Lecture *Hierarchical Planning*

Chapter:
Heuristics for (Hierarchical) Planning Problems

Dr. Pascal Bercher

Institute of Artificial Intelligence,
Ulm University, Germany

Winter Term 2018/2019

(Compiled on: February 20, 2019)

ulm university universität uulm

Overview:

Recap on Search in Non-hierarchical Planning

- In planning as search, we rely upon heuristics to guide search.

Recap on Search in Non-hierarchical Planning

- In planning as search, we rely upon heuristics to guide search.
- With the right combination of algorithm and heuristic, we can also provide optimality guarantees.

Recap on Search in Non-hierarchical Planning

- In planning as search, we rely upon heuristics to guide search.
- With the right combination of algorithm and heuristic, we can also provide optimality guarantees.
- In classical planning, heuristics estimate the number of actions (or their costs) that need to be applied to reach a goal state.

Recap on Search in Non-hierarchical Planning

- In planning as search, we rely upon heuristics to guide search.
- With the right combination of algorithm and heuristic, we can also provide optimality guarantees.
- In classical planning, heuristics estimate the number of actions (or their costs) that need to be applied to reach a goal state.
- In POCL planning, heuristics could also estimate the number of required modifications (which, in addition to task insertion, may estimate the number of ordering constraints and causal links that need to be added).

What about Search in Hierarchical Planning?

What's the different to hierarchical planning?

- In general, we don't have a goal description.

What about Search in Hierarchical Planning?

What's the different to hierarchical planning?

- In general, we don't have a goal description.
- In general, we don't allow task insertion, i.e., rather than estimating which tasks to insert we need to estimate how to decompose.

What about Search in Hierarchical Planning?

What's the different to hierarchical planning?

- In general, we don't have a goal description.
- In general, we don't allow task insertion, i.e., rather than estimating which tasks to insert we need to estimate how to decompose.
- We always have a *task network* (or *partial plan*), instead of/in addition to the initial state.

What about Search in Hierarchical Planning?

What's the different to hierarchical planning?

- In general, we don't have a goal description.
- In general, we don't allow task insertion, i.e., rather than estimating which tasks to insert we need to estimate how to decompose.
- We always have a *task network* (or *partial plan*), instead of/in addition to the initial state.
- We have to deal with the partial order.

What about Search in Hierarchical Planning?

What's the different to hierarchical planning?

- In general, we don't have a goal description.
- In general, we don't allow task insertion, i.e., rather than estimating which tasks to insert we need to estimate how to decompose.
- We always have a *task network* (or *partial plan*), instead of/in addition to the initial state.
- We have to deal with the partial order.
- We have to deal with abstract tasks (see above).

What about Search in Hierarchical Planning?

What's the different to hierarchical planning?

- In general, we don't have a goal description.
- In general, we don't allow task insertion, i.e., rather than estimating which tasks to insert we need to estimate how to decompose.
- We always have a *task network* (or *partial plan*), instead of/in addition to the initial state.
- We have to deal with the partial order.
- We have to deal with abstract tasks (see above).
- Additional challenges for decomposition-based planning:

What about Search in Hierarchical Planning?

What's the different to hierarchical planning?

- In general, we don't have a goal description.
- In general, we don't allow task insertion, i.e., rather than estimating which tasks to insert we need to estimate how to decompose.
- We always have a *task network* (or *partial plan*), instead of/in addition to the initial state.
- We have to deal with the partial order.
- We have to deal with abstract tasks (see above).
- Additional challenges for decomposition-based planning:
  - There is no current state, only the current partial plan.

What about Search in Hierarchical Planning?

What's the different to hierarchical planning?

- In general, we don't have a goal description.
- In general, we don't allow task insertion, i.e., rather than estimating which tasks to insert we need to estimate how to decompose.
- We always have a *task network* (or *partial plan*), instead of/in addition to the initial state.
- We have to deal with the partial order.
- We have to deal with abstract tasks (see above).
- Additional challenges for decomposition-based planning:
  - There is no current state, only the current partial plan.
  - Search nodes get bigger and bigger. Thus, paradoxically, the problem gets *harder* the closer we approach a solution.

Introduction

- Many heuristics base upon the so-called task decomposition graph (TDG).

Introduction

- Many heuristics base upon the so-called task decomposition graph (TDG).
- It is basically an explicit data structure showing how the (ground) methods interact.

Introduction

- Many heuristics base upon the so-called task decomposition graph (TDG).
- It is basically an explicit data structure showing how the (ground) methods interact.
- Recap: the *decomposition tree (DT)* is the representation of the decompositional structure of a single task network.

Introduction

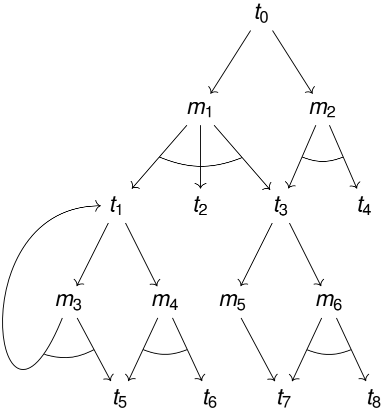
- Many heuristics base upon the so-called task decomposition graph (TDG).
- It is basically an explicit data structure showing how the (ground) methods interact.
- Recap: the *decomposition tree (DT)* is the representation of the decompositional structure of a single task network.
- In contrast, the decomposition graph represents the planning *domain* (cf. introductory chapter).

Introduction

- Many heuristics base upon the so-called task decomposition graph (TDG).
- It is basically an explicit data structure showing how the (ground) methods interact.
- Recap: the *decomposition tree (DT)* is the representation of the decompositional structure of a single task network.
- In contrast, the decomposition graph represents the planning *domain* (cf. introductory chapter).
- Due to possibly cyclic methods, DTs are in general no sub structures of TDGs.

Example

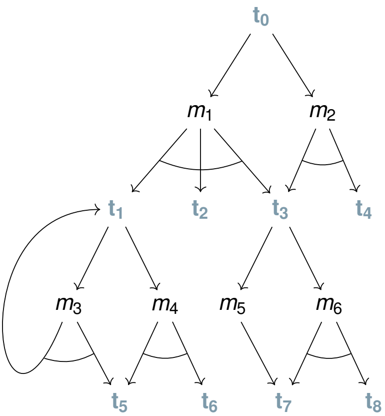
The Task Decomposition Graph (TDG) represents how tasks can be decomposed:



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

Example

The Task Decomposition Graph (TDG) represents how tasks can be decomposed:

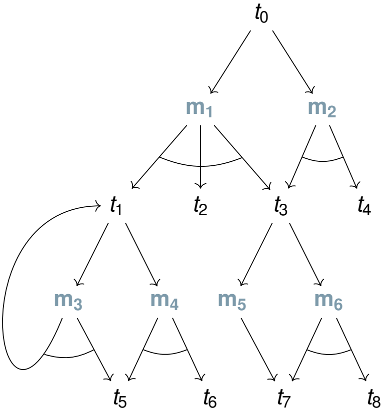


A TDG is a bipartite graph $\mathcal{G}$ $\langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

- N_T, the task nodes,

Example

The Task Decomposition Graph (TDG) represents how tasks can be decomposed:

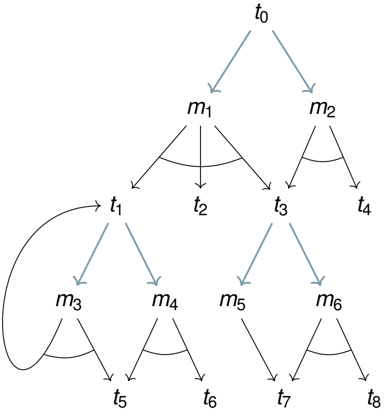


A TDG is a bipartite graph $\mathcal{G}$ $\langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

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

Example

The Task Decomposition Graph (TDG) represents how tasks can be decomposed:

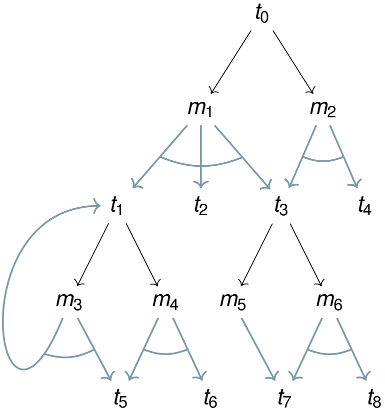


A TDG is a bipartite graph $\mathcal{G}$ $\langle N_T, N_M, E_{(T,M)}, E_{(M,T)} \rangle$ with

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

Example

The Task Decomposition Graph (TDG) represents how tasks can be decomposed:



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

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

Prerequisites of Formal Definition

- Although we primarily focus on propositional (or ground) (TI)HTN problems, we define the *ground* TDG based on a *lifted* model.

Prerequisites of Formal Definition

- Although we primarily focus on propositional (or ground) (TI)HTN problems, we define the *ground* TDG based on a *lifted* model.
- Here, the representation relies on a quantifier-free first-order predicate logic $\mathcal{L}$.

Prerequisites of Formal Definition

- Although we primarily focus on propositional (or ground) (TI)HTN problems, we define the *ground* TDG based on a *lifted* model.
- Here, the representation relies on a quantifier-free first-order predicate logic $\mathcal{L}$.
- Among others, decomposition methods are lifted: $(t(\bar{\tau}), tn_m, VC_m) \in M$ means:

Prerequisites of Formal Definition

- Although we primarily focus on propositional (or ground) (TI)HTN problems, we define the *ground* TDG based on a *lifted* model.
- Here, the representation relies on a quantifier-free first-order predicate logic $\mathcal{L}$.
- Among others, decomposition methods are lifted: $(t(\bar{\tau}), tn_m, VC_m) \in M$ means:
  - $t(\bar{\tau})$ is a task name followed by its parameter list (which are terms, i.e., variables or constants).

Prerequisites of Formal Definition

- Although we primarily focus on propositional (or ground) (TI)HTN problems, we define the *ground* TDG based on a *lifted* model.
- Here, the representation relies on a quantifier-free first-order predicate logic $\mathcal{L}$.
- Among others, decomposition methods are lifted: $(t(\bar{\tau}), tn_m, VC_m) \in M$ means:
  - $t(\bar{\tau})$ is a task name followed by its parameter list (which are terms, i.e., variables or constants).
  - tn_m is a task network of the form $(T, \prec, VC, \alpha)$, which now contains a set of variable constraints VC.

Prerequisites of Formal Definition

- Although we primarily focus on propositional (or ground) (TI)HTN problems, we define the *ground* TDG based on a *lifted* model.
- Here, the representation relies on a quantifier-free first-order predicate logic $\mathcal{L}$.
- Among others, decomposition methods are lifted: $(t(\bar{\tau}), tn_m, VC_m) \in M$ means:
  - $t(\bar{\tau})$ is a task name followed by its parameter list (which are terms, i.e., variables or constants).
  - tn_m is a task network of the form $(T, \prec, VC, \alpha)$, which now contains a set of variable constraints VC.
  - VC_m is also a set of variable constraints to relate the variables in $\bar{\tau}$ to the variables in tn_m.

Prerequisites of Formal Definition

- Although we primarily focus on propositional (or ground) (TI)HTN problems, we define the *ground* TDG based on a *lifted* model.
- Here, the representation relies on a quantifier-free first-order predicate logic $\mathcal{L}$.
- Among others, decomposition methods are lifted: $(t(\bar{\tau}), tn_m, VC_m) \in M$ means:
  - $t(\bar{\tau})$ is a task name followed by its parameter list (which are terms, i.e., variables or constants).
  - tn_m is a task network of the form $(T, \prec, VC, \alpha)$, which now contains a set of variable constraints VC.
  - VC_m is also a set of variable constraints to relate the variables in $\bar{\tau}$ to the variables in tn_m.
- $Ground_{VC}(tn)$ denotes the set of all possible groundings of tn by also taking into account the variable constraints VC.

Formal Definition

Let $\mathcal{P} = (\mathcal{D}, s_I, c_I)$ with $\mathcal{D} = (\mathcal{L}, P, \delta, C, M)$ be an HTN planning problem. The graph $\mathcal{G} = \langle V_T, V_M, E_{T \rightarrow M}, E_{M \rightarrow T} \rangle$ is called the TDG of $\mathcal{P}$ if it holds:

1. **base case** (task vertex for the given task)
   $c_I \in V_T$, the TDG's root.

Formal Definition

Let $\mathcal{P} = (\mathcal{D}, s_I, c_I)$ with $\mathcal{D} = (\mathcal{L}, P, \delta, C, M)$ be an HTN planning problem. The graph $\mathcal{G} = \langle V_T, V_M, E_{T \to M}, E_{M \to T} \rangle$ is called the TDG of $\mathcal{P}$ if it holds:

1. **base case** (task vertex for the given task)
   $c_I \in V_T$, the TDG's root.
2. **method vertices** (derived from task vertices)
   Let $v_t \in V_T$ with $v_t = t(\bar{c})$ and $(t(\bar{\tau}), tn_m, VC_m) \in M$.
   Then, for all $v_m \in Ground_{VC_m \cup \{\bar{\tau} = \bar{c}\}}(tn_m)$ holds:
   - $v_m \in V_M$
   - $(v_t, v_m) \in E_{T \to M}$.

Formal Definition

Let $\mathcal{P} = (\mathcal{D}, s_I, c_I)$ with $\mathcal{D} = (\mathcal{L}, P, \delta, C, M)$ be an HTN planning problem. The graph $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ is called the TDG of $\mathcal{P}$ if it holds:

1. **base case** (task vertex for the given task)
   $c_I \in V_T$, the TDG's root.
2. **method vertices** (derived from task vertices)
   Let $v_t \in V_T$ with $v_t = t(\bar{c})$ and $(t(\bar{\tau}), tn_m, VC_m) \in M$.
   Then, for all $v_m \in Ground_{VC_m \cup \{\bar{\tau}=\bar{c}\}}(tn_m)$ holds:
   • $v_m \in V_M$ • $(v_t, v_m) \in E_{T\to M}$.
3. **task vertices** (derived from method vertices)
   Let $v_m \in V_M$ with $v_m = (T, \prec, VC, \alpha)$. Then, for all task identifiers $t' \in T$ with $v_t = \alpha(t') = t(\bar{c})$, holds:
   • $v_t \in V_T$ • $(v_m, v_t) \in E_{M\to T}$.

Formal Definition

Let $\mathcal{P} = (\mathcal{D}, s_I, c_I)$ with $\mathcal{D} = (\mathcal{L}, P, \delta, C, M)$ be an HTN planning problem.
The graph $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ is called the TDG of $\mathcal{P}$ if it holds:

1. **base case** (task vertex for the given task)
   $c_I \in V_T$, the TDG's root.
2. **method vertices** (derived from task vertices)
   Let $v_t \in V_T$ with $v_t = t(\bar{c})$ and $(t(\bar{\tau}), tn_m, VC_m) \in M$.
   Then, for all $v_m \in Ground_{VC_m \cup \{\bar{\tau}=\bar{c}\}}(tn_m)$ holds:
   - $v_m \in V_M$ • $(v_t, v_m) \in E_{T\to M}$.
3. **task vertices** (derived from method vertices)
   Let $v_m \in V_M$ with $v_m = (T, \prec, VC, \alpha)$. Then, for all task identifiers $t' \in T$ with $v_t = \alpha(t') = t(\bar{c})$, holds:
   - $v_t \in V_T$ • $(v_m, v_t) \in E_{M\to T}$.
4. **tightness**
   $\mathcal{G}$ is minimal, such that 1. to 3. hold.

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required?

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)
  - What impact does it have on heuristic values?

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)
  - What impact does it have on heuristic values? We discuss this later.

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)
  - What impact does it have on heuristic values? We discuss this later.
- Even with a cyclic model, the TDG is finite. What's its size?

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)
  - What impact does it have on heuristic values? We discuss this later.
- Even with a cyclic model, the TDG is finite. What's its size?
  It's size is bounded by or identical to (depending on whether optimizations are incorporated) the number of ground methods.

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)
  - What impact does it have on heuristic values? We discuss this later.
- Even with a cyclic model, the TDG is finite. What's its size?
  It's size is bounded by or identical to (depending on whether optimizations are incorporated) the number of ground methods.
- What's its construction time?

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)
  - What impact does it have on heuristic values? We discuss this later.
- Even with a cyclic model, the TDG is finite. What's its size?
  It's size is bounded by or identical to (depending on whether optimizations are incorporated) the number of ground methods.
- What's its construction time? That depends on the input:

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)
  - What impact does it have on heuristic values? We discuss this later.
- Even with a cyclic model, the TDG is finite. What's its size? It's size is bounded by or identical to (depending on whether optimizations are incorporated) the number of ground methods.
- What's its construction time? That depends on the input:
  - Ground Model: Polynomial time.

Properties

- Every task occurs just once, even if present multiple times within the same method! Questions:
  - Is that required? No, we can easily extend the definition to incorporate duplicates. (But it becomes more complicated.)
  - What impact does it have on heuristic values? We discuss this later.
- Even with a cyclic model, the TDG is finite. What's its size?
  It's size is bounded by or identical to (depending on whether optimizations are incorporated) the number of ground methods.
- What's its construction time? That depends on the input:
  - Ground Model: Polynomial time.
  - Lifted Model: Exponential time (due to the grounding).

Further Notes

- Our formal definition relies on a planning *problem*.
  But we can also represent the entire *domain*:

Further Notes

- Our formal definition relies on a planning *problem*.
  But we can also represent the entire *domain*:
  - Simply start constructing the TDG with an arbitrary compound task until it is converged.

Further Notes

- Our formal definition relies on a planning *problem*.
  But we can also represent the entire *domain*:
  - Simply start constructing the TDG with an arbitrary compound task until it is converged.
  - If not every compound task (i.e., every grounding!) is in the TDG, add one such task and extend the TDG. Note that the old TDG(s) and the new one may be connected; but only in one direction.

Further Notes

- Our formal definition relies on a planning *problem*.
  But we can also represent the entire *domain*:
  - Simply start constructing the TDG with an arbitrary compound task until it is converged.
  - If not every compound task (i.e., every grounding!) is in the TDG, add one such task and extend the TDG. Note that the old TDG(s) and the new one may be connected; but only in one direction.
  - Repeat until every compound task (i.e., their groundings) is in the TDG.

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)

  → Top-down reachability analysis.

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)
  - → Top-down reachability analysis.
  - We can restrict the TDG's primitive tasks to those reachable from the initial (current?) state.

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)
  - → Top-down reachability analysis.
  - We can restrict the TDG's primitive tasks to those reachable from the initial (current?) state.
  - → Bottom-up reachability analysis.

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)

  → Top-down reachability analysis.
  - We can restrict the TDG's primitive tasks to those reachable from the initial (current?) state.

  → Bottom-up reachability analysis.
  - We can restrict to compound tasks with at least one method.

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)
  - → Top-down reachability analysis.
  - We can restrict the TDG's primitive tasks to those reachable from the initial (current?) state.
  - → Bottom-up reachability analysis.
  - We can restrict to compound tasks with at least one method.
  - We can restrict to compound tasks that allow a primitive decomposition.

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)
  - → Top-down reachability analysis.
  - We can restrict the TDG's primitive tasks to those reachable from the initial (current?) state.
  - → Bottom-up reachability analysis.
  - We can restrict to compound tasks with at least one method.
  - We can restrict to compound tasks that allow a primitive decomposition.
  - We can restrict to task networks without "eliminated elements".

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)

  → Top-down reachability analysis.
  - We can restrict the TDG's primitive tasks to those reachable from the initial (current?) state.

  → Bottom-up reachability analysis.
  - We can restrict to compound tasks with at least one method.
  - We can restrict to compound tasks that allow a primitive decomposition.
  - We can restrict to task networks without "eliminated elements".

→ More details on next slide.

Optimizations

- If a problem instance is given, we can perform a reachability analysis and thereby reducing its size:
  - We can restrict the TDG to those tasks reachable from the initial task (network). (This follows already from the definition.)
  - → Top-down reachability analysis.
  - We can restrict the TDG's primitive tasks to those reachable from the initial (current?) state.
  - → Bottom-up reachability analysis.
  - We can restrict to compound tasks with at least one method.
  - We can restrict to compound tasks that allow a primitive decomposition.
  - We can restrict to task networks without "eliminated elements".

→ More details on next slide.

Note: *Technically*, any modification to the TDG will violate its definition. We still refer to the resulting structures as TDGs, though.

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

- Mark all primitive tasks as reachable.

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

- Mark all primitive tasks as reachable.
- Iterate over all task networks in the TDG in which all tasks are marked as reachable (base case: primitive task networks). Mark their parent compound task as reachable.

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

- Mark all primitive tasks as reachable.
- Iterate over all task networks in the TDG in which all tasks are marked as reachable (base case: primitive task networks). Mark their parent compound task as reachable.
- Continue until no more tasks can be marked as reachable

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

- Mark all primitive tasks as reachable.
- Iterate over all task networks in the TDG in which all tasks are marked as reachable (base case: primitive task networks). Mark their parent compound task as reachable.
- Continue until no more tasks can be marked as reachable

Step 4: Restrict TDG:

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

- Mark all primitive tasks as reachable.
- Iterate over all task networks in the TDG in which all tasks are marked as reachable (base case: primitive task networks). Mark their parent compound task as reachable.
- Continue until no more tasks can be marked as reachable

Step 4: Restrict TDG:

- Remove all task networks with an unreachable compound task.

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

- Mark all primitive tasks as reachable.
- Iterate over all task networks in the TDG in which all tasks are marked as reachable (base case: primitive task networks). Mark their parent compound task as reachable.
- Continue until no more tasks can be marked as reachable

Step 4: Restrict TDG:

- Remove all task networks with an unreachable compound task.
- Remove all compound tasks without decomposition method.

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

- Mark all primitive tasks as reachable.
- Iterate over all task networks in the TDG in which all tasks are marked as reachable (base case: primitive task networks). Mark their parent compound task as reachable.
- Continue until no more tasks can be marked as reachable

Step 4: Restrict TDG:

- Remove all task networks with an unreachable compound task.
- Remove all compound tasks without decomposition method.
- Repeat until nothing can be deleted.

Restricting the TDG

Step 1: Construct PG to find reachable ground primitive tasks.

Step 2: Construct TDG top-down (ignoring task networks with unreachable primitive tasks) until converged.

Step 3: Bottom-Up reachability to eliminate tasks that do not admit a primitive decomposition:

- Mark all primitive tasks as reachable.
- Iterate over all task networks in the TDG in which all tasks are marked as reachable (base case: primitive task networks). Mark their parent compound task as reachable.
- Continue until no more tasks can be marked as reachable

Step 4: Restrict TDG:

- Remove all task networks with an unreachable compound task.
- Remove all compound tasks without decomposition method.
- Repeat until nothing can be deleted.

Step 5: Since the set of reachable primitive tasks may have changed, we can repeat all previous steps (possibly multiple times).
This step does usually not pay off empirically.

Restricting the TDG – Further Optimization

- Obviously, the order of steps 1 and 2 are somehow interchangeable:

Restricting the TDG – Further Optimization

- Obviously, the order of steps 1 and 2 are somehow interchangeable:
  - Should we first restrict to top-down-reachability thereby reducing the PG construction process?

Restricting the TDG – Further Optimization

- Obviously, the order of steps 1 and 2 are somehow interchangeable:
  - Should we first restrict to top-down-reachability thereby reducing the PG construction process?
  - Or should we first do a top-down reachability thereby reducing the TDG construction process?

Restricting the TDG – Further Optimization

- Obviously, the order of steps 1 and 2 are somehow interchangeable:
  - Should we first restrict to top-down-reachability thereby reducing the PG construction process?
  - Or should we first do a top-down reachability thereby reducing the TDG construction process?
- We deem the given order more useful, *but* we first perform a step 0:

Restricting the TDG – Further Optimization

- Obviously, the order of steps 1 and 2 are somehow interchangeable:
  - Should we first restrict to top-down-reachability thereby reducing the PG construction process?
  - Or should we first do a top-down reachability thereby reducing the TDG construction process?
- We deem the given order more useful, *but* we first perform a step 0:
  - We first perform a *parameter-relaxed* top-down analysis.

Restricting the TDG – Further Optimization

- Obviously, the order of steps 1 and 2 are somehow interchangeable:
  - Should we first restrict to top-down-reachability thereby reducing the PG construction process?
  - Or should we first do a top-down reachability thereby reducing the TDG construction process?
- We deem the given order more useful, *but* we first perform a step 0:
  - We first perform a *parameter-relaxed* top-down analysis.
  - We build a task parameter- and predicate parameter-free TDG.

Restricting the TDG – Further Optimization

- Obviously, the order of steps 1 and 2 are somehow interchangeable:
  - Should we first restrict to top-down-reachability thereby reducing the PG construction process?
  - Or should we first do a top-down reachability thereby reducing the TDG construction process?
- We deem the given order more useful, *but* we first perform a step 0:
  - We first perform a *parameter-relaxed* top-down analysis.
  - We build a task parameter- and predicate parameter-free TDG.
  - This TDG can be built very efficiently.

Restricting the TDG – Further Optimization

- Obviously, the order of steps 1 and 2 are somehow interchangeable:
  - Should we first restrict to top-down-reachability thereby reducing the PG construction process?
  - Or should we first do a top-down reachability thereby reducing the TDG construction process?
- We deem the given order more useful, *but* we first perform a step 0:
  - We first perform a *parameter-relaxed* top-down analysis.
  - We build a task parameter- and predicate parameter-free TDG.
  - This TDG can be built very efficiently.
  - This overestimates the number of reachable tasks, but already rules out some unreachable actions for the PG construction.

Non-Hierarchical Landmarks

- The concept of *landmarks* originates from *classical* planning, but it was transferred to hierarchical planning later on.

Non-Hierarchical Landmarks

- The concept of *landmarks* originates from *classical* planning, but it was transferred to hierarchical planning later on.
- A classical *fact landmark* is a state variable that has to be true at some point in every solution. Generalization: Conjunctions, disjunctions, or arbitrary formulae of state variables.

Non-Hierarchical Landmarks

- The concept of *landmarks* originates from *classical* planning, but it was transferred to hierarchical planning later on.
- A classical *fact landmark* is a state variable that has to be true at some point in every solution. Generalization: Conjunctions, disjunctions, or arbitrary formulae of state variables.
- What are *trivial* fact landmarks?

.

Non-Hierarchical Landmarks

- The concept of *landmarks* originates from *classical* planning, but it was transferred to hierarchical planning later on.
- A classical *fact landmark* is a state variable that has to be true at some point in every solution. Generalization: Conjunctions, disjunctions, or arbitrary formulae of state variables.
- What are *trivial* fact landmarks? The initial state and goal description.

Non-Hierarchical Landmarks

- The concept of *landmarks* originates from *classical* planning, but it was transferred to hierarchical planning later on.
- A classical *fact landmark* is a state variable that has to be true at some point in every solution. Generalization: Conjunctions, disjunctions, or arbitrary formulae of state variables.
- What are *trivial* fact landmarks? The initial state and goal description.
- A classical *action landmark* is an action that is part of every solution.

Hierarchical Landmarks

- A hierarchical landmark is a task (primitive or compound) that occurs on any sequence of decompositions from the initial task (network) to any solution.
- A formal definition will be provided or has to be found in the exercises.

Exploitation of Landmarks

- Why are we interested in landmarks?

Exploitation of Landmarks

- Why are we interested in landmarks?
- They are measurements of how “important” state variables and/or actions/tasks are.

Exploitation of Landmarks

- Why are we interested in landmarks?
- They are measurements of how “important” state variables and/or actions/tasks are.
- They can be exploited, probably among others, for explanations and heuristics.

Exploitation of Landmarks

- Why are we interested in landmarks?
- They are measurements of how "important" state variables and/or actions/tasks are.
- They can be exploited, probably among others, for explanations and heuristics.
  - Explanations: Explanations basing on landmarks might be more convincing.

Exploitation of Landmarks

- Why are we interested in landmarks?
- They are measurements of how “important” state variables and/or actions/tasks are.
- They can be exploited, probably among others, for explanations and heuristics.
  - Explanations: Explanations basing on landmarks might be more convincing.
  - Heuristics: Later in this section!

Classical Landmarks (Hardness)

- Determining whether a fact is a *classical landmark* is as hard as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?

Classical Landmarks (Hardness)

- Determining whether a fact is a *classical landmark* is as hard as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:

Classical Landmarks (Hardness)

- Determining whether a fact is a *classical landmark* is as hard as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial state s_I' and three new actions.

Classical Landmarks (Hardness)

- Determining whether a fact is a *classical landmark* is as hard as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial state s_I' and three new actions.
  - One action generates the original s_I.

Classical Landmarks (Hardness)

- Determining whether a fact is a *classical landmark* is as hard as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial state s_I' and three new actions.
  - One action generates the original s_I.
  - One action is a shortcut to the goal: It generates some $v^* \notin V$.

Classical Landmarks (Hardness)

- Determining whether a fact is a *classical landmark* is as hard as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial state s'_I and three new actions.
  - One action generates the original s_I.
  - One action is a shortcut to the goal: It generates some $v^* \notin V$.
  - One action exploits that shortcut: It uses v^* as precondition and generates the goal.

Classical Landmarks (Hardness)

- Determining whether a fact is a *classical landmark* is as hard as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial state s'_I and three new actions.
  - One action generates the original s_I.
  - One action is a shortcut to the goal: It generates some $v^* \notin V$.
  - One action exploits that shortcut: It uses v^* as precondition and generates the goal.
- If v^* is a landmark, then the original task was unsolvable.

Classical Landmarks (Hardness)

- Determining whether a fact is a *classical landmark* is as hard as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial state s_I' and three new actions.
  - One action generates the original s_I.
  - One action is a shortcut to the goal: It generates some $v^* \notin V$.
  - One action exploits that shortcut: It uses v^* as precondition and generates the goal.
- If v^* is a landmark, then the original task was unsolvable.

→ All in all: Reduction from the *Plan-Nonexistence decision problem* to the *Landmark decision problem*.

Classical Landmarks (Membership)

- Determining whether a fact is a *classical landmark* is as "easy" as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?

Classical Landmarks (Membership)

- Determining whether a fact is a *classical landmark* is as “easy” as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We again exploit that deciding the unsolvability is as hard as deciding the solvability:

Classical Landmarks (Membership)

- Determining whether a fact is a *classical landmark* is as “easy” as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We again exploit that deciding the unsolvability is as hard as deciding the solvability:
  - Given a fact $v \in V$, remove all actions that generate v and check whether the problem is still solvable.

Classical Landmarks (Membership)

- Determining whether a fact is a *classical landmark* is as “easy” as planning! (As we will see later: $\mathbb{PSPACE}$-complete.) Why?
- We again exploit that deciding the unsolvability is as hard as deciding the solvability:
  - Given a fact $v \in V$, remove all actions that generate v and check whether the problem is still solvable.
  - v is a landmark if and only if it has become unsolvable.

Hierarchical Landmarks (Hardness)

- Determining whether a task is a *hierarchical landmark* is as hard as hierarchical planning! (As we will see later: undecidable.) Why?

Hierarchical Landmarks (Hardness)

- Determining whether a task is a *hierarchical landmark* is as hard as hierarchical planning! (As we will see later: undecidable.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:

Hierarchical Landmarks (Hardness)

- Determining whether a task is a *hierarchical landmark* is as hard as hierarchical planning! (As we will see later: undecidable.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial task c_I' with two methods and another compound task with a new method.

Hierarchical Landmarks (Hardness)

- Determining whether a task is a *hierarchical landmark* is as hard as hierarchical planning! (As we will see later: undecidable.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial task c_I' with two methods and another compound task with a new method.
  - One method maps to the original c_I, the other to a network containing the other new compound task c^*.

Hierarchical Landmarks (Hardness)

- Determining whether a task is a *hierarchical landmark* is as hard as hierarchical planning! (As we will see later: undecidable.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial task c_I' with two methods and another compound task with a new method.
  - One method maps to the original c_I, the other to a network containing the other new compound task c^*.
  - The third method maps c^* to the empty task network thereby allowing a trivial solution.

Hierarchical Landmarks (Hardness)

- Determining whether a task is a *hierarchical landmark* is as hard as hierarchical planning! (As we will see later: undecidable.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial task c_I' with two methods and another compound task with a new method.
  - One method maps to the original c_I, the other to a network containing the other new compound task c^*.
  - The third method maps c^* to the empty task network thereby allowing a trivial solution.
- If $c^* \notin C$ is a landmark, then the original task was unsolvable.

Hierarchical Landmarks (Hardness)

- Determining whether a task is a *hierarchical landmark* is as hard as hierarchical planning! (As we will see later: undecidable.) Why?
- We exploit that deciding the unsolvability (Plan-Nonexistence decision problem) is as hard as deciding the solvability:
  - Construct an easier planning problem with a new initial task c_I' with two methods and another compound task with a new method.
  - One method maps to the original c_I, the other to a network containing the other new compound task c^*.
  - The third method maps c^* to the empty task network thereby allowing a trivial solution.
- If $c^* \notin C$ is a landmark, then the original task was unsolvable.

→ All in all: Reduction from the *Plan-Nonexistence decision problem* to the *Landmark decision problem*.

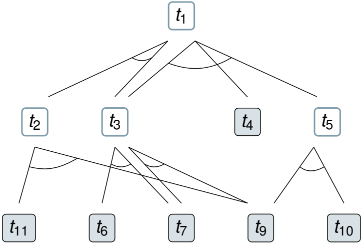
Hierarchical Landmarks (Membership)

- Is the problem semi-decidable?

$\rightarrow$ Exercise!

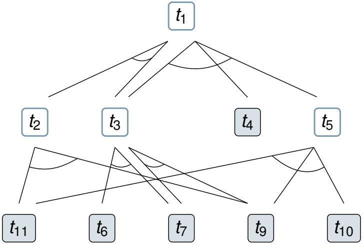
Landmark Computation

General idea: Compute the intersection of all partial plans that belong to the same compound task.



Landmark Computation, Improved

We can still do better than that, though...



Exploitation of Landmarks

- Landmarks were developed for *flaw selectors*: Choose to decompose a compound task with fewer landmarks (as a measure of its hardness).

Exploitation of Landmarks

- Landmarks were developed for *flaw selectors*: Choose to decompose a compound task with fewer landmarks (as a measure of its hardness).
- We can also choose a *task network* (from the search fringe, i.e., the search strategy) based on the number of its landmarks.

Exploitation of Landmarks

- Landmarks were developed for *flaw selectors*: Choose to decompose a compound task with fewer landmarks (as a measure of its hardness).
- We can also choose a *task network* (from the search fringe, i.e., the search strategy) based on the number of its landmarks.

→ Both approaches lack from the problem that the landmarks as computed are those that will be used no matter what, i.e., one *cannot prevent* having them in a sequence of decompositions – limiting their usefulness.

Exploitation of Landmarks

- Landmarks were developed for *flaw selectors*: Choose to decompose a compound task with fewer landmarks (as a measure of its hardness).
- We can also choose a *task network* (from the search fringe, i.e., the search strategy) based on the number of its landmarks.

→ Both approaches lack from the problem that the landmarks as computed are those that will be used no matter what, i.e., one *cannot prevent* having them in a sequence of decompositions – limiting their usefulness.

- We can use all primitive landmarks as the basis for state-based heuristics!

Exploitation of Landmarks

- Landmarks were developed for *flaw selectors*: Choose to decompose a compound task with fewer landmarks (as a measure of its hardness).
- We can also choose a *task network* (from the search fringe, i.e., the search strategy) based on the number of its landmarks.

→ Both approaches lack from the problem that the landmarks as computed are those that will be used no matter what, i.e., one *cannot prevent* having them in a sequence of decompositions – limiting their usefulness.

- We can use all primitive landmarks as the basis for state-based heuristics!

→ This allows to use *any* classical heuristic (or classical landmark technique!).

Introduction

- The landmark heuristics take only simple landmarks into account.

Introduction

- The landmark heuristics take only simple landmarks into account.
- More precisely: one can easily rewrite a model, such that *none* of its landmarks gets discovered.

Introduction

- The landmark heuristics take only simple landmarks into account.
- More precisely: one can easily rewrite a model, such that *none* of its landmarks gets discovered.
- Even in these cases, the TDG holds valuable information that can be exploited to estimate goal distances...

Introduction

- The landmark heuristics take only simple landmarks into account.
- More precisely: one can easily rewrite a model, such that *none* of its landmarks gets discovered.
- Even in these cases, the TDG holds valuable information that can be exploited to estimate goal distances...
- But how..? We know that:

Introduction

- The landmark heuristics take only simple landmarks into account.
- More precisely: one can easily rewrite a model, such that *none* of its landmarks gets discovered.
- Even in these cases, the TDG holds valuable information that can be exploited to estimate goal distances...
- But how..? We know that:
  - *all tasks within a method* need to be “accomplished” (applied or decomposed).

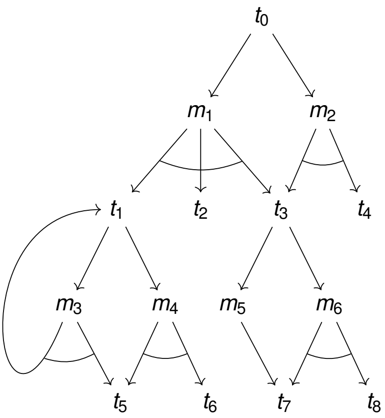
Introduction

- The landmark heuristics take only simple landmarks into account.
- More precisely: one can easily rewrite a model, such that *none* of its landmarks gets discovered.
- Even in these cases, the TDG holds valuable information that can be exploited to estimate goal distances...
- But how..? We know that:
  - *all tasks within a method* need to be “accomplished” (applied or decomposed).
  - For each compound task, only one of its methods needs to be applied.

Introduction

- The landmark heuristics take only simple landmarks into account.
- More precisely: one can easily rewrite a model, such that *none* of its landmarks gets discovered.
- Even in these cases, the TDG holds valuable information that can be exploited to estimate goal distances...
- But how..? We know that:
  - *all tasks within a method* need to be "accomplished" (applied or decomposed).
  - For each compound task, only one of its methods needs to be applied.
- Note: For convenience, we later write $t(\bar{\tau}) \in T$ as shorthand for $t(\bar{\tau}) = \alpha(t'), t' \in T$.

Overview



Exploit TDG for effort estimation.

Step 1:
Compute the TDG.

Step 2:
Compute TDG-based estimates $h_T(t)/h_M(t)$ for each task/method node in the TDG (*once* via preprocessing).

Step 3:
For search node (task network or partial plan) *tn* and its tasks T, compute $h(tn)$ based on the estimates for the $t \in T$.

- Via estimating the costs of missing actions → TDG-c.
- Via estimating the still required modifications → TDG-m.

Motivation

- To obtain *good* (or cheap) solutions heuristically, we need to estimate the costs of a plan that can be developed from a current task network.

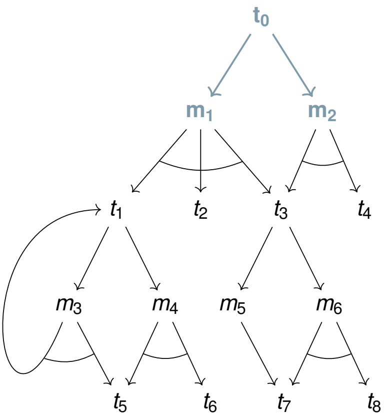
Motivation

- To obtain *good* (or cheap) solutions heuristically, we need to estimate the costs of a plan that can be developed from a current task network.
- We thus exploit the TDG and use its *action costs* as basis for estimates.

Motivation

- To obtain *good* (or cheap) solutions heuristically, we need to estimate the costs of a plan that can be developed from a current task network.
- We thus exploit the TDG and use its *action costs* as basis for estimates.
- The resulting heuristic will be admissible (trivial).

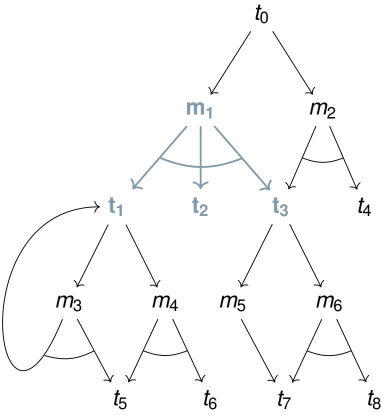
Illustration of TDG-c Computation



Example:

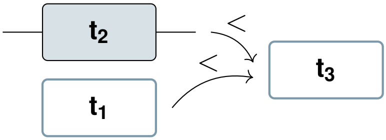
$$h_T(t_0) = min\{h_M(m_1), h_M(m_2)\}$$

Illustration of TDG-c Computation



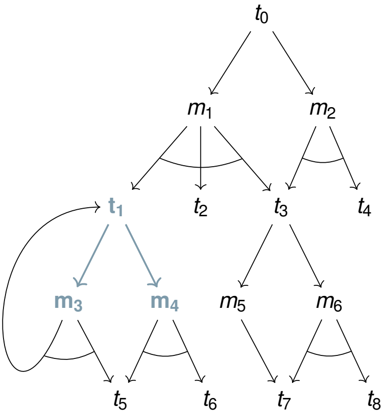
Example:

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



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

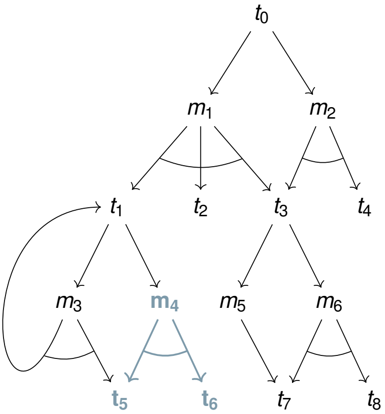
Illustration of TDG-c Computation



Example:

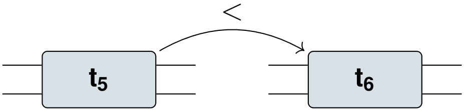
$$h_T(t_1) = min\{h_M(m_3), h_M(m_4)\}$$

Illustration of TDG-c Computation



Example:

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



$$h(m_4) = c(t_5) + c(t_6)$$

TDG-c Computation

Let $\mathcal{G} = \langle V_T, V_M, E_{T \to M}, E_{M \to T} \rangle$ be a TDG.
The TDG-c estimates of $\mathcal{G}$'s task nodes are given by:

$$h_T(v_t) := \begin{cases} cost(v_t) & \text{if } v_t \text{ primitive} \\ \min_{(v_t, v_m) \in E_{T \to M}} h_M(v_m) & \text{else} \end{cases}$$

TDG-c Computation

Let $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.
The TDG-c estimates of $\mathcal{G}$'s task nodes are given by:

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

For methods nodes $v_m = (T, \prec, VC, \alpha)$:

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

TDG-c Computation

Let $\mathcal{G} = \langle V_T, V_M, E_{T \to M}, E_{M \to T} \rangle$ be a TDG.
The TDG-c estimates of $\mathcal{G}$'s task nodes are given by:

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

For methods nodes $v_m = (T, \prec, VC, \alpha)$:

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

Heuristic value for $tn = (T, \prec, VC, \alpha)$:

decomposition-based: $h_{TDG-c}(tn) := \sum_{\substack{t(\bar{\tau}) \in T \\ t(\bar{\tau}) \text{ abstract}}} \left(\min\limits_{v_t \in Ground_{VC}(t(\bar{\tau}))} h_T(v_t) \right)$

TDG-c Computation

Let $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.
The TDG-c estimates of $\mathcal{G}$'s task nodes are given by:

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

For methods nodes $v_m = (T, \prec, VC, \alpha)$:

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

Heuristic value for $tn = (T, \prec, VC, \alpha)$:

decomposition-based: $$h_{TDG-c}(tn) := \sum_{\substack{t(\bar\tau)\in T \\ t(\bar\tau) \text{ abstract}}} \left(\min_{v_t\in Ground_{VC}(t(\bar\tau))} h_T(v_t) \right)$$

progression-based: $$h_{TDG-c}(tn) := \sum_{t(\bar\tau)\in T} \left(\min_{v_t\in Ground_{VC}(t(\bar\tau))} h_T(v_t) \right)$$

Notes

- The heuristic formulae were given for *lifted* planning and can be simplified for ground planning:

$$\sum_{\substack{t(\bar{\tau}) \in T \\ t(\bar{\tau}) \text{ abstract}}} \left(\min_{v_t \in Ground_{VC}(t(\bar{\tau}))} h_T(v_t) \right) \text{ becomes } \sum_{\substack{t(\bar{c}) \in T \\ t(\bar{c}) \text{ abstract}}} h_T(t(\bar{c})).$$

Notes

- The heuristic formulae were given for *lifted* planning and can be simplified for ground planning:

$$\sum_{\substack{t(\bar{\tau}) \in T \\ t(\bar{\tau}) \text{ abstract}}} \left(\min_{v_t \in Ground_{VC}(t(\bar{\tau}))} h_T(v_t) \right) \text{ becomes } \sum_{\substack{t(\bar{c}) \in T \\ t(\bar{c}) \text{ abstract}}} h_T(t(\bar{c})).$$

- Why do we have two different formulae for progression- and decomposition-based search?

Notes

- The heuristic formulae were given for *lifted* planning and can be simplified for ground planning:

$$\sum_{\substack{t(\bar{\tau}) \in T \\ t(\bar{\tau}) \text{ abstract}}} \left(\min_{v_t \in Ground_{VC}(t(\bar{\tau}))} h_T(v_t) \right) \text{ becomes } \sum_{\substack{t(\bar{c}) \in T \\ t(\bar{c}) \text{ abstract}}} h_T(t(\bar{c})).$$

- Why do we have two different formulae for progression- and decomposition-based search?
  - Because the costs of search nodes are usually computed differently.

Notes

- The heuristic formulae were given for *lifted* planning and can be simplified for ground planning:

$$\sum_{\substack{t(\overline{\tau})\in T \\ t(\overline{\tau}) \text{ abstract}}} \left(\min_{v_t \in Ground_{VC}(t(\overline{\tau}))} h_T(v_t) \right) \text{ becomes } \sum_{\substack{t(\overline{c})\in T \\ t(\overline{c}) \text{ abstract}}} h_T(t(\overline{c})).$$

- Why do we have two different formulae for progression- and decomposition-based search?
  - Because the costs of search nodes are usually computed differently.
  - In decomposition-based search, a search node's costs is given by its primitive tasks.

Notes

- The heuristic formulae were given for *lifted* planning and can be simplified for ground planning:

$$\sum_{\substack{t(\overline{\tau}) \in T \\ t(\overline{\tau}) \text{ abstract}}} \left(\min_{v_t \in Ground_{VC}(t(\overline{\tau}))} h_T(v_t) \right) \text{ becomes } \sum_{\substack{t(\overline{c}) \in T \\ t(\overline{c}) \text{ abstract}}} h_T(t(\overline{c})).$$

- Why do we have two different formulae for progression- and decomposition-based search?
  - Because the costs of search nodes are usually computed differently.
  - In decomposition-based search, a search node's costs is given by its primitive tasks.
  - In progression-based search, those costs are usually incorporated after an action has been progressed away.

Notes

- The heuristic formulae were given for *lifted* planning and can be simplified for ground planning:

$$\sum_{\substack{t(\bar{\tau}) \in T \\ t(\bar{\tau}) \text{ abstract}}} \left(\min_{v_t \in Ground_{VC}(t(\bar{\tau}))} h_T(v_t) \right) \text{ becomes } \sum_{\substack{t(\bar{c}) \in T \\ t(\bar{c}) \text{ abstract}}} h_T(t(\bar{c})).$$

- Why do we have two different formulae for progression- and decomposition-based search?
  - Because the costs of search nodes are usually computed differently.
  - In decomposition-based search, a search node's costs is given by its primitive tasks.
  - In progression-based search, those costs are usually incorporated after an action has been progressed away.
  - $\rightarrow$ Using only the abstract tasks for the heuristic in decomposition-based planning thus prevents taking those primitive costs into account twice.

Motivation

- Just estimating the final solution costs says little about the effort finding it. One can easily construct examples, where expensive solutions can be found easily (with only few decompositions), whereas cheap solutions need more search effort.

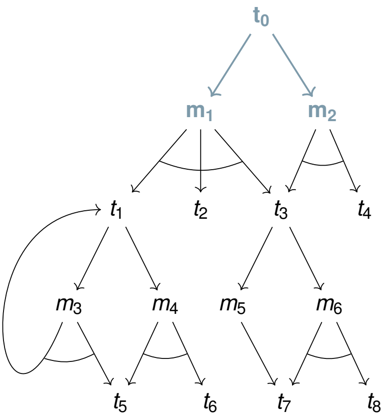
Motivation

- Just estimating the final solution costs says little about the effort finding it. One can easily construct examples, where expensive solutions can be found easily (with only few decompositions), whereas cheap solutions need more search effort.
- We thus exploit the TDG to estimate how many *modifications* we require for certain tasks.

Motivation

- Just estimating the final solution costs says little about the effort finding it. One can easily construct examples, where expensive solutions can be found easily (with only few decompositions), whereas cheap solutions need more search effort.
- We thus exploit the TDG to estimate how many *modifications* we require for certain tasks.
- The resulting heuristic will be *not be* admissible, but admissible in the number of required modifications (trivial). This means that any solution returned by A^* will have the property that no other solution can be created with fewer modifications. (This is not something we aim for, it's just a property we get.)

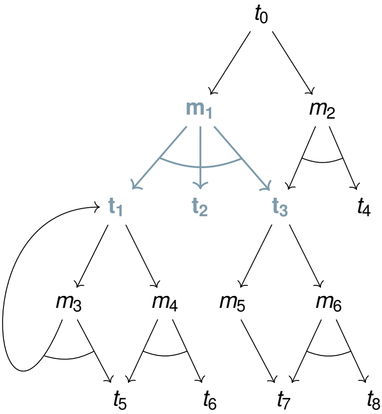
Illustration of TDG-m Computation – For Decomposition-based Search



Example:

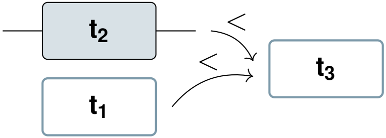
$$h_T(t_0) = 1 + min\{h_M(m_1), h_M(m_2)\}$$

Illustration of TDG-m Computation – For Decomposition-based Search



Example:

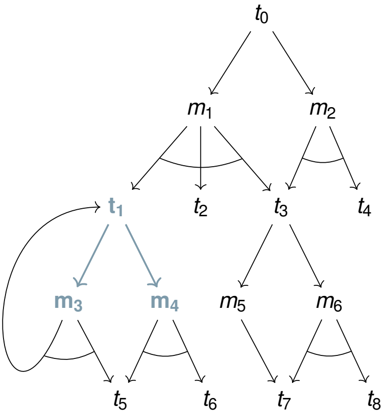
Method $m_1 = (t_0, tn)$ with task network tn:



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

(Also subtract $|CL|$ in case we have a partial plan containing causal links.)

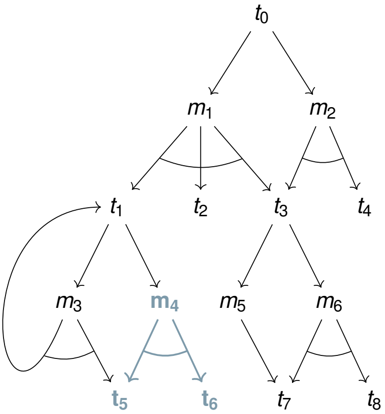
Illustration of TDG-m Computation – For Decomposition-based Search



Example:

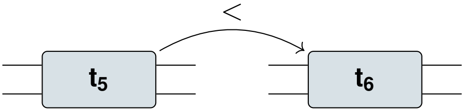
$$h_T(t_1) = 1 + min\{h_M(m_3), h_M(m_4)\}$$

Illustration of TDG-m Computation – For Decomposition-based Search



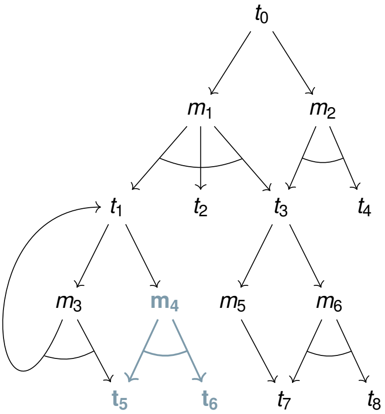
Example:

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



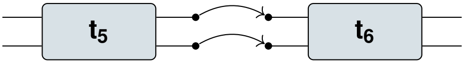
$$\begin{aligned} h_M(m_4) &= h_T(t_5) + h_T(t_6) \\ &= |pre(t_5)| + |pre(t_6)| \\ &= 2 + 2 = 4 \end{aligned}$$

Illustration of TDG-m Computation – For Decomposition-based Search



Example:

Method $m_4 = (t_1, P)$ with partial plan P:



$$\begin{aligned} h_M(m_4) &= h_T(t_5) + h_T(t_6) - |CL| \\ &= |pre(t_5)| + |pre(t_6)| - 2 \\ &= 2 + 2 - 2 = 2 \end{aligned}$$

TDG-m Computation

Let $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.
The TDG-c estimates of $\mathcal{G}$'s task nodes are given by:

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

TDG-m Computation

Let $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.
The TDG-c estimates of $\mathcal{G}$'s task nodes are given by:

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

For methods nodes $v_m = (T, \prec, VC, CL, \alpha)$:

$$h_M(v_m) := \sum_{(v_m,v_t)\in E_{M\to T}} h_T(v_t) - |CL|$$

TDG-m Computation

Let $\mathcal{G} = \langle V_T, V_M, E_{T\to M}, E_{M\to T}\rangle$ be a TDG.
The TDG-c estimates of $\mathcal{G}$'s task nodes are given by:

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

For methods nodes $v_m = (T, \prec, VC, CL, \alpha)$:

$$h_M(v_m) := \sum_{(v_m,v_t)\in E_{M\to T}} h_T(v_t) - |CL|$$

Heuristic value for partial plan $P = (T, \prec, VC, CL, \alpha)$:

decomposition-based: $h_{TDG-m}(P) := \sum_{t(\bar{\tau})\in T} \left(\min_{v_t\in Ground_{VC}(t(\bar{\tau}))} h_T(v_t) \right) - |CL|$

TDG-m Computation

Let $\mathcal{G} = \langle V_T, V_M, E_{T\rightarrow M}, E_{M\rightarrow T}\rangle$ be a TDG.
The TDG-c estimates of $\mathcal{G}$'s task nodes are given by:

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

For methods nodes $v_m = (T, \prec, VC, CL, \alpha)$:

$$h_M(v_m) := \sum_{(v_m,v_t)\in E_{M\rightarrow T}} h_T(v_t) - |CL|$$

Heuristic value for partial plan $P = (T, \prec, VC, CL, \alpha)$:

decomposition-based: $h_{TDG-m}(P) := \sum\limits_{t(\bar{\tau})\in T} \left(\min\limits_{v_t \in Ground_{VC}(t(\bar{\tau}))} h_T(v_t) \right) - |CL|$

progression-based: Ignore links and use 1 instead of $|pre(v_t)|$.

Properties

- Note that this heuristic does not compute executable plans. So what *does* it compute?

Properties

- Note that this heuristic does not compute executable plans. So what *does* it compute?
- The cheapest set of primitive tasks that can be made true by *any* primitive tasks.

Properties

- Note that this heuristic does not compute executable plans. So what *does* it compute?
- The cheapest set of primitive tasks that can be made true by *any* primitive tasks.
- However, the costs (or the effort) of these tasks is *not* reflected in the heuristic value!

Properties

- Note that this heuristic does not compute executable plans. So what *does* it compute?
- The cheapest set of primitive tasks that can be made true by *any* primitive tasks.
- However, the costs (or the effort) of these tasks is *not* reflected in the heuristic value!
- To illustrate what this means: What heuristic do we get if the only abstract task can be decomposed into an empty task network (which will not work as solution due to a goal description or subsequent primitive tasks).

Properties

- Note that this heuristic does not compute executable plans. So what *does* it compute?
- The cheapest set of primitive tasks that can be made true by *any* primitive tasks.
- However, the costs (or the effort) of these tasks is *not* reflected in the heuristic value!
- To illustrate what this means: What heuristic do we get if the only abstract task can be decomposed into an empty task network (which will not work as solution due to a goal description or subsequent primitive tasks).
- However, the heuristic can still come up with even exponentially large heuristic values. (This is true although every task occurs just once in the TDG. Why?)

Motivation

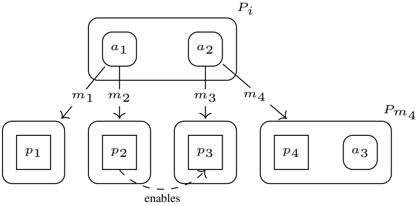
- Recall from the beginning (construction of the TDG) that the TDG can be recomputed as soon as any of its tasks is identified as unreachable.

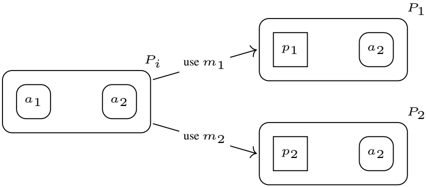
Motivation

- Recall from the beginning (construction of the TDG) that the TDG can be recomputed as soon as any of its tasks is identified as unreachable.
- So far, the TDG is only computed *once*. However, different search nodes might have different reachable action sets!

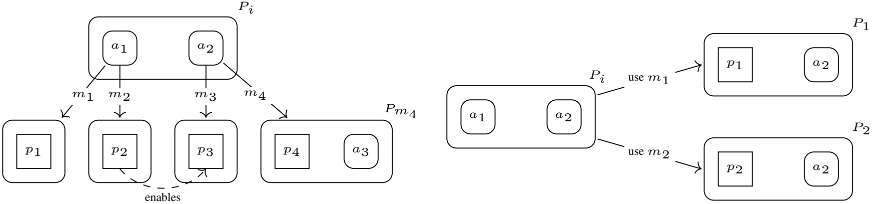
Motivation

- Recall from the beginning (construction of the TDG) that the TDG can be recomputed as soon as any of its tasks is identified as unreachable.
- So far, the TDG is only computed *once*. However, different search nodes might have different reachable action sets!



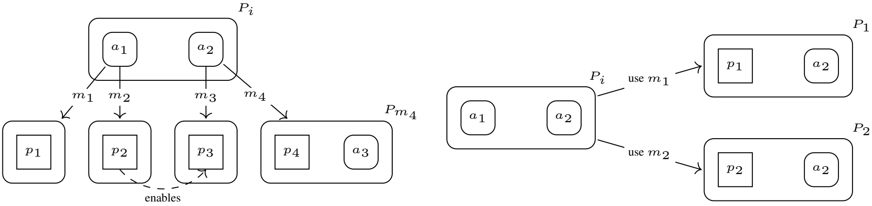


Example



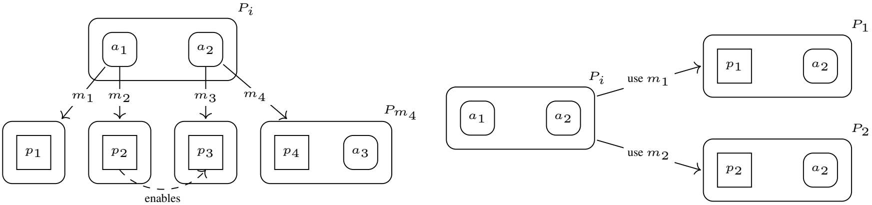
- Let $c(p_3) = i$ and $h_M(P_{m_4}) = h_T(p_4) + h_T(a_3) = j > i$.

Example



- Let $c(p_3) = i$ and $h_M(P_{m_4}) = h_T(p_4) + h_T(a_3) = j > i$.
- Then, we get $h_T(a_2) = i$. Let us now consider the heuristic values for P_1 and P_2 resulting from decomposing a_1 using m_1 or m_2, respectively.

Example



- Let $c(p_3) = i$ and $h_M(P_{m_4}) = h_T(p_4) + h_T(a_3) = j > i$.
- Then, we get $h_T(a_2) = i$. Let us now consider the heuristic values for P_1 and P_2 resulting from decomposing a_1 using m_1 or m_2, respectively.
- Without recomputation, we get $h(P_1) = h(P_2) = i$. With recomputation, we get $h(P_1) = j$ and $h(P_2) = i$, so we get improved heuristic accuracy due to updated reachability information in the TDG.

When to Recompute? – In Decomposition-based Planning

- Let P be a partial plan, *mod* a modification, and P' the partial plan resulting from applying *mod* to P.

When to Recompute? – In Decomposition-based Planning

- Let P be a partial plan, *mod* a modification, and P' the partial plan resulting from applying *mod* to P.
- In case *mod* is a decomposition $m = (t, P_m)$ and there are also further methods for t, then we recompute the TDG. Otherwise, we perform an incremental heuristic calculation.

When to Recompute? – In Decomposition-based Planning

- Let P be a partial plan, *mod* a modification, and P' the partial plan resulting from applying *mod* to P.
- In case *mod* is a decomposition $m = (t, P_m)$ and there are also further methods for t, then we recompute the TDG. Otherwise, we perform an incremental heuristic calculation.
- Why? If there is just one method, the reachable action set cannot possibly change.

When to Recompute? – In Decomposition-based Planning

- Let P be a partial plan, *mod* a modification, and P' the partial plan resulting from applying *mod* to P.
- In case *mod* is a decomposition $m = (t, P_m)$ and there are also further methods for t, then we recompute the TDG. Otherwise, we perform an incremental heuristic calculation.
- Why? If there is just one method, the reachable action set cannot possibly change.
- There are other cases, which are not yet handled, though. E.g., causal links might also limit the available actions.

When to Recompute? – In Progression-based Planning

- Here, we have just two modifications: method application and progression (i.e., action application).

When to Recompute? – In Progression-based Planning

- Here, we have just two modifications: method application and progression (i.e., action application).
- With methods, we have the same situation as in decomposition-based planning (since also here, decompositions restrict the reachable actions).

When to Recompute? – In Progression-based Planning

- Here, we have just two modifications: method application and progression (i.e., action application).
- With methods, we have the same situation as in decomposition-based planning (since also here, decompositions restrict the reachable actions).
- Also, each progression allows a recomputation.

What if we don't Recompute TDG-c?

- If *mod* is not a decomposition (i.e., an insertion of a causal link, an ordering, or a variable constraint), we get: $h_{TDG-c}(P) = h_{TDG-c}(P')$

What if we don't Recompute TDG-c?

- If *mod* is not a decomposition (i.e., an insertion of a causal link, an ordering, or a variable constraint), we get: $h_{TDG-c}(P) = h_{TDG-c}(P')$
- If *mod* is a method $m = (t, tn_m)$ (without alternatives), we can set: $h_{TDG-c}(P') = h_{TDG-c}(P) - \sum_{\substack{t(\bar{c}) \in T_m \\ t(\bar{c}) \text{ primitive}}} c(t(\bar{c}))$

What if we don't Recompute TDG-c?

- If *mod* is not a decomposition (i.e., an insertion of a causal link, an ordering, or a variable constraint), we get:
  $h_{TDG-c}(P) = h_{TDG-c}(P')$
- If *mod* is a method $m = (t, tn_m)$ (without alternatives), we can set:
  $$h_{TDG-c}(P') = h_{TDG-c}(P) - \sum_{\substack{t(\bar{c}) \in T_m \\ t(\bar{c}) \text{ primitive}}} c(t(\bar{c}))$$
- These equations are specific to decomposition-based planning. The latter is required because the method's primitive tasks were accounted by the heuristic, but are now covered by the cost value of the search node.

What if we don't Recompute TDG-m?

- If *mod* is an ordering or variable insertion, we get: $h_{TDG-m}(P) = h_{TDG-m}(P')$

What if we don't Recompute TDG-m?

- If *mod* is an ordering or variable insertion, we get:
  $h_{TDG-m}(P) = h_{TDG-m}(P')$
- If *mod* is a causal link insertion or a decomposition (without alternatives), we get:
  $h_{TDG-m}(P) = h_{TDG-m}(P') - 1$

What if we don't Recompute TDG-m?

- If *mod* is an ordering or variable insertion, we get: $h_{TDG-m}(P) = h_{TDG-m}(P')$
- If *mod* is a causal link insertion or a decomposition (without alternatives), we get: $h_{TDG-m}(P) = h_{TDG-m}(P') - 1$
- No difference between decomposition- and progression-based planning.

Motivation

- We would like to exploit existing classical planning heuristics in HTN planning.

Motivation

- We would like to exploit existing classical planning heuristics in HTN planning.
- Issues:

Motivation

- We would like to exploit existing classical planning heuristics in HTN planning.
- Issues:
  - More expressive formalism (cf. lecture on expressiveness and next lecture on computational complexity). In particular: Hierarchical problems are undecidable, so they cannot be translated into classical problems.

Motivation

- We would like to exploit existing classical planning heuristics in HTN planning.
- Issues:
  - More expressive formalism (cf. lecture on expressiveness and next lecture on computational complexity). In particular: Hierarchical problems are undecidable, so they cannot be translated into classical problems.
  - In general, HTN problems do not have a goal description, so what's the classical problem's goal?

Motivation

- We would like to exploit existing classical planning heuristics in HTN planning.
- Issues:
  - More expressive formalism (cf. lecture on expressiveness and next lecture on computational complexity). In particular: Hierarchical problems are undecidable, so they cannot be translated into classical problems.
  - In general, HTN problems do not have a goal description, so what's the classical problem's goal?
  - In classical planning, all actions can be applied, in hierarchical planning only those reachable from the initial task (network).

Classical Heuristics in HTN Planning

Similarities and differences to TDG heuristic:

- Both heuristics can estimate plan cost or modification effort.

Classical Heuristics in HTN Planning

Similarities and differences to TDG heuristic:

- Both heuristics can estimate plan cost or modification effort.
- Both heuristics work for progression- and decomposition-based planning.

Classical Heuristics in HTN Planning

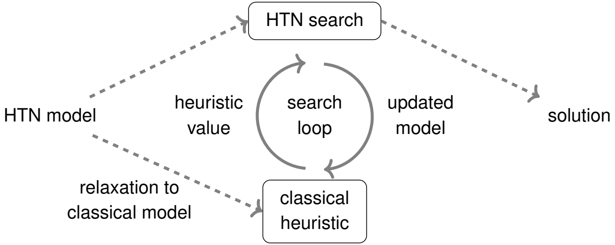
Similarities and differences to TDG heuristic:

- Both heuristics can estimate plan cost or modification effort.
- Both heuristics work for progression- and decomposition-based planning.
- They both, somehow, incorporate the TDG.

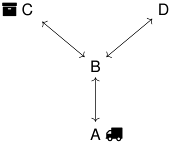
Classical Heuristics in HTN Planning

Similarities and differences to TDG heuristic:

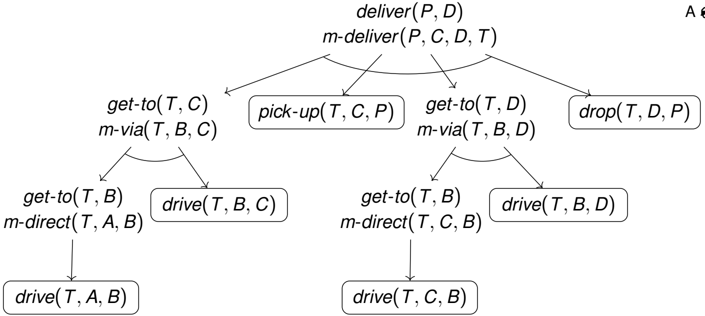
- Both heuristics can estimate plan cost or modification effort.
- Both heuristics work for progression- and decomposition-based planning.
- They both, somehow, incorporate the TDG.
- It is *not* a preprocessing heuristic: Its “heuristics model” gets adapted for every search node. In that way, it corresponds the previous heuristics with enabled recomputation.



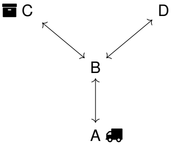
Simulating Composition



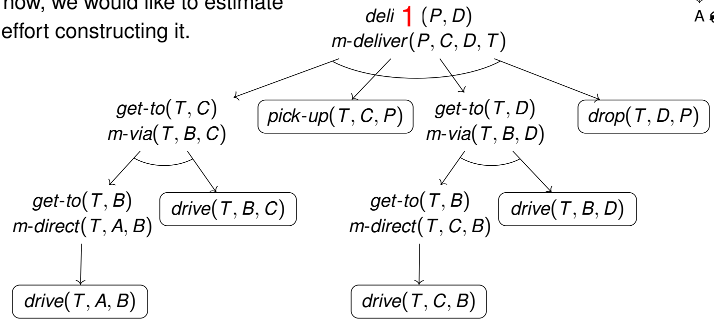
- Let's assume we want to "reconstruct" a decomposition tree via composition.



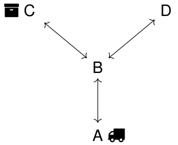
Simulating Composition



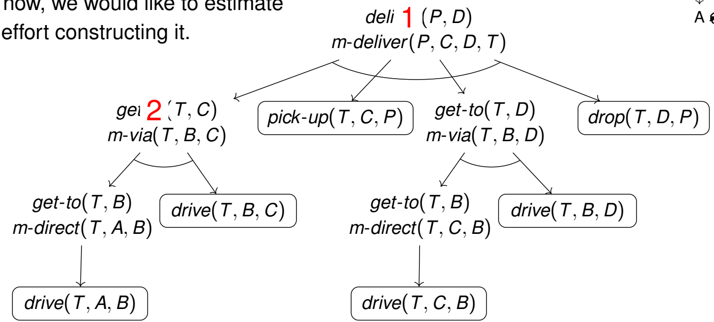
- Let's assume we want to "reconstruct" a decomposition tree via composition.
- For now, we would like to estimate the effort constructing it.



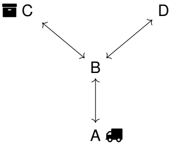
Simulating Composition



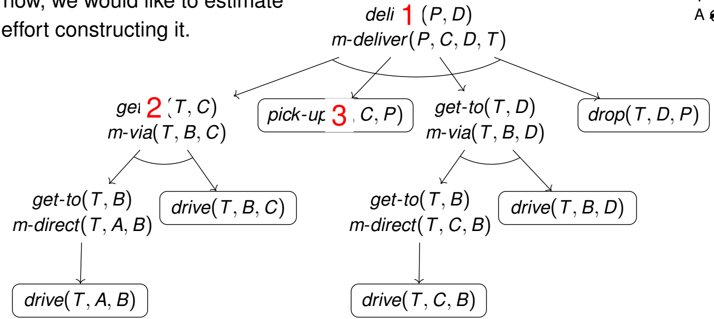
- Let's assume we want to "reconstruct" a decomposition tree via composition.
- For now, we would like to estimate the effort constructing it.



Simulating Composition

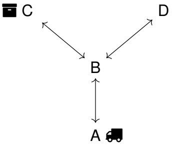


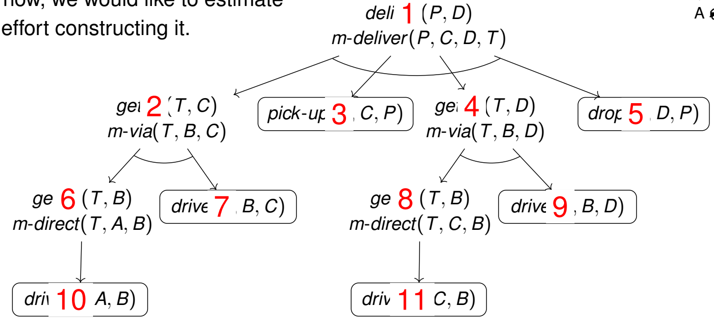
- Let's assume we want to "reconstruct" a decomposition tree via composition.
- For now, we would like to estimate the effort constructing it.



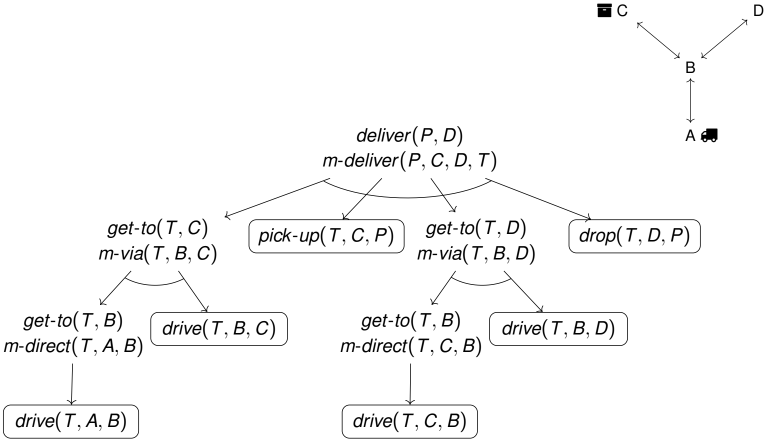
Simulating Composition

- Let's assume we want to "reconstruct" a decomposition tree via composition.
- For now, we would like to estimate the effort constructing it.

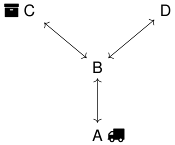




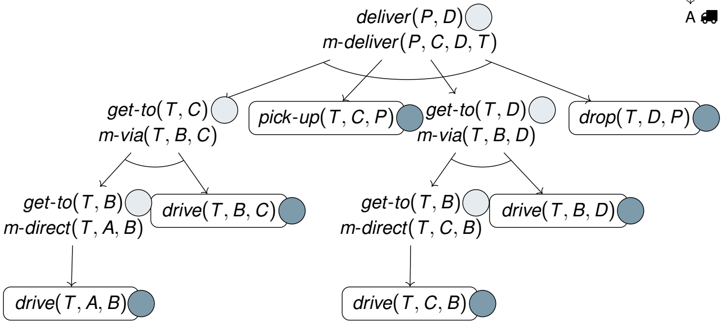
Simulating Composition



Simulating Composition

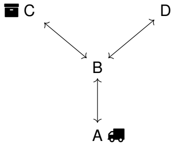


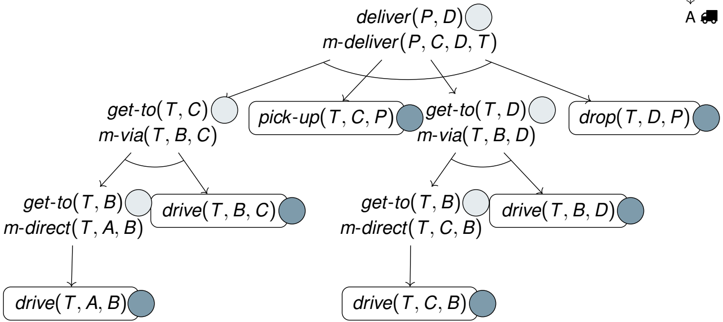
- Introduce new state features.

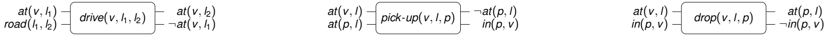


Simulating Composition

- Introduce new state features.

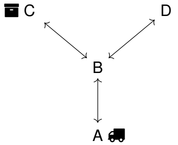


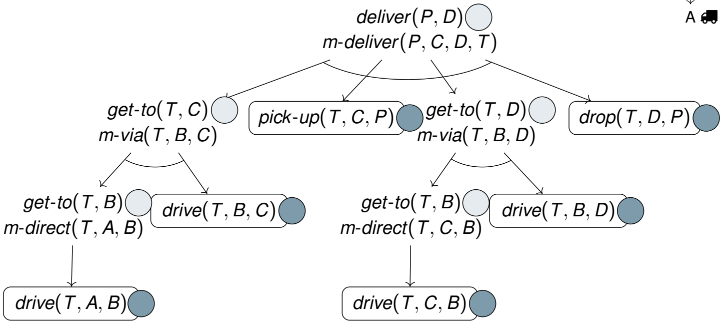


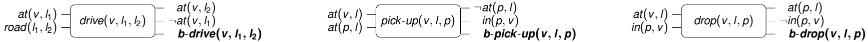


Simulating Composition

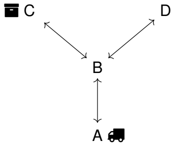
- Introduce new state features.
- Modify actions.



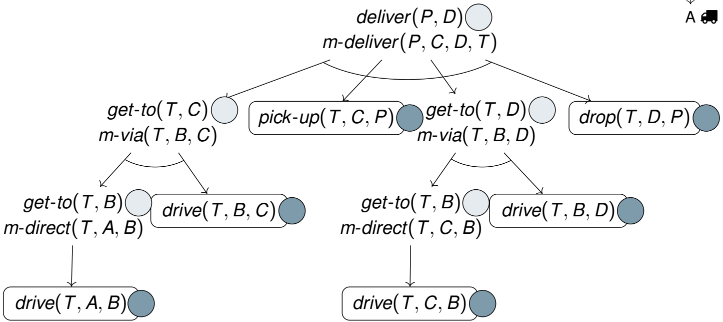


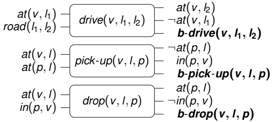


Simulating Composition

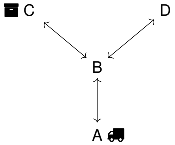


- Introduce new state features.
- Modify actions.

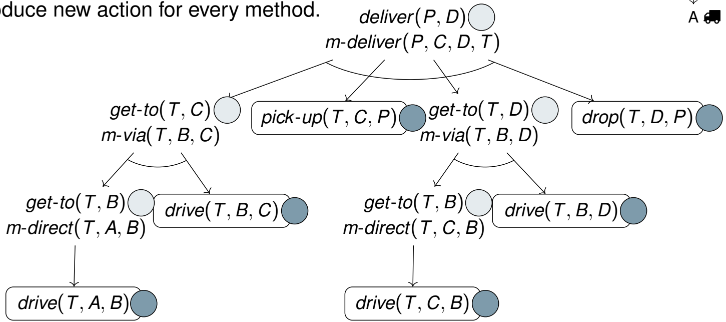


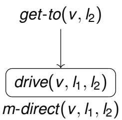


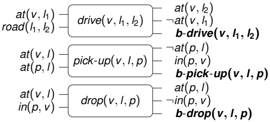
Simulating Composition



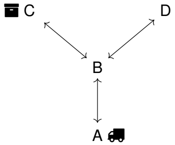
- Introduce new state features.
- Modify actions.
- Introduce new action for every method.



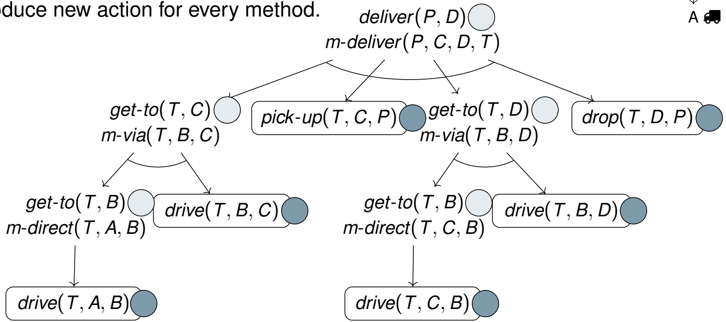


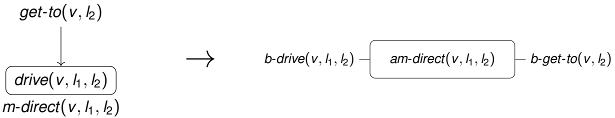


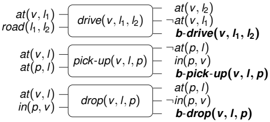
Simulating Composition



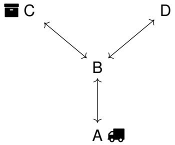
- Introduce new state features.
- Modify actions.
- Introduce new action for every method.



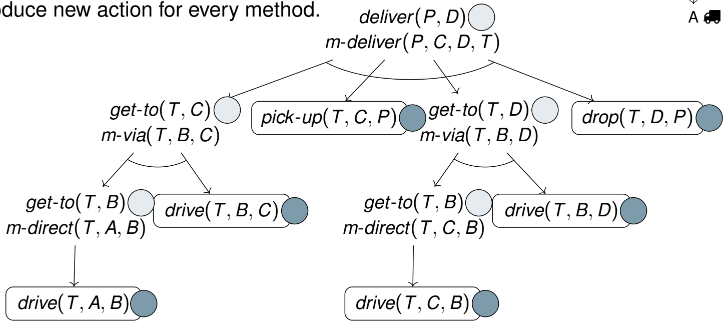


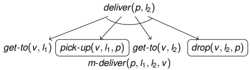


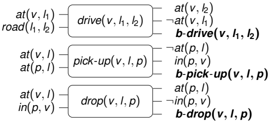
Simulating Composition



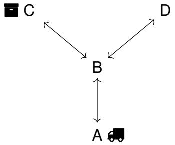
- Introduce new state features.
- Modify actions.
- Introduce new action for every method.



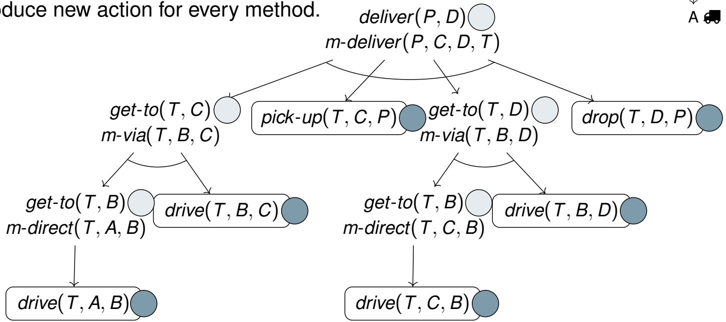


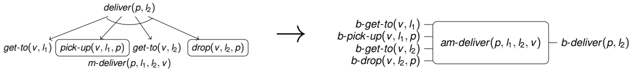


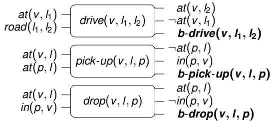
Simulating Composition



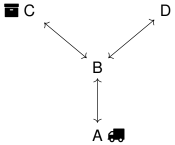
- Introduce new state features.
- Modify actions.
- Introduce new action for every method.



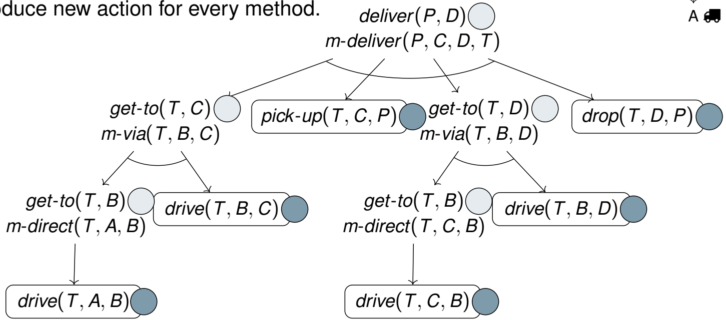




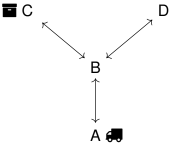
Simulating Composition



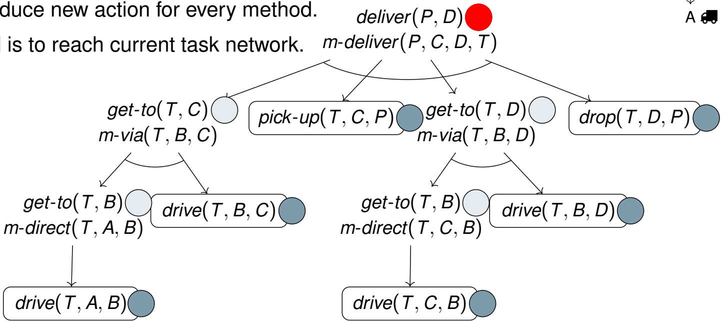
- Introduce new state features.
- Modify actions.
- Introduce new action for every method.



Simulating Composition

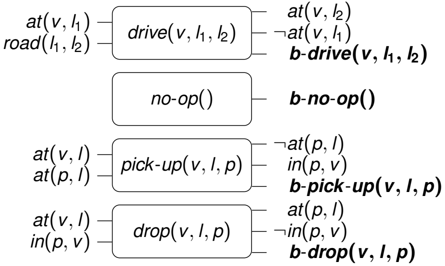


- Introduce new state features.
- Modify actions.
- Introduce new action for every method.
- Goal is to reach current task network.

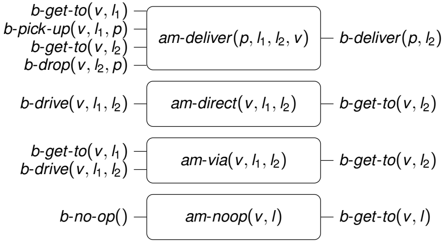


Simulating Composition – Resulting Model

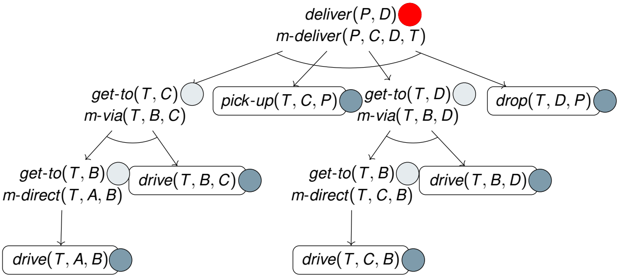
Altered action encodings:

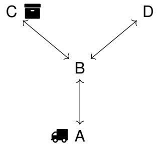


New actions encoding methods:



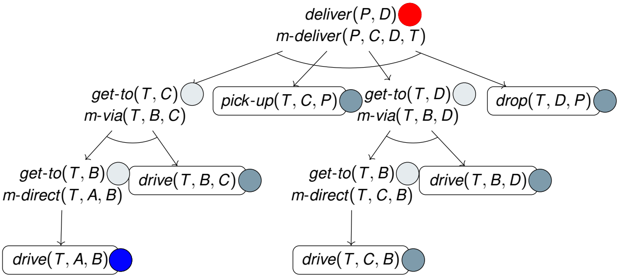
Unrelaxed Planning in the Transformed Model

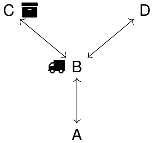


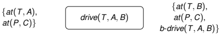


$\{at(T, A),$
$at(P, C)\}$

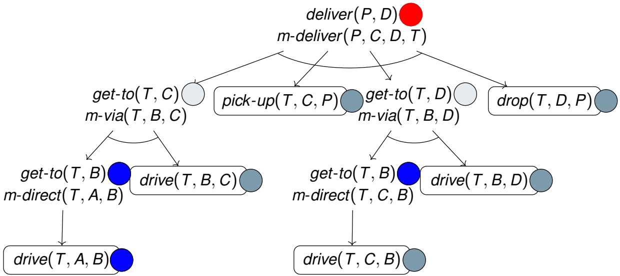
Unrelaxed Planning in the Transformed Model

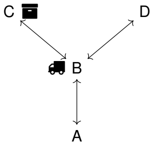


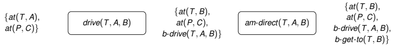




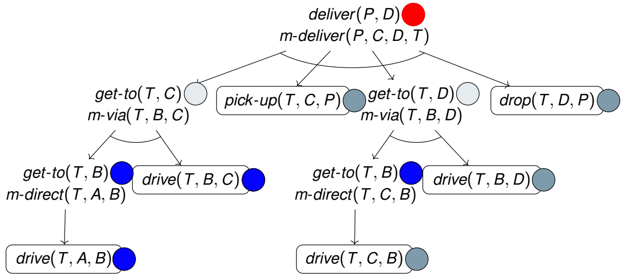
Unrelaxed Planning in the Transformed Model

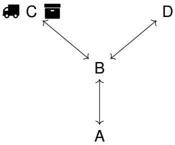


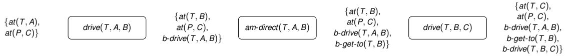




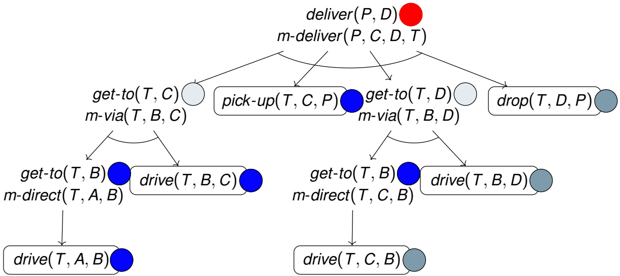
Unrelaxed Planning in the Transformed Model

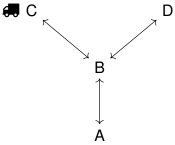


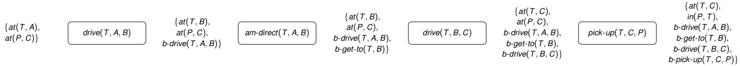




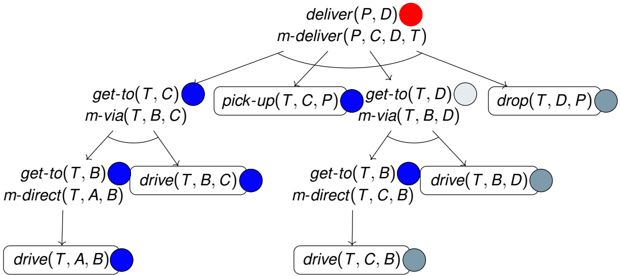
Unrelaxed Planning in the Transformed Model

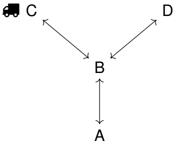


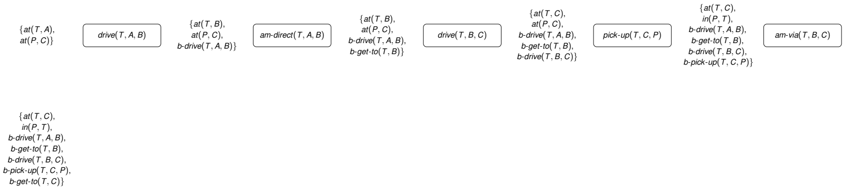




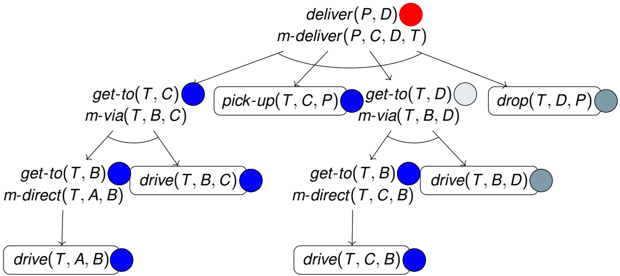
Unrelaxed Planning in the Transformed Model

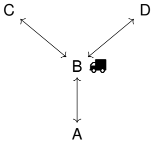


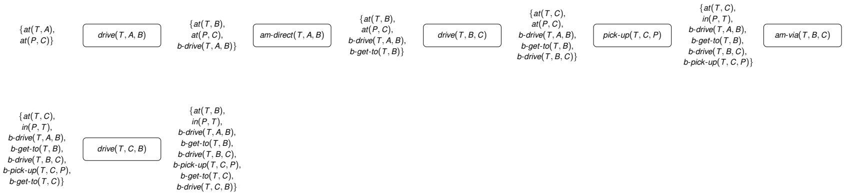




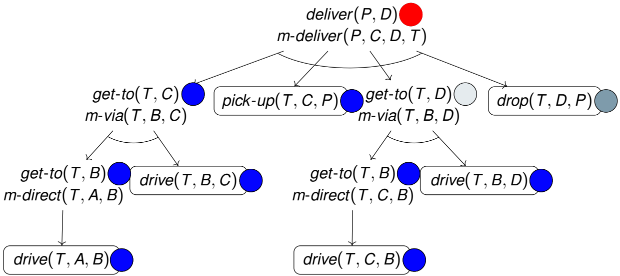
Unrelaxed Planning in the Transformed Model

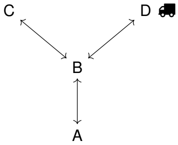


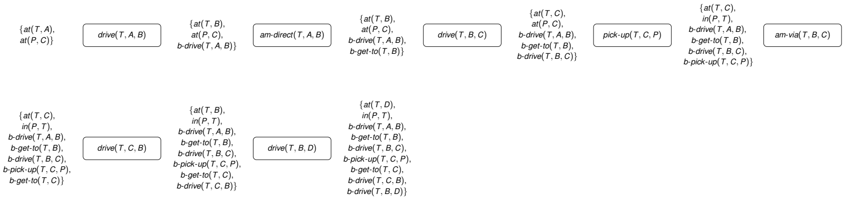




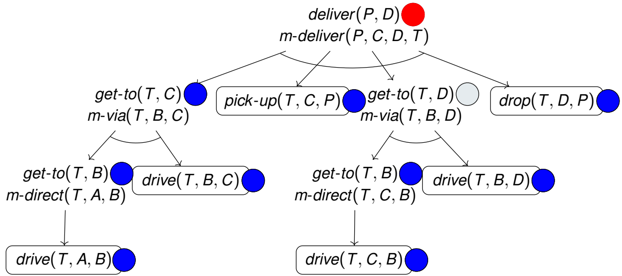
Unrelaxed Planning in the Transformed Model

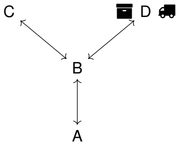


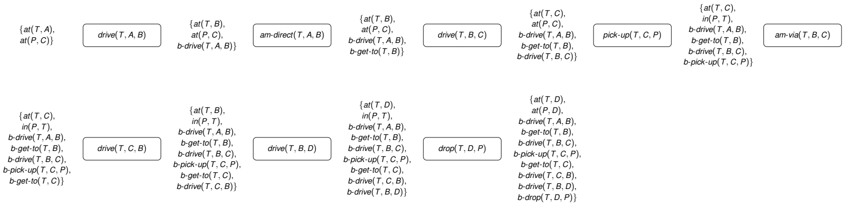




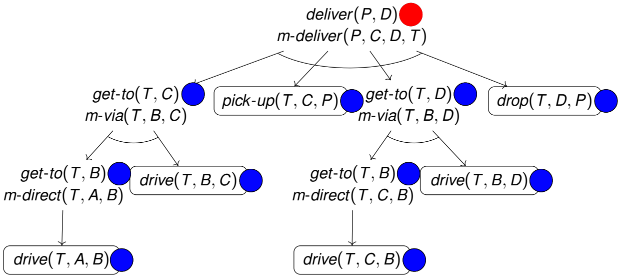
Unrelaxed Planning in the Transformed Model

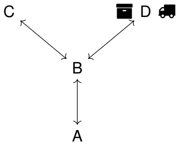


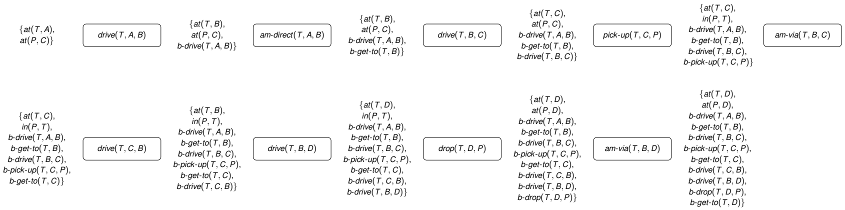




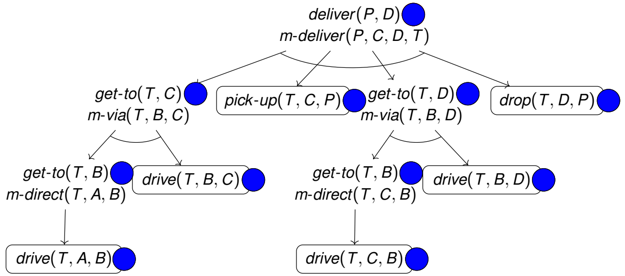
Unrelaxed Planning in the Transformed Model

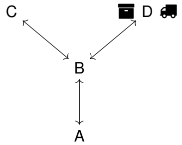


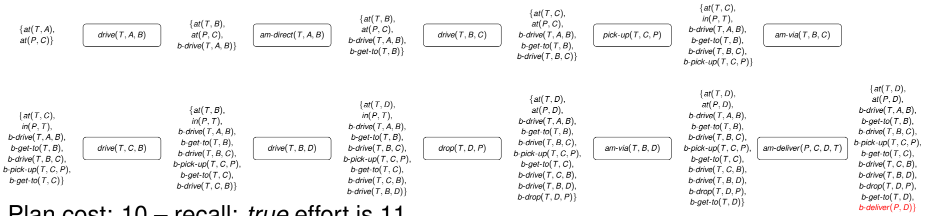




Unrelaxed Planning in the Transformed Model

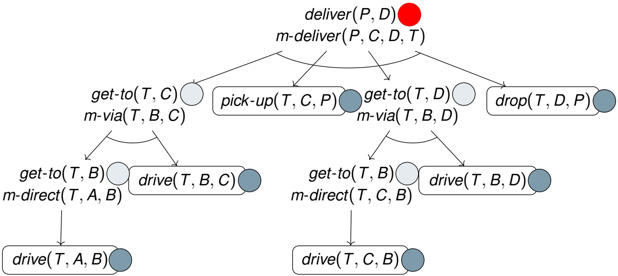


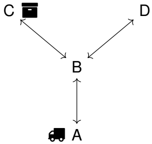




Plan cost: 10 – recall: *true* effort is 11.

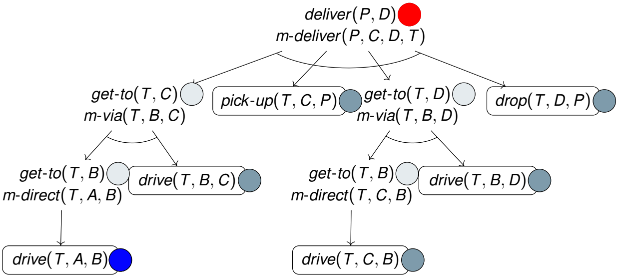
Relaxed Planning in the Transformed Model (Heuristic Computation)

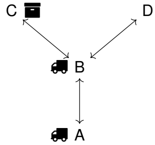


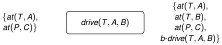


{at(T, A),
at(P, C)}

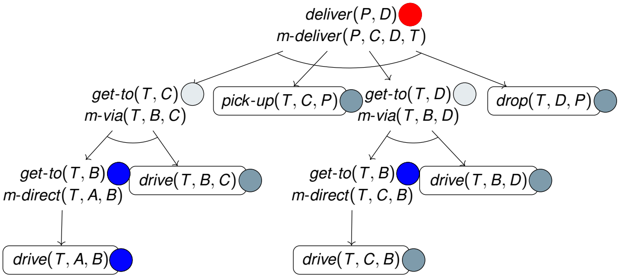
Relaxed Planning in the Transformed Model (Heuristic Computation)

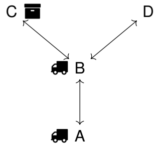


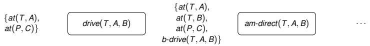




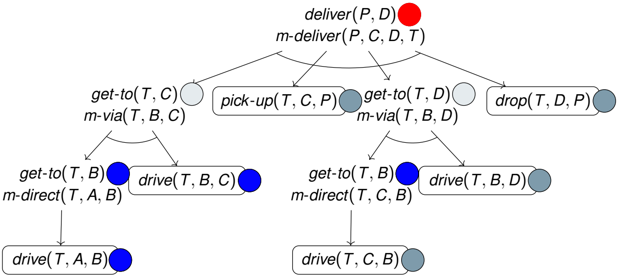
Relaxed Planning in the Transformed Model (Heuristic Computation)

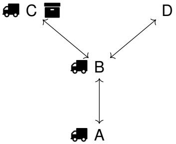


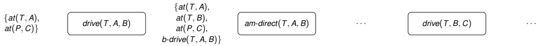




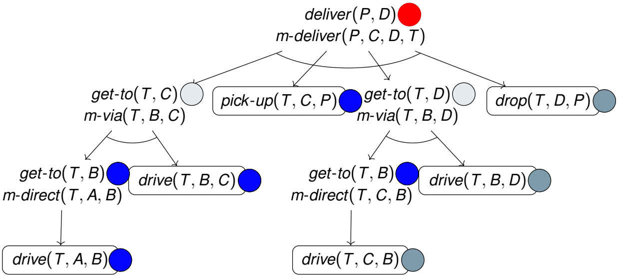
Relaxed Planning in the Transformed Model (Heuristic Computation)

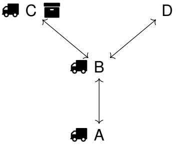




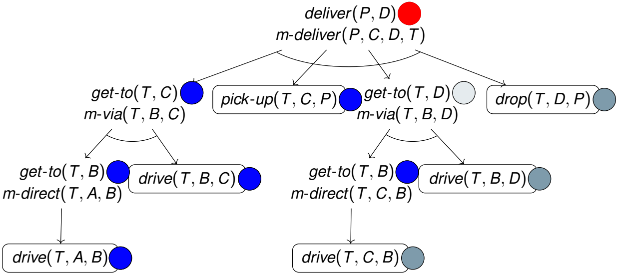


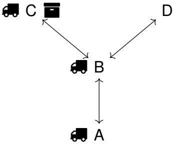
Relaxed Planning in the Transformed Model (Heuristic Computation)





Relaxed Planning in the Transformed Model (Heuristic Computation)

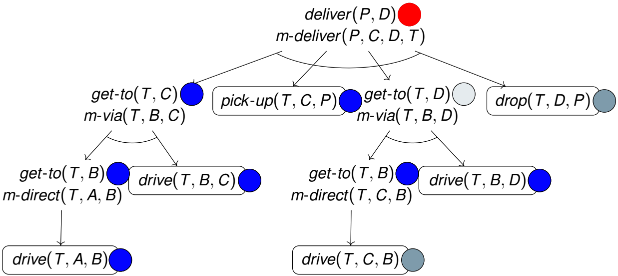


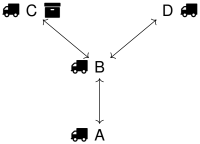


{*at*(*T*, *A*), *at*(*P*, *C*)} — *drive*(*T*, *A*, *B*) — {*at*(*T*, *A*), *at*(*T*, *B*), *at*(*P*, *C*), *b-drive*(*T*, *A*, *B*)} — *am-direct*(*T*, *A*, *B*) — … — *drive*(*T*, *B*, *C*) — … — *pick-up*(*T*, *C*, *P*) — … — *am-via*(*T*, *B*, *C*)

…

Relaxed Planning in the Transformed Model (Heuristic Computation)

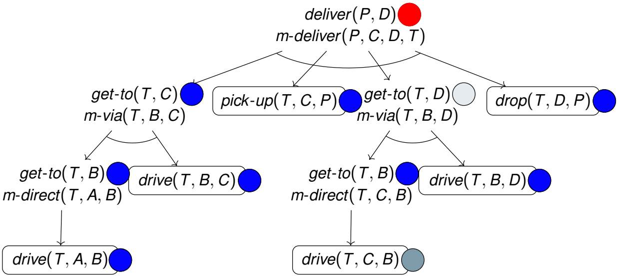


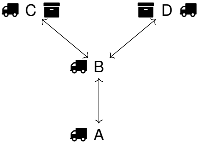


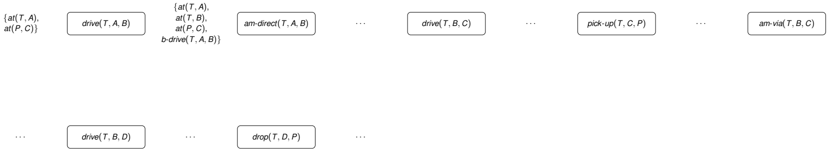
{$at(T, A)$, $at(P, C)$} | $drive(T, A, B)$ | {$at(T, A)$, $at(T, B)$, $at(P, C)$, $b\text{-}drive(T, A, B)$} | $am\text{-}direct(T, A, B)$ | … | $drive(T, B, C)$ | … | $pick\text{-}up(T, C, P)$ | … | $am\text{-}via(T, B, C)$

… | $drive(T, B, D)$ | …

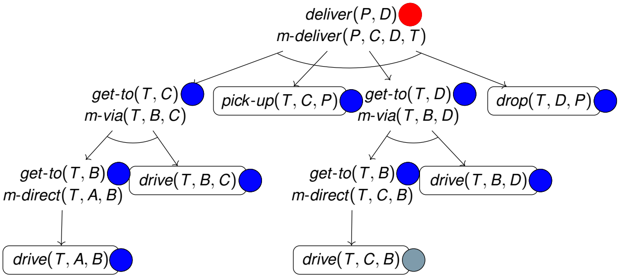
Relaxed Planning in the Transformed Model (Heuristic Computation)

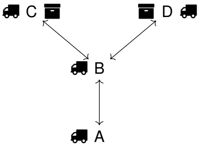


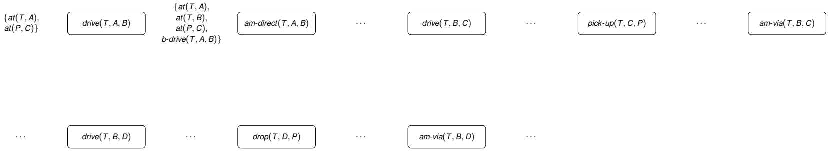




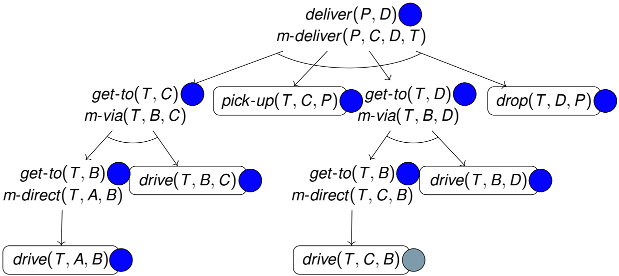
Relaxed Planning in the Transformed Model (Heuristic Computation)

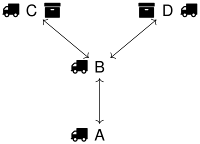






Relaxed Planning in the Transformed Model (Heuristic Computation)





{at(T, A), at(P, C)} → drive(T, A, B) → {at(T, A), at(T, B), at(P, C), b-drive(T, A, B)} → am-direct(T, A, B) → … → drive(T, B, C) → … → pick-up(T, C, P) → … → am-via(T, B, C)

… → drive(T, B, D) → … → drop(T, D, P) → … → am-via(T, B, D) → … → am-deliver(P, C, D, T) → …, b-deliver(P, D)

Plan cost: 9 – recall: *true* effort is 11 and unrelaxed plan cost is 10.

General Characteristics

- Technique simulates task composition,

General Characteristics

- Technique simulates task composition,
- ✓ incorporates hierarchical reachability information,
- ✓ combines it with information on state-based executability, and
- ✓ solves the problem of a missing goal description.

General Characteristics

- Technique simulates task composition,
- ✓ incorporates hierarchical reachability information,
- ✓ combines it with information on state-based executability, and
- ✓ solves the problem of a missing goal description.

- The transformation from an HTN problem to a classical problem is a relaxation.
- → The set of valid solutions increases.
- The new model essentially encodes the TDG.

General Characteristics

- Technique simulates task composition,
- ✓ incorporates hierarchical reachability information,
- ✓ combines it with information on state-based executability, and
- ✓ solves the problem of a missing goal description.

- The transformation from an HTN problem to a classical problem is a relaxation.
- → The set of valid solutions increases.
- The new model essentially encodes the TDG.

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

General Characteristics

- Technique simulates task composition,
- ✓ incorporates hierarchical reachability information,
- ✓ combines it with information on state-based executability, and
- ✓ solves the problem of a missing goal description.

- The transformation from an HTN problem to a classical problem is a relaxation.
- → The set of valid solutions increases.
- The new model essentially encodes the TDG.

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

Computational Aspects

- The size of the new model is *linear* in the input HTN domain.

Computational Aspects

- The size of the new model is *linear* in the input HTN domain.
- Most parts of the *model* are *static* during search. One only needs to update:

Computational Aspects

- The size of the new model is *linear* in the input HTN domain.
- Most parts of the *model* are *static* during search. One only needs to update:
  - Initial state.

Computational Aspects

- The size of the new model is *linear* in the input HTN domain.
- Most parts of the *model* are *static* during search. One only needs to update:
  - Initial state.
  - Goal description.

Resulting Heuristic Values

- If unit costs are used, the heuristic value encodes the search effort of a progression search.

Resulting Heuristic Values

- If unit costs are used, the heuristic value encodes the search effort of a progression search.
- If method actions cost one and all primitive tasks' cost equals their number of preconditions, then the heuristic value encodes the search effort of a decomposition search.

Resulting Heuristic Values

- If unit costs are used, the heuristic value encodes the search effort of a progression search.
- If method actions cost one and all primitive tasks' cost equals their number of preconditions, then the heuristic value encodes the search effort of a decomposition search.
- If all actions stemming from primitive tasks keep their original costs and all new actions cost 0, then the heuristic value estimates the resulting plan costs.

Resulting Heuristic Values

- If unit costs are used, the heuristic value encodes the search effort of a progression search.
- If method actions cost one and all primitive tasks' cost equals their number of preconditions, then the heuristic value encodes the search effort of a decomposition search.
- If all actions stemming from primitive tasks keep their original costs and all new actions cost 0, then the heuristic value estimates the resulting plan costs.
- When the used classical heuristic has one of the following properties, the resulting HTN heuristic has it, too:

Resulting Heuristic Values

- If unit costs are used, the heuristic value encodes the search effort of a progression search.
- If method actions cost one and all primitive tasks' cost equals their number of preconditions, then the heuristic value encodes the search effort of a decomposition search.
- If all actions stemming from primitive tasks keep their original costs and all new actions cost 0, then the heuristic value estimates the resulting plan costs.
- When the used classical heuristic has one of the following properties, the resulting HTN heuristic has it, too:
  - Safety.

Resulting Heuristic Values

- If unit costs are used, the heuristic value encodes the search effort of a progression search.
- If method actions cost one and all primitive tasks' cost equals their number of preconditions, then the heuristic value encodes the search effort of a decomposition search.
- If all actions stemming from primitive tasks keep their original costs and all new actions cost 0, then the heuristic value estimates the resulting plan costs.
- When the used classical heuristic has one of the following properties, the resulting HTN heuristic has it, too:
    - Safety.
    - Goal-awareness.

Resulting Heuristic Values

- If unit costs are used, the heuristic value encodes the search effort of a progression search.
- If method actions cost one and all primitive tasks' cost equals their number of preconditions, then the heuristic value encodes the search effort of a decomposition search.
- If all actions stemming from primitive tasks keep their original costs and all new actions cost 0, then the heuristic value estimates the resulting plan costs.
- When the used classical heuristic has one of the following properties, the resulting HTN heuristic has it, too:
  - Safety.
  - Goal-awareness.
  - Admissibility (only if costs are chosen as above).

Discussion

- Technique can be combined with *many* classical heuristics.

Discussion

- Technique can be combined with *many* classical heuristics.
- It thus profits *automatically* from any progress made in classical heuristic search.

Discussion

- Technique can be combined with *many* classical heuristics.
- It thus profits *automatically* from any progress made in classical heuristic search.
- Applicable to both decomposition-based and progression-based search.

Discussion

- Technique can be combined with *many* classical heuristics.
- It thus profits *automatically* from any progress made in classical heuristic search.
- Applicable to both decomposition-based and progression-based search.
- The new technique in general neither dominates the TDG heuristics nor gets dominated by it:

Discussion

- Technique can be combined with *many* classical heuristics.
- It thus profits *automatically* from any progress made in classical heuristic search.
- Applicable to both decomposition-based and progression-based search.
- The new technique in general neither dominates the TDG heuristics nor gets dominated by it:
    - The technique can clearly dominate TDG heuristics, because it incorporates costs of inserted actions (which the TDG heuristics do not). Recall what happens in domains in which tasks can be decomposed into empty task networks.

Discussion

- Technique can be combined with *many* classical heuristics.
- It thus profits *automatically* from any progress made in classical heuristic search.
- Applicable to both decomposition-based and progression-based search.
- The new technique in general neither dominates the TDG heuristics nor gets dominated by it:
  - The technique can clearly dominate TDG heuristics, because it incorporates costs of inserted actions (which the TDG heuristics do not). Recall what happens in domains in which tasks can be decomposed into empty task networks.
  - Many classical heuristics compute heuristic values that are polynomial in the the input. The TDG heuristics can come up with exponential heuristic values.

Summary

- Hierarchical planning as search relies, just as classical planning, on heuristics to guide search.

Summary

- Hierarchical planning as search relies, just as classical planning, on heuristics to guide search.
- There are still only a very heuristics and techniques known – most other approaches rely on domain-specific models.

Summary

- Hierarchical planning as search relies, just as classical planning, on heuristics to guide search.
- There are still only a very heuristics and techniques known – most other approaches rely on domain-specific models.
- We introduced the task decomposition graph (TDG) as a basis for many heuristics.

Summary

- Hierarchical planning as search relies, just as classical planning, on heuristics to guide search.
- There are still only a very heuristics and techniques known – most other approaches rely on domain-specific models.
- We introduced the task decomposition graph (TDG) as a basis for many heuristics.
- Landmarks – tasks that occur on any sequence from the initial task network to a solution – can be used as basis for heuristics.

Summary

- Hierarchical planning as search relies, just as classical planning, on heuristics to guide search.
- There are still only a very heuristics and techniques known – most other approaches rely on domain-specific models.
- We introduced the task decomposition graph (TDG) as a basis for many heuristics.
- Landmarks – tasks that occur on any sequence from the initial task network to a solution – can be used as basis for heuristics.
- The TDG heuristics compute admissible estimates, but take task insertion into account to only a limited extent.

Summary

- Hierarchical planning as search relies, just as classical planning, on heuristics to guide search.
- There are still only a very heuristics and techniques known – most other approaches rely on domain-specific models.
- We introduced the task decomposition graph (TDG) as a basis for many heuristics.
- Landmarks – tasks that occur on any sequence from the initial task network to a solution – can be used as basis for heuristics.
- The TDG heuristics compute admissible estimates, but take task insertion into account to only a limited extent.
- We can also exploit *classical* heuristics for hierarchical planning by a relaxing problem transformation.