Lecture *Hierarchical Planning*

Chapter:
Problem Compilations for (Non-Hierarchical) Planning

Dr. Pascal Bercher

Institute of Artificial Intelligence,
Ulm University, Germany

Winter Term 2018/2019

(Compiled on: February 20, 2019)

Overview:

Which Problem Representation is the Best?

Even when being interested in solving *classical* problems, there are still various choices regarding the representation.

Which Problem Representation is the Best?

Even when being interested in solving *classical* problems, there are still various choices regarding the representation.

- Lifted model vs. ground/propositional model.

Which Problem Representation is the Best?

Even when being interested in solving *classical* problems, there are still various choices regarding the representation.

- Lifted model vs. ground/propositional model.
- Language features:
  - Negative preconditions.
  - Derived Predicates (based on *axioms*).
  - Quantifiers in preconditions or effects.
  - Conditional effects.

Which Problem Representation is the Best?

Even when being interested in solving *classical* problems, there are still various choices regarding the representation.

- Lifted model vs. ground/propositional model.
- Language features:
  - Negative preconditions.
  - Derived Predicates (based on *axioms*).
  - Quantifiers in preconditions or effects.
  - Conditional effects.

$\rightarrow$ They can all be “compiled away”!

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let $\mathit{Sol}(\mathcal{P})$ its set of solutions.

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let $Sol(\mathcal{P})$ its set of solutions.
- We then (automatically) translate $\mathcal{P}$ to a new problem $\mathcal{P}'$ which does not use (or "know") this language feature. Let $Sol(\mathcal{P}')$ be its set of solutions.

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let $Sol(\mathcal{P})$ its set of solutions.
- We then (automatically) translate $\mathcal{P}$ to a new problem $\mathcal{P}'$ which does not use (or "know") this language feature. Let $Sol(\mathcal{P}')$ be its set of solutions.

$\rightarrow$ The translation (ordinarily) has the following properties:

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let *Sol*$(\mathcal{P})$ its set of solutions.
- We then (automatically) translate $\mathcal{P}$ to a new problem $\mathcal{P}'$ which does not use (or "know") this language feature. Let *Sol*$(\mathcal{P}')$ be its set of solutions.

$\rightarrow$ The translation (ordinarily) has the following properties:

- $|Sol(\mathcal{P})| \leq |Sol(\mathcal{P}')|$ (this is not necessarily the case, but the norm)

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let $Sol(\mathcal{P})$ its set of solutions.
- We then (automatically) translate $\mathcal{P}$ to a new problem $\mathcal{P}'$ which does not use (or "know") this language feature. Let $Sol(\mathcal{P}')$ be its set of solutions.

$\rightarrow$ The translation (ordinarily) has the following properties:

  - $|Sol(\mathcal{P})| \leq |Sol(\mathcal{P}')|$ (this is not necessarily the case, but the norm)
  - Very often the compilation function (mapping $\mathcal{P}$ to $\mathcal{P}'$) does *not* run in $\mathbb{P}$.

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let $Sol(\mathcal{P})$ its set of solutions.
- We then (automatically) translate $\mathcal{P}$ to a new problem $\mathcal{P}'$ which does not use (or "know") this language feature. Let $Sol(\mathcal{P}')$ be its set of solutions.

→ The translation (ordinarily) has the following properties:
  - $|Sol(\mathcal{P})| \leq |Sol(\mathcal{P}')|$ (this is not necessarily the case, but the norm)
  - Very often the compilation function (mapping $\mathcal{P}$ to $\mathcal{P}'$) does *not* run in $\mathbb{P}$.
- We have a function $f : Sol(\mathcal{P}') \rightarrow Sol(\mathcal{P})$.

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let $Sol(\mathcal{P})$ its set of solutions.
- We then (automatically) translate $\mathcal{P}$ to a new problem $\mathcal{P}'$ which does not use (or "know") this language feature. Let $Sol(\mathcal{P}')$ be its set of solutions.

$\rightarrow$ The translation (ordinarily) has the following properties:

  - $|Sol(\mathcal{P})| \leq |Sol(\mathcal{P}')|$ (this is not necessarily the case, but the norm)
  - Very often the compilation function (mapping $\mathcal{P}$ to $\mathcal{P}'$) does *not* run in $\mathbb{P}$.

- We have a function $f : Sol(\mathcal{P}') \rightarrow Sol(\mathcal{P})$.
  - Used to obtain the original solutions from the compiled problem.

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let $Sol(\mathcal{P})$ its set of solutions.
- We then (automatically) translate $\mathcal{P}$ to a new problem $\mathcal{P}'$ which does not use (or "know") this language feature. Let $Sol(\mathcal{P}')$ be its set of solutions.

$\rightarrow$ The translation (ordinarily) has the following properties:

  - $|Sol(\mathcal{P})| \leq |Sol(\mathcal{P}')|$ (this is not necessarily the case, but the norm)
  - Very often the compilation function (mapping $\mathcal{P}$ to $\mathcal{P}'$) does *not* run in $\mathbb{P}$.

- We have a function $f : Sol(\mathcal{P}') \rightarrow Sol(\mathcal{P})$.
  - Used to obtain the original solutions from the compiled problem.
  - Surjectiveness ensures that none of the original solutions got lost.

Problem Compilations – What's That?

"Compiling away" some language feature means:

- Given a planning problem $\mathcal{P}$ with some "high-level" language feature, let $\mathit{Sol}(\mathcal{P})$ its set of solutions.
- We then (automatically) translate $\mathcal{P}$ to a new problem $\mathcal{P}'$ which does not use (or "know") this language feature. Let $\mathit{Sol}(\mathcal{P}')$ be its set of solutions.

$\rightarrow$ The translation (ordinarily) has the following properties:
  - $|\mathit{Sol}(\mathcal{P})| \leq |\mathit{Sol}(\mathcal{P}')|$ (this is not necessarily the case, but the norm)
  - Very often the compilation function (mapping $\mathcal{P}$ to $\mathcal{P}'$) does *not* run in $\mathbb{P}$.
- We have a function $f : \mathit{Sol}(\mathcal{P}') \rightarrow \mathit{Sol}(\mathcal{P})$.
  - Used to obtain the original solutions from the compiled problem.
  - Surjectiveness ensures that none of the original solutions got lost.
  - Ordinarily, the runtime of f is in $\mathbb{P}$ (but in the size of its input, which is neither $\mathcal{P}$ nor $\mathcal{P}'$).

Problem Compilations – Why??

"Compiling away" some language feature brings many benefits:

- Algorithms do not need to be extended.

Problem Compilations – Why??

"Compiling away" some language feature brings many benefits:

- Algorithms do not need to be extended. E.g., POCL planning with negative preconditions? With *variables*?

Problem Compilations – Why??

"Compiling away" some language feature brings many benefits:

- Algorithms do not need to be extended. E.g., POCL planning with negative preconditions? With *variables*?
- Heuristics do not need to be extended.

Problem Compilations – Why??

“Compiling away” some language feature brings many benefits:

- Algorithms do not need to be extended. E.g., POCL planning with negative preconditions? With *variables*?
- Heuristics do not need to be extended. E.g., rPG with negative preconditions? What does delete relaxation even mean, then?

Problem Compilations – Why??

"Compiling away" some language feature brings many benefits:

- Algorithms do not need to be extended. E.g., POCL planning with negative preconditions? With *variables*?
- Heuristics do not need to be extended. E.g., rPG with negative preconditions? What does delete relaxation even mean, then?
- Many definitions are much easier.

Problem Compilations – Why??

"Compiling away" some language feature brings many benefits:

- Algorithms do not need to be extended. E.g., POCL planning with negative preconditions? With *variables*?
- Heuristics do not need to be extended. E.g., rPG with negative preconditions? What does delete relaxation even mean, then?
- Many definitions are much easier. Just consider the examples from above...

Problem Compilations – Why??

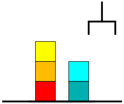
"Compiling away" some language feature brings many benefits:

- Algorithms do not need to be extended. E.g., POCL planning with negative preconditions? With *variables*?
- Heuristics do not need to be extended. E.g., rPG with negative preconditions? What does delete relaxation even mean, then?
- Many definitions are much easier. Just consider the examples from above...

However, *natively* dealing with some language feature might be more efficient (in particular if the compilation increases the problem size significantly) but may be (much) more complicated.

Motivation

Consider Blocksworld:

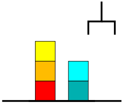


- *n* blocks, 1 gripper.
- A single action either takes a block with the gripper or puts a block we are holding onto some other block/the table.

blocks	states	blocks	states
1	1	10	58941091
2	3	11	824073141
3	13	12	12470162233
4	73	13	202976401213
5	501	14	3535017524403
6	4051	15	65573803186921
7	37633	16	1290434218669921
8	394353	17	26846616451246353
9	4596553	18	588633468315403843

Motivation

Consider Blocksworld:



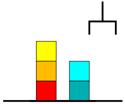
- *n* blocks, 1 gripper.
- A single action either takes a block with the gripper or puts a block we are holding onto some other block/the table.

blocks	states	blocks	states
1	1	10	58941091
2	3	11	824073141
3	13	12	12470162233
4	73	13	202976401213
5	501	14	3535017524403
6	4051	15	65573803186921
7	37633	16	1290434218669921
8	394353	17	26846616451246353
9	4596553	18	588633468315403843

$\rightarrow$ Exercise: Model the problem for 18 blocks with standard STRIPS.

Motivation

Consider Blocksworld:

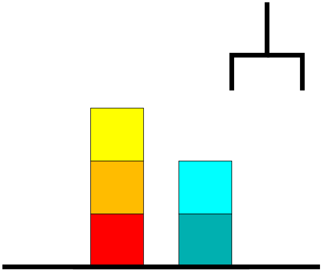


- n blocks, 1 gripper.
- A single action either takes a block with the gripper or puts a block we are holding onto some other block/the table.

blocks	states	blocks	states
1	1	10	58941091
2	3	11	824073141
3	13	12	12470162233
4	73	13	202976401213
5	501	14	3535017524403
6	4051	15	65573803186921
7	37633	16	1290434218669921
8	394353	17	26846616451246353
9	4596553	18	588633468315403843

$\rightarrow$ Exercise: Model the problem for 18 blocks with standard STRIPS. ;)

Motivation, cont'd



Predicates:

- $on(x, y)$ – x lies directly on y.
- $free(x)$ – x has no block above it.

Actions:

- $pickup(x)$ – pick up x, if it is free.
- $putdown(x, y)$ – put x on y, if y is free (the *table* is always free).

$\rightarrow$ Modeling in a lifted representation is much easier:

- Just 2 actions!
- Just 18+1 constants.

Definition of Lifted Models

- Lifted models are based upon a first-order predicate logic. We assume that you are familiar with predicate logics and only briefly recap.

Definition of Lifted Models

- Lifted models are based upon a first-order predicate logic. We assume that you are familiar with predicate logics and only briefly recap.
  - Instead of using state variables (which are equivalent to propositional variables), a set of sorted *predicates* is given, i.e., each predicate takes a sequence of parameters of a certain sort. E.g, $on_{Block,Block}(b_1, b_2)$ is a predicate with two variables b_1, b_2 both being of sort *Block*.

Definition of Lifted Models

- Lifted models are based upon a first-order predicate logic. We assume that you are familiar with predicate logics and only briefly recap.
  - Instead of using state variables (which are equivalent to propositional variables), a set of sorted *predicates* is given, i.e., each predicate takes a sequence of parameters of a certain sort. E.g, $on_{Block,Block}(b_1, b_2)$ is a predicate with two variables b_1, b_2 both being of sort *Block*.
  - There is a set of sorted *constants*. Predicates can be instantiated by constants of respective sorts.

Definition of Lifted Models

- Lifted models are based upon a first-order predicate logic. We assume that you are familiar with predicate logics and only briefly recap.
  - Instead of using state variables (which are equivalent to propositional variables), a set of sorted *predicates* is given, i.e., each predicate takes a sequence of parameters of a certain sort. E.g, $on_{Block,Block}(b_1, b_2)$ is a predicate with two variables b_1, b_2 both being of sort *Block*.
  - There is a set of sorted *constants*. Predicates can be instantiated by constants of respective sorts.
  - Actions take sorted parameters as well, which are all the variables used by all preconditions' and effects' variables.

Definition of Lifted Models

- Lifted models are based upon a first-order predicate logic. We assume that you are familiar with predicate logics and only briefly recap.
  - Instead of using state variables (which are equivalent to propositional variables), a set of sorted *predicates* is given, i.e., each predicate takes a sequence of parameters of a certain sort. E.g, $on_{Block,Block}(b_1, b_2)$ is a predicate with two variables b_1, b_2 both being of sort *Block*.
  - There is a set of sorted *constants*. Predicates can be instantiated by constants of respective sorts.
  - Actions take sorted parameters as well, which are all the variables used by all preconditions' and effects' variables.
  - Such *lifted* actions are compact representations of an up to exponential set of *instantiations* or *groundings* that are obtained via grounding.

Definition of Lifted Models

- Lifted models are based upon a first-order predicate logic. We assume that you are familiar with predicate logics and only briefly recap.
  - Instead of using state variables (which are equivalent to propositional variables), a set of sorted *predicates* is given, i.e., each predicate takes a sequence of parameters of a certain sort. E.g, $on_{Block,Block}(b_1, b_2)$ is a predicate with two variables b_1, b_2 both being of sort *Block*.
  - There is a set of sorted *constants*. Predicates can be instantiated by constants of respective sorts.
  - Actions take sorted parameters as well, which are all the variables used by all preconditions' and effects' variables.
  - Such *lifted* actions are compact representations of an up to exponential set of *instantiations* or *groundings* that are obtained via grounding.
- The standard description language for planning problems (PDDL: planning domain description language) relies upon such a lifted formalism.

Definition of Lifted Models

- Lifted models are based upon a first-order predicate logic. We assume that you are familiar with predicate logics and only briefly recap.
  - Instead of using state variables (which are equivalent to propositional variables), a set of sorted *predicates* is given, i.e., each predicate takes a sequence of parameters of a certain sort. E.g, $on_{Block,Block}(b_1, b_2)$ is a predicate with two variables b_1, b_2 both being of sort *Block*.
  - There is a set of sorted *constants*. Predicates can be instantiated by constants of respective sorts.
  - Actions take sorted parameters as well, which are all the variables used by all preconditions' and effects' variables.
  - Such *lifted* actions are compact representations of an up to exponential set of *instantiations* or *groundings* that are obtained via grounding.
- The standard description language for planning problems (PDDL: planning domain description language) relies upon such a lifted formalism.
- For example, consider the *Blocksworld domain* from the IPC 2000: https://github.com/potassco/pddl-instances/tree/master/ipc-2000/domains/blocks-strips-typed (see live demo)

Compilation

- While models get *written* in a lifted fashion, many (in fact, most) planning systems rely on a standard STRIPS model.

Compilation

- While models get *written* in a lifted fashion, many (in fact, most) planning systems rely on a standard STRIPS model.
- For translation, the model gets *grounded* via instantiating all variables by all constants of the correct sort resulting into an exponentially larger model.

Compilation

- While models get *written* in a lifted fashion, many (in fact, most) planning systems rely on a standard STRIPS model.
- For translation, the model gets *grounded* via instantiating all variables by all constants of the correct sort resulting into an exponentially larger model.
- Rather than computing *all possible* groundings a more reasonable approach (resulting into much smaller models) is to exploit the (relaxed) planning graph:

Compilation

- While models get *written* in a lifted fashion, many (in fact, most) planning systems rely on a standard STRIPS model.
- For translation, the model gets *grounded* via instantiating all variables by all constants of the correct sort resulting into an exponentially larger model.
- Rather than computing *all possible* groundings a more reasonable approach (resulting into much smaller models) is to exploit the (relaxed) planning graph:
  - Start with the initial state and create all groundings necessary to build the first fact layer following the initial state.

Compilation

- While models get *written* in a lifted fashion, many (in fact, most) planning systems rely on a standard STRIPS model.
- For translation, the model gets *grounded* via instantiating all variables by all constants of the correct sort resulting into an exponentially larger model.
- Rather than computing *all possible* groundings a more reasonable approach (resulting into much smaller models) is to exploit the (relaxed) planning graph:
  - Start with the initial state and create all groundings necessary to build the first fact layer following the initial state.
  - Then continue until a fixed point is reached. The actions in the final fact layer are all groundings required.

Compilation

- While models get *written* in a lifted fashion, many (in fact, most) planning systems rely on a standard STRIPS model.
- For translation, the model gets *grounded* via instantiating all variables by all constants of the correct sort resulting into an exponentially larger model.
- Rather than computing *all possible* groundings a more reasonable approach (resulting into much smaller models) is to exploit the (relaxed) planning graph:
  - Start with the initial state and create all groundings necessary to build the first fact layer following the initial state.
  - Then continue until a fixed point is reached. The actions in the final fact layer are all groundings required.
  - → Using the PG instead of the rPG results into less actions, but may be too expensive empirically.

Motivation

- The standard STRIPS representation relies on *positive preconditions*.

Motivation

- The standard STRIPS representation relies on *positive preconditions*.
- This makes *everything* easier!

Motivation

- The standard STRIPS representation relies on *positive preconditions*.
- This makes *everything* easier!
  - Algorithms, heuristics.

Motivation

- The standard STRIPS representation relies on *positive preconditions*.
- This makes *everything* easier!
  - Algorithms, heuristics.
  - Complexity Analysis, proofs.

Motivation

- The standard STRIPS representation relies on *positive preconditions*.
- This makes *everything* easier!
  - Algorithms, heuristics.
  - Complexity Analysis, proofs.
- Most heuristics rely on positive preconditions. What means delete relaxation if we have positive preconditions?

Problem Definition (given Negative Effects)

Blackboard/Whiteboard.
(See also exercise sheet.)

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem in the standard STRIPS formalism, i.e., with only positive preconditions. Then, deciding whether $\mathcal{P}^+$ has a solution can be decided in $\mathbb{P}$.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem in the standard STRIPS formalism, i.e., with only positive preconditions. Then, deciding whether $\mathcal{P}^+$ has a solution can be decided in $\mathbb{P}$.

Proof:

- Execute the following algorithm exploiting that no action needs to applied more than once:

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem in the standard STRIPS formalism, i.e., with only positive preconditions. Then, deciding whether $\mathcal{P}^+$ has a solution can be decided in $\mathbb{P}$.

Proof:

- Execute the following algorithm exploiting that no action needs to applied more than once:

(1) Try to apply actions to the initial state that were not yet applied.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem in the standard STRIPS formalism, i.e., with only positive preconditions. Then, deciding whether $\mathcal{P}^+$ has a solution can be decided in $\mathbb{P}$.

Proof:

- Execute the following algorithm exploiting that no action needs to applied more than once:

(1) Try to apply actions to the initial state that were not yet applied.

(2) As long as at least one action was applied, repeat. Also stop if the goal is generated.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Membership Proof:

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Membership Proof:

- Each action needs to be applied at most once.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Membership Proof:

- Each action needs to be applied at most once.
- Thus, the maximum *required* plan length (to achieve any goal description) is bounded by $b \leq |A|$.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Membership Proof:

- Each action needs to be applied at most once.
- Thus, the maximum *required* plan length (to achieve any goal description) is bounded by $b \leq |A|$.
- Now, guess a sequence of b actions and verify in linear time whether it's applicable.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Membership Proof:

- Each action needs to be applied at most once.
- Thus, the maximum *required* plan length (to achieve any goal description) is bounded by $b \leq |A|$.
- Now, guess a sequence of b actions and verify in linear time whether it's applicable.
- Return true or false (depending on whether all goals hold in the final state and the guessed plan is executable).

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Hardness Proof:

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Hardness Proof: Reduction from CNF-SAT:

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Hardness Proof: Reduction from CNF-SAT:

- Let $\varphi = \underbrace{\{C_1, \dots, C_m\}}_{\text{clauses}}$, $C_j = \underbrace{\{\varphi_{j_1}, \dots, \varphi_{j_k}\}}_{\text{literals}}$, and $V = \underbrace{\{x_1, \dots, x_n\}}_{\text{variables}}$.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Hardness Proof: Reduction from CNF-SAT:

- Let $\varphi = \underbrace{\{C_1, \ldots, C_m\}}_{\text{clauses}}$, $C_j = \underbrace{\{\varphi_{j_1}, \ldots, \varphi_{j_k}\}}_{\text{literals}}$, and $V = \underbrace{\{x_1, \ldots, x_n\}}_{\text{variables}}$.
- For each boolean variable $x_i \in V$ add two actions to A:

$$\neg x_i{-}\bot \;\; \boxed{x_i \mapsto \top} \;\; x_i{-}\top \quad \text{and} \quad \neg x_i{-}\top \;\; \boxed{x_i \mapsto \bot} \;\; x_i{-}\bot$$

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Hardness Proof: Reduction from CNF-SAT:

- Let $\varphi = \underbrace{\{C_1, \ldots, C_m\}}_{\text{clauses}}$, $C_j = \underbrace{\{\varphi_{j_1}, \ldots, \varphi_{j_k}\}}_{\text{literals}}$, and $V = \underbrace{\{x_1, \ldots, x_n\}}_{\text{variables}}$.
- For each boolean variable $x_i \in V$ add two actions to A:

$\neg x_i - \bot$ $\boxed{x_i \mapsto \top}$ $x_i - \top$ and $\neg x_i - \top$ $\boxed{x_i \mapsto \bot}$ $x_i - \bot$

- For each *positive* $\varphi_{j_i} = x_{j_i}$ or *negative* $\varphi_{j_i} = \neg x_{j_i}$ add

$x_{j_i} - \top$ $\boxed{\begin{array}{c}\text{“}x_{j_i} = \top\text{”} \\ C_j \mapsto \top\end{array}}$ $C_j - \top$ or $x_{j_i} - \bot$ $\boxed{\begin{array}{c}\text{“}x_{j_i} = \bot\text{”} \\ C_j \mapsto \top\end{array}}$ $C_j - \top$

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Hardness Proof: Reduction from CNF-SAT:

- Let $\varphi = \underbrace{\{C_1, \ldots, C_m\}}_{\text{clauses}}$, $C_j = \underbrace{\{\varphi_{j_1}, \ldots, \varphi_{j_k}\}}_{\text{literals}}$, and $V = \underbrace{\{x_1, \ldots, x_n\}}_{\text{variables}}$.
- For each boolean variable $x_i \in V$ add two actions to A:

  $\neg x_i - \bot \; \boxed{x_i \mapsto \top} \; x_i - \top$ and $\neg x_i - \top \; \boxed{x_i \mapsto \bot} \; x_i - \bot$

- For each *positive* $\varphi_{j_i} = x_{j_i}$ or *negative* $\varphi_{j_i} = \neg x_{j_i}$ add

  $x_{j_i} - \top \; \boxed{\begin{array}{c}\text{“}x_{j_i} = \top\text{”} \\ C_j \mapsto \top\end{array}} \; C_j - \top$ or $x_{j_i} - \bot \; \boxed{\begin{array}{c}\text{“}x_{j_i} = \bot\text{”} \\ C_j \mapsto \top\end{array}} \; C_j - \top$

- $g = \{C_j - \top \mid 1 \leq j \leq m\}$.

Delete-free Planning Problems, Complexities

Theorem

Let $\mathcal{P}$ be a planning problem with negative preconditions (in addition to the positive ones). Then, deciding whether $\mathcal{P}^+$ has a solution is $\mathbb{NP}$-complete.

Hardness Proof: Reduction from CNF-SAT:

- Let $\varphi = \underbrace{\{C_1, \dots, C_m\}}_{\text{clauses}}$, $C_j = \underbrace{\{\varphi_{j_1}, \dots, \varphi_{j_k}\}}_{\text{literals}}$, and $V = \underbrace{\{x_1, \dots, x_n\}}_{\text{variables}}$.
- For each boolean variable $x_i \in V$ add two actions to A:

  $\neg x_i - \bot$ $\boxed{x_i \mapsto \top}$ $x_i - \top$ and $\neg x_i - \top$ $\boxed{x_i \mapsto \bot}$ $x_i - \bot$

- For each *positive* $\varphi_{j_i} = x_{j_i}$ or *negative* $\varphi_{j_i} = \neg x_{j_i}$ add

  $x_{j_i} - \top$ $\boxed{\begin{array}{c}\text{“}x_{j_i} = \top\text{”} \\ C_j \mapsto \top\end{array}}$ $C_j - \top$ or $x_{j_i} - \bot$ $\boxed{\begin{array}{c}\text{“}x_{j_i} = \bot\text{”} \\ C_j \mapsto \top\end{array}}$ $C_j - \top$

- $g = \{C_j - \top \mid 1 \leq j \leq m\}$. φ is satisfiable if and only if a plan exists.

Delete-free Planning Problems, Complexities, cont'd

Recap:

- Deciding whether $\mathcal{P}$ (with or without negative preconditions) has a solution is $\mathbb{PSPACE}$-complete (shown later).

Delete-free Planning Problems, Complexities, cont'd

Recap:

- Deciding whether $\mathcal{P}$ (with or without negative preconditions) has a solution is $\mathbb{PSPACE}$-complete (shown later).
- Deciding whether $\mathcal{P}^+$ (without negative preconditions) has a solution is in $\mathbb{P}$.

Delete-free Planning Problems, Complexities, cont'd

Recap:

- Deciding whether $\mathcal{P}$ (with or without negative preconditions) has a solution is $\mathbb{PSPACE}$-complete (shown later).
- Deciding whether $\mathcal{P}^+$ (without negative preconditions) has a solution is in $\mathbb{P}$.
- Deciding whether $\mathcal{P}^+$ (with negative preconditions) has a solution is $\mathbb{NP}$-complete.

Delete-free Planning Problems, Complexities, cont'd

Recap:

- Deciding whether $\mathcal{P}$ (with or without negative preconditions) has a solution is $\mathbb{PSPACE}$-complete (shown later).
- Deciding whether $\mathcal{P}^+$ (without negative preconditions) has a solution is in $\mathbb{P}$.
- Deciding whether $\mathcal{P}^+$ (with negative preconditions) has a solution is $\mathbb{NP}$-complete.

Question:
Since $\mathbb{NP} \subseteq \mathbb{PSPACE}$ (and presumably $\mathbb{NP} \subsetneq \mathbb{PSPACE}$), using $\mathcal{P}^+$ with negative preconditions as basis for heuristics is a (n expensive, but) useful relaxation heuristic, right?

Delete-free Planning Problems, Complexities, cont'd

Recap:

- Deciding whether $\mathcal{P}$ (with or without negative preconditions) has a solution is $\mathbb{PSPACE}$-complete (shown later).
- Deciding whether $\mathcal{P}^+$ (without negative preconditions) has a solution is in $\mathbb{P}$.
- Deciding whether $\mathcal{P}^+$ (with negative preconditions) has a solution is $\mathbb{NP}$-complete.

Question:
Since $\mathbb{NP} \subseteq \mathbb{PSPACE}$ (and presumably $\mathbb{NP} \subsetneq \mathbb{PSPACE}$), using $\mathcal{P}^+$ with negative preconditions as basis for heuristics is a (n expensive, but) useful relaxation heuristic, right?

No! Although deciding it is easier, it's not a relaxation. *Why?*

Blackboard/Whiteboard.

How to Combine Negative Preconditions and Delete-Relaxation?

If we can't use delete-relaxation for heuristics in case we have negative preconditions, what should we do?

How to Combine Negative Preconditions and Delete-Relaxation?

If we can't use delete-relaxation for heuristics in case we have negative preconditions, what should we do?

$\rightarrow$ Compile them away!

Compilation Technique (easy)

Blackboard/Whiteboard (see also exercise sheet).

- Easy to understand and to implement.
- Number of additional variables is often unnecessarily high.

Compilation Technique (fancy)

Blackboard/Whiteboard (see also exercise sheet).

- Much more complicated to understand and to implement.
- Number of additional variables is minimal.

Motivation

- Some actions can be modeled more canonically if their effects depend on the current state $\rightarrow$ their effects are *conditional* – given the current state.

Motivation

- Some actions can be modeled more canonically if their effects depend on the current state $\rightarrow$ their effects are *conditional* – given the current state.
- We already know one example from the lecture:

Motivation

- Some actions can be modeled more canonically if their effects depend on the current state $\rightarrow$ their effects are *conditional* – given the current state.
- We already know one example from the lecture: the *move* action from the *Cranes in the Harbor* domain. We had to model it with two distinct actions: *moveLeft* and *moveRight*.

Motivation

- Some actions can be modeled more canonically if their effects depend on the current state $\rightarrow$ their effects are *conditional* – given the current state.
- We already know one example from the lecture: the *move* action from the *Cranes in the Harbor* domain. We had to model it with two distinct actions: *moveLeft* and *moveRight*.
- There are many such examples:

Motivation

- Some actions can be modeled more canonically if their effects depend on the current state $\rightarrow$ their effects are *conditional* – given the current state.
- We already know one example from the lecture: the *move* action from the *Cranes in the Harbor* domain. We had to model it with two distinct actions: *moveLeft* and *moveRight*.
- There are many such examples:
  - Use a light switch (rather than turn on/turn off).

Motivation

- Some actions can be modeled more canonically if their effects depend on the current state $\rightarrow$ their effects are *conditional* – given the current state.
- We already know one example from the lecture: the *move* action from the *Cranes in the Harbor* domain. We had to model it with two distinct actions: *moveLeft* and *moveRight*.
- There are many such examples:
  - Use a light switch (rather than turn on/turn off).
  - . . . (You can also check the IPC.)

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.
- Formally, an action $a \in A$ with conditional effects is given by:

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.
- Formally, an action $a \in A$ with conditional effects is given by:
  - $a = (\mathit{pre}, \mathit{effs}, c)$ with $\mathit{pre} \subseteq V$, $c \in \mathbb{R}_0$ and each $\mathit{eff} \in \mathit{effs}$ is a tuple $(\mathit{prec}, \mathit{add}, \mathit{del}) \in 2^V \times 2^V \times 2^V$.

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.
- Formally, an action $a \in A$ with conditional effects is given by:
  - $a = (pre, effs, c)$ with $pre \subseteq V$, $c \in \mathbb{R}_0$ and each $eff \in effs$ is a tuple $(prec, add, del) \in 2^V \times 2^V \times 2^V$.
  - Note: We also display conditional effects as $prec \rightarrow (add, del)$.

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.
- Formally, an action $a \in A$ with conditional effects is given by:
  - $a = (pre, effs, c)$ with $pre \subseteq V$, $c \in \mathbb{R}_0$ and each $eff \in effs$ is a tuple $(prec, add, del) \in 2^V \times 2^V \times 2^V$.
  - Note: We also display conditional effects as $prec \rightarrow (add, del)$.
  - a is applicable in $s \in S$, $\tau(a, s) = \top$, if and only if $pre \subseteq s$.

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.
- Formally, an action $a \in A$ with conditional effects is given by:
  - $a = (pre, effs, c)$ with $pre \subseteq V$, $c \in \mathbb{R}_0$ and each $eff \in effs$ is a tuple $(prec, add, del) \in 2^V \times 2^V \times 2^V$.
  - Note: We also display conditional effects as $prec \rightarrow (add, del)$.
  - a is applicable in $s \in S$, $\tau(a, s) = \top$, if and only if $pre \subseteq s$.
  - If $\tau(a, s) = \top$, then the application of a in s results into the following successor state $\gamma(a, s) = s'$:

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.
- Formally, an action $a \in A$ with conditional effects is given by:
  - $a = (pre, effs, c)$ with $pre \subseteq V$, $c \in \mathbb{R}_0$ and each $eff \in effs$ is a tuple $(prec, add, del) \in 2^V \times 2^V \times 2^V$.
  - Note: We also display conditional effects as $prec \rightarrow (add, del)$.
  - a is applicable in $s \in S$, $\tau(a, s) = \top$, if and only if $pre \subseteq s$.
  - If $\tau(a, s) = \top$, then the application of a in s results into the following successor state $\gamma(a, s) = s'$:
    - Let $add' = \bigcup_{(prec, add, del) \in effs, prec \subseteq s} add$.

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.
- Formally, an action $a \in A$ with conditional effects is given by:
  - $a = (pre, effs, c)$ with $pre \subseteq V$, $c \in \mathbb{R}_0$ and each $eff \in effs$ is a tuple $(prec, add, del) \in 2^V \times 2^V \times 2^V$.
  - Note: We also display conditional effects as $prec \to (add, del)$.
  - a is applicable in $s \in S$, $\tau(a, s) = \top$, if and only if $pre \subseteq s$.
  - If $\tau(a, s) = \top$, then the application of a in s results into the following successor state $\gamma(a, s) = s'$:
    - Let $add' = \bigcup_{(prec, add, del) \in effs, prec \subseteq s} add$.
    - Let $del' = \bigcup_{(prec, add, del) \in effs, prec \subseteq s} del$.

Problem Definition

- When actions have conditional effects, they have a set of two-tuples rather than an add and delete list. Each head of the tuple is a precondition set (analogous to STRIPS with negative preconditions, we can use negative preconditions here as well) and the body consists of an add and delete list.
- Formally, an action $a \in A$ with conditional effects is given by:
  - $a = (pre, effs, c)$ with $pre \subseteq V$, $c \in \mathbb{R}_0$ and each $eff \in effs$ is a tuple $(prec, add, del) \in 2^V \times 2^V \times 2^V$.
  - Note: We also display conditional effects as $prec \rightarrow (add, del)$.
  - a is applicable in $s \in S$, $\tau(a, s) = \top$, if and only if $pre \subseteq s$.
  - If $\tau(a, s) = \top$, then the application of a in s results into the following successor state $\gamma(a, s) = s'$:
    - Let $add' = \bigcup_{(prec, add, del) \in effs, prec \subseteq s} add$.
    - Let $del' = \bigcup_{(prec, add, del) \in effs, prec \subseteq s} del$.
    - Then, $s' = (s \setminus del') \cup add'$.

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$
  - $c \wedge a \wedge \neg a \wedge \neg b$

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$
  - $c \wedge a \wedge \neg a \wedge \neg b$
  - $c \wedge a \wedge \neg\neg a \wedge b$

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$
  - $c \wedge a \wedge \neg a \wedge \neg b$
  - $c \wedge a \wedge \neg\neg a \wedge b$
  - $c \wedge a \wedge \neg\neg a \wedge \neg b$

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$
  - $c \wedge a \wedge \neg a \wedge \neg b$
  - $c \wedge a \wedge \neg\neg a \wedge b$
  - $c \wedge a \wedge \neg\neg a \wedge \neg b$
  - $c \wedge \neg a \wedge \neg a \wedge b$

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$
  - $c \wedge a \wedge \neg a \wedge \neg b$
  - $c \wedge a \wedge \neg\neg a \wedge b$
  - $c \wedge a \wedge \neg\neg a \wedge \neg b$
  - $c \wedge \neg a \wedge \neg a \wedge b$
  - $c \wedge \neg a \wedge \neg a \wedge \neg b$

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$
  - $c \wedge a \wedge \neg a \wedge \neg b$
  - $c \wedge a \wedge \neg\neg a \wedge b$
  - $c \wedge a \wedge \neg\neg a \wedge \neg b$
  - $c \wedge \neg a \wedge \neg a \wedge b$
  - $c \wedge \neg a \wedge \neg a \wedge \neg b$
  - $c \wedge \neg a \wedge \neg\neg a \wedge b$

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
  Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$
  - $c \wedge a \wedge \neg a \wedge \neg b$
  - $c \wedge a \wedge \neg\neg a \wedge b$
  - $c \wedge a \wedge \neg\neg a \wedge \neg b$
  - $c \wedge \neg a \wedge \neg a \wedge b$
  - $c \wedge \neg a \wedge \neg a \wedge \neg b$
  - $c \wedge \neg a \wedge \neg\neg a \wedge b$
  - $c \wedge \neg a \wedge \neg\neg a \wedge \neg b$

Compilation Technique (easy)

- We can simply compute all possible combinations of the conditional effects' preconditions and create a new action for each of them.
- In the following example, we also use negative preconditions:
Let $a = (\{c\}, \{a \rightarrow b, \neg a \rightarrow c, b \rightarrow \neg c\})$. New preconditions:
  - $c \wedge a \wedge \neg a \wedge b$
  - $c \wedge a \wedge \neg a \wedge \neg b$
  - $c \wedge a \wedge \neg\neg a \wedge b$
  - $c \wedge a \wedge \neg\neg a \wedge \neg b$
  - $c \wedge \neg a \wedge \neg a \wedge b$
  - $c \wedge \neg a \wedge \neg a \wedge \neg b$
  - $c \wedge \neg a \wedge \neg\neg a \wedge b$
  - $c \wedge \neg a \wedge \neg\neg a \wedge \neg b$

  $\rightarrow$ Note that $\neg\neg\varphi = \varphi$ and that mutex relations can be exploited, i.e., actions with a precondition $\neg a \wedge a$ can be ignored.

Compilation Technique (fancy)

- The previous compilation induces an exponential blowup of the model.

Compilation Technique (fancy)

- The previous compilation induces an exponential blowup of the model.
- Instead, we can include additional state variables that prevent "standard actions" to be executed. Then, "synchonization actions" become applicable that produce the correct successor state and make standard actions applicable again.

Compilation Technique (fancy)

- The previous compilation induces an exponential blowup of the model.
- Instead, we can include additional state variables that prevent "standard actions" to be executed. Then, "synchonization actions" become applicable that produce the correct successor state and make standard actions applicable again.
- This technique only requires a linear space increase of the model, but required more effort for the planner.

Formal Definition

- Preconditions can involve disjunctions. You can assume that preconditions are given in conjunctive normal form.

Compilation

- Create one single action instance for every evaluation of the action's precondition evaluating to true.

Compilation

- Create one single action instance for every evaluation of the action's precondition evaluating to true.
- Example: The precondition $(a \vee b) \wedge c \wedge (\neg a \vee d)$ of an action A gets translated into the following actions:

Compilation

- Create one single action instance for every evaluation of the action's precondition evaluating to true.
- Example: The precondition $(a \lor b) \land c \land (\neg a \lor d)$ of an action A gets translated into the following actions:

  A_1 $a \land c \land \neg a$

Compilation

- Create one single action instance for every evaluation of the action's precondition evaluating to true.
- Example: The precondition $(a \vee b) \wedge c \wedge (\neg a \vee d)$ of an action A gets translated into the following actions:

  A_1 $a \wedge c \wedge \neg a$

  A_2 $a \wedge c \wedge \neg d$

Compilation

- Create one single action instance for every evaluation of the action's precondition evaluating to true.
- Example: The precondition $(a \lor b) \land c \land (\neg a \lor d)$ of an action A gets translated into the following actions:

  A_1 $a \land c \land \neg a$

  A_2 $a \land c \land \neg d$

  A_3 $b \land c \land \neg a$

Compilation

- Create one single action instance for every evaluation of the action's precondition evaluating to true.
- Example: The precondition $(a \vee b) \wedge c \wedge (\neg a \vee d)$ of an action A gets translated into the following actions:

  A_1 $a \wedge c \wedge \neg a$
  A_2 $a \wedge c \wedge \neg d$
  A_3 $b \wedge c \wedge \neg a$
  A_4 $b \wedge c \wedge \neg d$

Compilation

- Create one single action instance for every evaluation of the action's precondition evaluating to true.
- Example: The precondition $(a \vee b) \wedge c \wedge (\neg a \vee d)$ of an action A gets translated into the following actions:

  A_1 $a \wedge c \wedge \neg a$

  A_2 $a \wedge c \wedge \neg d$

  A_3 $b \wedge c \wedge \neg a$

  A_4 $b \wedge c \wedge \neg d$

  $\rightarrow$ Actions with preconditions that are mutex to each other (such as A_1: $a \wedge \neg a$) can be ignored.

Motivation

Some actions depend on specific (some/all) objects, e.g.

- Paint all blocks (all quantifier in effects).

Motivation

Some actions depend on specific (some/all) objects, e.g.

- Paint all blocks (all quantifier in effects).
- Do something if all blocks lie on the table (all quantifier in precondition).

Motivation

Some actions depend on specific (some/all) objects, e.g.

- Paint all blocks (all quantifier in effects).
- Do something if all blocks lie on the table (all quantifier in precondition).
- Take a block that has no block above it (negative existence quantified precondition)

Motivation

Some actions depend on specific (some/all) objects, e.g.

- Paint all blocks (all quantifier in effects).
- Do something if all blocks lie on the table (all quantifier in precondition).
- Take a block that has no block above it (negative existence quantified precondition)
- Paint all blocks that don't have another block above them (all-quantified conditional effect).

Motivation

Some actions depend on specific (some/all) objects, e.g.

- Paint all blocks (all quantifier in effects).
- Do something if all blocks lie on the table (all quantifier in precondition).
- Take a block that has no block above it (negative existence quantified precondition)
- Paint all blocks that don't have another block above them (all-quantified conditional effect).
  More realistic example:
  If a truck moves from one location to another, all objects it has loaded change their location as well.

Formal Definition

- Simple: Preconditions can be arbitrary first-order formulae.

Formal Definition

- Simple: Preconditions can be arbitrary first-order formulae.
- Please note that a *universal* quantifier ($\forall$) refers to all existing objects (of the respective sort), whereas *exists* quantifiers ($\exists$) are redundant, since free variables are implicitly existence quantified (both propositions only hold if there is no negation in front of the quantifier).

Formal Definition

- Simple: Preconditions can be arbitrary first-order formulae.
- Please note that a *universal* quantifier ($\forall$) refers to all existing objects (of the respective sort), whereas *exists* quantifiers ($\exists$) are redundant, since free variables are implicitly existence quantified (both propositions only hold if there is no negation in front of the quantifier).
- In principle, also effects can make use of quantifiers, but they cannot be applied in *arbitrary* formulae, since disjunctions violate the standard action semantics. (Disjunctions correspond to non-deterministic effects.)

Formal Definition

- Simple: Preconditions can be arbitrary first-order formulae.
- Please note that a *universal* quantifier ($\forall$) refers to all existing objects (of the respective sort), whereas *exists* quantifiers ($\exists$) are redundant, since free variables are implicitly existence quantified (both propositions only hold if there is no negation in front of the quantifier).
- In principle, also effects can make use of quantifiers, but they cannot be applied in *arbitrary* formulae, since disjunctions violate the standard action semantics. (Disjunctions correspond to non-deterministic effects.)
- For more standardized restrictions on how to use quantifiers, you can investigate PDDL.

Compilation

- First, bring the formulae in prenex normal form (quantifiers are in front of the formula, negations are only in front of literals).

Compilation

- First, bring the formulae in prenex normal form (quantifiers are in front of the formula, negations are only in front of literals).
- All quantifiers can be eliminated by duplicating the respective literal and substituting the quantified variable by the respective objects. For example, given there are three blocks in a blocksworld problem instance, A, B, C, then $\forall b \, \neg on(b_1, b_2)$ (encoding that block b_2 is the top-most block of its stack) gets replaced by $\neg on(A, b_2) \wedge \neg on(B, b_2) \wedge \neg on(C, b_2)$.

Compilation

- First, bring the formulae in prenex normal form (quantifiers are in front of the formula, negations are only in front of literals).
- All quantifiers can be eliminated by duplicating the respective literal and substituting the quantified variable by the respective objects. For example, given there are three blocks in a blocksworld problem instance, A, B, C, then $\forall b \neg on(b_1, b_2)$ (encoding that block b_2 is the top-most block of its stack) gets replaced by $\neg on(A, b_2) \wedge \neg on(B, b_2) \wedge \neg on(C, b_2)$.
- Existence quantifiers can simply be ignored, but the variable might have to be renamed if it's already occurring as a free variable or as another existence quantified variable.

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.
  - Theoretical studies, like the hardness for the plan existence problem.

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.
  - Theoretical studies, like the hardness for the plan existence problem.
  - Description of algorithms, heuristics, and other techniques (cf. first lecture).

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.
  - Theoretical studies, like the hardness for the plan existence problem.
  - Description of algorithms, heuristics, and other techniques (cf. first lecture).
- For *practical purposes*, STRIPS is *never* used. Instead, we have:

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.
  - Theoretical studies, like the hardness for the plan existence problem.
  - Description of algorithms, heuristics, and other techniques (cf. first lecture).
- For *practical purposes*, STRIPS is *never* used. Instead, we have:
  - A *lifted* model, which is based on a first-order predicate logic rather than on propositional logic.

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.
  - Theoretical studies, like the hardness for the plan existence problem.
  - Description of algorithms, heuristics, and other techniques (cf. first lecture).
- For *practical purposes*, STRIPS is *never* used. Instead, we have:
  - A *lifted* model, which is based on a first-order predicate logic rather than on propositional logic.
  - One such standard is PDDL (the planning domain description language). It differentiates between a planning *domain* and a planning *problem* (or planning *instance*).

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.
  - Theoretical studies, like the hardness for the plan existence problem.
  - Description of algorithms, heuristics, and other techniques (cf. first lecture).
- For *practical purposes*, STRIPS is *never* used. Instead, we have:
  - A *lifted* model, which is based on a first-order predicate logic rather than on propositional logic.
  - One such standard is PDDL (the planning domain description language). It differentiates between a planning *domain* and a planning *problem* (or planning *instance*).
  - Many extensions to the most basic language level exist, such as negative preconditions, disjunctive preconditions, quantifiers, and conditional effects.

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.
  - Theoretical studies, like the hardness for the plan existence problem.
  - Description of algorithms, heuristics, and other techniques (cf. first lecture).
- For *practical purposes*, STRIPS is *never* used. Instead, we have:
  - A *lifted* model, which is based on a first-order predicate logic rather than on propositional logic.
  - One such standard is PDDL (the planning domain description language). It differentiates between a planning *domain* and a planning *problem* (or planning *instance*).
  - Many extensions to the most basic language level exist, such as negative preconditions, disjunctive preconditions, quantifiers, and conditional effects.

→ One way of dealing with additional features is compilation.

Summary

- The STRIPS formalism is primarily used for *theoretical purposes*.
  - Theoretical studies, like the hardness for the plan existence problem.
  - Description of algorithms, heuristics, and other techniques (cf. first lecture).
- For *practical purposes*, STRIPS is *never* used. Instead, we have:
  - A *lifted* model, which is based on a first-order predicate logic rather than on propositional logic.
  - One such standard is PDDL (the planning domain description language). It differentiates between a planning *domain* and a planning *problem* (or planning *instance*).
  - Many extensions to the most basic language level exist, such as negative preconditions, disjunctive preconditions, quantifiers, and conditional effects.

$\rightarrow$ One way of dealing with additional features is compilation.

$\rightarrow$ One can also deal with them natively. This has the potential to be much more efficient for the respective algorithms, but all techniques (algorithm, heuristic, pruning techniques, reachability analysis, etc.) may have to be adapted by hand.