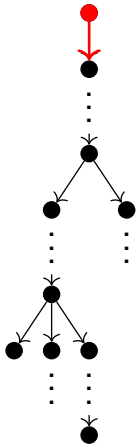


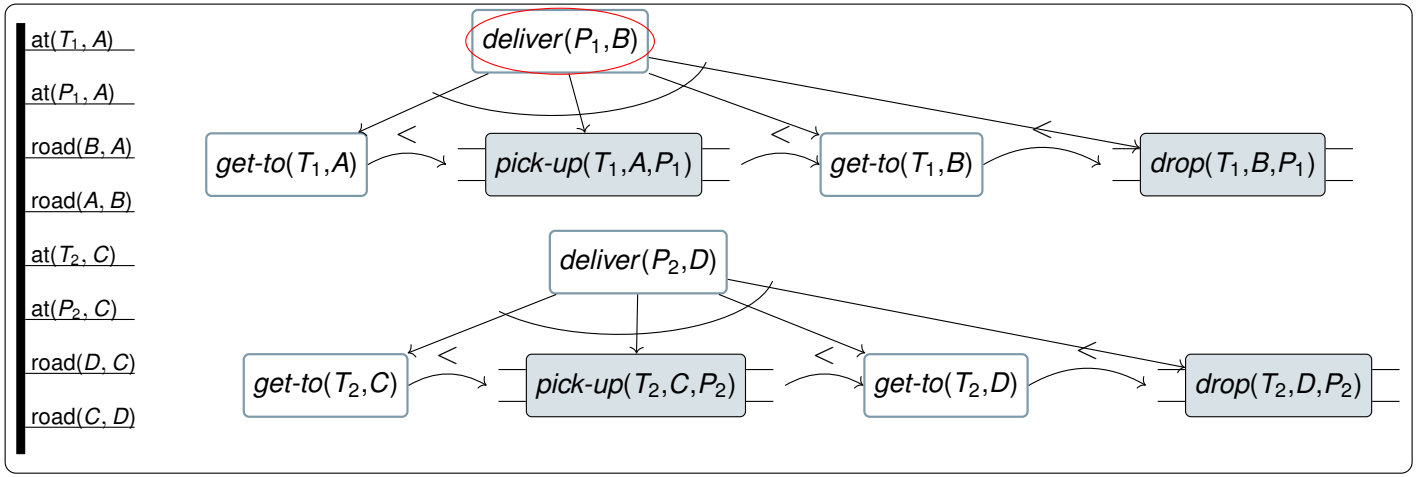


Flaws	**Modifications**
compound task: deliver(P_1, B)	decompose with *m-deliver(P_1, A, B, T_1)*
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*

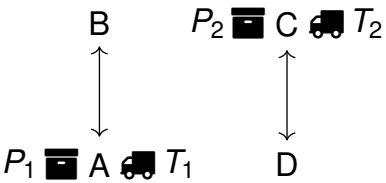


Plan Space-based HTN Planning, Example

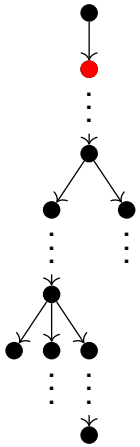


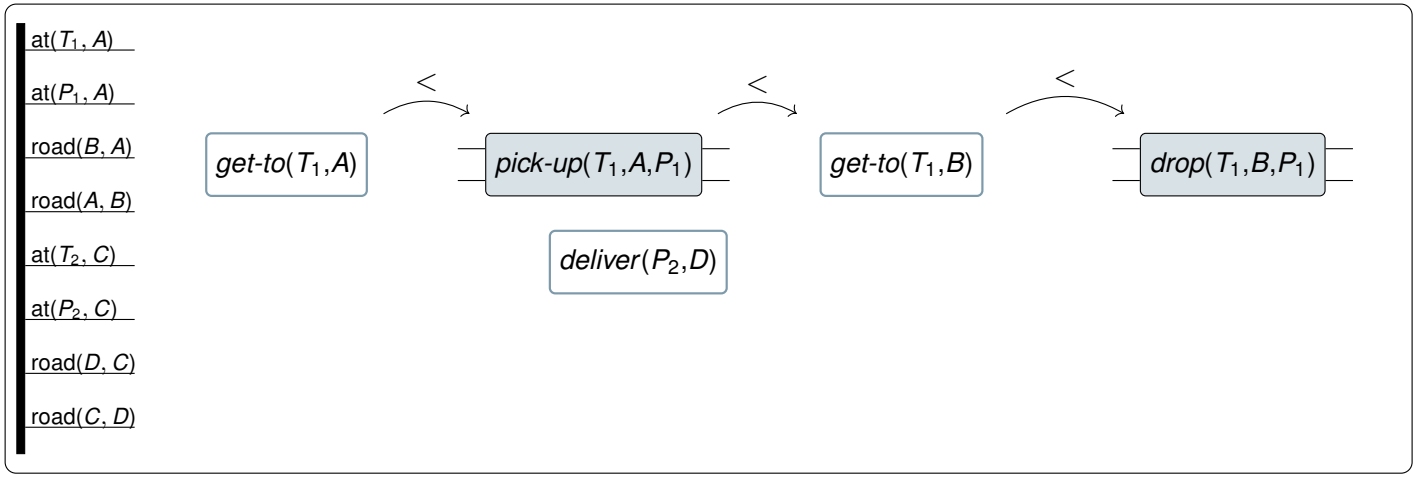


Flaws	**Modifications**
compound task: deliver(P_1, B)	**decompose with *m-deliver(P_1, A, B, T_1)***
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*

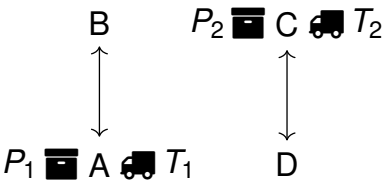


Plan Space-based HTN Planning, Example

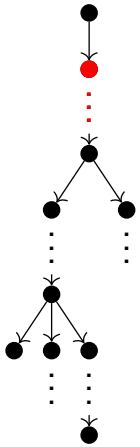


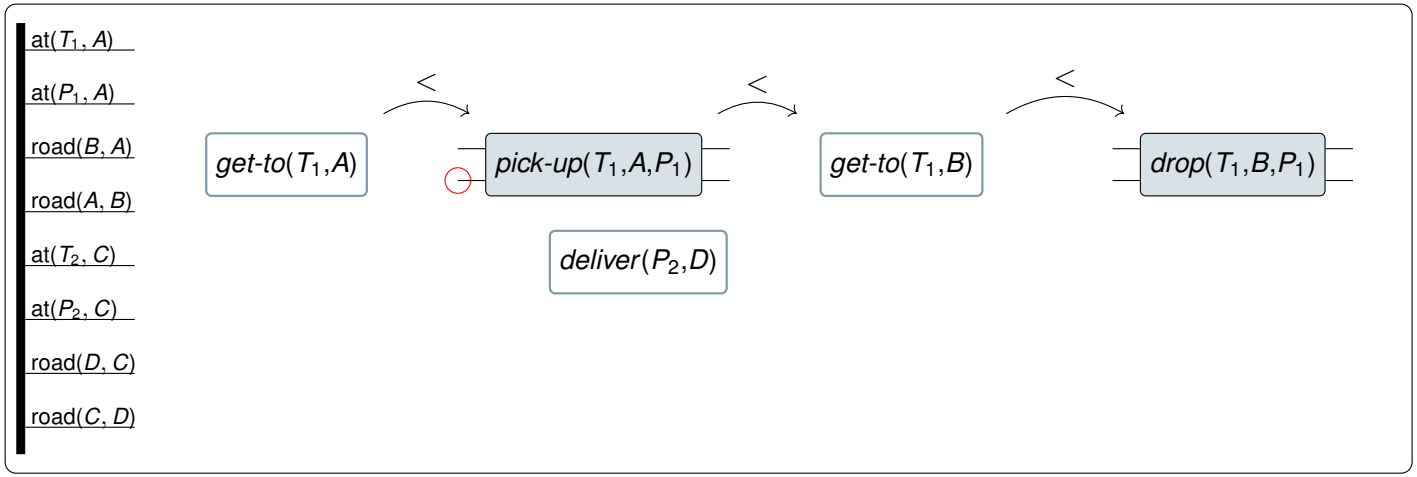


Flaws	**Modifications**
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from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

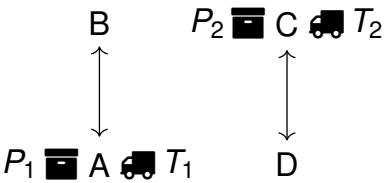


Plan Space-based HTN Planning, Example

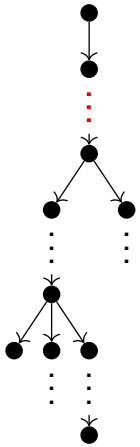


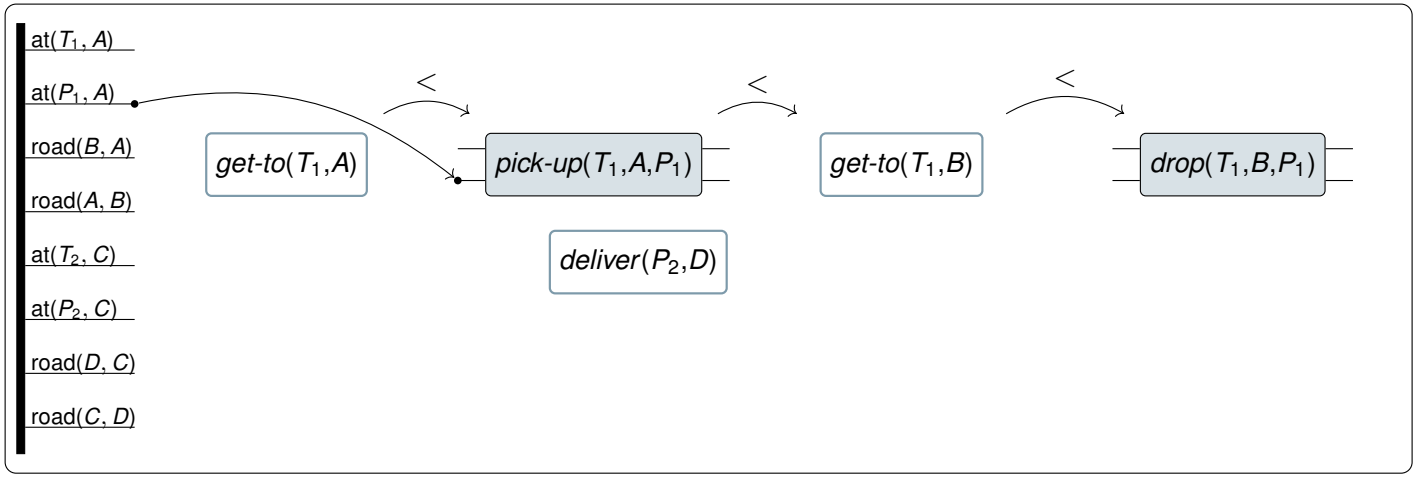


Flaws	**Modifications**
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from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

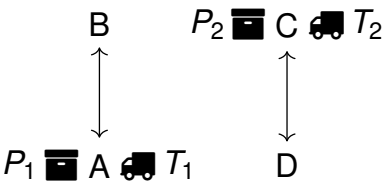


Plan Space-based HTN Planning, Example

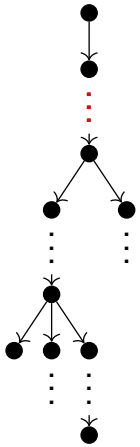


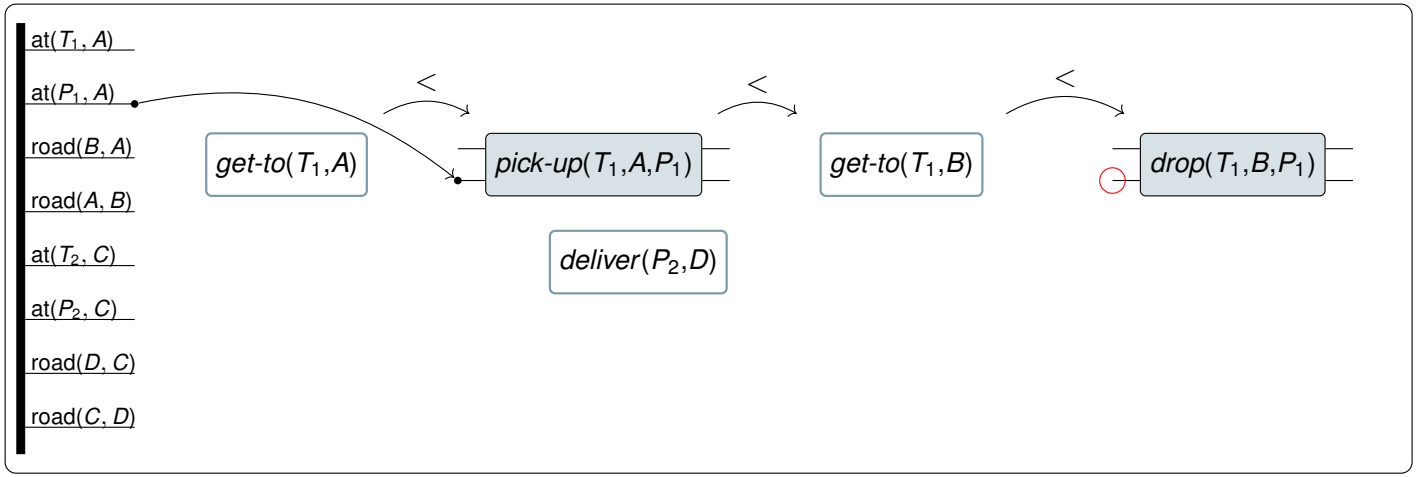


Flaws	**Modifications**
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from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

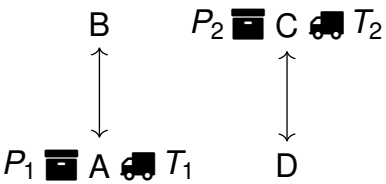


Plan Space-based HTN Planning, Example

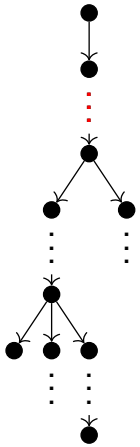


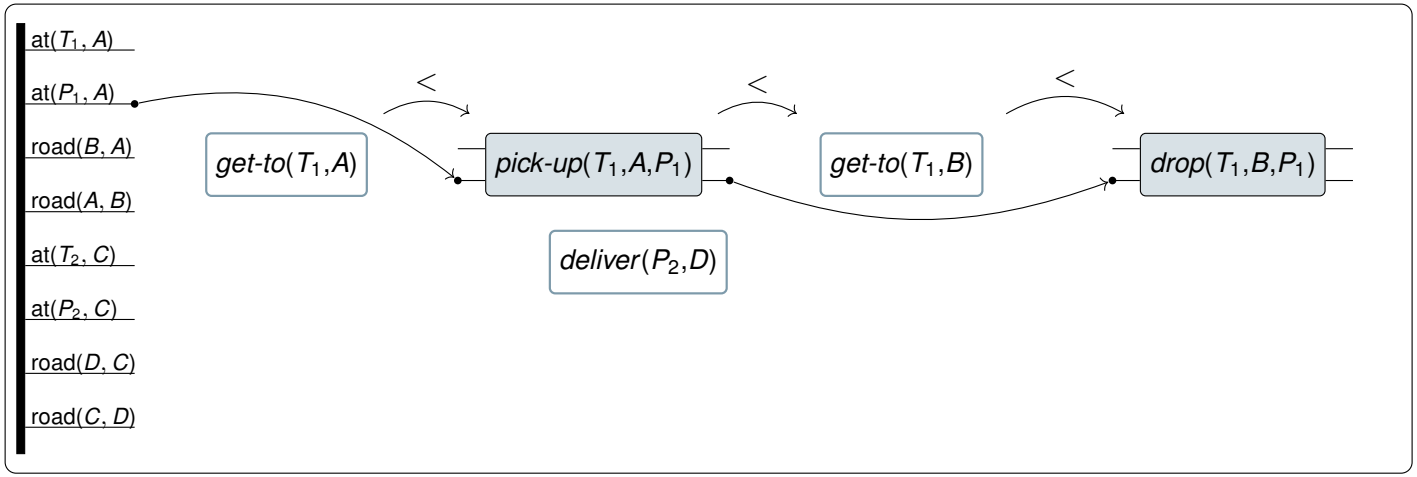


Flaws	**Modifications**
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from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

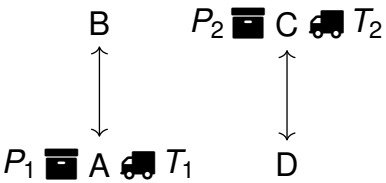


Plan Space-based HTN Planning, Example

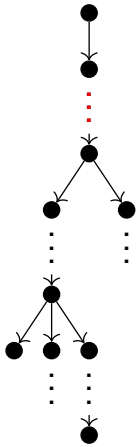


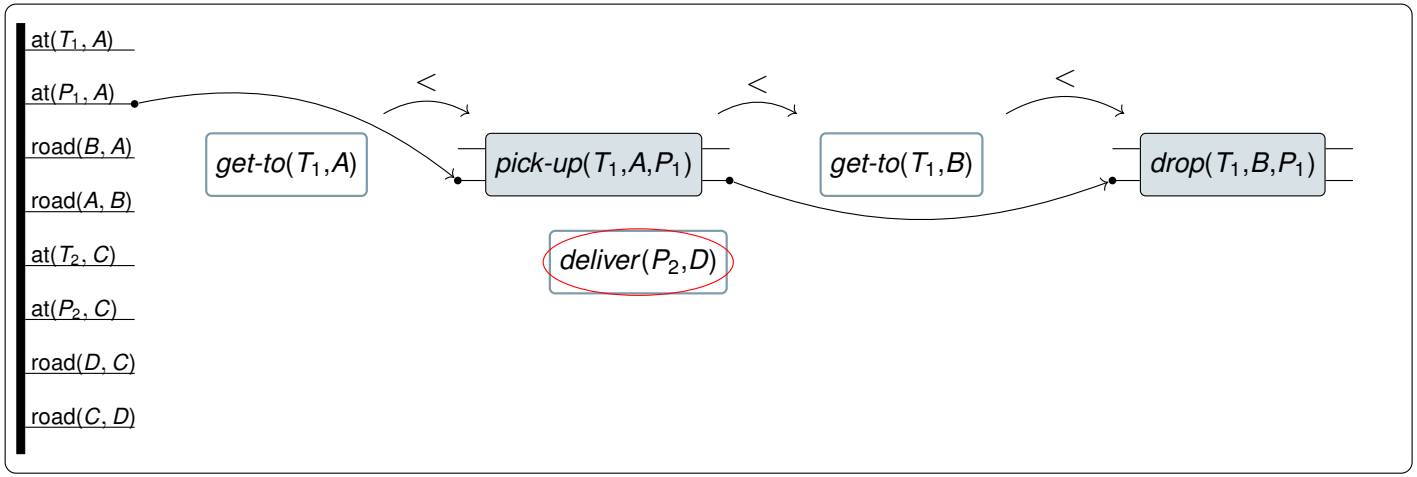


Flaws	**Modifications**
compound task: deliver(P_2, D)	decompose with *m-deliver(P_2, C, D, T_2)*
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from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

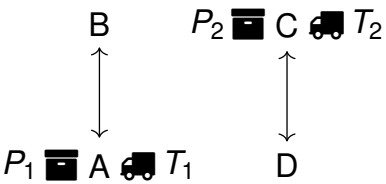


Plan Space-based HTN Planning, Example

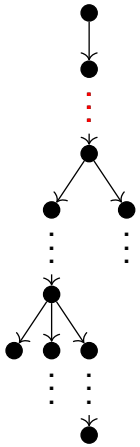


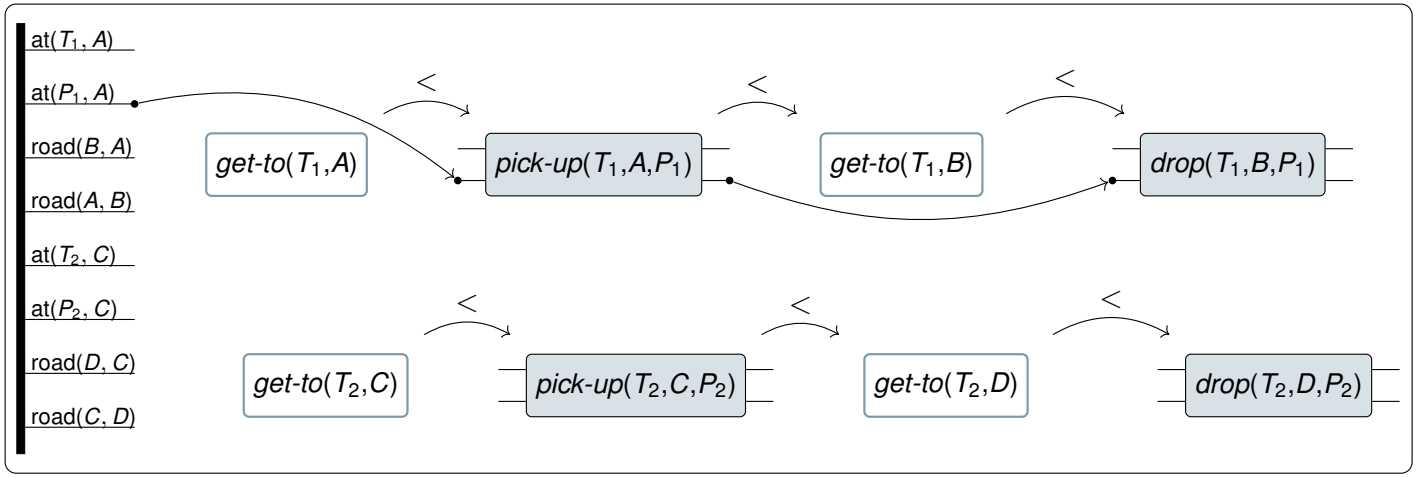


Flaws	**Modifications**
compound task: deliver(P_2, D)	**decompose with *m-deliver***(P_2, C, D, T_2)
compound task: get-to(T_1, A)	decompose with *m-direct*(T_1, B, A) decompose with *m-via*(T_1, B, A) decompose with *m-noop*(T_1, A)
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link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

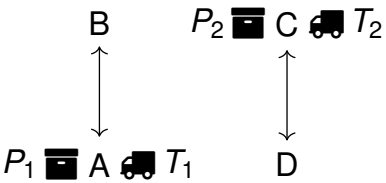


Plan Space-based HTN Planning, Example

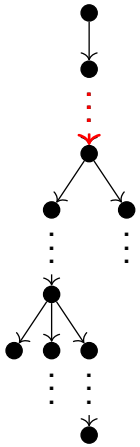


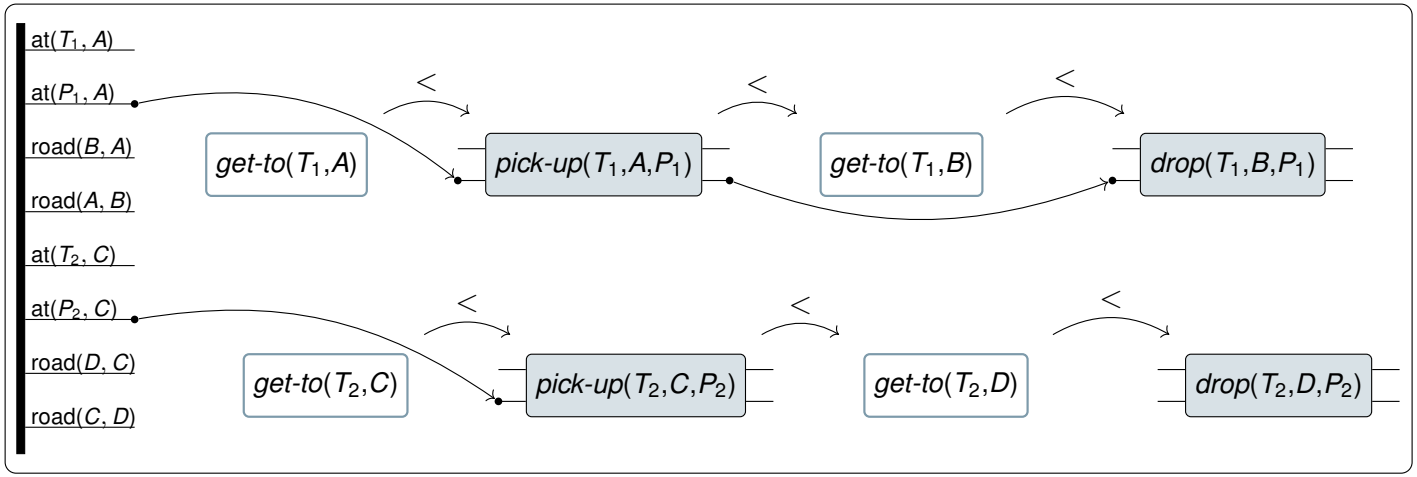


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

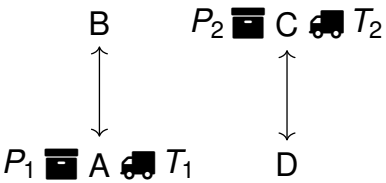


Plan Space-based HTN Planning, Example

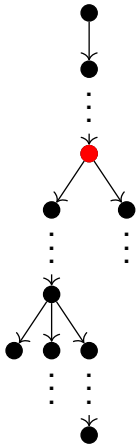


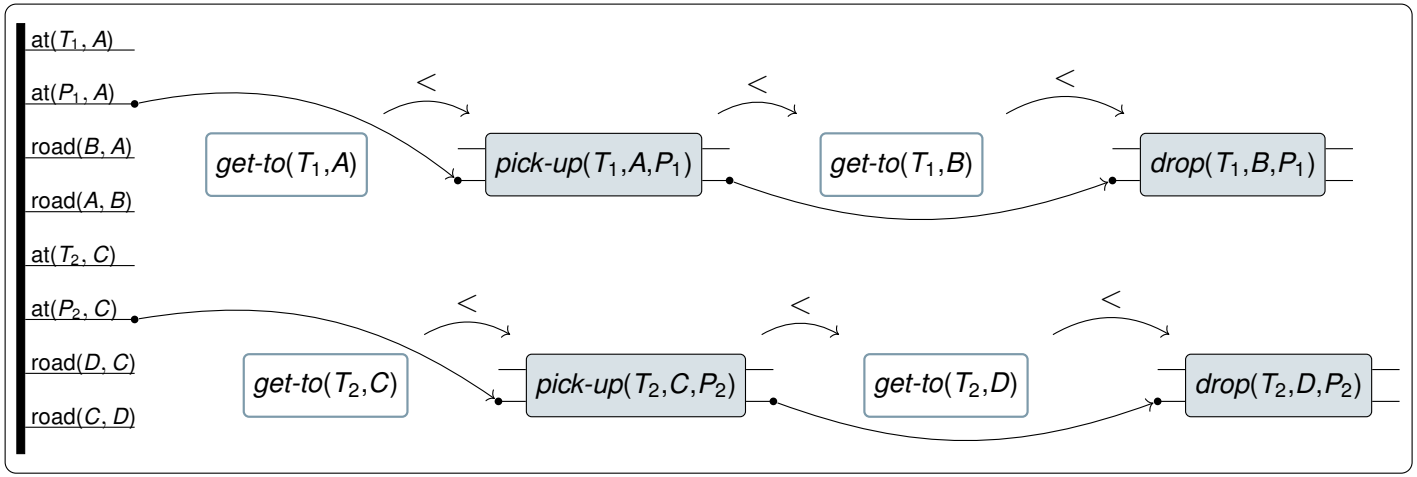


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

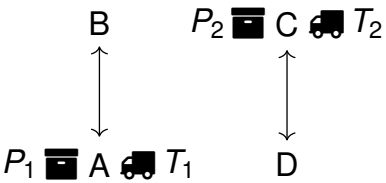


Plan Space-based HTN Planning, Example

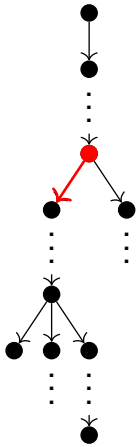


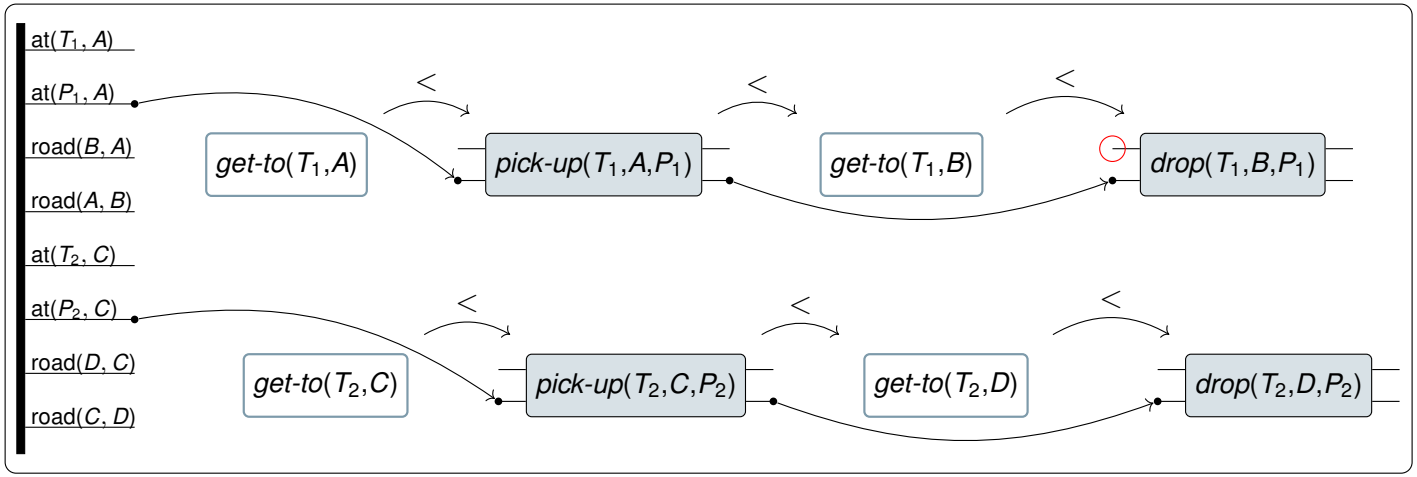


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

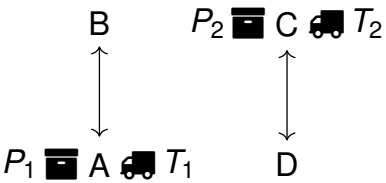


Plan Space-based HTN Planning, Example

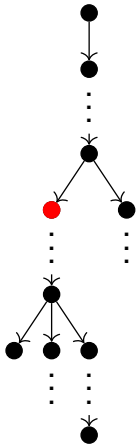


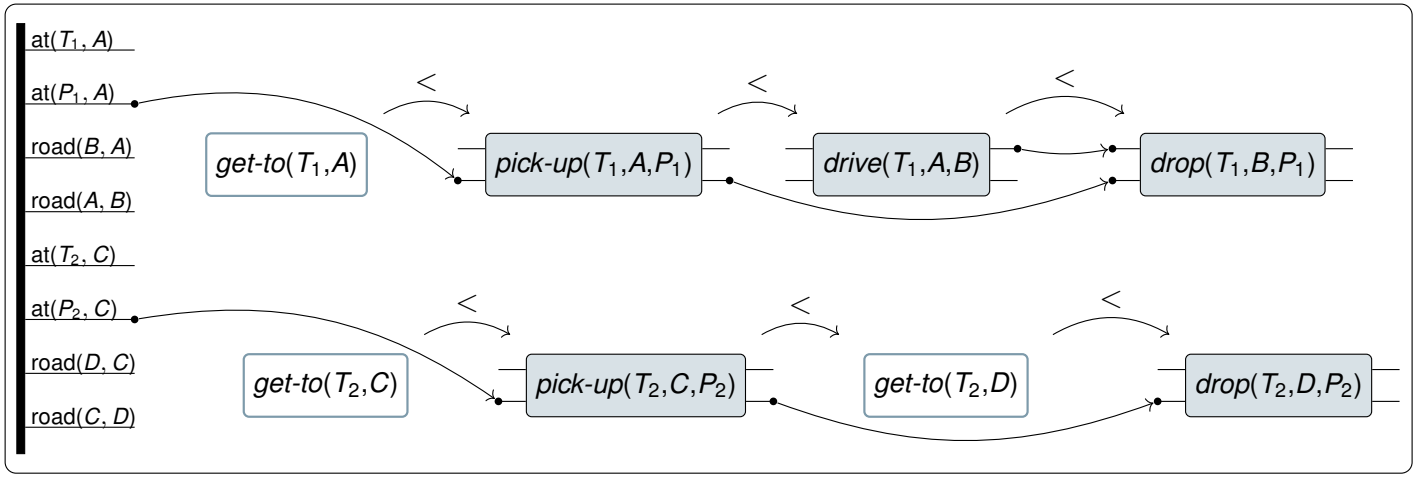


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

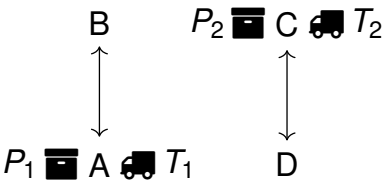


Plan Space-based HTN Planning, Example

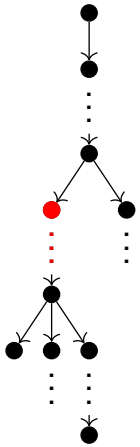


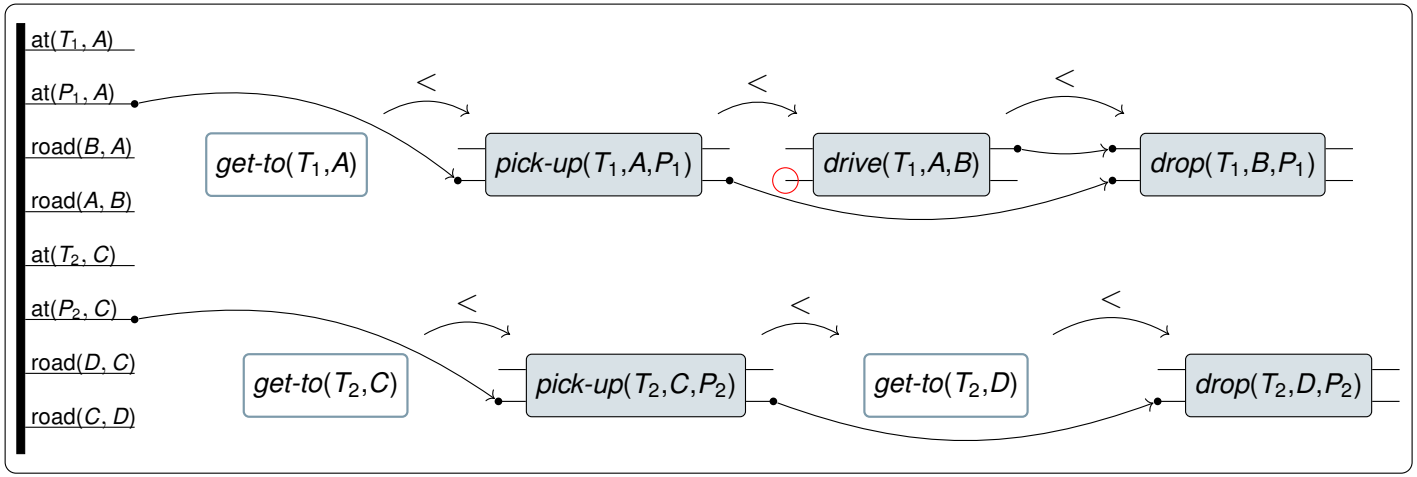


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

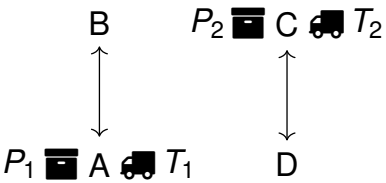


Plan Space-based HTN Planning, Example

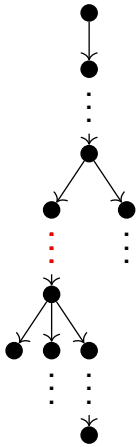


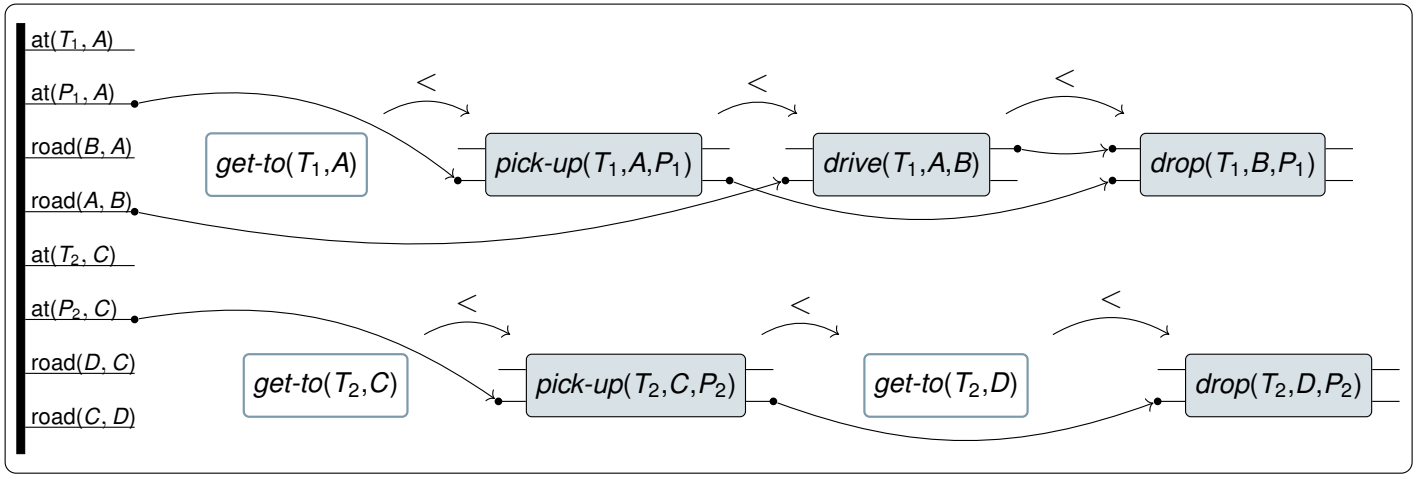


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

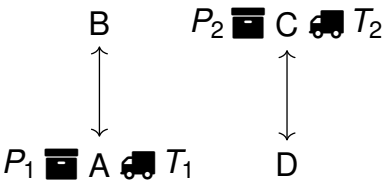


Plan Space-based HTN Planning, Example

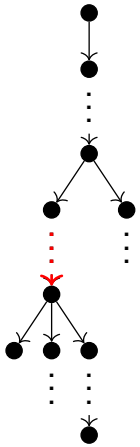


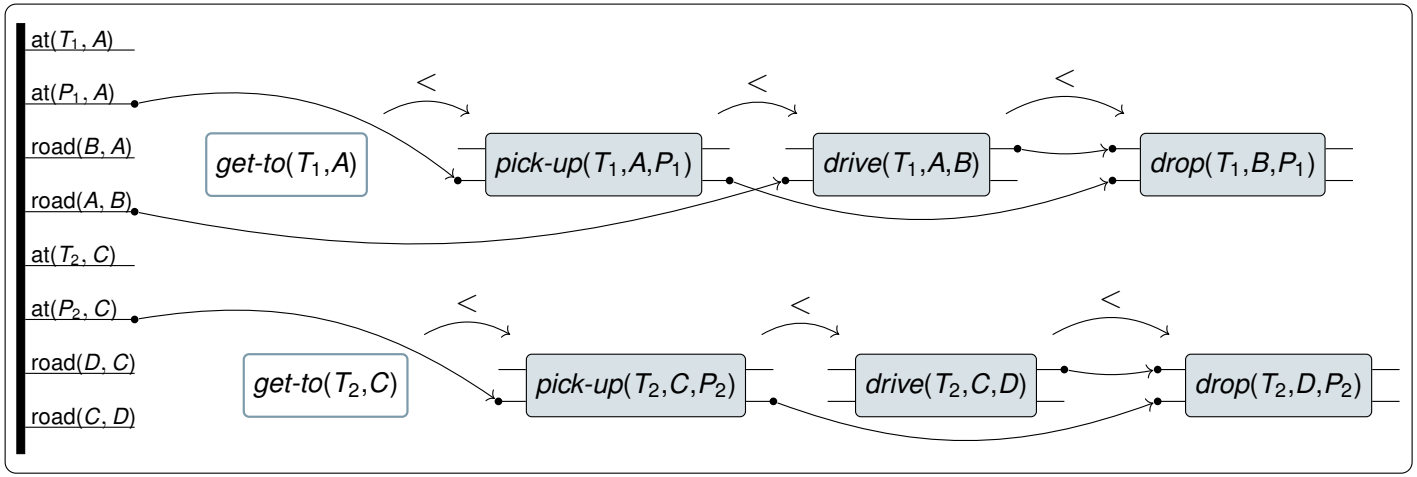


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

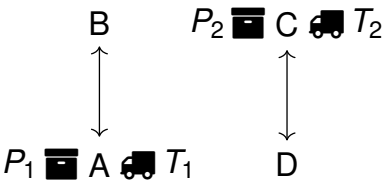


Plan Space-based HTN Planning, Example

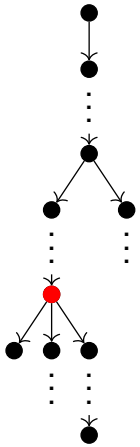


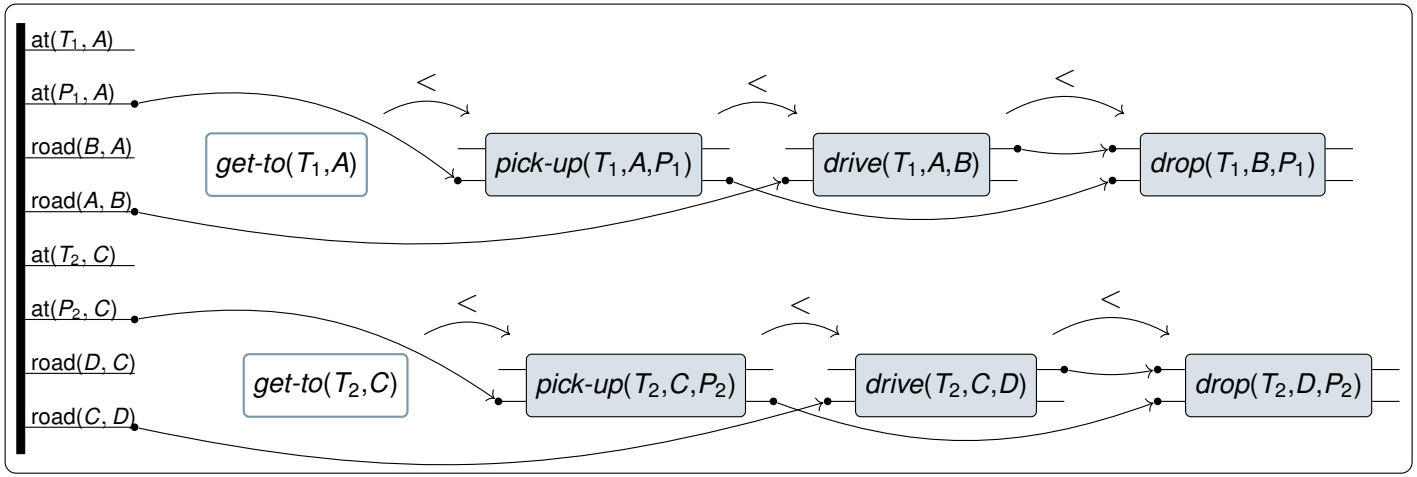


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

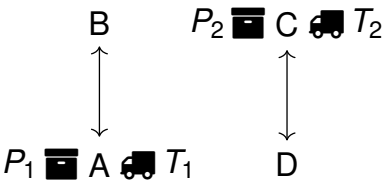


Plan Space-based HTN Planning, Example

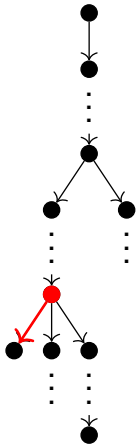


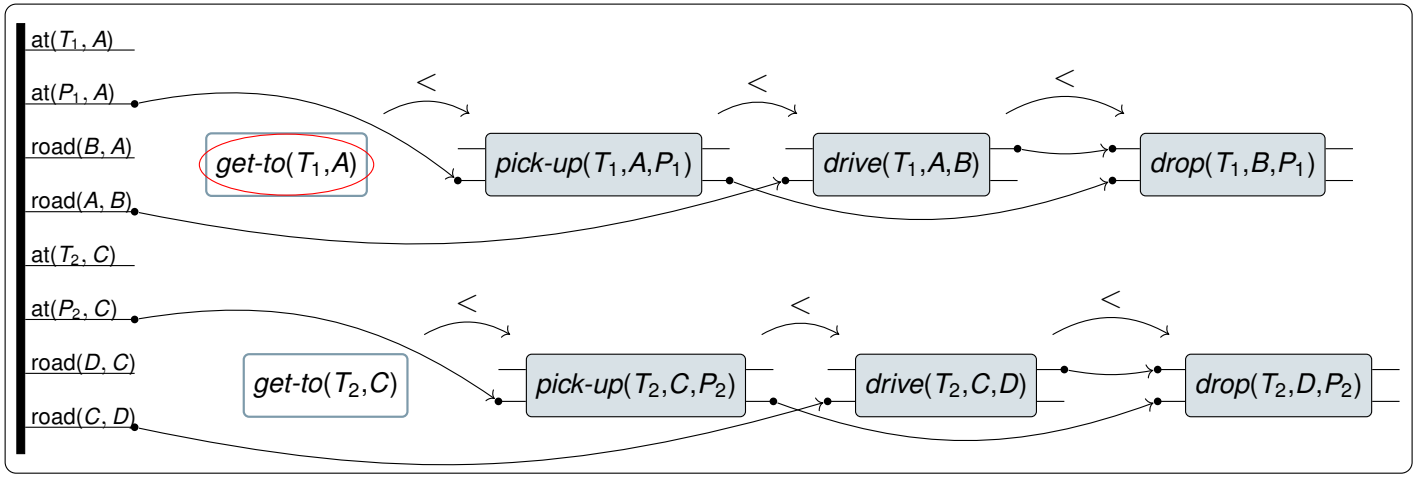


Flaws	**Modifications**
compound task: get-to(T_1, A)	decompose with *m-direct(T_1, B, A)* decompose with *m-via(T_1, B, A)* decompose with *m-noop(T_1, A)*
open prec.: at(T_1, A) of *pick-up(T_1, A, P_1)*	insert causal link from *init* decompose *get-to(T_1, A)* with *m-direct(T_1, B, A)* decompose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

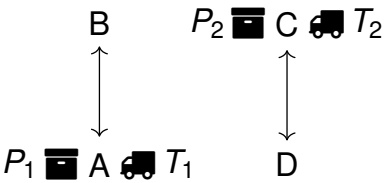


Plan Space-based HTN Planning, Example

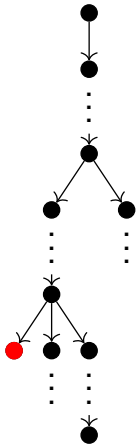


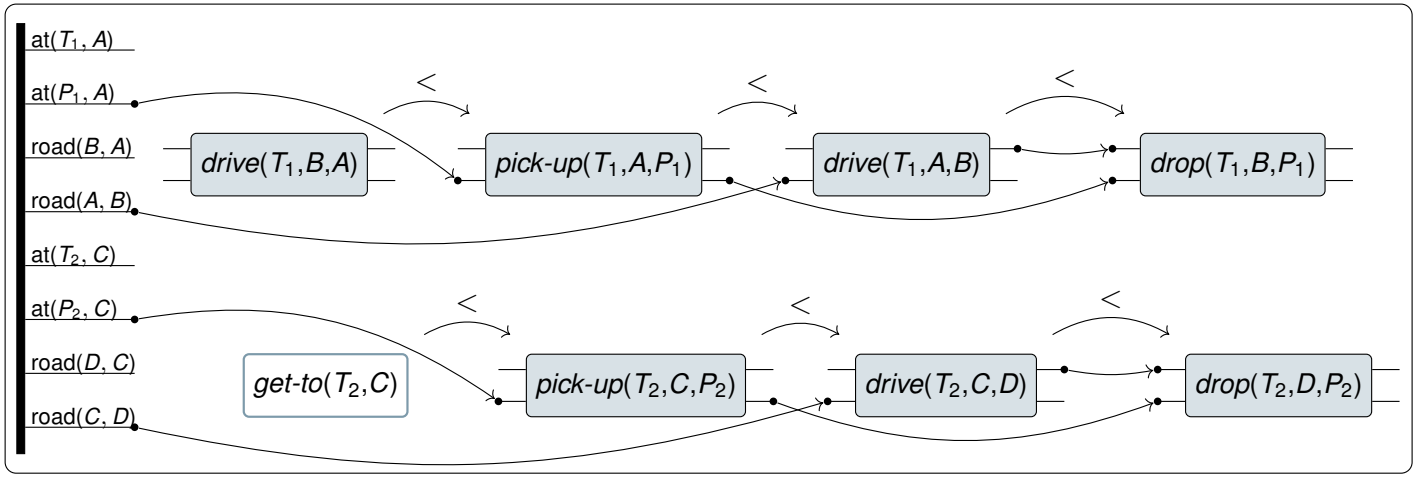


Flaws	**Modifications**
compound task: get-to(T_1, A)	**decompose with *m-direct***(T_1, B, A) decompose with *m-via*(T_1, B, A) decompose with *m-noop*(T_1, A)
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init* decompose *get-to*(T_1, A) with *m-direct*(T_1, B, A) decompose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

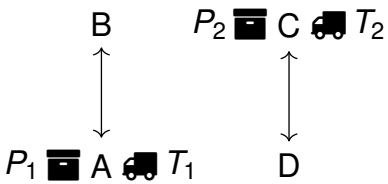


Plan Space-based HTN Planning, Example

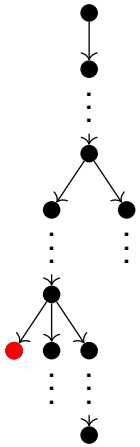


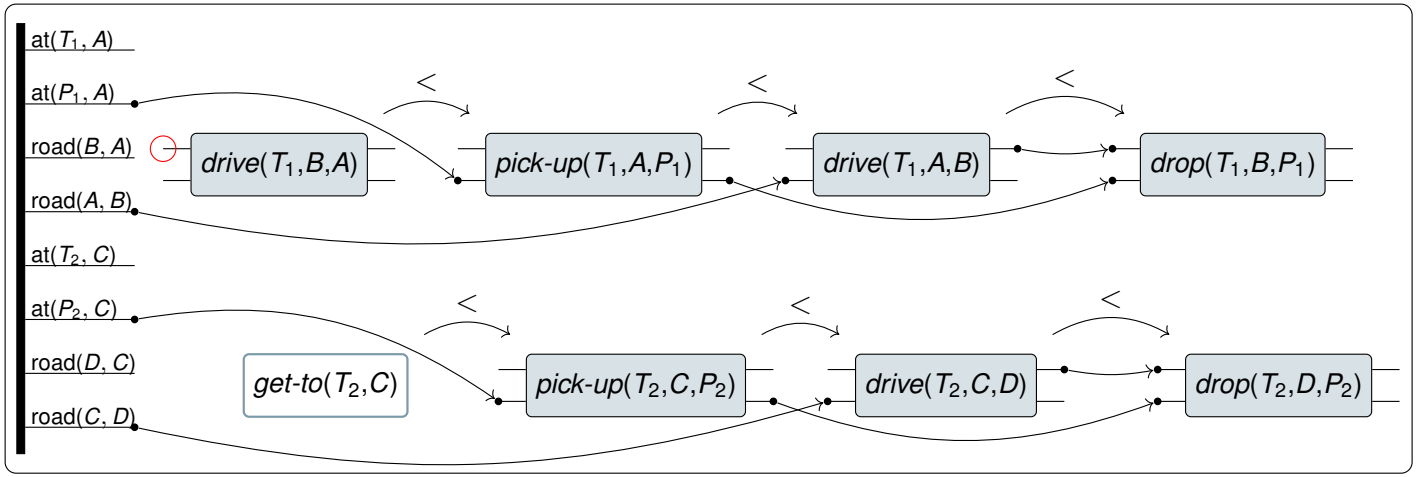


Flaws	**Modifications**
open prec: at(T_1, B) of *drive*(T_1, B, A)	—
open prec.: road(B, A) of *drive*(T_1, B, A)	insert causal link from *init*
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init* insert causal link from *drive*(T_1, B, A)
open prec.: at(T_1, A) of *drive*(T_1,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



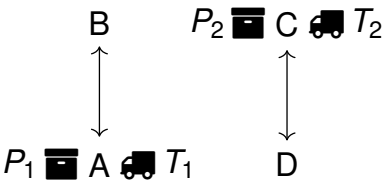
Plan Space-based HTN Planning, Example



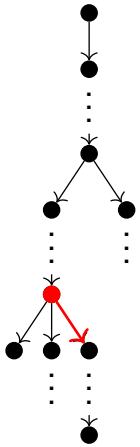


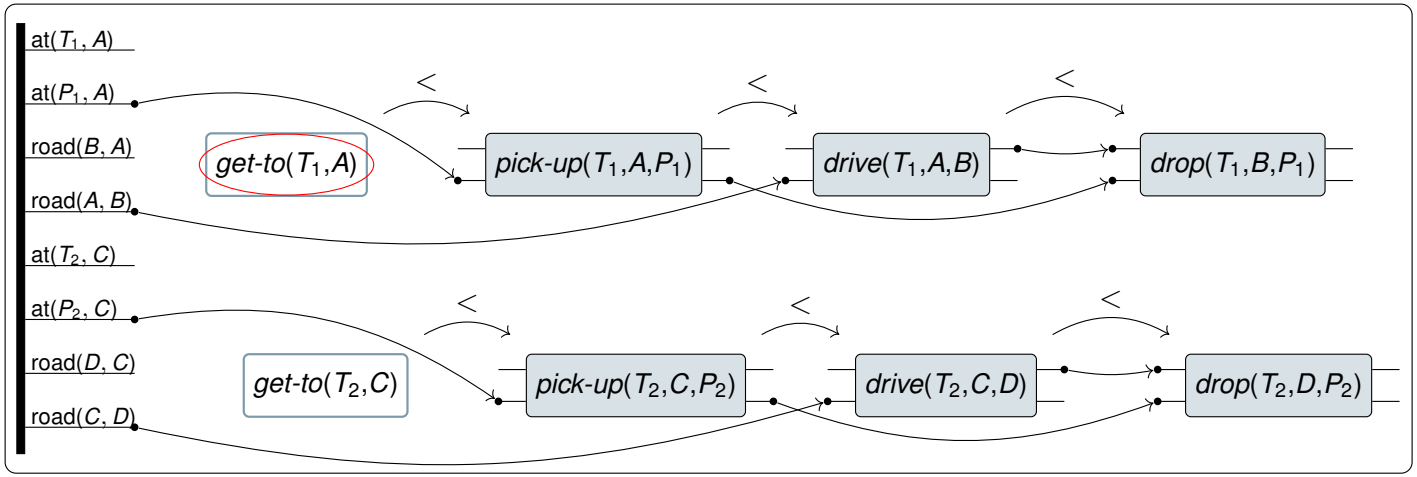
Flaws	**Modifications**
***open prec:* at(T_1, B) of *drive*(T_1, B, A)**	**—**
open prec.: road(B, A) of *drive*(T_1, B, A)	insert causal link from *init*
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init* insert causal link from *drive*(T_1, B, A)
open prec.: at(T_1, A)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

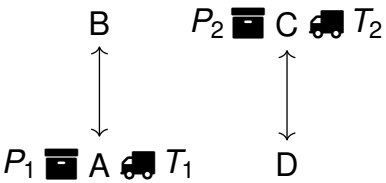


Plan Space-based HTN Planning, Example

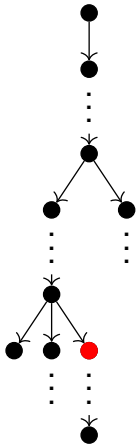


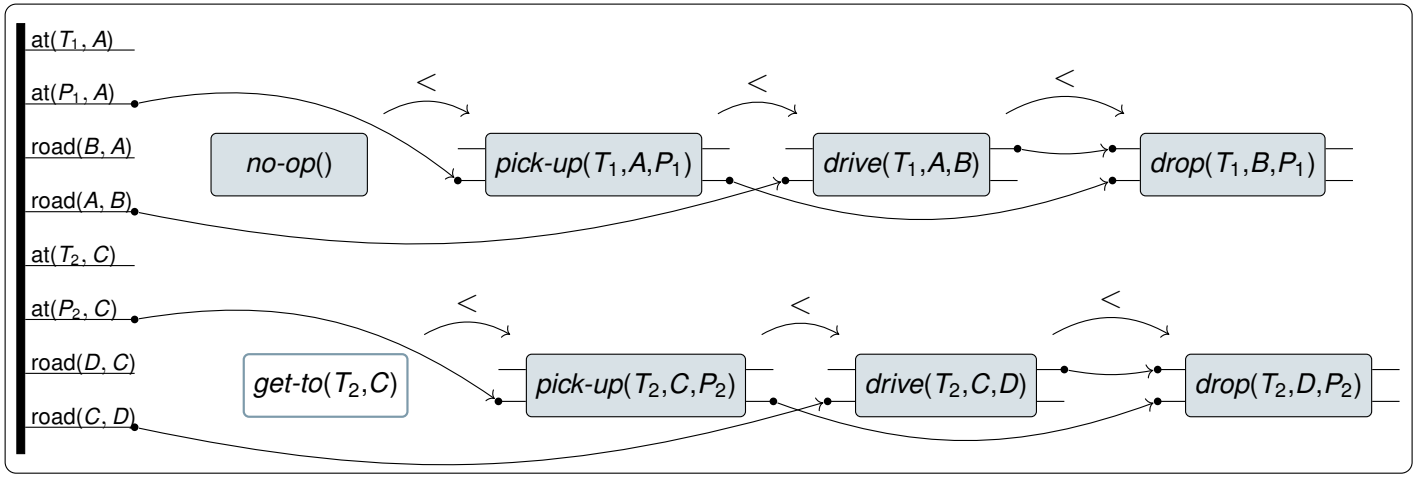


Flaws	**Modifications**
compound task: get-to(*T_1, A*)	decompose with *m-direct(*T_1, B, A*)* decompose with *m-via(*T_1, B, A*)* **decompose with *m-noop(*T_1, A*)***
open prec.: at(T_1, A) of *pick-up(*T_1, A, P_1*)*	insert causal link from *init* decompose *get-to(*T_1, A*)* with *m-direct(*T_1,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

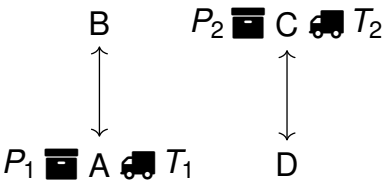


Plan Space-based HTN Planning, Example

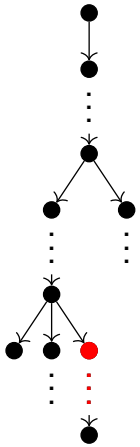


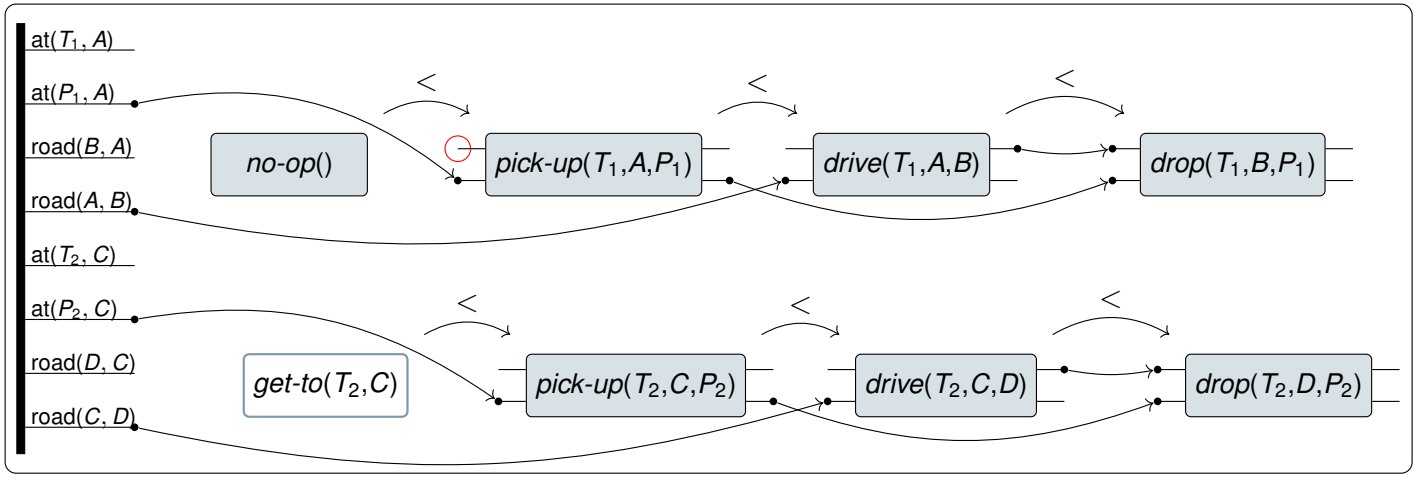


Flaws	**Modifications**
open prec.: at(T_1, A) of *pick-up*(T_1, A, P_1)	insert causal link from *init*
open prec.: at(T_1, A) of *drive*(T_1, A, B)	insert causal link from *init*
...	...

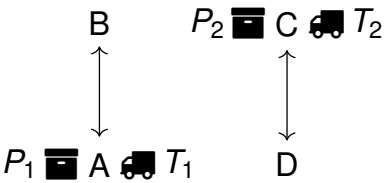


Plan Space-based HTN Planning, Example

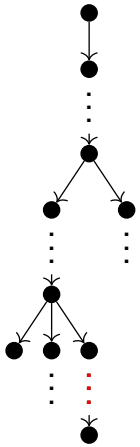


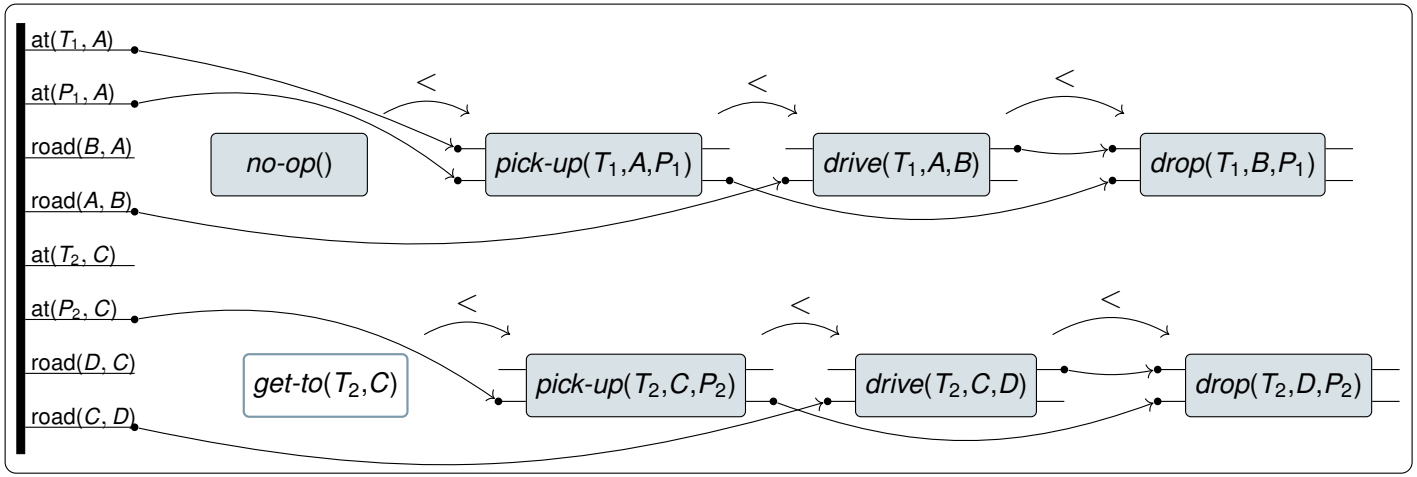


Flaws	**Modifications**
***open prec.:* at(T_1, A) of *pick-up*(T_1, A, P_1)**	**insert causal link from *init***
open prec.: at(T_1, A) of *drive*(T_1, A, B)	insert causal link from *init*
...	...

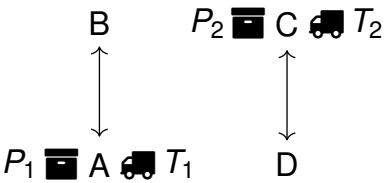


Plan Space-based HTN Planning, Example

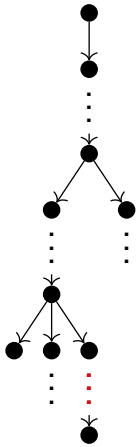


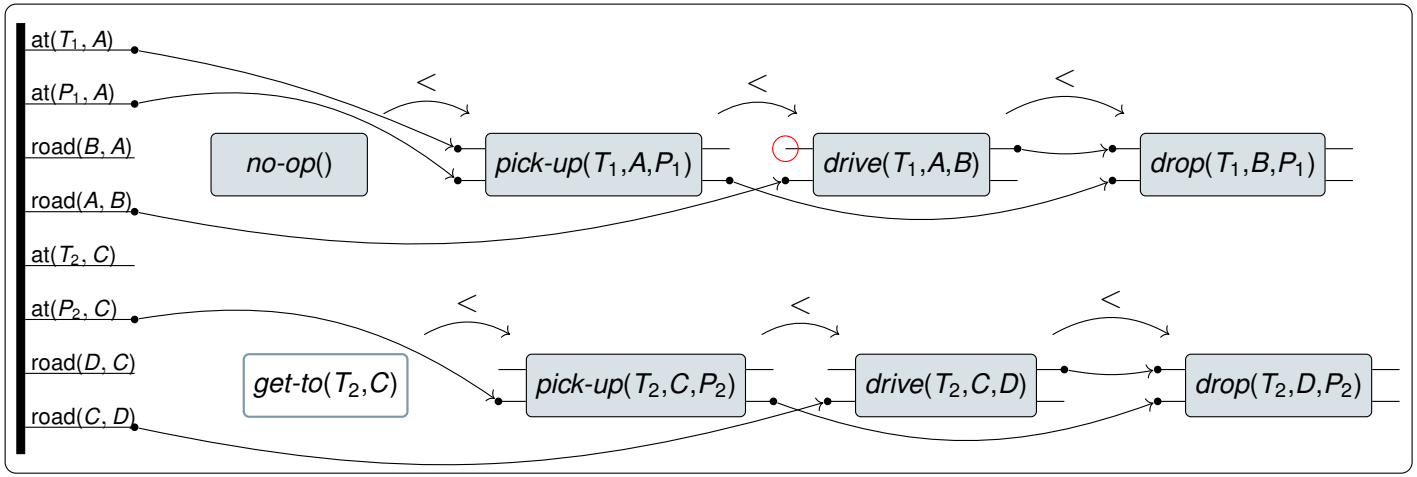


Flaws	**Modifications**
open prec.: at(T_1, A) of *drive*(T_1, A, B)	insert causal link from *init*
...	...

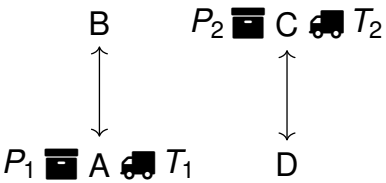


Plan Space-based HTN Planning, Example

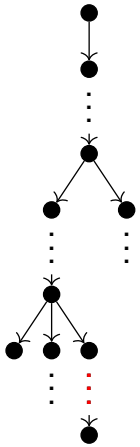


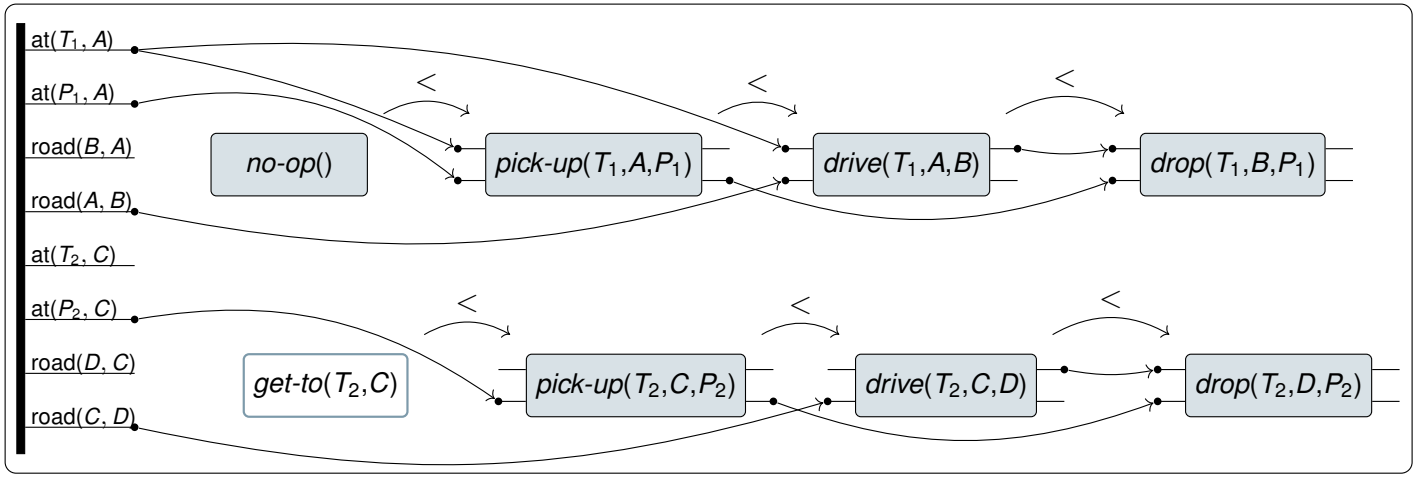


Flaws	**Modifications**
***open prec.:* at(T_1, A) of *drive*(T_1, A, B)**	**insert causal link from *init***
...	...

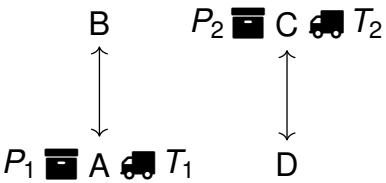


Plan Space-based HTN Planning, Example

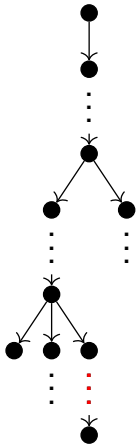


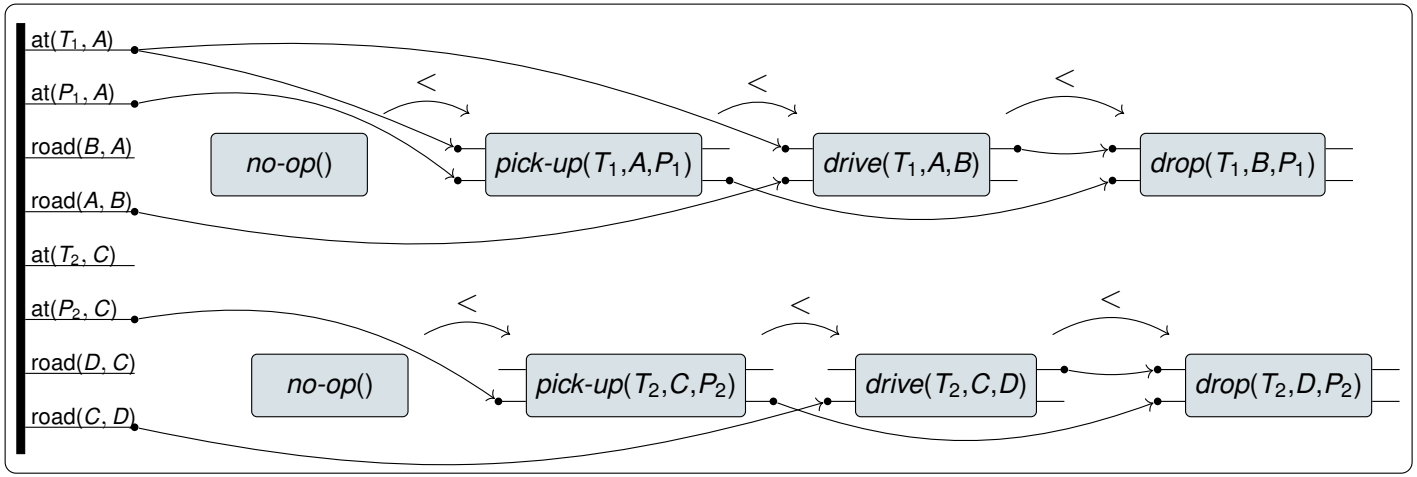


Flaws	Modifications
...	...

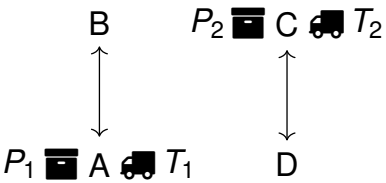


Plan Space-based HTN Planning, Example

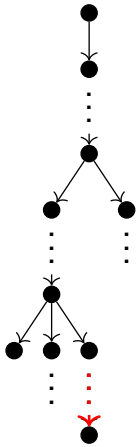


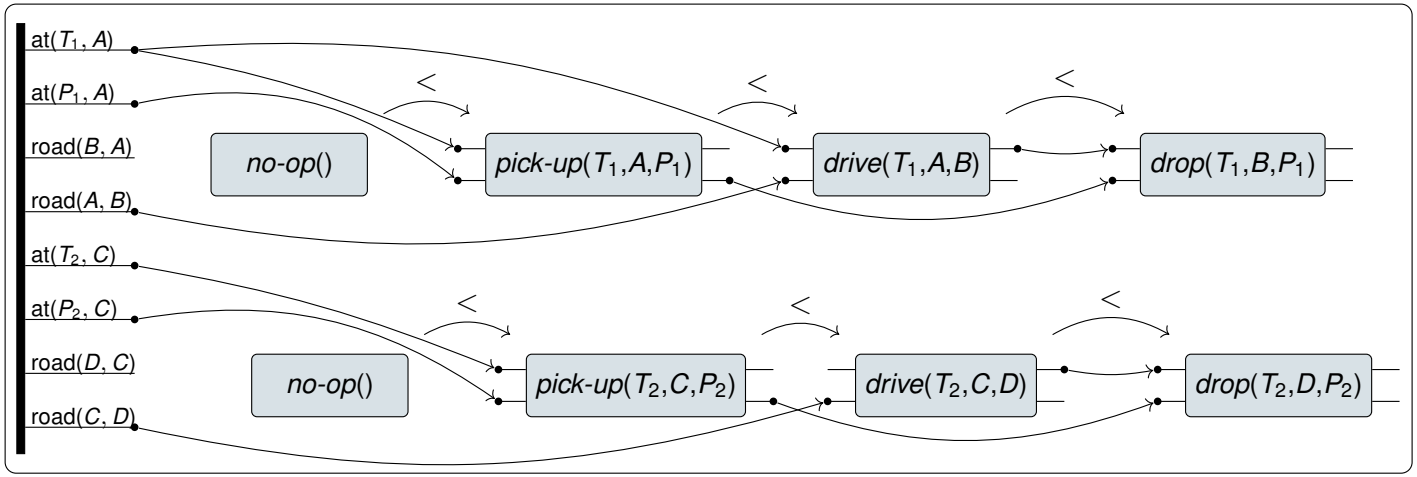


Flaws	Modifications
...	...

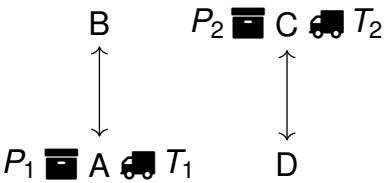


Plan Space-based HTN Planning, Example

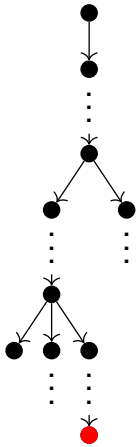


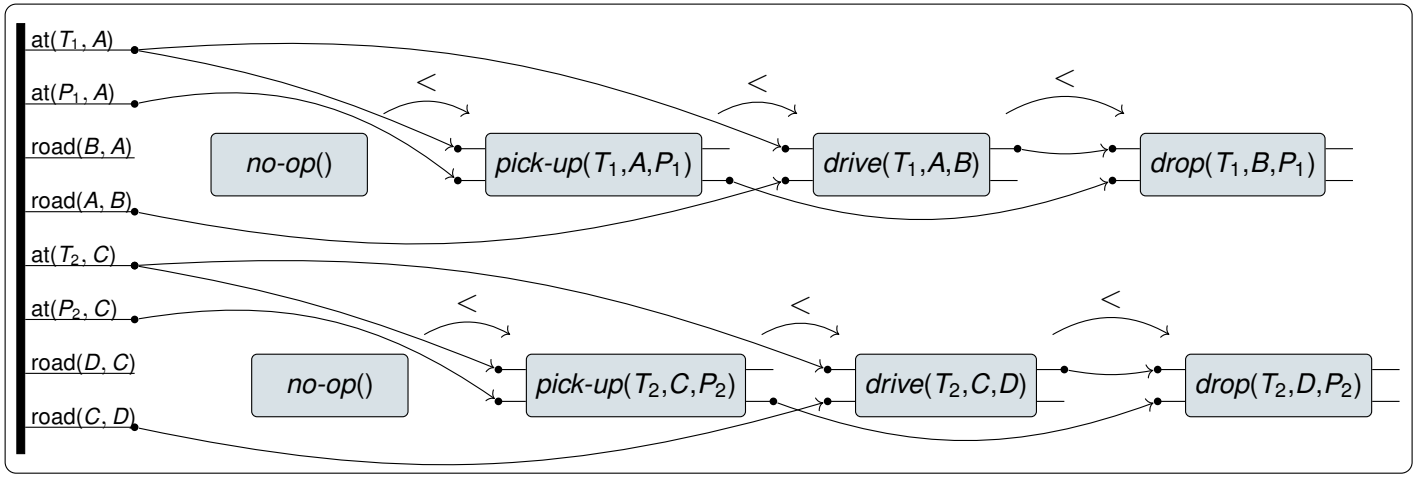


Flaws	Modifications
...	...



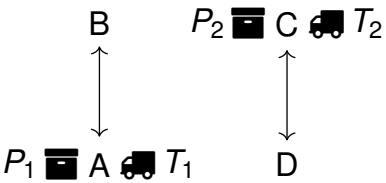
Plan Space-based HTN Planning, Example





Flaws **Modifications**

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



Flaw Selection Strategies

- Many of the flaw selection strategies for POCL planning can be reused for plan space-based HTN planning.
- As for POCL planning, one good possibility is LCFR. Further strategies might be discussed in the exercises.

Properties

Theorem

Plan space-based search is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*().

Properties

Theorem

Plan space-based search is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*().

Proof:
Follows from the properties of the underlying search algorithm.
However:

- Be aware that the transition system is not finite!
- We had to show that for each flaw, *all* possible ways to resolve it are generated and that no unintended side effects occur such as being overly restrictive thereby unintentionally ruling out solutions.

Extensions

- Method preconditions:

Extensions

- Method preconditions: They can be handled via compilation. How?

Extensions

- Method preconditions: They can be handled via compilation.
  How? *Exercise!*

Extensions

- Method preconditions: They can be handled via compilation. How? *Exercise!*
- TIHTN problems:

Extensions

- Method preconditions: They can be handled via compilation. How? *Exercise!*
- TIHTN problems:
  - Plan space-based search is also applicable for TIHTN problems.

Extensions

- Method preconditions: They can be handled via compilation. How? *Exercise!*
- TIHTN problems:
  - Plan space-based search is also applicable for TIHTN problems.
  - The only required extension is to re-enable action insertion as in POCL planning.

Extensions

- Method preconditions: They can be handled via compilation. How? *Exercise!*
- TIHTN problems:
  - Plan space-based search is also applicable for TIHTN problems.
  - The only required extension is to re-enable action insertion as in POCL planning.
- Goal description:

Extensions

- Method preconditions: They can be handled via compilation. How? *Exercise!*
- TIHTN problems:
  - Plan space-based search is also applicable for TIHTN problems.
  - The only required extension is to re-enable action insertion as in POCL planning.
- Goal description: Just add the artificial goal action as in POCL planning.

Extensions

- Method preconditions: They can be handled via compilation. How? *Exercise!*
- TIHTN problems:
  - Plan space-based search is also applicable for TIHTN problems.
  - The only required extension is to re-enable action insertion as in POCL planning.
- Goal description: Just add the artificial goal action as in POCL planning.
- State constraints:

Extensions

- Method preconditions: They can be handled via compilation. How? *Exercise!*
- TIHTN problems:
  - Plan space-based search is also applicable for TIHTN problems.
  - The only required extension is to re-enable action insertion as in POCL planning.
- Goal description: Just add the artificial goal action as in POCL planning.
- State constraints: Unclear/not yet implemented/published.

Extensions

- Method preconditions: They can be handled via compilation. How? *Exercise!*
- TIHTN problems:
  - Plan space-based search is also applicable for TIHTN problems.
  - The only required extension is to re-enable action insertion as in POCL planning.
- Goal description: Just add the artificial goal action as in POCL planning.
- State constraints: Unclear/not yet implemented/published.
- Extension to hybrid planning, where compound tasks show preconditions and effects as well: Discussed at the end of the lecture if time.

Summary

- Again, do not mistake hierarchical planning algorithms as “just another algorithm for solving planning problems” – they are required to solve hierarchical problems, which are more expressive than non-hierarchical ones (confer last lecture!).

Summary

- Again, do not mistake hierarchical planning algorithms as “just another algorithm for solving planning problems” – they are required to solve hierarchical problems, which are more expressive than non-hierarchical ones (confer last lecture!).
- Similar to solving non-hierarchical problems,

Summary

- Again, do not mistake hierarchical planning algorithms as “just another algorithm for solving planning problems” – they are required to solve hierarchical problems, which are more expressive than non-hierarchical ones (confer last lecture!).
- Similar to solving non-hierarchical problems,
  - planning as search is one of the standard approaches for solving hierarchical planning problems,

Summary

- Again, do not mistake hierarchical planning algorithms as “just another algorithm for solving planning problems” – they are required to solve hierarchical problems, which are more expressive than non-hierarchical ones (confer last lecture!).
- Similar to solving non-hierarchical problems,
  - planning as search is one of the standard approaches for solving hierarchical planning problems,
  - there is progression-based search in the space of states (plus the remaining task network),

Summary

- Again, do not mistake hierarchical planning algorithms as “just another algorithm for solving planning problems” – they are required to solve hierarchical problems, which are more expressive than non-hierarchical ones (confer last lecture!).
- Similar to solving non-hierarchical problems,
  - planning as search is one of the standard approaches for solving hierarchical planning problems,
  - there is progression-based search in the space of states (plus the remaining task network),
  - search in the space of partial plans, and

Summary

- Again, do not mistake hierarchical planning algorithms as “just another algorithm for solving planning problems” – they are required to solve hierarchical problems, which are more expressive than non-hierarchical ones (confer last lecture!).
- Similar to solving non-hierarchical problems,
  - planning as search is one of the standard approaches for solving hierarchical planning problems,
  - there is progression-based search in the space of states (plus the remaining task network),
  - search in the space of partial plans, and
  - both approaches rely on heuristics to guide search (next lecture).