Change the Plan - How hard can that be?

Gregor Behnke, Daniel Höller, Pascal Bercher, Susanne Biundo

Ulm University, Institute of Artificial Intelligence

June 17, 2016

ICAPS 2016 – London
Changing Plans

Planning doesn't take place in a vacuum

Planners can generate solutions users might not like

Preferences can be infeasible

Users might not know their preferences

... or cannot be expected to be asked about them

⇒ Integrate the user into the planning process

⇒ We have to allow for changes to plans
Changing Plans

- Planning doesn’t take place in a vacuum
Changing Plans

- Planning doesn’t take place in a vacuum
- Planners can generate solutions users might not like
Changing Plans

- Planning doesn’t take place in a vacuum
- Planners can generate solutions users might not like
- Preferences can be infeasible
  - Users might not know their preferences
  - ... or cannot be expected to be asked about them
Changing Plans

- Planning doesn’t take place in a vacuum
- Planners can generate solutions users might not like
- Preferences can be infeasible
  - Users might not know their preferences
  - ... or cannot be expected to be asked about them

⇒ Integrate the user into the planning process
Changing Plans

- Planning doesn’t take place in a vacuum
- Planners can generate solutions users might not like
- Preferences can be infeasible
  - Users might not know their preferences
  - ... or cannot be expected to be asked about them

⇒ Integrate the user into the planning process
⇒ We have to allow for changes to plans
Changing Plans

Changing plans is important for user-centred planning applications, e.g., mixed-initiative planning.
Changing Plans

Changing plans is important for user-centred planning applications, e.g., mixed-initiative planning

We want to understand its theoretical foundations
Changing Plans

Changing plans is important for user-centred planning applications, e.g., mixed-initiative planning

We want to understand its theoretical foundations

- Discuss what changing plans means in an HTN context
- Provide formal descriptions of several change operations
- Investigate their computational complexity
Hierarchical Task Network (HTN) Planning

\( \mathcal{P} = (P, C, c_i, M, L, s_i) \)
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_i, M, L, s_i) \]

- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_i, M, L, s_i) \]
- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
- \( c_i \in C \) the initial task

A solution \( t_n \in Sol(\mathcal{P}) \) must
- be a refinement of the initial task
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_I, M, L, s_I) \]

- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
- \( c_I \in C \) the initial task
- \( M \subseteq C \times 2^{TN} \) the methods

A solution \( t_n \in Sol(\mathcal{P}) \) must

- be a refinement of the initial task
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_I, M, L, s_I) \]
- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
- \( c_I \in C \) the initial task
- \( M \subseteq C \times 2^{TN} \) the methods

A solution \( tn \in Sol(\mathcal{P}) \) must
- be a refinement of the initial task
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_I, M, L, s_I) \]

- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
- \( c_I \in C \) the initial task
- \( M \subseteq C \times 2^{TN} \) the methods

A solution \( tn \in \text{Sol}(\mathcal{P}) \) must

- be a refinement of the initial task
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_I, M, L, s_I) \]
- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
- \( c_I \in C \) the initial task
- \( M \subseteq C \times 2^{TN} \) the methods

A solution \( tn \in Sol(\mathcal{P}) \) must
- be a refinement of the initial task
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_i, M, L, s_i) \]

- \( P \): a set of primitive tasks
- \( C \): a set of compound tasks
- \( c_i \in C \): the initial task
- \( M \subseteq C \times 2^{TN} \): the methods
- \( L \): a set of variables
- \( s_i \subseteq L \): the initial state

A solution \( t_n \in \text{Sol}(\mathcal{P}) \) must
- be a refinement of the initial task
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_I, M, L, s_I) \]

- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
- \( c_I \in C \) the initial task
- \( M \subseteq C \times 2^{TN} \) the methods

A solution \( tn \in Sol(\mathcal{P}) \) must
- be a refinement of the initial task
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (\mathcal{P}, \mathcal{C}, c_i, M, L, s_i) \]

- \( \mathcal{P} \) a set of primitive tasks
- \( \mathcal{C} \) a set of compound tasks
- \( c_i \in \mathcal{C} \) the initial task
- \( M \subseteq \mathcal{C} \times 2^{\mathcal{TN}} \) the methods

A solution \( t_n \in Sol(\mathcal{P}) \) must

- be a refinement of the initial task
- only contain primitive tasks
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_i, M, L, s_i) \]

- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
- \( c_i \in C \) the initial task
- \( M \subseteq C \times 2^{TN} \) the methods
- \( L \) a set of variables
- \( s_i \subseteq L \) the initial state

A solution \( tn \in Sol(\mathcal{P}) \) must

- be a refinement of the initial task
- only contain primitive tasks
Hierarchical Task Network (HTN) Planning

\[ \mathcal{P} = (P, C, c_I, M, L, s_I) \]

- \( P \) a set of primitive tasks
- \( C \) a set of compound tasks
- \( c_I \in C \) the initial task
- \( M \subseteq C \times 2^{TN} \) the methods
- \( L \) a set of variables
- \( s_I \subseteq L \) the initial state

A solution \( tn \in Sol(\mathcal{P}) \) must
- be a refinement of the initial task
- only contain primitive tasks
- have a linearization, executable from the initial state
Motivation of Hierarchies

HTN planning problems can pose restrictions that classical planning cannot
Motivation of Hierarchies

HTN planning problems can pose restrictions that classical planning cannot

- Every plan must contain the same amount of a’s and b’s
- a can be executed twice in a row, but not thrice
- HTNs can express all context free and some context sensitive language, while classical planning is limited to regular structures
- Precondition-free HTNs can express classical planning
Motivation of Hierarchies

HTN planning problems can pose restrictions that classical planning cannot

- Every plan must contain the same amount of $a$’s and $b$’s
- $a$ can be executed twice in a row, but not thrice
- HTNs can express all context free and some context sensitive language, while classical planning is limited to regular structures
- Precondition-free HTNs can express classical planning

When changing plans, we can either:

- Ignore the domain’s hierarchy and just try to find an executable solution
- Find a solution adhering to the hierarchy, s.t. we keep all restrictions
Motivation of Hierarchies

HTN planning problems can pose restrictions that classical planning cannot:

- Every plan must contain the same amount of a’s and b’s
- a can be executed twice in a row, but not thrice
- HTNs can express all context free and some context sensitive language, while classical planning is limited to regular structures
- Precondition-free HTNs can express classical planning

When changing plans, we can either:

- Ignore the domain’s hierarchy and just try to find an executable solution
- Find a solution adhering to the hierarchy, s.t. we keep all restrictions
We’ve investigated a wide range of change requests

- 5 request objectives
- 3 request restrictions

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>delete</th>
<th>exchange</th>
<th>order</th>
<th>avoid effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>no changes</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td>(k) changes</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
</tr>
<tr>
<td>any changes</td>
<td>un-dec</td>
<td>un-dec</td>
<td>un-dec</td>
<td>un-dec</td>
<td>un-dec</td>
</tr>
</tbody>
</table>
Results

We’ve investigated a wide range of change requests

- 5 request objectives
- 3 request restrictions

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>delete</th>
<th>exchange</th>
<th>order</th>
<th>avoid effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>no changes</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td>(k) changes</td>
<td>(\text{NEXPTIME})</td>
<td>(\text{NEXPTIME})</td>
<td>(\text{NEXPTIME})</td>
<td>(\text{NEXPTIME})</td>
<td>(\text{NEXPTIME})</td>
</tr>
<tr>
<td>any changes</td>
<td>un-dec</td>
<td>un-dec</td>
<td>un-dec</td>
<td>un-dec</td>
<td>un-dec</td>
</tr>
</tbody>
</table>

- Most proofs are structurally similar
- We will only show one from each group
Add Task – no changes

**Definition (ADD-NO-CHANGE)**

Given a planning problem $\mathcal{P}$, a solution $tn \in \text{Sol}(\mathcal{P})$, and task $t$. ADD-NO-CHANGE is to decide whether the task network $tn'$, which is $tn$ with an additional task $t$ and some ordering constraints added, is still a solution.
Add Task – no changes

Definition (ADD-NO-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in \text{Sol}(\mathcal{P})$, and task $t$. ADD-NO-CHANGE is to decide whether the task network $tn'$, which is $tn$ with an additional task $t$ and some ordering constraints added, is still a solution.
Add Task – no changes

Definition (ADD-NO-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in Sol(\mathcal{P})$, and task $t$. ADD-NO-CHANGE is to decide whether the task network $tn'$, which is $tn$ with an additional task $t$ and some ordering constraints added, is still a solution.
Add Task – no changes

Definition (ADD-NO-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in \text{Sol}(\mathcal{P})$, and task $t$. ADD-NO-CHANGE is to decide whether the task network $tn'$, which is $tn$ with an additional task $t$ and some ordering constraints added, is still a solution.

- Decomposition becomes invalid
Add Task – no changes

Definition (ADD-NO-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in \text{Sol}(\mathcal{P})$, and task $t$. ADD-NO-CHANGE is to decide whether the task network $tn'$, which is $tn$ with an additional task $t$ and some ordering constraints added, is still a solution.

- Decomposition becomes invalid
Add Task – no changes

Definition (ADD-NO-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in \text{Sol}(\mathcal{P})$, and task $t$. ADD-NO-CHANGE is to decide whether the task network $tn'$, which is $tn$ with an additional task $t$ and some ordering constraints added, is still a solution.

- Decomposition becomes invalid
- We (potentially) have to find a new linearisation
Plan Verification

Definition (VERIFYTN)

Given a planning problem $\mathcal{P}$ and a task network $tn$. Is $tn \in Sol(\mathcal{P})$?
Plan Verification

Definition (VERIFYTN)

Given a planning problem $\mathcal{P}$ and a task network $tn$. Is $tn \in Sol(\mathcal{P})$?

What do we have to check?
Plan Verification

Definition (VERIFYTN)
Given a planning problem \( \mathcal{P} \) and a task network \( tn \). Is \( tn \in Sol(\mathcal{P}) \)?

What do we have to check?

- Refinement
Plan Verification

Definition (VERIFYTN)
Given a planning problem $\mathcal{P}$ and a task network $tn$. Is $tn \in Sol(\mathcal{P})$?

What do we have to check?

- Refinement
- Primitive
Plan Verification

Definition (\textsc{VERIFYTN})

Given a planning problem $\mathcal{P}$ and a task network $tn$. Is $tn \in \text{Sol}(\mathcal{P})$?

What do we have to check?

- Refinement
- Primitive
- Executability
Plan Verification

Definition (\textsc{VerifyTN})

Given a planning problem $\mathcal{P}$ and a task network $tn$. Is $tn \in Sol(\mathcal{P})$?

What do we have to check?

- Refinement
- Primitive
- Executability

Theorem

\textsc{VerifyTN} is \textbf{NP}-complete
Add Task – no changes

Theorem

ADD-NO-CHANGE is NP-complete.

Proof:
Add Task – no changes

Theorem

**ADD-NO-CHANGE is NP-complete.**

Proof: **Membership:**

- Add the new task $t$ and guess some additional ordering constraints
- Check the resulting task network using the NP algorithm for VERIFYTN
Add Task – no changes

Theorem

ADD\-NO\-CHANGE is \textbf{NP}\-complete.

Proof: Hardness: Reduction from VERIFYTN.
Add Task – no changes

Theorem

ADD-NO-CHANGE is NP-complete.

Proof: Hardness: Reduction from VERIFYTN.
Add Task – no changes

Theorem

**ADD-NO-CHANGE** is **NP-complete**.

Proof: **Hardness:** Reduction from **VERIFYTN**.

![Diagram](image)
Add Task – no changes

**Theorem**

**ADD-NO-CHANGE** is **NP-complete**.

**Proof:** **Hardness:** Reduction from **VERIFYTN**.

- Can we add \( t_n \) to \( t_n \)?
Add Task – no changes

Theorem

ADD-NO-CHANGE is NP-complete.

Proof: **Hardness:** Reduction from VERIFYTN. Also holds if the domain does not contain preconditions and effects.
Add Task – no changes

Theorem

**ADD-NO-CHANGE** is **NP**-complete.

Proof: **Hardness:** Reduction from **VERIFYTN**.
Also holds if the domain does not contain preconditions and effects.
Add Task – no changes

Theorem

**ADD-NO-CHANGE is NP-complete.**

Proof: **Hardness:** Reduction from VERIFYTN.
Also holds if the domain does not contain preconditions and effects.

- Can we add $t_a$ to $tn$?
Add Ordering – $k$ changes

**Definition (ORDERING-K-CHANGE)**

Given a planning problem $\mathcal{P}$, a solution $tn \in Sol(\mathcal{P})$, and two tasks $t_1, t_2$ from $tn$. ORDERING-K-CHANGE is to decide whether another solution $tn'$ can be obtained from $tn$ by at most $k$ of the following operations:

- Adding/removing a primitive task
- Adding/removing an ordering constraint
- Such that $t_1 < t_2$ holds in $tn'$ and neither $t_1$ nor $t_2$ are deleted.
Add Ordering – $k$ changes

**Definition (ORDERING-K-CHANGE)**

Given a planning problem $\mathcal{P}$, a solution $tn \in \text{Sol}(\mathcal{P})$, and two tasks $t_1, t_2$ from $tn$. ORDERING-K-CHANGE is to decide whether another solution $tn'$ can be obtained from $tn$ by at most $k$ of the following operations

- Adding/removing a primitive task
- Adding/removing an ordering constraint such that $t_1 < t_2$ holds in $tn'$ and neither $t_1$ nor $t_2$ are deleted.
Add Ordering – \( k \) changes

**Definition (ORDERING-K-CHANGE)**

Given a planning problem \( \mathcal{P} \), a solution \( t_n \in \text{Sol}(\mathcal{P}) \), and two tasks \( t_1, t_2 \) from \( t_n \). ORDERING-K-CHANGE is to decide whether another solution \( t_n' \) can be obtained from \( t_n \) by at most \( k \) of the following operations:

- Adding/removing a primitive task
- Adding/removing an ordering constraint

such that \( t_1 < t_2 \) holds in \( t_n' \) and neither \( t_1 \) nor \( t_2 \) are deleted.
Add Ordering – $k$ changes

**Definition (ORDERING-K-CHANGE)**

Given a planning problem $\mathcal{P}$, a solution $tn \in Sol(\mathcal{P})$, and two tasks $t_1, t_2$ from $tn$. ORDERING-K-CHANGE is to decide whether another solution $tn'$ can be obtained from $tn$ by at most $k$ of the following operations

- Adding/removing a primitive task
- Adding/removing an ordering constraint such that $t_1 < t_2$ holds in $tn'$ and neither $t_1$ nor $t_2$ are deleted.

- Remove tasks
Add Ordering – $k$ changes

Definition (ORDERING-K-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in Sol(\mathcal{P})$, and two tasks $t_1$, $t_2$ from $tn$. ORDERING-K-CHANGE is to decide whether another solution $tn'$ can be obtained from $tn$ by at most $k$ of the following operations:

- Adding/removing a primitive task
- Adding/removing an ordering constraint such that $t_1 < t_2$ holds in $tn'$ and neither $t_1$ nor $t_2$ are deleted.

- Remove tasks
- Old decomposition becomes invalid
Add Ordering – $k$ changes

Definition (ORDERING-K-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in \text{Sol}(\mathcal{P})$, and two tasks $t_1, t_2$ from $tn$. ORDERING-K-CHANGE is to decide whether another solution $tn'$ can be obtained from $tn$ by at most $k$ of the following operations:

- Adding/removing a primitive task
- Adding/removing an ordering constraint such that $t_1 < t_2$ holds in $tn'$ and neither $t_1$ nor $t_2$ are deleted.
- Remove tasks
- Old decomposition becomes invalid
- Add new tasks
Add Ordering – $k$ changes

Definition (ORDERING-K-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in Sol(\mathcal{P})$, and two tasks $t_1, t_2$ from $tn$. ORDERING-K-CHANGE is to decide whether another solution $tn'$ can be obtained from $tn$ by at most $k$ of the following operations:

- Adding/removing a primitive task
- Adding/removing an ordering constraint such that $t_1 < t_2$ holds in $tn'$ and neither $t_1$ nor $t_2$ are deleted.
- Remove tasks
- Old decomposition becomes invalid
- Add new tasks
- Add new ordering constraints
Add Ordering – $k$ changes

**Definition (ORDERING-K-CHANGE)**

Given a planning problem $\mathcal{P}$, a solution $tn \in Sol(\mathcal{P})$, and two tasks $t_1, t_2$ from $tn$. ORDERING-K-CHANGE is to decide whether another solution $tn'$ can be obtained from $tn$ by at most $k$ of the following operations

- Adding/removing a primitive task
- Adding/removing an ordering constraint such that $t_1 < t_2$ holds in $tn'$ and neither $t_1$ nor $t_2$ are deleted.

- Remove tasks
- Old decomposition becomes invalid
- Add new tasks
- Add new ordering constraints
- Find new decomposition
Add Ordering – $k$ changes

Theorem

ORDERING-$K$-CHANGE is **NEXPTIME-complete**.

Proof:
Add Ordering – $k$ changes

Theorem

ORDERING-K-CHANGE is NEXPTIME-complete.

Proof: Membership:

- Guess a number $l \leq k$
- Apply $l$ allowed operations to the task network $tn$
- Check the resulting task network using the NP algorithm for VERIFYTN
Add Ordering – $k$ changes

**Theorem**

**ORDERING-K-CHANGE** is **NEXPTIME-complete**.

Proof: **Hardness:** Reduction from **SOLUTION** in acyclic HTNs.
Add Ordering – $k$ changes

**Theorem**

ORDERING-K-CHANGE is **NEXPTIME-complete**.

**Proof**: Hardness: Reduction from SOLUTION in acyclic HTNs.
Add Ordering – $k$ changes

### Theorem

**ORDERING-K-CHANGE** is **NEXPTIME-complete**.

**Proof:** **Hardness:** Reduction from **SOLUTION** in acyclic HTNs.

- Any plan has length $\leq m^{|C|}$
Add Ordering – $k$ changes

Theorem

**ORDERING-K-CHANGE** is **NEXPTIME-complete**.

Proof: **Hardness:** Reduction from **SOLUTION** in acyclic HTNs.

- Any plan has length $\leq m^{|C|}$
- Choose $k = m^{|C|} + (m^{|C|})^2$
Add Ordering – any changes

Definition (ORDERING-ANY-CHANGE)
Given a planning problem $\mathcal{P}$, a solution $tn \in Sol(\mathcal{P})$, and two tasks $t_1, t_2$. Is there any solution to $\mathcal{P}$ containing $t_1$ and $t_2$ and the ordering constraint $t_1 < t_2$?
Definition (ORDERING-ANY-CHANGE)

Given a planning problem $\mathcal{P}$, a solution $tn \in Sol(\mathcal{P})$, and two tasks $t_1, t_2$. Is there any solution to $\mathcal{P}$ containing $t_1$ and $t_2$ and the ordering constraint $t_1 < t_2$?

- The solution $tn$ does not really help
Add Ordering – any changes

Theorem

ORDERING-ANY-SOLUTION is undecidable.

Proof:

\[ c_l \]
Theorem

ORDERING-ANY-SOLUTION is undecidable.

Proof:

![Task network diagram]

- The task network containing only $a$ is a solution.
- Ask whether a solution containing $t_1 < t_2$ exists.
Add Ordering – any changes

Theorem

ORDERING-ANY-SOLUTION is undecidable.

Proof:

- The task network containing only a is a solution
- Ask whether a solution containing $t_1 < t_2$ exists
Conclusion

- Adding ordering constraints or actions to HTN Plan Verification is
  - **NP-complete** if we can’t alter the plan otherwise
  - **NEXPTIME-complete** if we can perform up to \( k \) changing operations
  - **Undecidable** if we can alter the plan arbitrarily
Conclusion

- Adding ordering constraints or actions to HTN Plan Verification is
  - \textit{NP-complete} if we can’t alter the plan otherwise
  - \textit{NEXPTIME-complete} if we can perform up to \( k \) changing operations
  - \textit{Undecidable} if we can alter the plan arbitrarily

Further results can be combined to obtain the following classification

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>delete</th>
<th>exchange</th>
<th>order</th>
<th>avoid effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>no changes</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
<td>NP</td>
</tr>
<tr>
<td>( k ) changes</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
</tr>
<tr>
<td>any changes</td>
<td>\textit{un-dec}</td>
<td>\textit{un-dec}</td>
<td>\textit{un-dec}</td>
<td>\textit{un-dec}</td>
<td>\textit{un-dec}</td>
</tr>
</tbody>
</table>

Provided the first theoretical investigation of MIP requests to change plan