Principles of AI Planning

Planning by heuristic search

Malte Helmert Bernhard Nebel

Albert-Ludwigs-Universität Freiburg

November 15th, 2006

Plan search with heuristic search algorithms

- For forward and backward search (progression, regression) the search space consists of incomplete plans that are respectively prefixes of possible plans and suffixes of possible plans.
- Search starts from the empty plan.
- The neighbors/children of an incomplete plan in the search space are those that are obtained by
  1. adding an operator to the incomplete plan, or
  2. removing an operator from the incomplete plan.
- Systematic search algorithms (like A*) keep track of the incomplete plans generated so far, and therefore can go back to them.

  Hence removing operators from incomplete plans is only needed for local search algorithms which do not keep track of the history of the search process.

Plan search with heuristic search algorithms

- For forward and backward search (progression, regression) the search space consists of incomplete plans that are respectively prefixes of possible plans and suffixes of possible plans.
- Search starts from the empty plan.
- The neighbors/children of an incomplete plan in the search space are those that are obtained by
  1. adding an operator to the incomplete plan, or
  2. removing an operator from the incomplete plan.
- Systematic search algorithms (like A*) keep track of the incomplete plans generated so far, and therefore can go back to them.
  Hence removing operators from incomplete plans is only needed for local search algorithms which do not keep track of the history of the search process.

Plan search with heuristic search algorithms

- For forward and backward search (progression, regression) the search space consists of incomplete plans that are respectively prefixes of possible plans and suffixes of possible plans.
- Search starts from the empty plan.
- The neighbors/children of an incomplete plan in the search space are those that are obtained by
  1. adding an operator to the incomplete plan, or
  2. removing an operator from the incomplete plan.
- Systematic search algorithms (like A*) keep track of the incomplete plans generated so far, and therefore can go back to them.

  Hence removing operators from incomplete plans is only needed for local search algorithms which do not keep track of the history of the search process.

Plan search: incomplete plans for progression

For progression, the incomplete plans are prefixes $o_1, o_2, \ldots, o_n$ of potential plans.

An incomplete plan is extended by

1. adding an operator after the last operator, from $o_1, \ldots, o_n$ to $o_1, o_2, \ldots, o_n, o$ for some $o \in O$, or
2. removing one or more of the last operators, from $o_1, \ldots, o_n$ to $o_1, \ldots, o_i$ for some $i < n$. This is for local search algorithms only.

$o_1, o_2, \ldots, o_n$ is a plan if $app_{o_n}(app_{o_{n-1}}(\cdots app_{o_1}(I) \cdots)) \models G$.

Plan search: incomplete plans for progression

For progression, the incomplete plans are prefixes $o_1, o_2, \ldots, o_n$ of potential plans.

An incomplete plan is extended by

1. adding an operator after the last operator, from $o_1, \ldots, o_n$ to $o_1, o_2, \ldots, o_n, o$ for some $o \in O$, or
2. removing one or more of the last operators, from $o_1, \ldots, o_n$ to $o_1, \ldots, o_i$ for some $i < n$. This is for local search algorithms only.

$o_1, o_2, \ldots, o_n$ is a plan if $app_{o_n}(app_{o_{n-1}}(\cdots app_{o_1}(I)\cdots)) \models G$.

Plan search: incomplete plans for progression

For progression, the incomplete plans are prefixes $o_1, o_2, \ldots, o_n$ of potential plans.

An incomplete plan is extended by

1. adding an operator after the last operator,
   from $o_1, \ldots, o_n$ to $o_1, o_2, \ldots, o_n, o$ for some $o \in O$, or
2. removing one or more of the last operators,
   from $o_1, \ldots, o_n$ to $o_1, \ldots, o_i$ for some $i < n$.
   This is for local search algorithms only.

$o_1, o_2, \ldots, o_n$ is a plan if $app_{o_n}(app_{o_{n-1}}(\cdots app_{o_1}(I) \cdots)) \models G$.

Plan search: incomplete plans for progression

For progression, the incomplete plans are prefixes $o_1, o_2, \ldots, o_n$ of potential plans.

An incomplete plan is extended by

1. adding an operator after the last operator, from $o_1, \ldots, o_n$ to $o_1, o_2, \ldots, o_n, o$ for some $o \in O$, or
2. removing one or more of the last operators, from $o_1, \ldots, o_n$ to $o_1, \ldots, o_i$ for some $i < n$. This is for local search algorithms only.

$o_1, o_2, \ldots, o_n$ is a plan if $app_{o_n}(app_{o_{n-1}}(\cdots app_{o_1}(I)\cdots)) \models G$.

Plan search: incomplete plans for regression

For regression, the incomplete plans are suffixes $o_n, \ldots, o_1$ of potential plans.

An incomplete plan is extended by

1. adding an operator in front of the first operator, from $o_n, \ldots, o_1$ to $o, o_n, \ldots, o_1$ for $o \in O$, or
2. deleting one or more of the first operators, from $o_n, \ldots, o_1$ to $o_i, \ldots, o_1$ for some $i < n$. This is for local search algorithms only.

$o_n, \ldots, o_1$ is a plan if $I \models \mathit{regr}_{o_n}(\cdots \mathit{regr}_{o_2}(\mathit{regr}_{o_1}(G)) \cdots)$.

Remark

The above is for the simplest case when formulae are not split. With splitting the formalization is slightly trickier.

Plan search: incomplete plans for regression

For regression, the incomplete plans are suffixes $o_n, \dots, o_1$ of potential plans.

An incomplete plan is extended by

1. adding an operator in front of the first operator, from $o_n, \dots, o_1$ to $o, o_n, \dots, o_1$ for $o \in O$, or
2. deleting one or more of the first operators, from $o_n, \dots, o_1$ to $o_i, \dots, o_1$ for some $i < n$. This is for local search algorithms only.

$o_n, \dots, o_1$ is a plan if $I \models \mathit{regr}_{o_n}(\cdots \mathit{regr}_{o_2}(\mathit{regr}_{o_1}(G)) \cdots)$.

Remark

The above is for the simplest case when formulae are not split. With splitting the formalization is slightly trickier.

Plan search: incomplete plans for regression

For regression, the incomplete plans are suffixes $o_n, \ldots, o_1$ of potential plans.

An incomplete plan is extended by

1. adding an operator in front of the first operator, from $o_n, \ldots, o_1$ to $o, o_n, \ldots, o_1$ for $o \in O$, or
2. deleting one or more of the first operators, from $o_n, \ldots, o_1$ to $o_i, \ldots, o_1$ for some $i < n$. This is for local search algorithms only.

$o_n, \ldots, o_1$ is a plan if $I \models \mathit{regr}_{o_n}(\cdots \mathit{regr}_{o_2}(\mathit{regr}_{o_1}(G)) \cdots)$.

Remark

The above is for the simplest case when formulae are not split. With splitting the formalization is slightly trickier.

Plan search: incomplete plans for regression

For regression, the incomplete plans are suffixes $o_n, \ldots, o_1$ of potential plans.

An incomplete plan is extended by

1. adding an operator in front of the first operator, from $o_n, \ldots, o_1$ to $o, o_n, \ldots, o_1$ for $o \in O$, or
2. deleting one or more of the first operators, from $o_n, \ldots, o_1$ to $o_i, \ldots, o_1$ for some $i < n$. This is for local search algorithms only.

$o_n, \ldots, o_1$ is a plan if $I \models \mathit{regr}_{o_n}(\cdots \mathit{regr}_{o_2}(\mathit{regr}_{o_1}(G)) \cdots)$.

Remark

The above is for the simplest case when formulae are not split. With splitting the formalization is slightly trickier.

Plan search: incomplete plans for regression

For regression, the incomplete plans are suffixes $o_n, \ldots, o_1$ of potential plans.

An incomplete plan is extended by

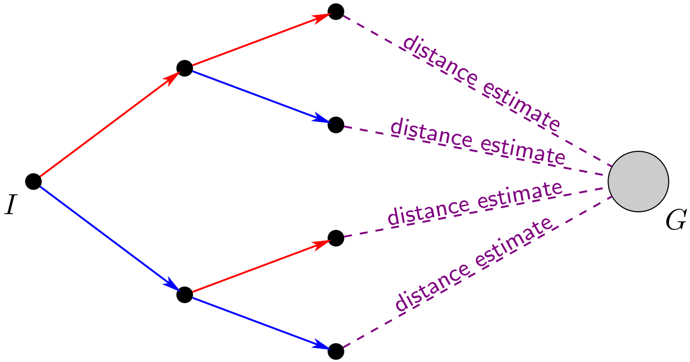
1. adding an operator in front of the first operator, from $o_n, \ldots, o_1$ to $o, o_n, \ldots, o_1$ for $o \in O$, or
2. deleting one or more of the first operators, from $o_n, \ldots, o_1$ to $o_i, \ldots, o_1$ for some $i < n$. This is for local search algorithms only.

$o_n, \ldots, o_1$ is a plan if $I \models \mathit{regr}_{o_n}(\cdots \mathit{regr}_{o_2}(\mathit{regr}_{o_1}(G)) \cdots)$.

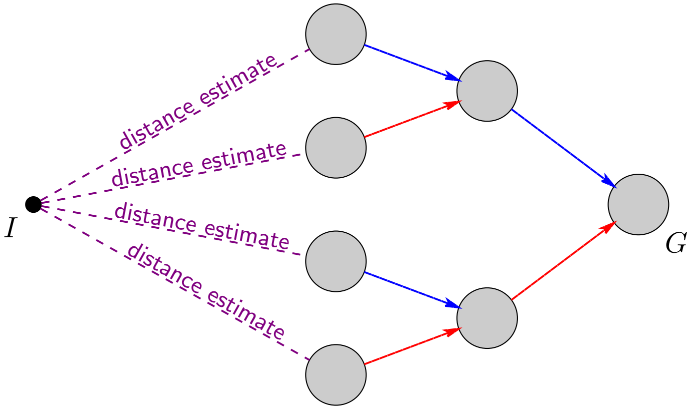
Remark

The above is for the simplest case when formulae are not split. With splitting the formalization is slightly trickier.

Planning by heuristic search
Forward search



Planning by heuristic search
Backward search



Select next operator $o \in O$ based on the estimated distance (number of operators) between

1. $\mathit{app}_o(\mathit{app}_{o_n}(\mathit{app}_{o_{n-1}}(\cdots \mathit{app}_{o_1}(I)\cdots)))$ and G, for forward search.
2. I and $\mathit{regr}_o(\mathit{regr}_{o_n}(\cdots \mathit{regr}_{o_2}(\mathit{regr}_{o_1}(G))\cdots))$, for backward search.

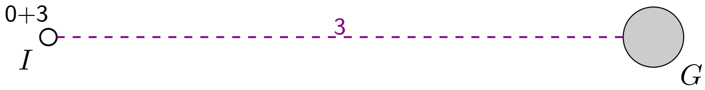
Search algorithms: A*

Search control of A*

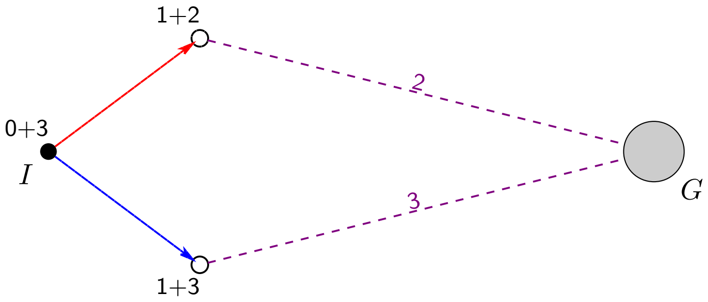
A* uses the function $f(\sigma) = g(\sigma) + h(\sigma)$ to guide search:

- $g(\sigma) =$ cost so far, i. e. number of operators in σ
- $h(\sigma) =$ estimated remaining cost (distance)
- admissibility: $h(\sigma)$ must be less than or equal to the actual remaining cost $h^*(\sigma)$ (distance), otherwise A* is not guaranteed to find an optimal solution.

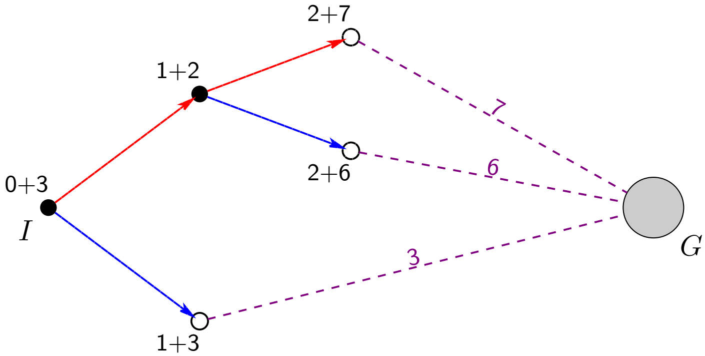
Search algorithms: A*
Example



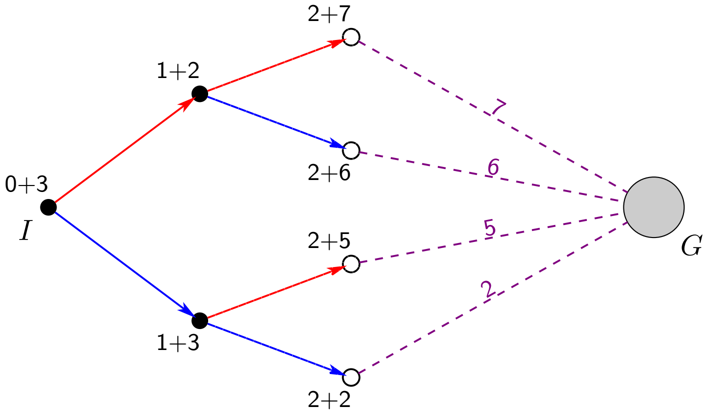
Search algorithms: A*
Example



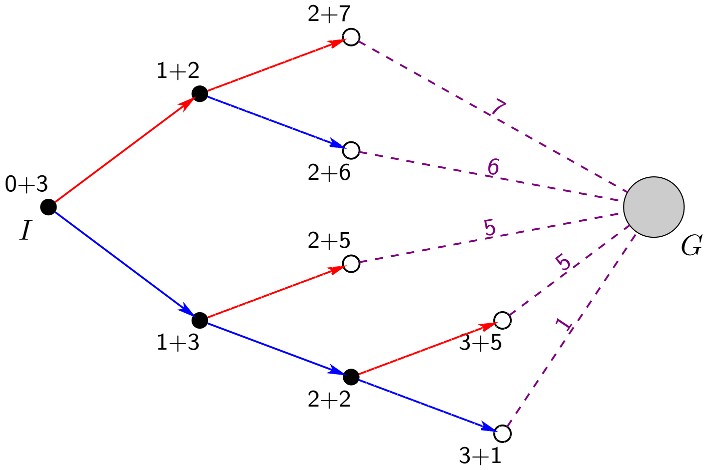
Search algorithms: A*
Example



Search algorithms: A*
Example

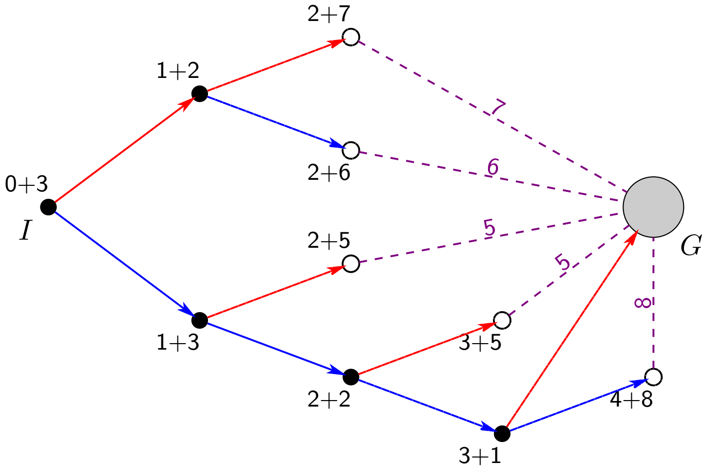


Search algorithms: A*
Example



Search algorithms: A*

Example



Search algorithms: A*
Definition

Notation for operator sequences

$\mathit{app}_{o_1;o_2;\ldots;o_n}(s)$ denotes $\mathit{app}_{o_n}(\ldots\mathit{app}_{o_2}(\mathit{app}_{o_1}(s))\ldots)$ and ϵ denotes the empty sequence for which $\mathit{app}_\epsilon(s) = s$.

Algorithm A*

Forward search with A* works as follows.

$\mathit{open} := \{\epsilon\}$, $\mathit{closed} := \emptyset$

loop:

    if $\mathit{open} = \emptyset$:

        return unsolvable

    Choose an element $\sigma \in \mathit{open}$ with the least $f(\sigma)$.

    if $\mathit{app}_\sigma(I) \models G$:

        return σ

    $\mathit{open} := \mathit{open} \setminus \{\sigma\}$; $\mathit{closed} := \mathit{closed} \cup \{\sigma\}$

    $\mathit{open} := \mathit{open} \cup (\{\sigma; o \mid o \in O\} \setminus \mathit{closed})$

Local search: random walk

Random walk

$\sigma := \epsilon$

loop:

  if *app*$_\sigma(I) \models G$:

    return σ

  Randomly choose a neighbor σ' of σ.

  $\sigma := \sigma'$

Remark

The algorithm usually does not find any solutions, unless almost every sequence of actions is a plan.

Local search: steepest descent hill-climbing

Hill-climbing

$\sigma := \epsilon$
loop:
    if $app_\sigma(I) \models G$:
        return σ
    Randomly choose a neighbor σ' of σ with the least $h(\sigma')$.
    $\sigma := \sigma'$

Remark

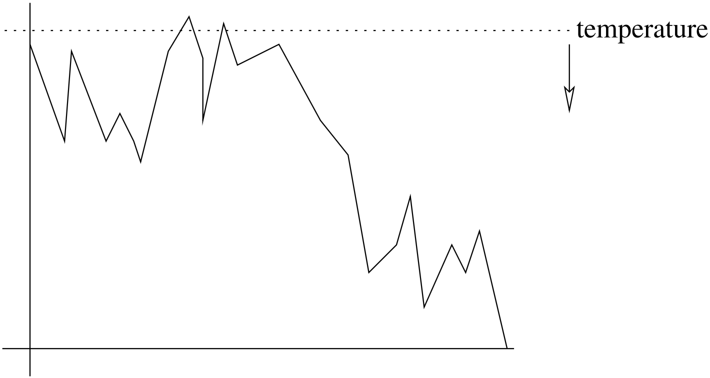
The algorithm gets stuck in local minima if no neighbor σ' has a better heuristic value than the current incomplete plan σ.

Local search: simulated annealing

Simulated annealing

$\sigma := \epsilon$
loop:
  if $\mathit{app}_\sigma(I) \models G$:
    return σ
  Randomly choose a neighbor σ' of σ.
  if $h(\sigma') < h(\sigma)$:
    $\sigma := \sigma'$
  else with probability $\exp\left(-\frac{h(\sigma')-h(\sigma)}{T}\right)$:
    $\sigma := \sigma'$
  Decrease T. (Different possible strategies!)

The temperature T is initially high and then gradually decreased.


temperature

How to obtain heuristics?

General procedure for obtaining a heuristic

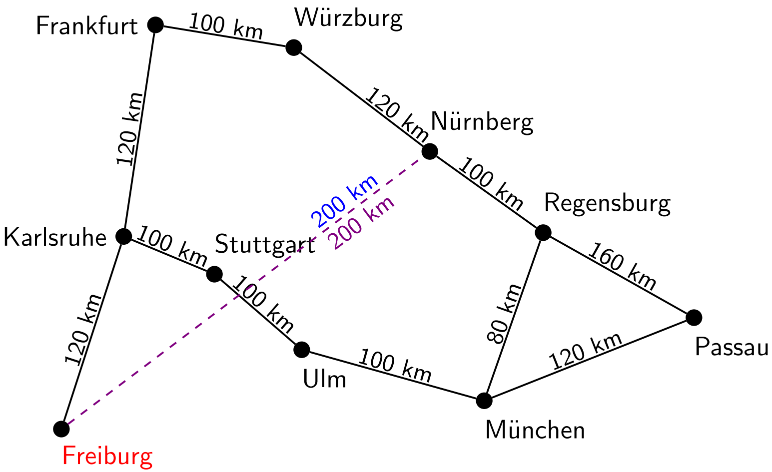
Solve a simplified / less restricted version of the problem.

Example (Route planning for a road network)

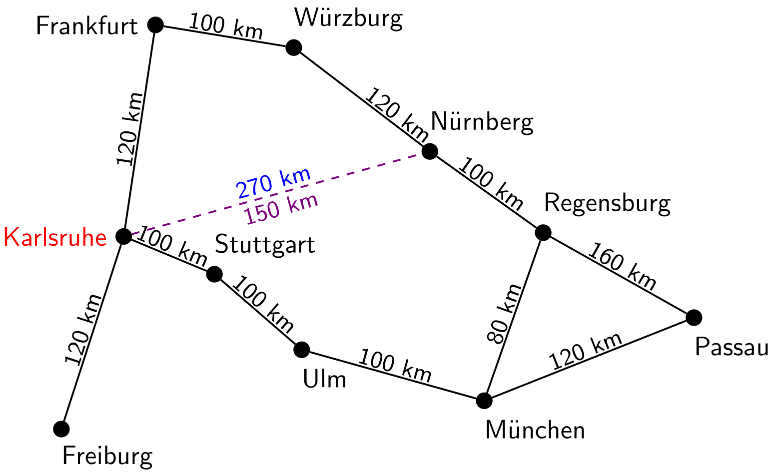
The road network is formalized as a weighted graph where the weight of an edge is the road distance between two locations. A heuristic is obtained from the Euclidean distance $\sqrt{|x_1 - x_2|^2 + |y_1 - y_2|^2}$. It is a lower bound on the road distance between (x_1, y_1) and (x_2, y_2).

An admissible heuristic for route planning

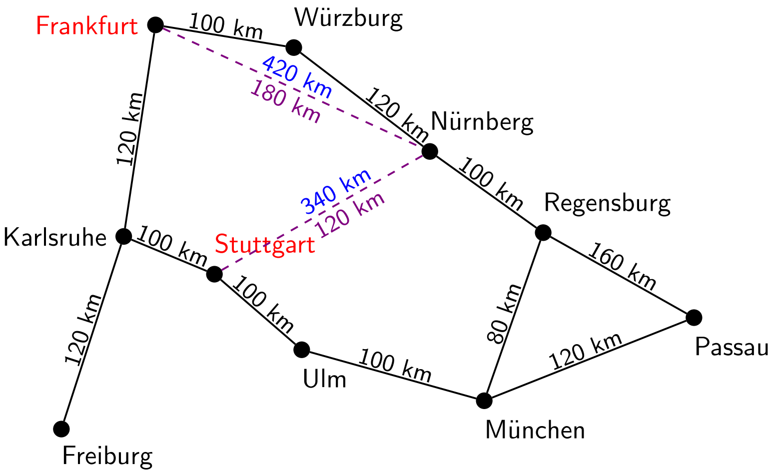
Example



An admissible heuristic for route planning
Example

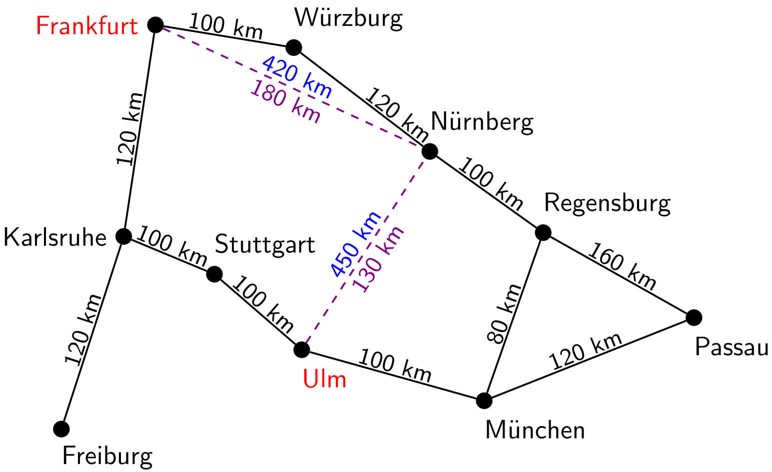


An admissible heuristic for route planning
Example

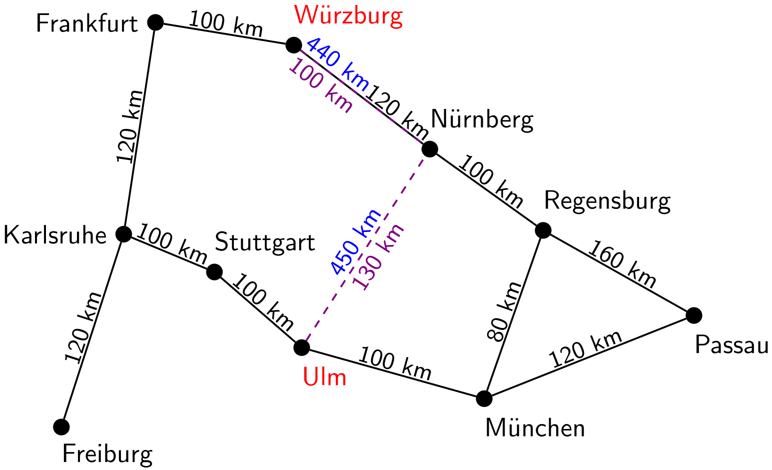


An admissible heuristic for route planning

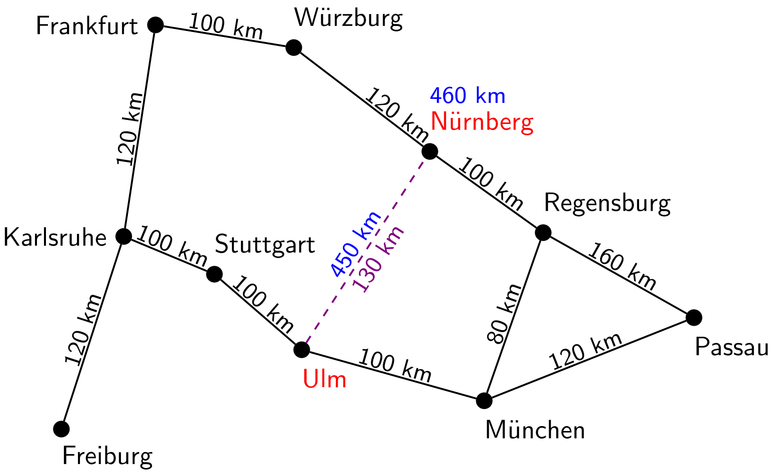
Example



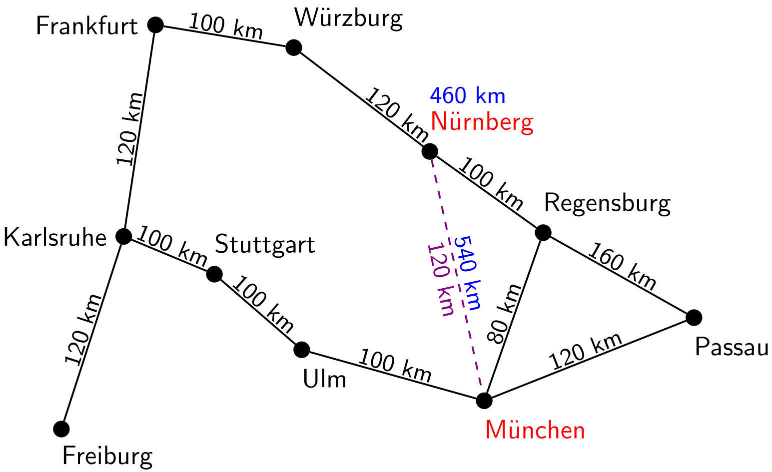
An admissible heuristic for route planning
Example



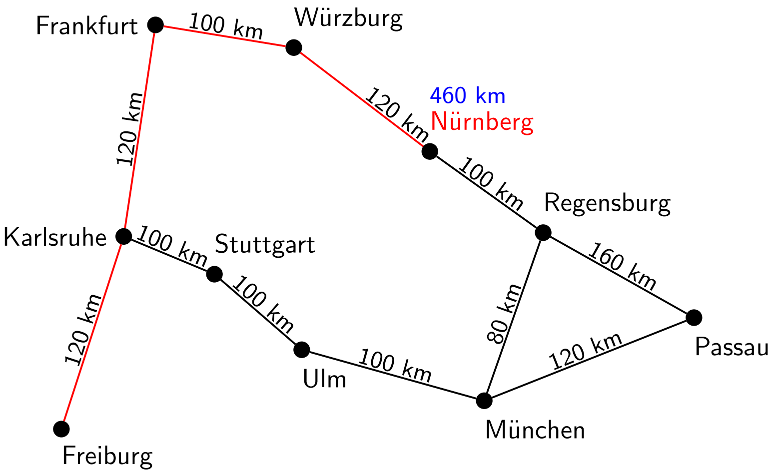
An admissible heuristic for route planning
Example



An admissible heuristic for route planning
Example



An admissible heuristic for route planning
Example



- STRIPS (Fikes & Nilsson, 1971) used the number of state variables that differ in current state s and a goal state s':

$$|\{a \in A \mid s(a) \neq s'(a)\}|.$$

  "The more goal literals an operator makes true, the more useful the operator is."

- The above heuristic is not admissible because one operator may reduce this measure by more than one. Instead,

$$\frac{|\{a \in A \mid s(a) \neq s'(a)\}|}{n}$$

  is admissible when no operator has $> n$ atomic effects.

Intuition

- To compute heuristics for planning tasks, we consider a relaxed version of the original problem, where some difficult aspects of the original problem are ignored.
- This is a general technique for heuristic design:
  - Straight-line heuristic (route planning): Ignore the fact that one must stay on roads.
  - Manhattan heuristic (15-puzzle): Ignore the fact that one cannot move through occupied tiles.
- For general planning problems, we will ignore negative interactions. Informally, we ignore "bad side effects" of applying operators.

Intuition

Question: Which operator effects are good, and which are bad?

This is difficult to answer in general, because it depends on context:

- If we want to prevent burglars from breaking into our flat, locking the entrance door is good.
- If we want to pass through it, locking the entrance door is bad.

We will now consider a reformulation of planning problems that makes the distinction between good and bad effects obvious.

Positive normal form

Definition

An operator $o = \langle c, e \rangle$ is in positive normal form if it is in normal form, no negation symbols appear in c, and no negation symbols appear in any effect condition in e.

A succinct deterministic transition system $\langle A, I, O, G \rangle$ is in positive normal form if all operators in O are in positive normal form and no negation symbols occur in the goal G.

Theorem

For every succinct deterministic transition system, an equivalent succinct deterministic transition system in positive normal form can be computed in polynomial time.

Equivalence here means that the represented (non-succinct) deterministic transition systems are isomorphic.

Positive normal form: algorithm

Transformation of $\langle A, I, O, G \rangle$ to positive normal form

Convert all operators $o \in O$ to normal form.
Convert all conditions to negation normal form (NNF).
while any condition contains a negative literal $\neg a$:
  Let a be a variable which occurs negatively in a condition.
  $A := A \cup \{\hat{a}\}$ for some new state variable $\hat{a}$
  $I(\hat{a}) := 1 - I(a)$
  Replace the effect $\neg a$ by $(\neg a \wedge \hat{a})$ in all operators $o \in O$.
  Replace the effect a by $(a \wedge \neg \hat{a})$ in all operators $o \in O$.
  Replace $\neg a$ by $\hat{a}$ in all conditions.
Convert all operators $o \in O$ to normal form (again).

Here, *all conditions* refers to all operator preconditions, operator effect conditions and the goal.

Positive normal form: Example

Example

$$A = \{\textit{home}, \textit{uni}, \textit{lecture}, \textit{bike}, \textit{bike-locked}\}$$

$$I = \{\textit{home} \mapsto 1, \textit{bike} \mapsto 1, \textit{bike-locked} \mapsto 1,$$
$$\textit{uni} \mapsto 0, \textit{lecture} \mapsto 0\}$$

$$O = \{\langle \textit{home} \land \textit{bike} \land \neg \textit{bike-locked}, \neg \textit{home} \land \textit{uni}\rangle,$$
$$\langle \textit{bike} \land \textit{bike-locked}, \neg \textit{bike-locked}\rangle,$$
$$\langle \textit{bike} \land \neg \textit{bike-locked}, \textit{bike-locked}\rangle,$$
$$\langle \textit{uni}, \textit{lecture} \land ((\textit{bike} \land \neg \textit{bike-locked}) \rhd \neg \textit{bike})\rangle\}$$

$$G = \textit{lecture} \land \textit{bike}$$

Positive normal form: Example

Example

$$
\begin{aligned}
A &= \{\textit{home}, \textit{uni}, \textit{lecture}, \textit{bike}, \textit{bike-locked}\} \\
I &= \{\textit{home} \mapsto 1, \textit{bike} \mapsto 1, \textit{bike-locked} \mapsto 1, \\
&\qquad \textit{uni} \mapsto 0, \textit{lecture} \mapsto 0\} \\
O &= \{\langle \textit{home} \land \textit{bike} \land \neg\textit{bike-locked}, \neg\textit{home} \land \textit{uni}\rangle, \\
&\qquad \langle \textit{bike} \land \textit{bike-locked}, \neg\textit{bike-locked}\rangle, \\
&\qquad \langle \textit{bike} \land \neg\textit{bike-locked}, \textit{bike-locked}\rangle, \\
&\qquad \langle \textit{uni}, \textit{lecture} \land ((\textit{bike} \land \neg\textit{bike-locked}) \rhd \neg\textit{bike})\rangle\} \\
G &= \textit{lecture} \land \textit{bike}
\end{aligned}
$$

Positive normal form: Example

Example

$$A = \{\textit{home}, \textit{uni}, \textit{lecture}, \textit{bike}, \textit{bike-locked}, \textit{bike-unlocked}\}$$

$$I = \{\textit{home} \mapsto 1, \textit{bike} \mapsto 1, \textit{bike-locked} \mapsto 1,$$
$$\textit{uni} \mapsto 0, \textit{lecture} \mapsto 0, \textit{bike-unlocked} \mapsto 0\}$$

$$O = \{\langle \textit{home} \land \textit{bike} \land \neg \textit{bike-locked}, \neg \textit{home} \land \textit{uni}\rangle,$$
$$\langle \textit{bike} \land \textit{bike-locked}, \neg \textit{bike-locked}\rangle,$$
$$\langle \textit{bike} \land \neg \textit{bike-locked}, \textit{bike-locked}\rangle,$$
$$\langle \textit{uni}, \textit{lecture} \land ((\textit{bike} \land \neg \textit{bike-locked}) \rhd \neg \textit{bike})\rangle\}$$

$$G = \textit{lecture} \land \textit{bike}$$

Positive normal form: Example

Example

$$A = \{\textit{home}, \textit{uni}, \textit{lecture}, \textit{bike}, \textit{bike-locked}, \textit{bike-unlocked}\}$$

$$I = \{\textit{home} \mapsto 1, \textit{bike} \mapsto 1, \textit{bike-locked} \mapsto 1, \textit{uni} \mapsto 0, \textit{lecture} \mapsto 0, \textit{bike-unlocked} \mapsto 0\}$$

$$O = \{\langle \textit{home} \land \textit{bike} \land \neg \textit{bike-locked}, \neg \textit{home} \land \textit{uni}\rangle, \langle \textit{bike} \land \textit{bike-locked}, \neg \textit{bike-locked}\rangle, \langle \textit{bike} \land \neg \textit{bike-locked}, \textit{bike-locked}\rangle, \langle \textit{uni}, \textit{lecture} \land ((\textit{bike} \land \neg \textit{bike-locked}) \rhd \neg \textit{bike})\rangle\}$$

$$G = \textit{lecture} \land \textit{bike}$$

Positive normal form: Example

Example

$$
\begin{aligned}
A &= \{\textit{home}, \textit{uni}, \textit{lecture}, \textit{bike}, \textit{bike-locked}, \textit{bike-unlocked}\} \\
I &= \{\textit{home} \mapsto 1, \textit{bike} \mapsto 1, \textit{bike-locked} \mapsto 1, \\
&\qquad \textit{uni} \mapsto 0, \textit{lecture} \mapsto 0, \textit{bike-unlocked} \mapsto 0\} \\
O &= \{\langle \textit{home} \land \textit{bike} \land \neg\textit{bike-locked}, \neg\textit{home} \land \textit{uni}\rangle, \\
&\qquad \langle \textit{bike} \land \textit{bike-locked}, \neg\textit{bike-locked} \land \textit{bike-unlocked}\rangle, \\
&\qquad \langle \textit{bike} \land \neg\textit{bike-locked}, \textit{bike-locked} \land \neg\textit{bike-unlocked}\rangle, \\
&\qquad \langle \textit{uni}, \textit{lecture} \land ((\textit{bike} \land \neg\textit{bike-locked}) \rhd \neg\textit{bike})\rangle\} \\
G &= \textit{lecture} \land \textit{bike}
\end{aligned}
$$

Positive normal form: Example

Example

$$
\begin{aligned}
A &= \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}, \mathit{bike\text{-}unlocked}\} \\
I &= \{\mathit{home} \mapsto 1, \mathit{bike} \mapsto 1, \mathit{bike\text{-}locked} \mapsto 1, \\
&\qquad \mathit{uni} \mapsto 0, \mathit{lecture} \mapsto 0, \mathit{bike\text{-}unlocked} \mapsto 0\} \\
O &= \{\langle \mathit{home} \wedge \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \neg \mathit{home} \wedge \mathit{uni}\rangle, \\
&\qquad \langle \mathit{bike} \wedge \mathit{bike\text{-}locked}, \neg \mathit{bike\text{-}locked} \wedge \mathit{bike\text{-}unlocked}\rangle, \\
&\qquad \langle \mathit{bike} \wedge \neg \mathit{bike\text{-}locked}, \mathit{bike\text{-}locked} \wedge \neg \mathit{bike\text{-}unlocked}\rangle, \\
&\qquad \langle \mathit{uni}, \mathit{lecture} \wedge ((\mathit{bike} \wedge \neg \mathit{bike\text{-}locked}) \rhd \neg \mathit{bike})\rangle\} \\
G &= \mathit{lecture} \wedge \mathit{bike}
\end{aligned}
$$

Positive normal form: Example

Example

$$
\begin{aligned}
A &= \{\mathit{home}, \mathit{uni}, \mathit{lecture}, \mathit{bike}, \mathit{bike\text{-}locked}, \mathit{bike\text{-}unlocked}\} \\
I &= \{\mathit{home} \mapsto 1, \mathit{bike} \mapsto 1, \mathit{bike\text{-}locked} \mapsto 1, \\
&\qquad \mathit{uni} \mapsto 0, \mathit{lecture} \mapsto 0, \mathit{bike\text{-}unlocked} \mapsto 0\} \\
O &= \{\langle \mathit{home} \land \mathit{bike} \land \mathit{bike\text{-}unlocked}, \neg \mathit{home} \land \mathit{uni}\rangle, \\
&\qquad \langle \mathit{bike} \land \mathit{bike\text{-}locked}, \neg \mathit{bike\text{-}locked} \land \mathit{bike\text{-}unlocked}\rangle, \\
&\qquad \langle \mathit{bike} \land \mathit{bike\text{-}unlocked}, \mathit{bike\text{-}locked} \land \neg \mathit{bike\text{-}unlocked}\rangle, \\
&\qquad \langle \mathit{uni}, \mathit{lecture} \land ((\mathit{bike} \land \mathit{bike\text{-}unlocked}) \rhd \neg \mathit{bike})\rangle\} \\
G &= \mathit{lecture} \land \mathit{bike}
\end{aligned}
$$

Positive normal form: Example

Example

$$A = \{\textit{home}, \textit{uni}, \textit{lecture}, \textit{bike}, \textit{bike-locked}, \textit{bike-unlocked}\}$$

$$I = \{\textit{home} \mapsto 1, \textit{bike} \mapsto 1, \textit{bike-locked} \mapsto 1, \textit{uni} \mapsto 0, \textit{lecture} \mapsto 0, \textit{bike-unlocked} \mapsto 0\}$$

$$O = \{\langle \textit{home} \wedge \textit{bike} \wedge \textit{bike-unlocked}, \neg\textit{home} \wedge \textit{uni}\rangle, \langle \textit{bike} \wedge \textit{bike-locked}, \neg\textit{bike-locked} \wedge \textit{bike-unlocked}\rangle, \langle \textit{bike} \wedge \textit{bike-unlocked}, \textit{bike-locked} \wedge \neg\textit{bike-unlocked}\rangle, \langle \textit{uni}, \textit{lecture} \wedge ((\textit{bike} \wedge \textit{bike-unlocked}) \rhd \neg\textit{bike})\rangle\}$$

$$G = \textit{lecture} \wedge \textit{bike}$$

Intuition

In positive normal form, good and bad effects are easy to distinguish:

- Effects that make state variables true are good (add effects).
- Effects that make state variables false are bad (delete effects).

Idea for the heuristic: Ignore all delete effects.

Relaxation

Definition

The relaxation o^+ of an operator $o = \langle c, e \rangle$ in positive normal form is the operator which is obtained by replacing all negative effects $\neg a$ within e by the do-nothing effect $\top$.

The relaxation $\mathcal{P}^+$ of a succinct deterministic transition system $\mathcal{P} = \langle A, I, O, G \rangle$ in positive normal form is the succinct deterministic transition system $\mathcal{P}^+ := \langle A, I, \{o^+ \mid o \in O\}, G \rangle$.

The relaxation of an operator sequence $\pi = o_1, \ldots, o_n$ is the operator sequence $\pi^+ := {o_1}^+, \ldots, {o_n}^+$.

Relaxation: properties

The on-set $on(s)$ of a state s is the set of true state variables in s, i.e. $on(s) = s^{-1}(\{1\})$.
A state s' dominates another state s iff $on(s) \subseteq on(s')$.

Lemma (domination)

Let s and s' be valuations of a set of propositional variables and let χ be a propositional formula which does not contain negation symbols.
If $s \models \chi$ and s' dominates s, then $s' \models \chi$.

Proof by induction over the structure of χ (exercise).

Relaxation: properties

Lemma (relaxation leads to dominated states)

Let s be a state, and let π be an operator sequence which is applicable in s.
Then π^+ is applicable in s and $\mathit{app}_{\pi^+}(s)$ dominates $\mathit{app}_{\pi}(s)$.

Proof.

Induction on the length of π.

Base case: $\pi = \epsilon$
Trivial.

Relaxation: properties

Lemma (relaxation leads to dominated states)

Let s be a state, and let π be an operator sequence which is applicable in s.
Then π^+ is applicable in s and $app_{\pi^+}(s)$ dominates $app_\pi(s)$.

Proof.

Induction on the length of π.

Base case: $\pi = \epsilon$
Trivial.

Relaxation: properties

Proof continues.

Inductive case: $\pi = {o_1}^+ \dots {o_{n+1}}^+$

By the induction hypothesis, ${o_1}^+ \dots {o_n}^+$ is applicable in s, and $t^+ = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s)$ dominates $t = \mathit{app}_{o_1 \dots o_n}(s)$.

Let $o := o_{n+1} = \langle c, e \rangle$ and $o^+ = \langle c, e^+ \rangle$. By assumption, o is applicable in t, and thus $t \models c$. By the domination lemma, we get $t^+ \models c$ and hence o^+ is applicable in t^+. Therefore, π^+ is applicable in s.

Because o is in positive normal form, all effect conditions satisfied by t are also satisfied by t^+ (by the domination lemma). Therefore, $([e]_t \cap A) \subseteq [e^+]_{t^+}$ (where A is the set of state variables, or positive literals).

We get $\mathit{on}(\mathit{app}_\pi(s)) \subseteq \mathit{on}(t) \cup ([e]_t \cap A) \subseteq \mathit{on}(t^+) \cup [e^+]_{t^+} = \mathit{on}(\mathit{app}_{\pi^+}(s))$, and thus $\mathit{app}_{\pi^+}(s)$ dominates $\mathit{app}_\pi(s)$.

Relaxation: properties

Proof continues.

Inductive case: $\pi = {o_1}^+ \dots {o_{n+1}}^+$

By the induction hypothesis, ${o_1}^+ \dots {o_n}^+$ is applicable in s, and $t^+ = app_{{o_1}^+ \dots {o_n}^+}(s)$ dominates $t = app_{o_1 \dots o_n}(s)$.

Let $o := o_{n+1} = \langle c, e \rangle$ and $o^+ = \langle c, e^+ \rangle$. By assumption, o is applicable in t, and thus $t \models c$. By the domination lemma, we get $t^+ \models c$ and hence o^+ is applicable in t^+. Therefore, π^+ is applicable in s.

Because o is in positive normal form, all effect conditions satisfied by t are also satisfied by t^+ (by the domination lemma). Therefore, $([e]_t \cap A) \subseteq [e^+]_{t^+}$ (where A is the set of state variables, or positive literals).

We get $on(app_\pi(s)) \subseteq on(t) \cup ([e]_t \cap A) \subseteq on(t^+) \cup [e^+]_{t^+} = on(app_{\pi^+}(s))$, and thus $app_{\pi^+}(s)$ dominates $app_\pi(s)$.

Relaxation: properties

Proof continues.

Inductive case: $\pi = {o_1}^+ \dots {o_{n+1}}^+$

By the induction hypothesis, ${o_1}^+ \dots {o_n}^+$ is applicable in s, and $t^+ = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s)$ dominates $t = \mathit{app}_{o_1 \dots o_n}(s)$.

Let $o := o_{n+1} = \langle c, e \rangle$ and $o^+ = \langle c, e^+ \rangle$. By assumption, o is applicable in t, and thus $t \models c$. By the domination lemma, we get $t^+ \models c$ and hence o^+ is applicable in t^+. Therefore, π^+ is applicable in s.

Because o is in positive normal form, all effect conditions satisfied by t are also satisfied by t^+ (by the domination lemma). Therefore, $([e]_t \cap A) \subseteq [e^+]_{t^+}$ (where A is the set of state variables, or positive literals).

We get $\mathit{on}(\mathit{app}_\pi(s)) \subseteq \mathit{on}(t) \cup ([e]_t \cap A) \subseteq \mathit{on}(t^+) \cup [e^+]_{t^+} = \mathit{on}(\mathit{app}_{\pi^+}(s))$, and thus $\mathit{app}_{\pi^+}(s)$ dominates $\mathit{app}_\pi(s)$.

Relaxation: properties

Proof continues.

Inductive case: $\pi = {o_1}^+ \dots {o_{n+1}}^+$

By the induction hypothesis, ${o_1}^+ \dots {o_n}^+$ is applicable in s, and $t^+ = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s)$ dominates $t = \mathit{app}_{o_1 \dots o_n}(s)$.

Let $o := o_{n+1} = \langle c, e \rangle$ and $o^+ = \langle c, e^+ \rangle$. By assumption, o is applicable in t, and thus $t \models c$. By the domination lemma, we get $t^+ \models c$ and hence o^+ is applicable in t^+. Therefore, π^+ is applicable in s.

Because o is in positive normal form, all effect conditions satisfied by t are also satisfied by t^+ (by the domination lemma). Therefore, $([e]_t \cap A) \subseteq [e^+]_{t^+}$ (where A is the set of state variables, or positive literals).

We get $\mathit{on}(\mathit{app}_\pi(s)) \subseteq \mathit{on}(t) \cup ([e]_t \cap A) \subseteq \mathit{on}(t^+) \cup [e^+]_{t^+} = \mathit{on}(\mathit{app}_{\pi^+}(s))$, and thus $\mathit{app}_{\pi^+}(s)$ dominates $\mathit{app}_\pi(s)$.

Relaxation: properties

Theorem (solution preservation)

Let π be a plan for a succinct deterministic transition system $\mathcal{T}$ in positive normal form.
Then π^+ is a plan for $\mathcal{T}^+$.

Proof.

Let $\mathcal{T} = \langle A, I, O, G \rangle$ and thus $\mathcal{T}^+ = \langle A, I, O^+, G \rangle$.

Since π is applicable in I, π^+ is also applicable in I (by the previous lemma).

Also by the previous lemma, the resulting state $s^+ := app_{\pi^+}(I)$ dominates the state $s := app_{\pi}(I)$. Because $s \models G$ and G is negation-free, we get $s^+ \models G$ by the domination lemma.

Thus π^+ is indeed a plan for $\mathcal{T}^+$.

Relaxation: properties

Theorem (solution preservation)

Let π be a plan for a succinct deterministic transition system $\mathcal{T}$ in positive normal form.
Then π^+ is a plan for $\mathcal{T}^+$.

Proof.

Let $\mathcal{T} = \langle A, I, O, G \rangle$ and thus $\mathcal{T}^+ = \langle A, I, O^+, G \rangle$.

Since π is applicable in I, π^+ is also applicable in I (by the previous lemma).

Also by the previous lemma, the resulting state $s^+ := \mathit{app}_{\pi^+}(I)$ dominates the state $s := \mathit{app}_{\pi}(I)$. Because $s \models G$ and G is negation-free, we get $s^+ \models G$ by the domination lemma.

Thus π^+ is indeed a plan for $\mathcal{T}^+$.

Relaxation: properties

Theorem (solution preservation)

Let π be a plan for a succinct deterministic transition system $\mathcal{T}$ in positive normal form.
Then π^+ is a plan for $\mathcal{T}^+$.

Proof.

Let $\mathcal{T} = \langle A, I, O, G \rangle$ and thus $\mathcal{T}^+ = \langle A, I, O^+, G \rangle$.

Since π is applicable in I, π^+ is also applicable in I (by the previous lemma).

Also by the previous lemma, the resulting state $s^+ := \mathit{app}_{\pi^+}(I)$ dominates the state $s := \mathit{app}_{\pi}(I)$. Because $s \models G$ and G is negation-free, we get $s^+ \models G$ by the domination lemma.

Thus π^+ is indeed a plan for $\mathcal{T}^+$.

Relaxation: properties

Theorem (solution preservation)

Let π be a plan for a succinct deterministic transition system $\mathcal{T}$ in positive normal form.
Then π^+ is a plan for $\mathcal{T}^+$.

Proof.

Let $\mathcal{T} = \langle A, I, O, G \rangle$ and thus $\mathcal{T}^+ = \langle A, I, O^+, G \rangle$.

Since π is applicable in I, π^+ is also applicable in I (by the previous lemma).

Also by the previous lemma, the resulting state $s^+ := app_{\pi^+}(I)$ dominates the state $s := app_{\pi}(I)$. Because $s \models G$ and G is negation-free, we get $s^+ \models G$ by the domination lemma.

Thus π^+ is indeed a plan for $\mathcal{T}^+$.

Relaxation: properties

Consequences of the solution preservation theorem:

- Relaxations are never harder to solve than the original problem.
- Optimal solutions to relaxations are never longer than optimal solutions to the original problem.

In fact, relaxations are much easier to solve than the original problems, which makes them suitable as the basis for heuristic functions.

We will now consider the problem of solving relaxations.

Solving relaxations: properties

Lemma (dominating states are better)

Let s be a state, let s' be a state that dominates s, and let π^+ be a relaxed operator sequence which is applicable in s.
Then π^+ is applicable in s' and $app_{\pi^+}(s')$ dominates $app_{\pi^+}(s)$.

Proof.

Induction on the length of π^+.

Base case: $\pi^+ = \epsilon$

The empty plan ϵ is applicable in s'.
Moreover, $app_{\pi^+}(s') = app_\epsilon(s') = s'$ and $app_{\pi^+}(s) = app_\epsilon(s) = s$, and s' dominates s.

Solving relaxations: properties

Lemma (dominating states are better)

Let s be a state, let s' be a state that dominates s, and let π^+ be a relaxed operator sequence which is applicable in s.
Then π^+ is applicable in s' and $app_{\pi^+}(s')$ dominates $app_{\pi^+}(s)$.

Proof.

Induction on the length of π^+.

Base case: $\pi^+ = \epsilon$
The empty plan ϵ is applicable in s'.
Moreover, $app_{\pi^+}(s') = app_\epsilon(s') = s'$ and $app_{\pi^+}(s) = app_\epsilon(s) = s$, and s' dominates s.

Solving relaxations: properties

Lemma (dominating states are better)

Let s be a state, let s' be a state that dominates s, and let π^+ be a relaxed operator sequence which is applicable in s. Then π^+ is applicable in s' and $app_{\pi^+}(s')$ dominates $app_{\pi^+}(s)$.

Proof.

Induction on the length of π^+.

Base case: $\pi^+ = \epsilon$
The empty plan ϵ is applicable in s'.
Moreover, $app_{\pi^+}(s') = app_\epsilon(s') = s'$ and $app_{\pi^+}(s) = app_\epsilon(s) = s$, and s' dominates s.

Solving relaxations: properties

Proof continues.

Inductive case: $\pi^+ = {o_1}^+ \dots {o_{n+1}}^+$

By the induction hypothesis, ${o_1}^+ \dots {o_n}^+$ is applicable in s', and $t' = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s')$ dominates $t = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s)$.

Let $o^+ := {o_{n+1}}^+ = \langle c, e \rangle$. By assumption, o^+ is applicable in t, and thus $t \models c$. By the domination lemma, we get $t' \models c$ and hence o^+ is applicable in t'. Therefore, π^+ is applicable in s'.

Because o^+ is in positive normal form, all effect conditions which are satisfied in t are also satisfied in t' (by the domination lemma). Therefore, $[e]_t \subseteq [e]_{t'}$.

Because all atomic effects in o^+ are positive, $[e]_t$ and $[e]_{t'}$ are sets of positive literals. We thus get $\mathit{on}(\mathit{app}_{\pi^+}(s)) = \mathit{on}(t) \cup [e]_t \subseteq \mathit{on}(t') \cup [e]_{t'} = \mathit{on}(\mathit{app}_{\pi^+}(s'))$, and thus $\mathit{app}_{\pi^+}(s')$ dominates $\mathit{app}_{\pi^+}(s)$.

Solving relaxations: properties

Proof continues.

Inductive case: $\pi^+ = {o_1}^+ \dots {o_{n+1}}^+$

By the induction hypothesis, ${o_1}^+ \dots {o_n}^+$ is applicable in s', and $t' = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s')$ dominates $t = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s)$.

Let $o^+ := {o_{n+1}}^+ = \langle c, e \rangle$. By assumption, o^+ is applicable in t, and thus $t \models c$. By the domination lemma, we get $t' \models c$ and hence o^+ is applicable in t'. Therefore, π^+ is applicable in s'.

Because o^+ is in positive normal form, all effect conditions which are satisfied in t are also satisfied in t' (by the domination lemma). Therefore, $[e]_t \subseteq [e]_{t'}$.

Because all atomic effects in o^+ are positive, $[e]_t$ and $[e]_{t'}$ are sets of positive literals. We thus get $\mathit{on}(\mathit{app}_{\pi^+}(s)) = \mathit{on}(t) \cup [e]_t \subseteq \mathit{on}(t') \cup [e]_{t'} = \mathit{on}(\mathit{app}_{\pi^+}(s'))$, and thus $\mathit{app}_{\pi^+}(s')$ dominates $\mathit{app}_{\pi^+}(s)$.

Solving relaxations: properties

Proof continues.

Inductive case: $\pi^+ = {o_1}^+ \dots {o_{n+1}}^+$

By the induction hypothesis, ${o_1}^+ \dots {o_n}^+$ is applicable in s', and $t' = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s')$ dominates $t = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s)$.

Let $o^+ := {o_{n+1}}^+ = \langle c, e \rangle$. By assumption, o^+ is applicable in t, and thus $t \models c$. By the domination lemma, we get $t' \models c$ and hence o^+ is applicable in t'. Therefore, π^+ is applicable in s'.

Because o^+ is in positive normal form, all effect conditions which are satisfied in t are also satisfied in t' (by the domination lemma). Therefore, $[e]_t \subseteq [e]_{t'}$.

Because all atomic effects in o^+ are positive, $[e]_t$ and $[e]_{t'}$ are sets of positive literals. We thus get $\mathit{on}(\mathit{app}_{\pi^+}(s)) = \mathit{on}(t) \cup [e]_t \subseteq \mathit{on}(t') \cup [e]_{t'} = \mathit{on}(\mathit{app}_{\pi^+}(s'))$, and thus $\mathit{app}_{\pi^+}(s')$ dominates $\mathit{app}_{\pi^+}(s)$.

Solving relaxations: properties

Proof continues.

Inductive case: $\pi^+ = {o_1}^+ \dots {o_{n+1}}^+$

By the induction hypothesis, ${o_1}^+ \dots {o_n}^+$ is applicable in s', and $t' = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s')$ dominates $t = \mathit{app}_{{o_1}^+ \dots {o_n}^+}(s)$.

Let $o^+ := {o_{n+1}}^+ = \langle c, e \rangle$. By assumption, o^+ is applicable in t, and thus $t \models c$. By the domination lemma, we get $t' \models c$ and hence o^+ is applicable in t'. Therefore, π^+ is applicable in s'.

Because o^+ is in positive normal form, all effect conditions which are satisfied in t are also satisfied in t' (by the domination lemma). Therefore, $[e]_t \subseteq [e]_{t'}$.

Because all atomic effects in o^+ are positive, $[e]_t$ and $[e]_{t'}$ are sets of positive literals. We thus get $\mathit{on}(\mathit{app}_{\pi^+}(s)) = \mathit{on}(t) \cup [e]_t \subseteq \mathit{on}(t') \cup [e]_{t'} = \mathit{on}(\mathit{app}_{\pi^+}(s'))$, and thus $\mathit{app}_{\pi^+}(s')$ dominates $\mathit{app}_{\pi^+}(s)$.

Solving relaxations: properties

Consequences of the lemma:

- If we can find a solution starting from a state s, the same solution can be used when starting from a dominating state s'.
- Thus, making a transition to a dominating state never hurts.

Lemma (monotonicity)

Let o^+ be a relaxed operator and let s be a state in which o^+ is applicable.
Then $\mathit{app}_{o^+}(s)$ dominates s.

Proof.

Since relaxed operators only have positive effects, we have $\mathit{on}(s) \subseteq \mathit{on}(s) \cup [e]_{o^+} = \mathit{on}(\mathit{app}_{o^+}(s))$. $\square$

Solving relaxations: properties

Consequences of the lemma:

- If we can find a solution starting from a state s, the same solution can be used when starting from a dominating state s'.
- Thus, making a transition to a dominating state never hurts.

Lemma (monotonicity)

Let o^+ be a relaxed operator and let s be a state in which o^+ is applicable.
Then $app_{o^+}(s)$ dominates s.

Proof.

Since relaxed operators only have positive effects, we have $on(s) \subseteq on(s) \cup [e]_{o^+} = on(app_{o^+}(s))$. □

Solving relaxations: properties

Consequences of the lemma:

- If we can find a solution starting from a state s, the same solution can be used when starting from a dominating state s'.
- Thus, making a transition to a dominating state never hurts.

Lemma (monotonicity)

Let o^+ be a relaxed operator and let s be a state in which o^+ is applicable.
Then $app_{o^+}(s)$ dominates s.

Proof.

Since relaxed operators only have positive effects, we have $on(s) \subseteq on(s) \cup [e]_{o^+} = on(app_{o^+}(s))$. □

Solving relaxations: algorithm

- Together, the two lemmas imply that making a transition never hurts.
- This suggests the following algorithm.

Greedy planning algorithm for $\langle A, I, O^+, G \rangle$

$s := I$
$\pi^+ := \epsilon$
loop:
  if $s \models G$:
    return π^+
  else if there is an operator $o^+ \in O^+$ with $app_{o^+}(s) \neq s$:
    Append such an operator o^+ to π^+.
    $s := app_{o^+}(s)$
  else:
    return unsolvable

Solving relaxations: algorithm

- Together, the two lemmas imply that making a transition never hurts.
- This suggests the following algorithm.

Greedy planning algorithm for $\langle A, I, O^+, G \rangle$

$s := I$
$\pi^+ := \epsilon$
loop:
  if $s \models G$:
    return π^+
  else if there is an operator $o^+ \in O^+$ with $app_{o^+}(s) \neq s$:
    Append such an operator o^+ to π^+.
    $s := app_{o^+}(s)$
  else:
    return unsolvable

Solving relaxations: algorithm

The algorithm is sound:

- If it returns a plan, this is indeed a correct solution.
- If it returns "unsolvable", the task is indeed unsolvable (by the two lemmas).

What about completeness (termination) and runtime?

- Each iteration of the loop adds at least one atom to $on(s)$.
- This guarantees termination after at most $|A|$ iterations.
- Thus, the algorithm can clearly run in polynomial time.

Solving relaxations: algorithm

The algorithm is sound:

- If it returns a plan, this is indeed a correct solution.
- If it returns "unsolvable", the task is indeed unsolvable (by the two lemmas).

What about completeness (termination) and runtime?

- Each iteration of the loop adds at least one atom to $\mathit{on}(s)$.
- This guarantees termination after at most $|A|$ iterations.
- Thus, the algorithm can clearly run in polynomial time.

Solving relaxations: optimality

One could use the solution algorithm as a heuristic estimator in a progression search for general planning tasks as follows:

- In a search node that corresponds to state s, solve the relaxation of the planning task with s as the initial state.
- Use the length of the relaxed plan as a heuristic estimate.

Is this an admissible heuristic?

- Yes if the relaxed plans are optimal (because of the solution preservation theorem).
- However, usually they are not, because our greedy planning algorithm is very poor.

Solving relaxations: optimality

One could use the solution algorithm as a heuristic estimator in a progression search for general planning tasks as follows:

- In a search node that corresponds to state s, solve the relaxation of the planning task with s as the initial state.
- Use the length of the relaxed plan as a heuristic estimate.

Is this an admissible heuristic?

- Yes if the relaxed plans are optimal (because of the solution preservation theorem).
- However, usually they are not, because our greedy planning algorithm is very poor.

Solving relaxations: optimality

One could use the solution algorithm as a heuristic estimator in a progression search for general planning tasks as follows:

- In a search node that corresponds to state s, solve the relaxation of the planning task with s as the initial state.
- Use the length of the relaxed plan as a heuristic estimate.

Is this an admissible heuristic?

- Yes if the relaxed plans are optimal (because of the solution preservation theorem).
- However, usually they are not, because our greedy planning algorithm is very poor.

Solving relaxations: optimality

So how do we use relaxations for heuristic planning?

Different possibilities:

- Implement an optimal planner for relaxed planning tasks and use its solution lengths as an estimate (h^+ heuristic).
  However, optimal planning for relaxed tasks is NP-hard.
- Do not actually solve the relaxed planning task, but compute an estimate of its difficulty in a different way (h_{max} heuristic, h_{add} heuristic).
- Compute a solution for relaxed planning tasks which is not necessarily optimal, but "reasonable" (h_{FF} heuristic).

Solving relaxations: optimality

So how do we use relaxations for heuristic planning?

Different possibilities:

- Implement an optimal planner for relaxed planning tasks and use its solution lengths as an estimate (h^+ heuristic). However, optimal planning for relaxed tasks is NP-hard.
- Do not actually solve the relaxed planning task, but compute an estimate of its difficulty in a different way (h_{max} heuristic, h_{add} heuristic).
- Compute a solution for relaxed planning tasks which is not necessarily optimal, but "reasonable" (h_{FF} heuristic).

Solving relaxations: optimality

So how do we use relaxations for heuristic planning?

Different possibilities:

- Implement an optimal planner for relaxed planning tasks and use its solution lengths as an estimate (h^+ heuristic). However, optimal planning for relaxed tasks is NP-hard.
- Do not actually solve the relaxed planning task, but compute an estimate of its difficulty in a different way (h_{max} heuristic, h_{add} heuristic).
- Compute a solution for relaxed planning tasks which is not necessarily optimal, but "reasonable" (h_{FF} heuristic).

Solving relaxations: optimality

So how do we use relaxations for heuristic planning?

Different possibilities:

- Implement an optimal planner for relaxed planning tasks and use its solution lengths as an estimate (h^+ heuristic). However, optimal planning for relaxed tasks is NP-hard.
- Do not actually solve the relaxed planning task, but compute an estimate of its difficulty in a different way (h_{max} heuristic, h_{add} heuristic).
- Compute a solution for relaxed planning tasks which is not necessarily optimal, but "reasonable" (h_{FF} heuristic).

Intutition

Why does the greedy algorithm compute low-quality plans?

- It may apply many operators which are not goal-directed.

How can this problem be fixed?

- Reaching the goal of a relaxed planning task is most easily achieved with forward search.
- Analyzing relevance of an operator for achieving a goal (or subgoal) is most easily achieved with backward search.

Idea: Use a forward-backward algorithm that first finds a path to the goal greedily, then prunes it to a relevant subplan.

Intutition

Why does the greedy algorithm compute low-quality plans?

- It may apply many operators which are not goal-directed.

How can this problem be fixed?

- Reaching the goal of a relaxed planning task is most easily achieved with forward search.
- Analyzing relevance of an operator for achieving a goal (or subgoal) is most easily achieved with backward search.

Idea: Use a forward-backward algorithm that first finds a path to the goal greedily, then prunes it to a relevant subplan.

Intutition

Why does the greedy algorithm compute low-quality plans?

- It may apply many operators which are not goal-directed.

How can this problem be fixed?

- Reaching the goal of a relaxed planning task is most easily achieved with forward search.
- Analyzing relevance of an operator for achieving a goal (or subgoal) is most easily achieved with backward search.

Idea: Use a forward-backward algorithm that first finds a path to the goal greedily, then prunes it to a relevant subplan.

Parallel plans

How do we decide which operators to apply in the forward direction?

- We avoid such a decision by applying all applicable operators simultaneously.

Definition

A plan step is a set of operators $\sigma = \{o_1, \ldots, o_n\}$.

A plan step $\sigma = \{o_1, \ldots, o_n\}$ with $o_i = \langle c_i, e_i \rangle$ of a relaxed planning task is applicable in a state s iff each operator $o_i \in \sigma$ is applicable in s.

The result of applying σ to s, in symbols $app_\sigma(s)$, is defined as the state s' with $on(s') = on(s) \cup \bigcup_{i=1}^{n} [e_i]_s$.

Parallel plans

How do we decide which operators to apply in the forward direction?

- We avoid such a decision by applying all applicable operators simultaneously.

Definition

A plan step is a set of operators $\sigma = \{o_1, \dots, o_n\}$.

A plan step $\sigma = \{o_1, \dots, o_n\}$ with $o_i = \langle c_i, e_i \rangle$ of a relaxed planning task is applicable in a state s iff each operator $o_i \in \sigma$ is applicable in s.

The result of applying σ to s, in symbols $\mathit{app}_\sigma(s)$, is defined as the state s' with $\mathit{on}(s') = \mathit{on}(s) \cup \bigcup_{i=1}^{n} [e_i]_s$.

Applying plan steps: Examples

In all cases, $s = \{a \mapsto 0, b \mapsto 0, c \mapsto 1, d \mapsto 0\}$.

- $\sigma = \{\langle c, a\rangle, \langle \top, b\rangle\}$
- $\sigma = \{\langle c, a\rangle, \langle c, a \rhd b\rangle\}$
- $\sigma = \{\langle c, a \wedge b\rangle, \langle a, b \rhd d\rangle\}$
- $\sigma = \{\langle c, a \wedge (b \rhd d)\rangle, \langle c, b \wedge (a \rhd d)\rangle\}$

Applying plan steps: Examples

In all cases, $s = \{a \mapsto 0, b \mapsto 0, c \mapsto 1, d \mapsto 0\}$.

- $\sigma = \{\langle c, a\rangle, \langle \top, b\rangle\}$
- $\sigma = \{\langle c, a\rangle, \langle c, a \rhd b\rangle\}$
- $\sigma = \{\langle c, a \wedge b\rangle, \langle a, b \rhd d\rangle\}$
- $\sigma = \{\langle c, a \wedge (b \rhd d)\rangle, \langle c, b \wedge (a \rhd d)\rangle\}$

Applying plan steps: Examples

In all cases, $s = \{a \mapsto 0, b \mapsto 0, c \mapsto 1, d \mapsto 0\}$.

- $\sigma = \{\langle c, a\rangle, \langle \top, b\rangle\}$
- $\sigma = \{\langle c, a\rangle, \langle c, a \rhd b\rangle\}$
- $\sigma = \{\langle c, a \wedge b\rangle, \langle a, b \rhd d\rangle\}$
- $\sigma = \{\langle c, a \wedge (b \rhd d)\rangle, \langle c, b \wedge (a \rhd d)\rangle\}$

Applying plan steps: Examples

In all cases, $s = \{a \mapsto 0, b \mapsto 0, c \mapsto 1, d \mapsto 0\}$.

- $\sigma = \{\langle c, a\rangle, \langle \top, b\rangle\}$
- $\sigma = \{\langle c, a\rangle, \langle c, a \rhd b\rangle\}$
- $\sigma = \{\langle c, a \wedge b\rangle, \langle a, b \rhd d\rangle\}$
- $\sigma = \{\langle c, a \wedge (b \rhd d)\rangle, \langle c, b \wedge (a \rhd d)\rangle\}$

Serializations

Applying a plan step to a state is related to applying the actions in the step to a state in sequence.

Definition

A serialization of plan step $\sigma = \{o_1, \dots, o_n\}$ is a sequence $o_{\pi(1)}, \dots, o_{\pi(n)}$ where π is a permutation of $\{1, \dots, n\}$.

Lemma

If σ is a plan step applicable in a state s of a relaxed planning task, then each serialization $o'_1, \dots, o'_n$ of σ is applicable in s and $app_{o'_1,\dots,o'_n}(s)$ dominates $app_\sigma(s)$.

- Does equality hold for all serializations?
- Does equality hold for some serialization?
- What if there are no conditional effects?
- What if the planning task is not relaxed?

Serializations

Applying a plan step to a state is related to applying the actions in the step to a state in sequence.

Definition

A serialization of plan step $\sigma = \{o_1, \ldots, o_n\}$ is a sequence $o_{\pi(1)}, \ldots, o_{\pi(n)}$ where π is a permutation of $\{1, \ldots, n\}$.

Lemma

If σ is a plan step applicable in a state s of a relaxed planning task, then each serialization $o'_1, \ldots, o'_n$ of σ is applicable in s and $app_{o'_1,\ldots,o'_n}(s)$ dominates $app_\sigma(s)$.

- Does equality hold for all serializations?
- Does equality hold for some serialization?
- What if there are no conditional effects?
- What if the planning task is not relaxed?

Serializations

Applying a plan step to a state is related to applying the actions in the step to a state in sequence.

Definition

A serialization of plan step $\sigma = \{o_1, \ldots, o_n\}$ is a sequence $o_{\pi(1)}, \ldots, o_{\pi(n)}$ where π is a permutation of $\{1, \ldots, n\}$.

Lemma

If σ is a plan step applicable in a state s of a relaxed planning task, then each serialization $o'_1, \ldots, o'_n$ of σ is applicable in s and app$_{o'_1,\ldots,o'_n}(s)$ dominates app$_\sigma(s)$.

- Does equality hold for all serializations?
- Does equality hold for some serialization?
- What if there are no conditional effects?
- What if the planning task is not relaxed?

Serializations

Applying a plan step to a state is related to applying the actions in the step to a state in sequence.

Definition

A serialization of plan step $\sigma = \{o_1, \dots, o_n\}$ is a sequence $o_{\pi(1)}, \dots, o_{\pi(n)}$ where π is a permutation of $\{1, \dots, n\}$.

Lemma

If σ is a plan step applicable in a state s of a relaxed planning task, then each serialization $o'_1, \dots, o'_n$ of σ is applicable in s and app$_{o'_1,\dots,o'_n}(s)$ dominates app$_\sigma(s)$.

- Does equality hold for all serializations?
- Does equality hold for some serialization?
- What if there are no conditional effects?
- What if the planning task is not relaxed?

Serializations

Applying a plan step to a state is related to applying the actions in the step to a state in sequence.

Definition

A serialization of plan step $\sigma = \{o_1, \dots, o_n\}$ is a sequence $o_{\pi(1)}, \dots, o_{\pi(n)}$ where π is a permutation of $\{1, \dots, n\}$.

Lemma

If σ is a plan step applicable in a state s of a relaxed planning task, then each serialization $o'_1, \dots, o'_n$ of σ is applicable in s and $app_{o'_1,\dots,o'_n}(s)$ dominates $app_\sigma(s)$.

- Does equality hold for all serializations?
- Does equality hold for some serialization?
- What if there are no conditional effects?
- What if the planning task is not relaxed?

Serializations

Applying a plan step to a state is related to applying the actions in the step to a state in sequence.

Definition

A serialization of plan step $\sigma = \{o_1, \dots, o_n\}$ is a sequence $o_{\pi(1)}, \dots, o_{\pi(n)}$ where π is a permutation of $\{1, \dots, n\}$.

Lemma

If σ is a plan step applicable in a state s of a relaxed planning task, then each serialization $o'_1, \dots, o'_n$ of σ is applicable in s and app$_{o'_1,\dots,o'_n}(s)$ dominates app$_\sigma(s)$.

- Does equality hold for all serializations?
- Does equality hold for some serialization?
- What if there are no conditional effects?
- What if the planning task is not relaxed?

Parallel plans

Definition

A parallel plan for a relaxed planning task $\langle A, I, O^+, G\rangle$ is a sequence of plan steps $\sigma_1, \dots, \sigma_n$ of operators in O^+ with:

- $s_0 := I$
- For $i = 1, \dots, n$, step σ_i is applicable in s_{i-1} and $s_i := app_{\sigma_i}(s_{i-1})$.
- $s_n \models G$

Remark: By ordering the operators within each single step arbitrarily, we obtain a (regular, non-parallel) plan.

Parallel plans

Definition

A parallel plan for a relaxed planning task $\langle A, I, O^+, G \rangle$ is a sequence of plan steps $\sigma_1, \dots, \sigma_n$ of operators in O^+ with:

- $s_0 := I$
- For $i = 1, \dots, n$, step σ_i is applicable in s_{i-1} and $s_i := app_{\sigma_i}(s_{i-1})$.
- $s_n \models G$

Remark: By ordering the operators within each single step arbitrarily, we obtain a (regular, non-parallel) plan.

Forward states and operator sets

Idea: In the forward phase of the heuristic computation, we first apply the plan step consisting of all initially applicable operators, then the plan step consisting of all operators applicable in the resulting state, etc.

Definition

The 0-th parallel forward state, in symbols s_0^{F}, of a relaxed planning task $\langle A, I, O^+, G\rangle$ is defined as $s_0^{\mathsf{F}} := s$.
For $n \in \mathbb{N}_1$, the n-th forward plan step, in symbols σ_n^{F}, is the set of operators applicable in s_{n-1}^{F}, and the n-th parallel forward state, in symbols s_n^{F}, is defined as $s_n^{\mathsf{F}} := app_{\sigma_n^{\mathsf{F}}}(s_{n-1}^{\mathsf{F}})$.
For $n \in \mathbb{N}_0$, the n-th parallel forward set, in symbols S_n^{F}, is defined as $S_n^{\mathsf{F}} := on(s_n^{\mathsf{F}})$.

Forward states and operator sets

Idea: In the forward phase of the heuristic computation, we first apply the plan step consisting of all initially applicable operators, then the plan step consisting of all operators applicable in the resulting state, etc.

Definition

The 0-th parallel forward state, in symbols s_0^{F}, of a relaxed planning task $\langle A, I, O^+, G\rangle$ is defined as $s_0^{\mathsf{F}} := s$.
For $n \in \mathbb{N}_1$, the n-th forward plan step, in symbols σ_n^{F}, is the set of operators applicable in s_{n-1}^{F}, and the n-th parallel forward state, in symbols s_n^{F}, is defined as $s_n^{\mathsf{F}} := \mathit{app}_{\sigma_n^{\mathsf{F}}}(s_{n-1}^{\mathsf{F}})$.
For $n \in \mathbb{N}_0$, the n-th parallel forward set, in symbols S_n^{F}, is defined as $S_n^{\mathsf{F}} := \mathit{on}(s_n^{\mathsf{F}})$.

Parallel forward distances

Definition

The parallel forward distance of a relaxed planning task $\langle A, I, O^+, G\rangle$ is the lowest number $n \in \mathbb{N}_0$ such that $s^{\mathsf{F}}_n \models G$, or ∞ if no parallel forward state satisfies G.

Remark: The parallel forward distance can be computed in polynomial time. (How?)

Definition

The $h_{\max}$ estimate of a state s in a planning task $\mathcal{P} = \langle A, I, O, G\rangle$ in positive normal form is the parallel forward distance of the relaxed planning task $\langle A, s, O^+, G\rangle$.

Remark: The $h_{\max}$ estimate is admissible. (Why?)

Parallel forward distances

Definition

The parallel forward distance of a relaxed planning task $\langle A, I, O^+, G\rangle$ is the lowest number $n \in \mathbb{N}_0$ such that $s_n^{\mathsf{F}} \models G$, or ∞ if no parallel forward state satisfies G.

Remark: The parallel forward distance can be computed in polynomial time. (How?)

Definition

The h_{max} estimate of a state s in a planning task $\mathcal{P} = \langle A, I, O, G\rangle$ in positive normal form is the parallel forward distance of the relaxed planning task $\langle A, s, O^+, G\rangle$.

Remark: The h_{max} estimate is admissible. (Why?)

Parallel forward distances

Definition

The parallel forward distance of a relaxed planning task $\langle A, I, O^+, G \rangle$ is the lowest number $n \in \mathbb{N}_0$ such that $s_n^{\mathsf{F}} \models G$, or ∞ if no parallel forward state satisfies G.

Remark: The parallel forward distance can be computed in polynomial time. (How?)

Definition

The $h_{\max}$ estimate of a state s in a planning task $\mathcal{P} = \langle A, I, O, G \rangle$ in positive normal form is the parallel forward distance of the relaxed planning task $\langle A, s, O^+, G \rangle$.

Remark: The $h_{\max}$ estimate is admissible. (Why?)

Parallel forward distances

Definition

The parallel forward distance of a relaxed planning task $\langle A, I, O^+, G\rangle$ is the lowest number $n \in \mathbb{N}_0$ such that $s_n^{\mathsf{F}} \models G$, or ∞ if no parallel forward state satisfies G.

Remark: The parallel forward distance can be computed in polynomial time. (How?)

Definition

The h_{max} estimate of a state s in a planning task $\mathcal{P} = \langle A, I, O, G\rangle$ in positive normal form is the parallel forward distance of the relaxed planning task $\langle A, s, O^+, G\rangle$.

Remark: The h_{max} estimate is admissible. (Why?)

So far, so good…

- We have seen how systematic computation of forward states leads to an admissible estimate for heuristic planning.
- However, this estimate is very coarse.
- To improve it, we need to include backward propagation of information.

For this purpose, we use a data structure called a relaxed planning graph.

Relaxed planning graphs: running example

As a running example, consider the relaxed planning task $\langle A, I, \{o_1, o_2, o_3, o_4\}, G\rangle$ with

$$\begin{aligned}
A &= \{a, b, c, d, e, f, g, h\} \\
I &= \{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 1, \\
&\qquad e \mapsto 0, f \mapsto 0, g \mapsto 0, h \mapsto 0\} \\
o_1 &= \langle b \vee (c \wedge d), b \wedge ((a \wedge b) \rhd e)\rangle \\
o_2 &= \langle \top, f\rangle \\
o_3 &= \langle f, g\rangle \\
o_4 &= \langle f, h\rangle \\
G &= e \wedge (g \wedge h)
\end{aligned}$$

Relaxed planning graphs

Relaxed planning graphs encode

- which propositions can be made true in a given number of plan steps,
- and how they can be made true.

They consist of four kinds of components:

- Proposition nodes represent the truth value of propositions after applying a certain number of plan steps.
- Idle arcs represent state variables that do not change their value when applying a plan step.
- Action subgraphs represent the application of a given action in a given plan step.
- Goal subgraphs represent the truth value of the goal condition after applying a certain number of plan steps.

Relaxed planning graph: proposition layers

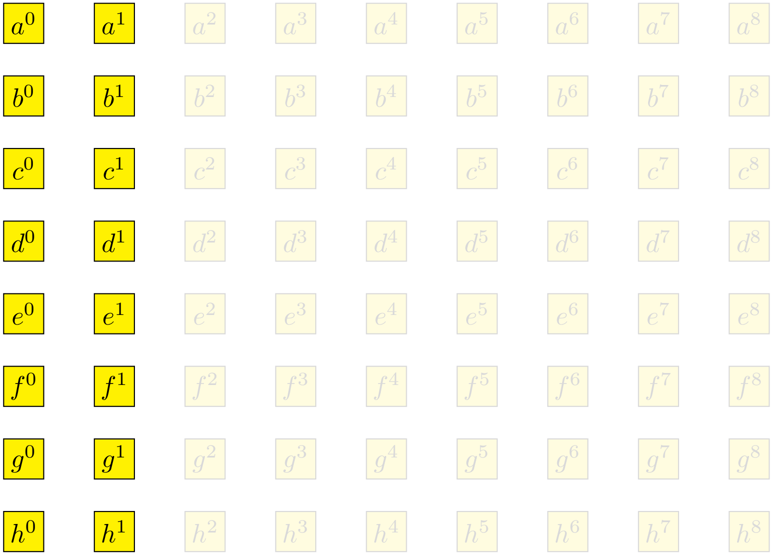
Let $\mathcal{P} = \langle A, I, O^+, G \rangle$ be a relaxed planning task, let $N \in \mathbb{N}_0$.

For each $i \in \{0, \dots, N\}$, the relaxed planning graph of depth N contains one proposition layer which consists of:

- a proposition node a^i for each state variable $a \in A$.

Relaxed planning graph: proposition layers

Relaxed planning graph: proposition layers



Relaxed planning graph: proposition layers

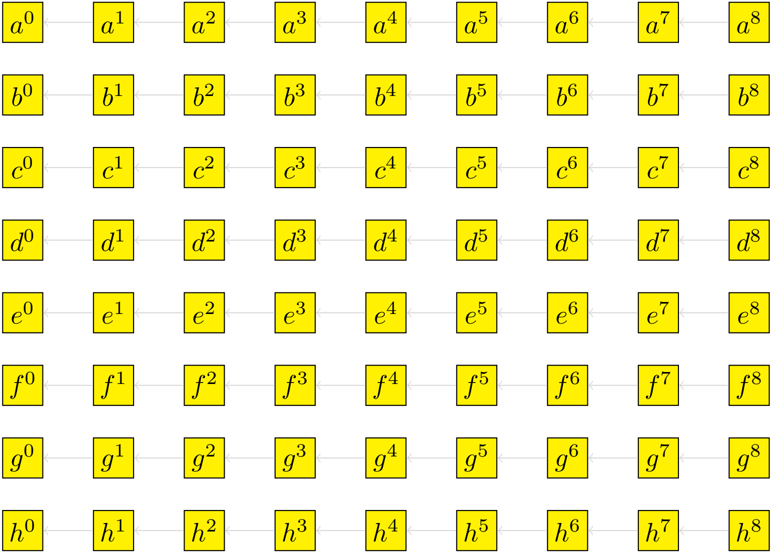
a^0	a^1	a^2	a^3	a^4	a^5	a^6	a^7	a^8
b^0	b^1	b^2	b^3	b^4	b^5	b^6	b^7	b^8
c^0	c^1	c^2	c^3	c^4	c^5	c^6	c^7	c^8
d^0	d^1	d^2	d^3	d^4	d^5	d^6	d^7	d^8
e^0	e^1	e^2	e^3	e^4	e^5	e^6	e^7	e^8
f^0	f^1	f^2	f^3	f^4	f^5	f^6	f^7	f^8
g^0	g^1	g^2	g^3	g^4	g^5	g^6	g^7	g^8
h^0	h^1	h^2	h^3	h^4	h^5	h^6	h^7	h^8

Relaxed planning graph: idle arcs

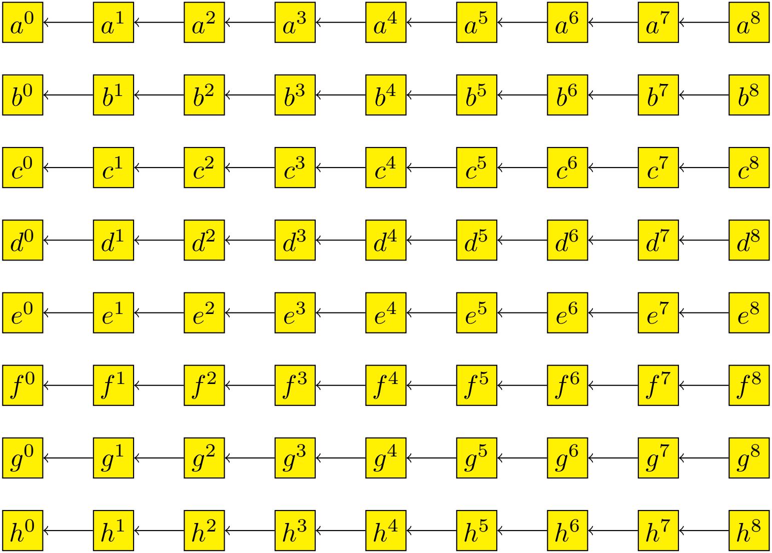
For each proposition node a^i with $i \in \{1, \dots, N\}$, the relaxed planning graph of depth N contains an arc from a^i to a^{i-1} (idle arcs).

Intuition: If a state variable is true in step i, one of the possible reasons is that it was already previously true.

Relaxed planning graph: idle arcs



Relaxed planning graph: idle arcs



Relaxed planning graph: action subgraphs

For each $i \in \{1, \dots, N\}$ and each operator $o = \langle c, e \rangle \in O$, the relaxed planning graph of depth N contains a subgraph called an action subgraph with the following parts:

- one formula node n^i_χ for each formula χ which is a subformula of c or of some effect condition in e:
  - If $\chi = a$ for some atom a, n^i_χ is the proposition node a^{i-1}.
  - If $\chi = \top$, n^i_χ is a new node labeled $(\top)$.
  - If $\chi = \bot$, n^i_χ is a new node labeled $(\bot)$.
  - If $\chi = (\phi \wedge \psi)$, n^i_χ is a new node labeled $(\wedge)$ with outgoing arcs to n^i_ϕ and n^i_ψ.
  - If $\chi = (\phi \vee \psi)$, n^i_χ is a new node labeled $(\vee)$ with outgoing arcs to n^i_ϕ and n^i_ψ.

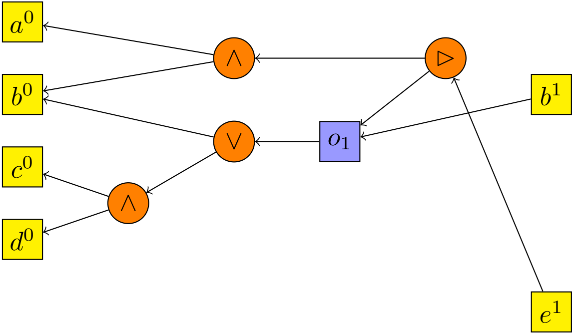
Relaxed planning graph: action subgraphs

For each $i \in \{1, \dots, N\}$ and each operator $o = \langle c, e \rangle \in O$, the relaxed planning graph of depth N contains a subgraph called an action subgraph with the following parts:

- an action node o^i with an outgoing arc to the precondition formula node n_c^i
- for each add effect a in e which does not occur within a conditional effect, an arc from proposition node a^{i+1} to action node o^i.
- for each conditional effect $(c' \rhd a)$ in e, a node $n^i_{c' \rhd a}$ labeled $(\rhd)$ with an incoming arc from proposition node a^{i+1} and outgoing arcs to action node o^i and formula node $n^i_{c'}$.

Relaxed planning graph: action subgraphs

Action subgraph for $o_1 = \langle b \vee (c \wedge d), b \wedge ((a \wedge b) \rhd e)\rangle$ for layer $i = 0$.



Relaxed planning graph: goal subgraphs

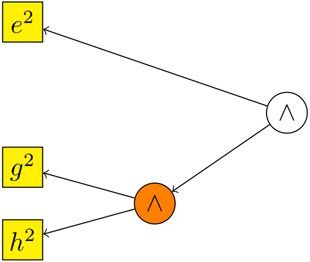
For each $i \in \{0, \ldots, N\}$, the relaxed planning graph of depth N contains a subgraph called a goal subgraph with the following parts:

- one formula node n^i_χ for each formula χ which is a subformula of G:
  - If $\chi = a$ for some atom a, n^i_χ is the proposition node a^i.
  - If $\chi = \top$, n^i_χ is a new node labeled $(\top)$.
  - If $\chi = \bot$, n^i_χ is a new node labeled $(\bot)$.
  - If $\chi = (\phi \wedge \psi)$, n^i_χ is a new node labeled $(\wedge)$ with outgoing arcs to n^i_ϕ and n^i_ψ.
  - If $\chi = (\phi \vee \psi)$, n^i_χ is a new node labeled $(\vee)$ with outgoing arcs to n^i_ϕ and n^i_ψ.

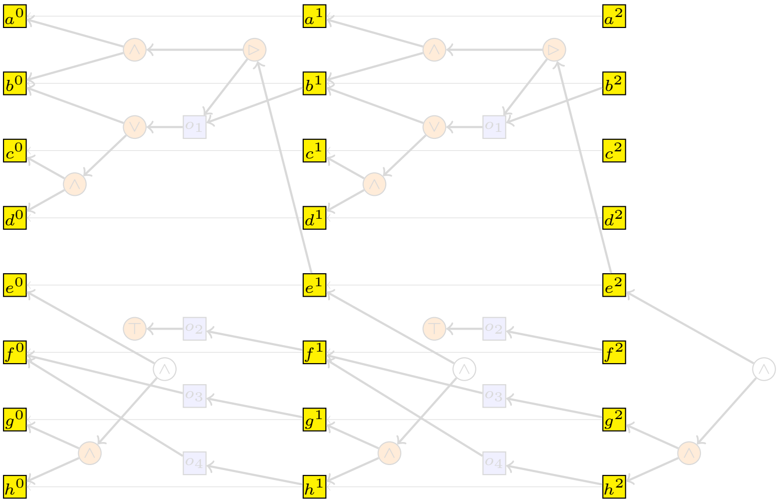
The node n^i_G is called a goal node.

Relaxed planning graph: goal subgraphs

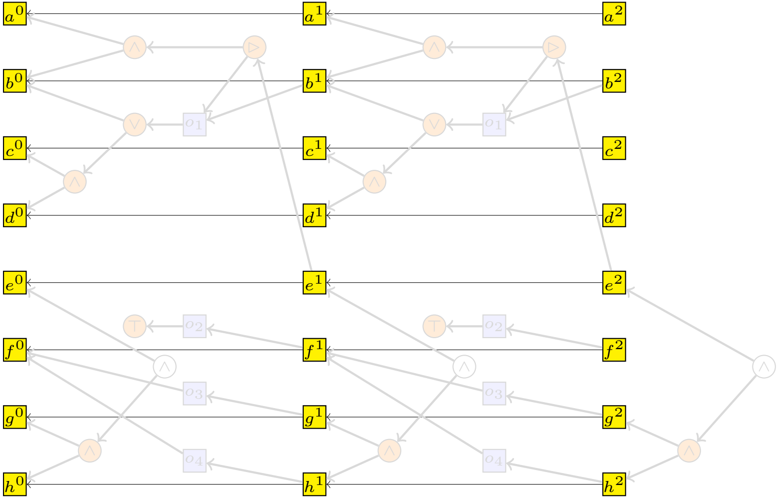
Goal subgraph for $G = e \wedge (g \wedge h)$
and layer $i = 2$:



Relaxed planning graph: complete example
(depth 2)

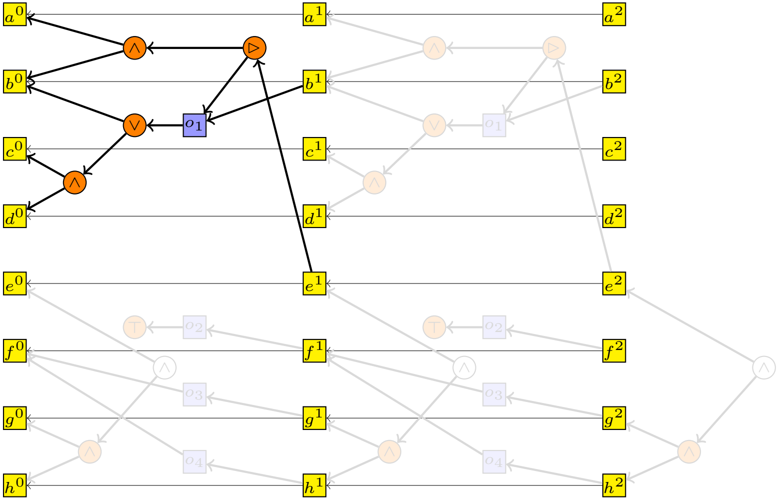


Relaxed planning graph: complete example
(depth 2)



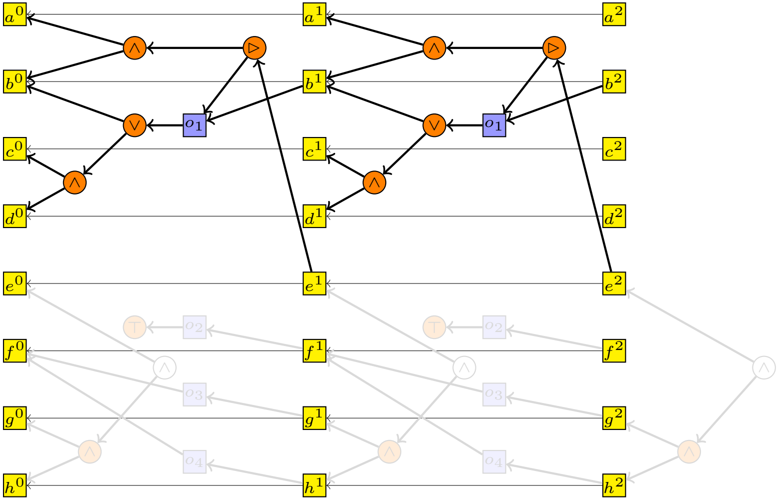
Relaxed planning graph: complete example
(depth 2)

$$o_1 = \langle b \vee (c \wedge d), b \wedge ((a \wedge b) \rhd e) \rangle$$



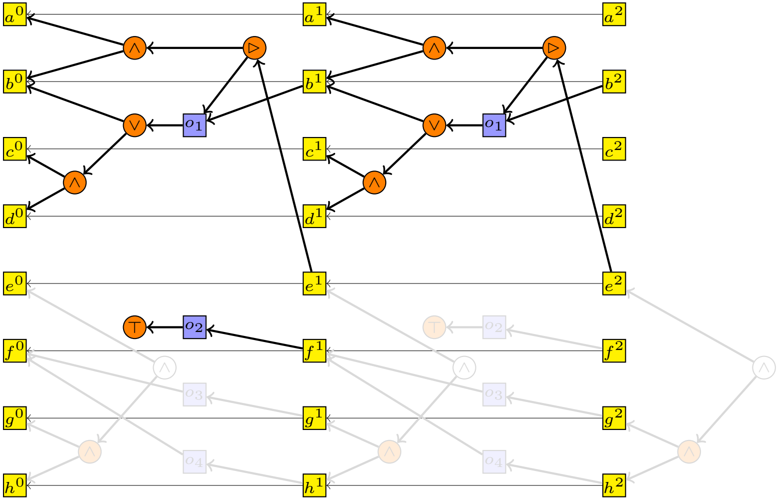
Relaxed planning graph: complete example
(depth 2)

$$o_1 = \langle b \vee (c \wedge d), b \wedge ((a \wedge b) \rhd e) \rangle$$

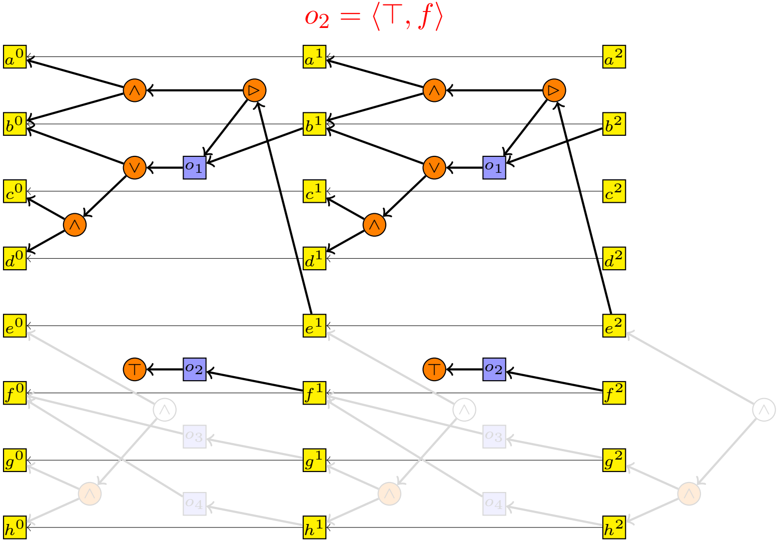


Relaxed planning graph: complete example
(depth 2)

$o_2 = \langle \top, f \rangle$

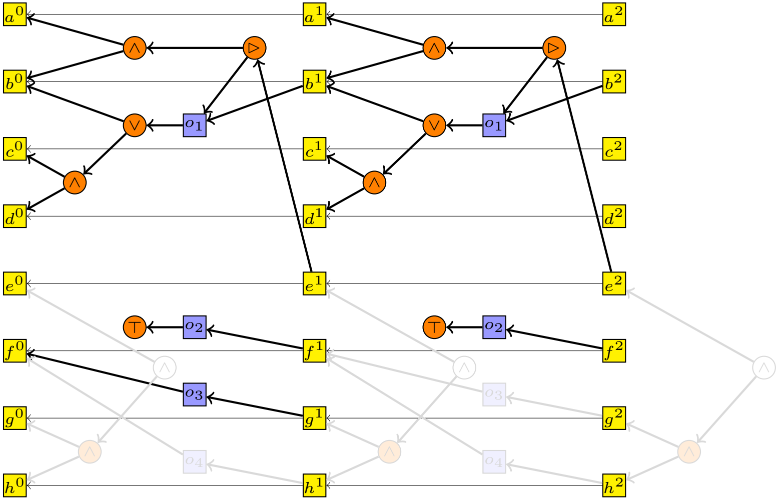


Relaxed planning graph: complete example
(depth 2)



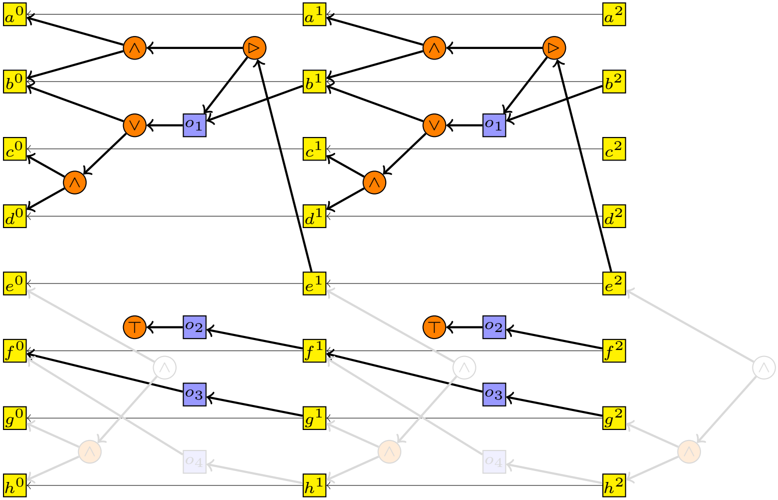
Relaxed planning graph: complete example
(depth 2)

$o_3 = \langle f, g \rangle$



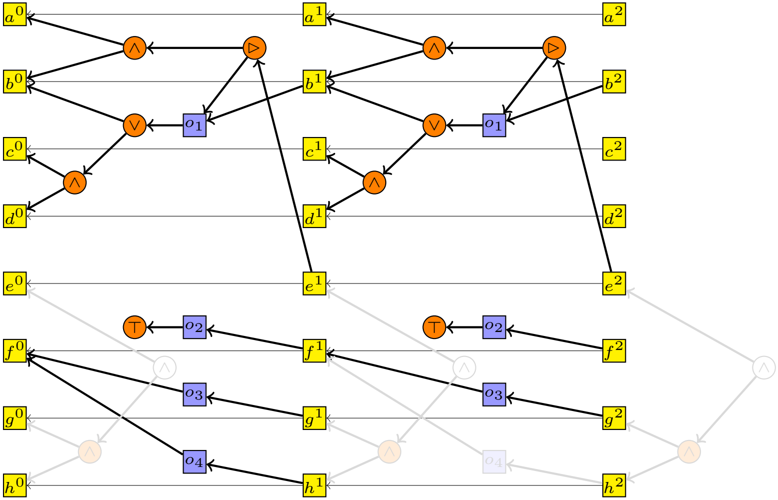
Relaxed planning graph: complete example
(depth 2)

$o_3 = \langle f, g \rangle$



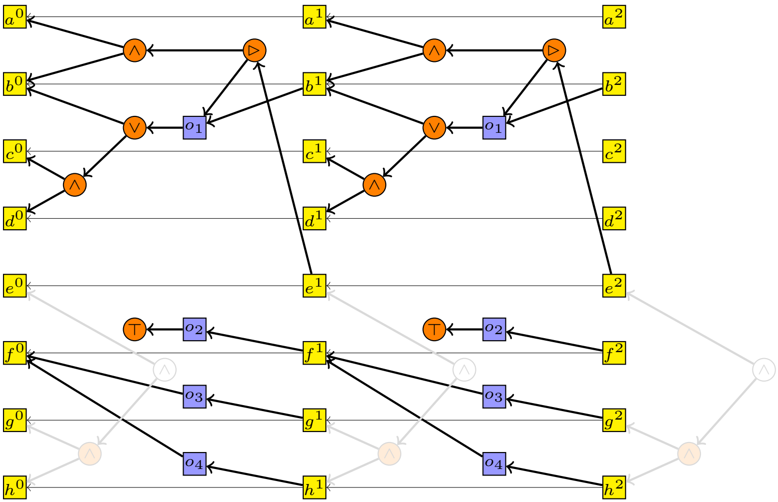
Relaxed planning graph: complete example
(depth 2)

$$o_4 = \langle f, h \rangle$$



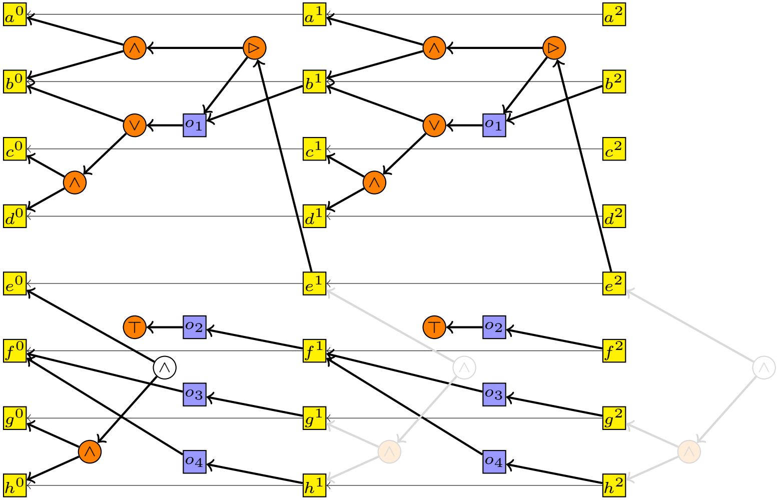
Relaxed planning graph: complete example
(depth 2)

$o_4 = \langle f, h \rangle$



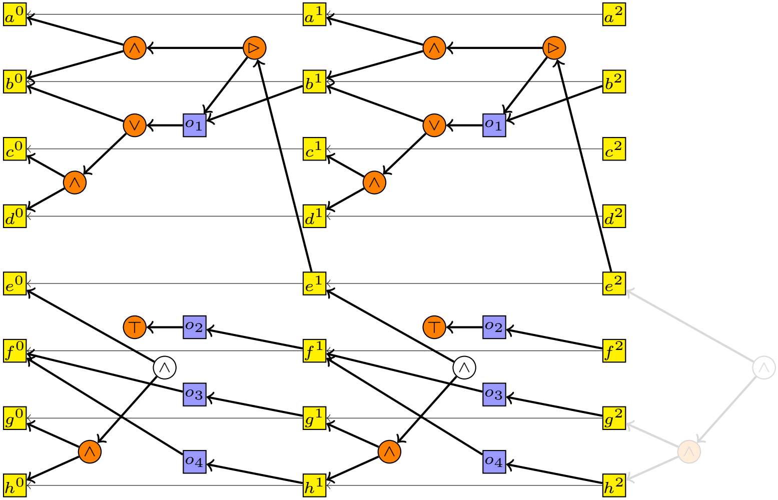
Relaxed planning graph: complete example
(depth 2)

$$G = e \land (g \land h)$$



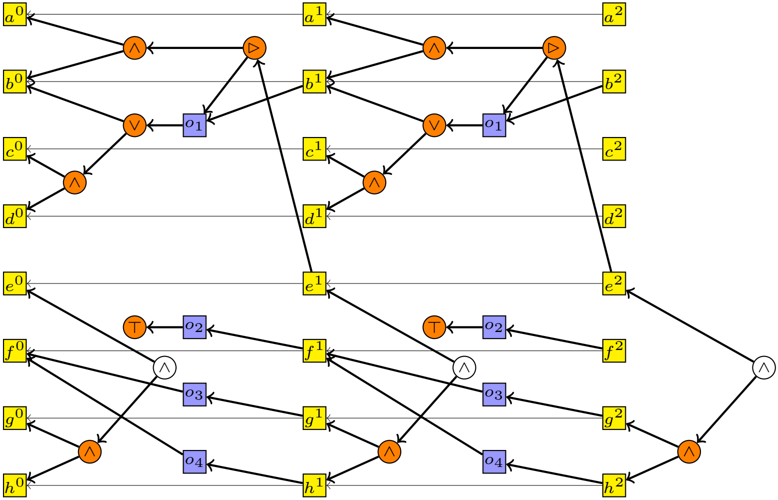
Relaxed planning graph: complete example
(depth 2)

$G = e \land (g \land h)$

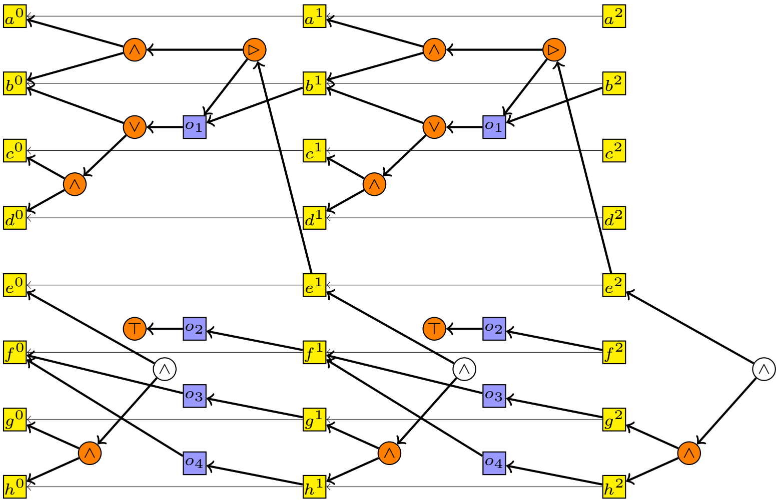


Relaxed planning graph: complete example
(depth 2)

$G = e \wedge (g \wedge h)$



Relaxed planning graph: complete example
(depth 2)



Boolean circuits

Definition

A Boolean circuit is a directed acyclic graph $G = (V, E)$, where the nodes V are called gates. Each gate $v \in V$ has a type $type(v) \in \{\neg, \wedge, \vee, \top, \bot, \} \cup \{a, b, c, \ldots\}$. The gates with $type(v) \in \{\top, \bot, a, b, c, \ldots\}$ have in-degree zero, the gates with $type(v) \in \{\neg\}$ have in-degree one, and the gates with $type(v) \in \{\vee, \wedge\}$ have in-degree two. The gates with no outgoing edge are called output gates. The gates with no incoming edges are called input gates.

Definition

Given a value assignment to the input gates, the circuit computes the value of gates in the obvious way.

What is the relation between circuits and relaxed planning graphs?

Boolean circuits

Definition

A Boolean circuit is a directed acyclic graph $G = (V, E)$, where the nodes V are called gates. Each gate $v \in V$ has a type $type(v) \in \{\neg, \wedge, \vee, \top, \bot, \} \cup \{a, b, c, \dots\}$. The gates with $type(v) \in \{\top, \bot, a, b, c, \dots\}$ have in-degree zero, the gates with $type(v) \in \{\neg\}$ have in-degree one, and the gates with $type(v) \in \{\vee, \wedge\}$ have in-degree two. The gates with no outgoing edge are called output gates. The gates with no incoming edges are called input gates.

Definition

Given a value assignment to the input gates, the circuit computes the value of gates in the obvious way.

What is the relation between circuits and relaxed planning graphs?

Boolean circuits

Definition

A Boolean circuit is a directed acyclic graph $G = (V, E)$, where the nodes V are called gates. Each gate $v \in V$ has a type $type(v) \in \{\neg, \wedge, \vee, \top, \bot, \} \cup \{a, b, c, \ldots\}$. The gates with $type(v) \in \{\top, \bot, a, b, c, \ldots\}$ have in-degree zero, the gates with $type(v) \in \{\neg\}$ have in-degree one, and the gates with $type(v) \in \{\vee, \wedge\}$ have in-degree two. The gates with no outgoing edge are called output gates. The gates with no incoming edges are called input gates.

Definition

Given a value assignment to the input gates, the circuit computes the value of gates in the obvious way.

What is the relation between circuits and relaxed planning graphs?

Relaxed planning graphs and Boolean circuits

Observations:

- Relaxed planning graphs can be understood as special (*monotone*) Boolean circuits, where
  - the direction of the arrows has to be *inverted*
  - proposition nodes in the 0-th layer are $\perp$ gates or $\top$ gates, depending on their initial value.
  - proposition nodes outside the 0-th layer are $\vee$ gates
  - action nodes are $\wedge$ gates (or $\vee$ gates)
  - $\triangleright$-nodes are $\wedge$ gates
- A parallel plan solves the planning task with n steps iff the value of the goal node on layer n has the value 1.
- The plan consists of all action nodes that have a value of 1 and that are "on a path to the goal node", which has a value of 1.

Relaxed planning graphs and Boolean circuits

Observations:

- Relaxed planning graphs can be understood as special (*monotone*) Boolean circuits, where
  - the direction of the arrows has to be *inverted*
  - proposition nodes in the 0-th layer are $\perp$ gates or $\top$ gates, depending on their initial value.
  - proposition nodes outside the 0-th layer are $\vee$ gates
  - action nodes are $\wedge$ gates (or $\vee$ gates)
  - $\triangleright$-nodes are $\wedge$ gates
- A parallel plan solves the planning task with n steps iff the value of the goal node on layer n has the value 1.
- The plan consists of all action nodes that have a value of 1 and that are “on a path to the goal node”, which has a value of 1.

Relaxed planning graphs and Boolean circuits

Observations:

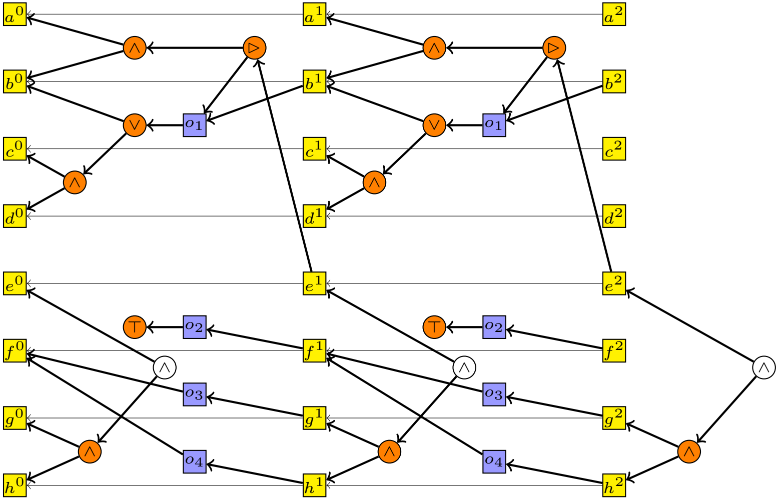
- Relaxed planning graphs can be understood as special (*monotone*) Boolean circuits, where
  - the direction of the arrows has to be *inverted*
  - proposition nodes in the 0-th layer are $\bot$ gates or $\top$ gates, depending on their initial value.
  - proposition nodes outside the 0-th layer are $\vee$ gates
  - action nodes are $\wedge$ gates (or $\vee$ gates)
  - $\triangleright$-nodes are $\wedge$ gates
- A parallel plan solves the planning task with n steps iff the value of the goal node on layer n has the value 1.
- The plan consists of all action nodes that have a value of 1 and that are "on a path to the goal node", which has a value of 1.

Relaxed planning graphs and Boolean circuits

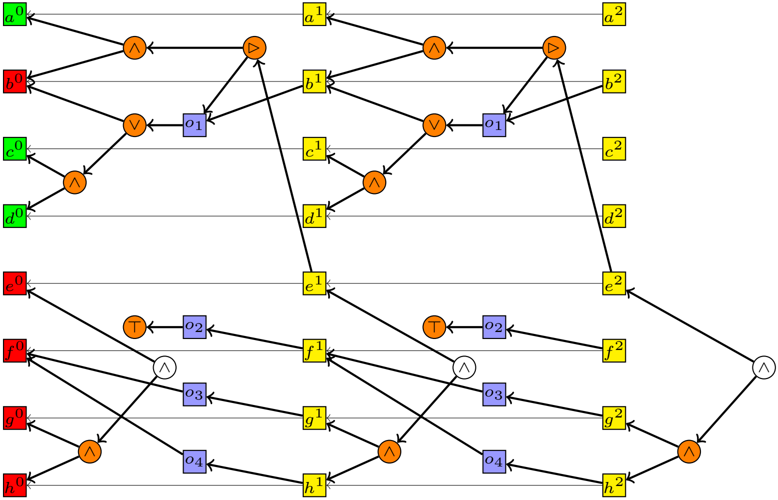
Observations:

- Relaxed planning graphs can be understood as special (*monotone*) Boolean circuits, where
  - the direction of the arrows has to be *inverted*
  - proposition nodes in the 0-th layer are $\perp$ gates or $\top$ gates, depending on their initial value.
  - proposition nodes outside the 0-th layer are $\vee$ gates
  - action nodes are $\wedge$ gates (or $\vee$ gates)
  - $\triangleright$-nodes are $\wedge$ gates
- A parallel plan solves the planning task with n steps iff the value of the goal node on layer n has the value 1.
- The plan consists of all action nodes that have a value of 1 and that are "on a path to the goal node", which has a value of 1.

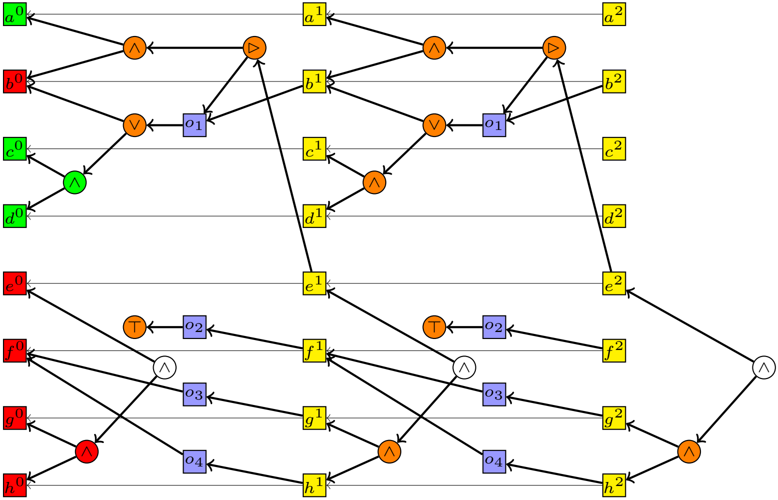
Computing gate values



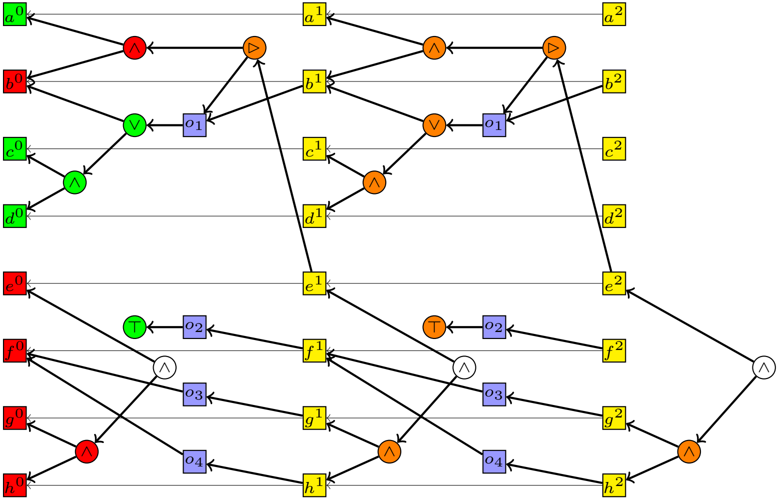
Computing gate values



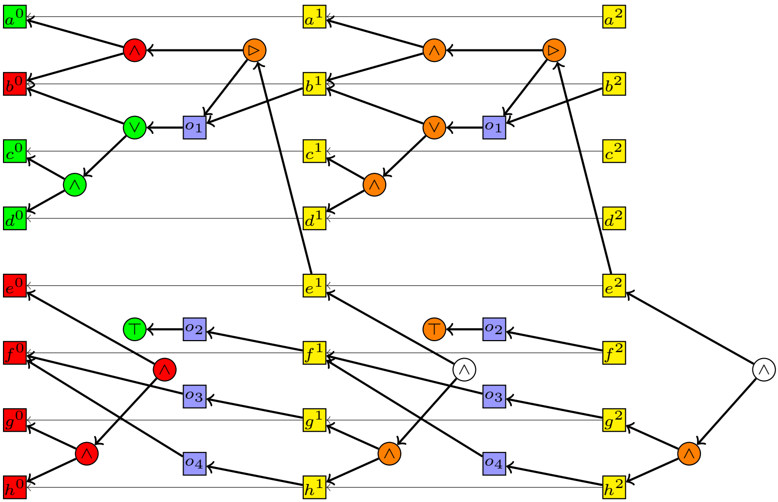
Computing gate values



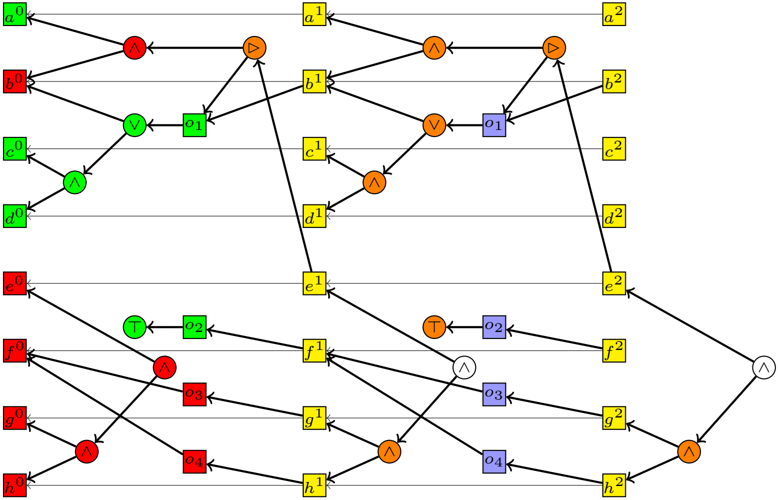
Computing gate values



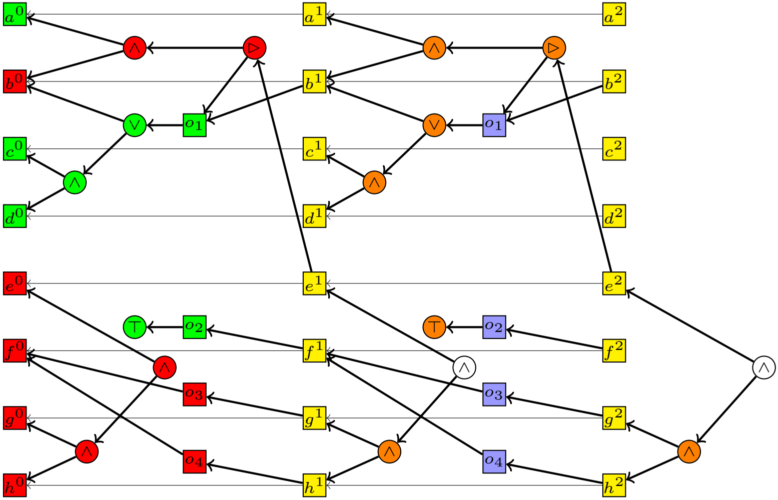
Computing gate values



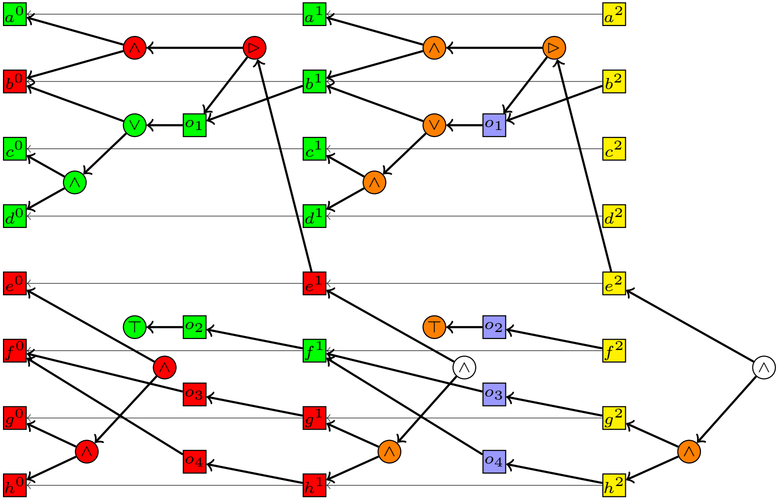
Computing gate values



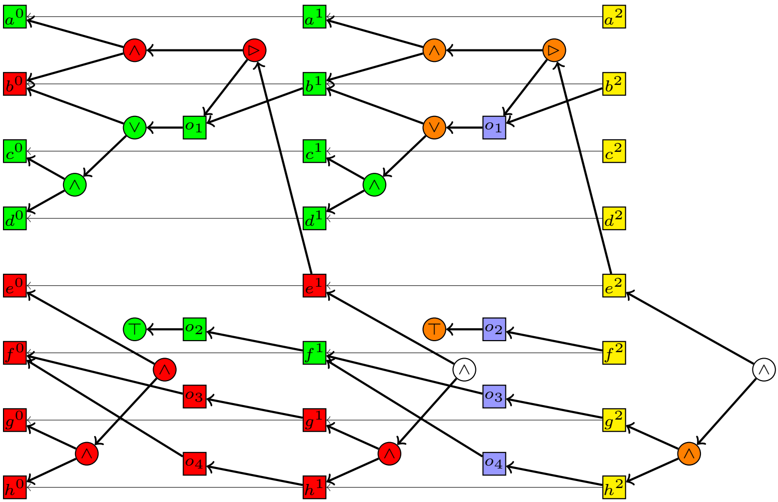
Computing gate values



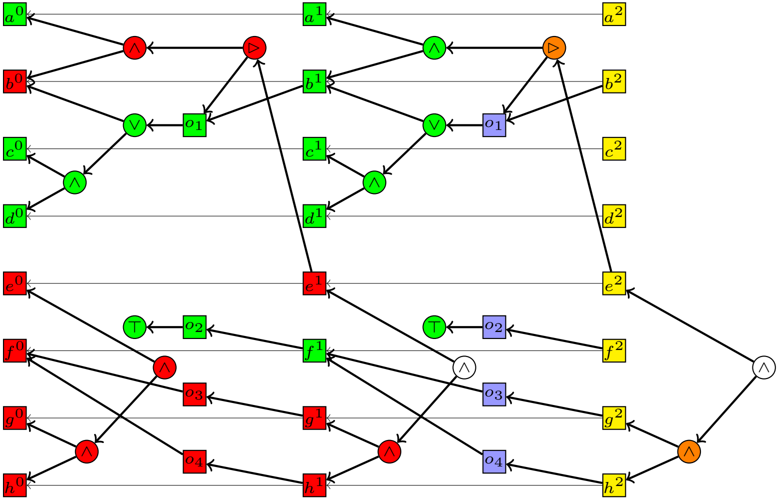
Computing gate values



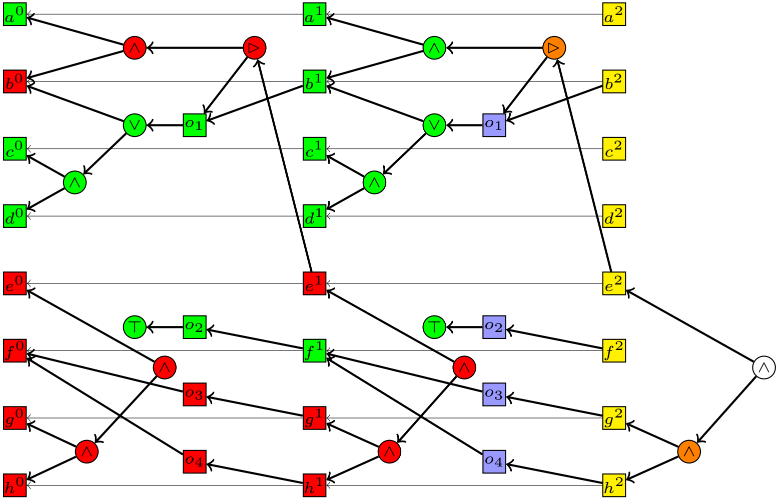
Computing gate values



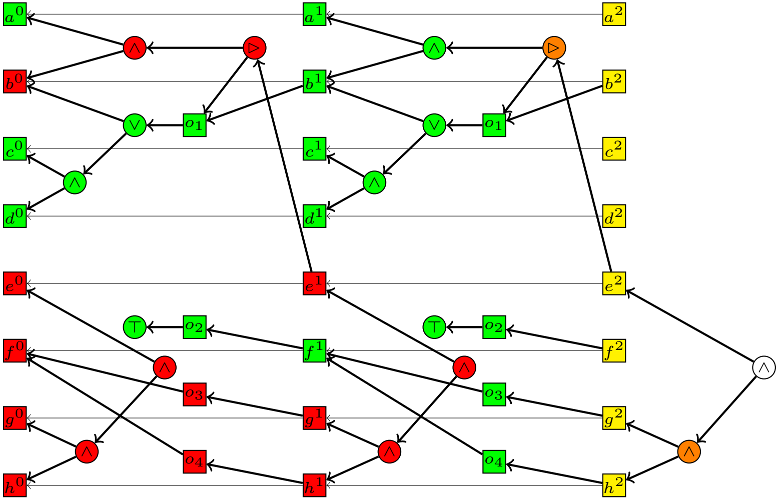
Computing gate values



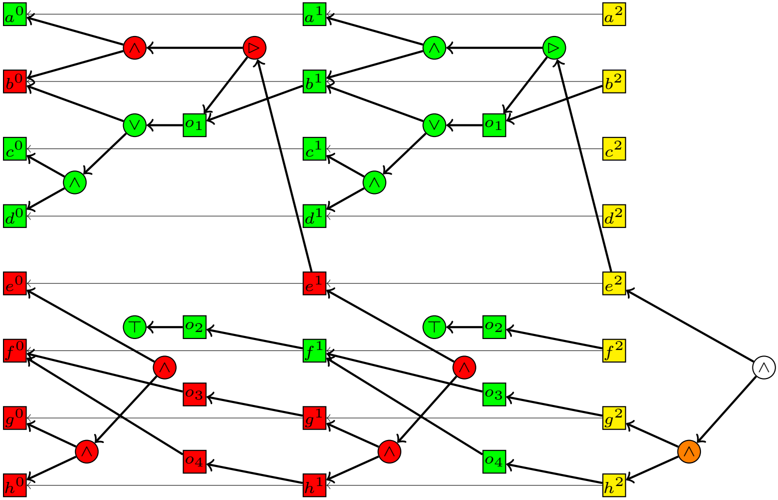
Computing gate values



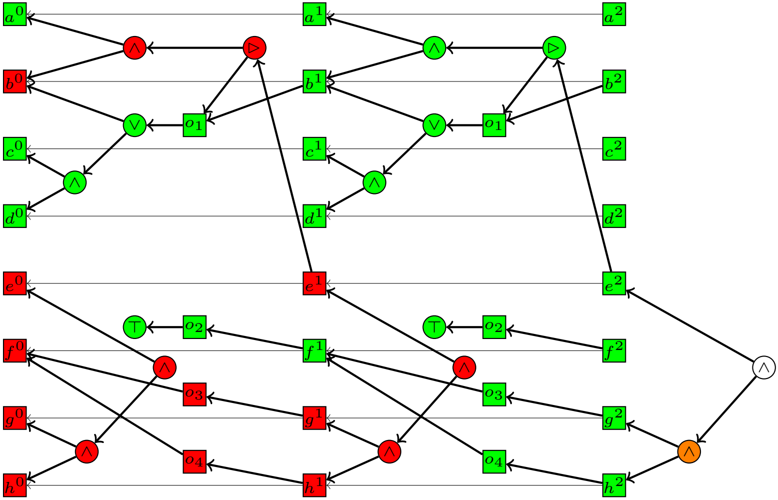
Computing gate values



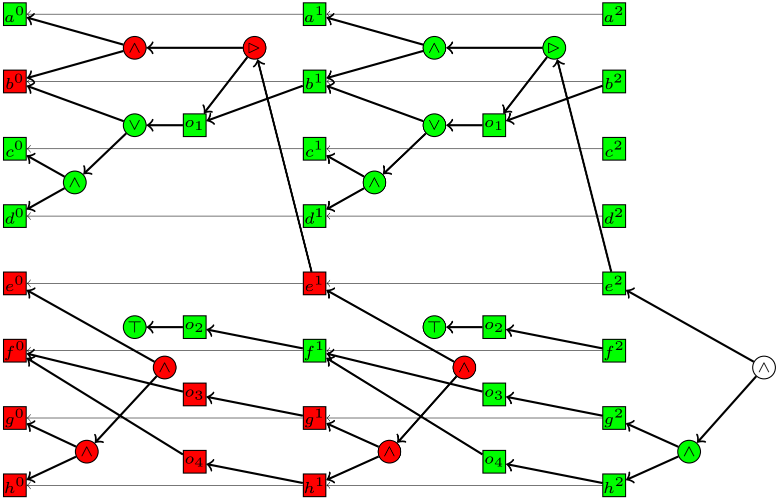
Computing gate values



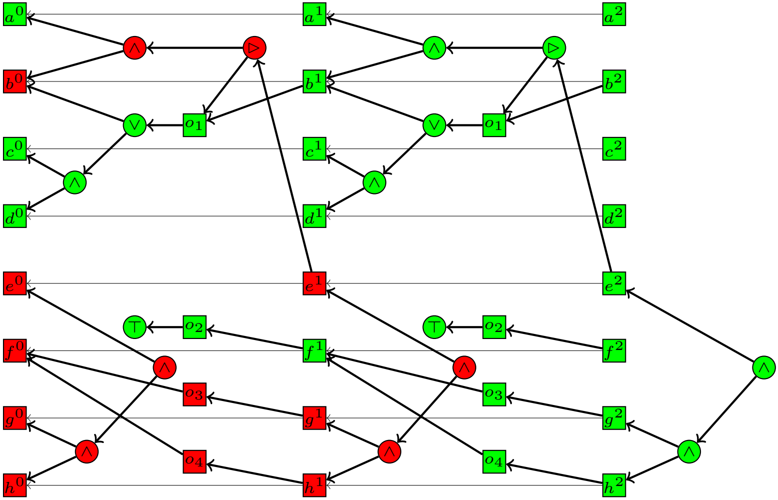
Computing gate values



Computing gate values



Computing gate values



Heuristic estimate h_{max}

- Using relaxed planning graphs, how can we compute the h_{max} heuristic?
- Solution:
  - Create relaxed planning graph of depth n, n being the number of state variables.
  - Compute gate values based on initial state values.
  - The h_{max} value is the lowest layer where the goal gate evaluates to 1!

Heuristic estimate h_{max}

- Using relaxed planning graphs, how can we compute the h_{max} heuristic?
- Solution:
  - Create relaxed planning graph of depth n, n being the number of state variables.
  - Compute gate values based on initial state values.
  - The h_{max} value is the lowest layer where the goal gate evaluates to 1!

Heuristic estimate h_{max}

- Using relaxed planning graphs, how can we compute the h_{max} heuristic?
- Solution:
  - Create relaxed planning graph of depth n, n being the number of state variables.
  - Compute gate values based on initial state values.
  - The h_{max} value is the lowest layer where the goal gate evaluates to 1!

Heuristic estimate h_{max}

- Using relaxed planning graphs, how can we compute the h_{max} heuristic?
- Solution:
  - Create relaxed planning graph of depth n, n being the number of state variables.
  - Compute gate values based on initial state values.
  - The h_{max} value is the lowest layer where the goal gate evaluates to 1!

Heuristic estimate h_{add}

- While h_{max} is admissible, it is not very informative (it does not distinguish between different states).
- Estimate how hard it is to make a proposition true.
- Estimate costs under the assumption that making a proposition true is independent from making any other proposition true (i.e., relaxed planning graph is a tree).
  - Any proposition already true in the *initial state* has cost 0.
  - *Conjunctions* (if true) have the sum of the costs of the conjunctions.
  - *Disjunctions* (if true) have the minimum of the costs of the disjuncts.
  - *Actions* (if executable) add one cost unit.
- This may, of course, over-estimate the real costs!

Heuristic estimate h_{add}

- While h_{max} is admissible, it is not very informative (it does not distinguish between different states).
- Estimate how hard it is to make a proposition true.
- Estimate costs under the assumption that making a proposition true is independent from making any other proposition true (i.e., relaxed planning graph is a tree).
  - Any proposition already true in the *initial state* has cost 0.
  - *Conjunctions* (if true) have the sum of the costs of the conjunctions.
  - *Disjunctions* (if true) have the minimum of the costs of the disjuncts.
  - *Actions* (if executable) add one cost unit.
- This may, of course, over-estimate the real costs!

Heuristic estimate h_{add}

- While $h_{\max}$ is admissible, it is not very informative (it does not distinguish between different states).
- Estimate how hard it is to make a proposition true.
- Estimate costs under the assumption that making a proposition true is independent from making any other proposition true (i.e., relaxed planning graph is a tree).
  - Any proposition already true in the *initial state* has cost 0.
  - *Conjunctions* (if true) have the sum of the costs of the conjunctions.
  - *Disjunctions* (if true) have the minimum of the costs of the disjuncts.
  - *Actions* (if executable) add one cost unit.
- This may, of course, over-estimate the real costs!

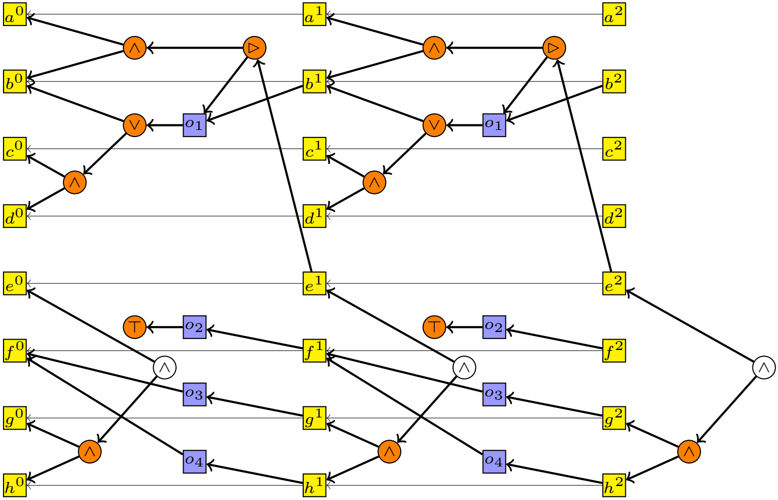
Heuristic estimate h_{add}

- While h_{max} is admissible, it is not very informative (it does not distinguish between different states).
- Estimate how hard it is to make a proposition true.
- Estimate costs under the assumption that making a proposition true is independent from making any other proposition true (i.e., relaxed planning graph is a tree).
  - Any proposition already true in the *initial state* has cost 0.
  - *Conjunctions* (if true) have the sum of the costs of the conjunctions.
  - *Disjunctions* (if true) have the minimum of the costs of the disjuncts.
  - *Actions* (if executable) add one cost unit.
- This may, of course, over-estimate the real costs!

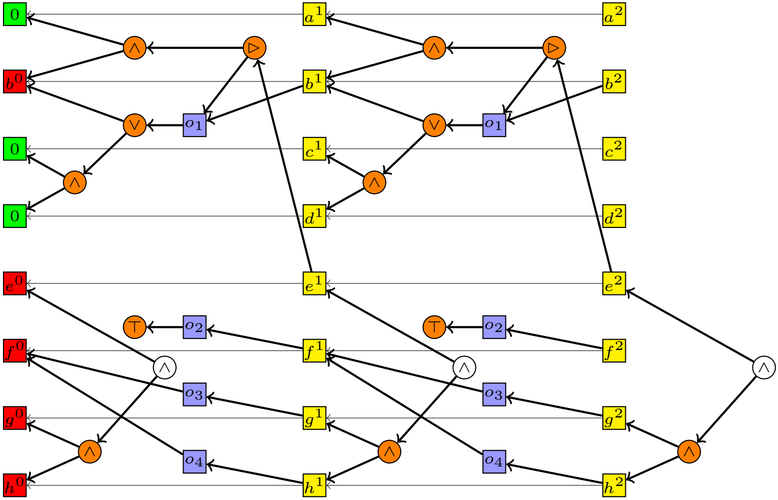
Heuristic estimate h_{add}

- While h_{max} is admissible, it is not very informative (it does not distinguish between different states).
- Estimate how hard it is to make a proposition true.
- Estimate costs under the assumption that making a proposition true is independent from making any other proposition true (i.e., relaxed planning graph is a tree).
  - Any proposition already true in the *initial state* has cost 0.
  - *Conjunctions* (if true) have the sum of the costs of the conjunctions.
  - *Disjunctions* (if true) have the minimum of the costs of the disjuncts.
  - *Actions* (if executable) add one cost unit.
- This may, of course, over-estimate the real costs!

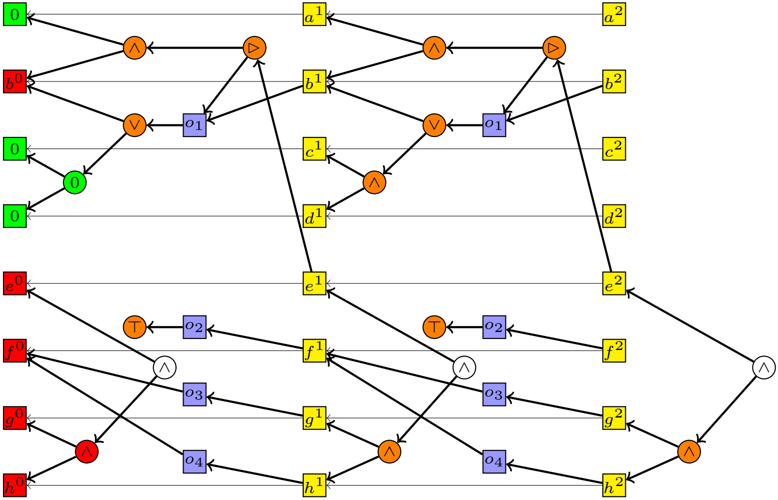
Computing h_{add}



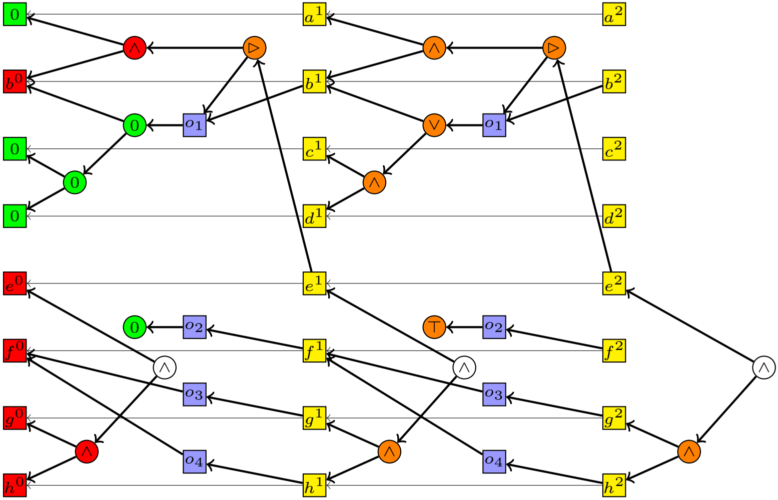
Computing h_{add}



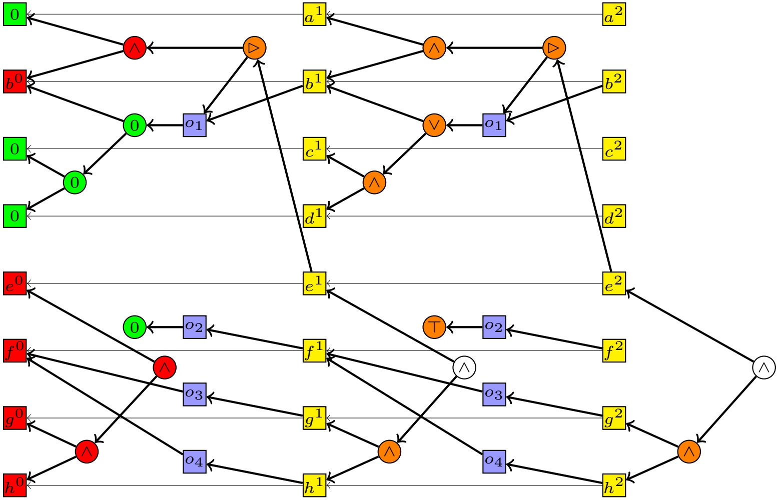
Computing h_{add}



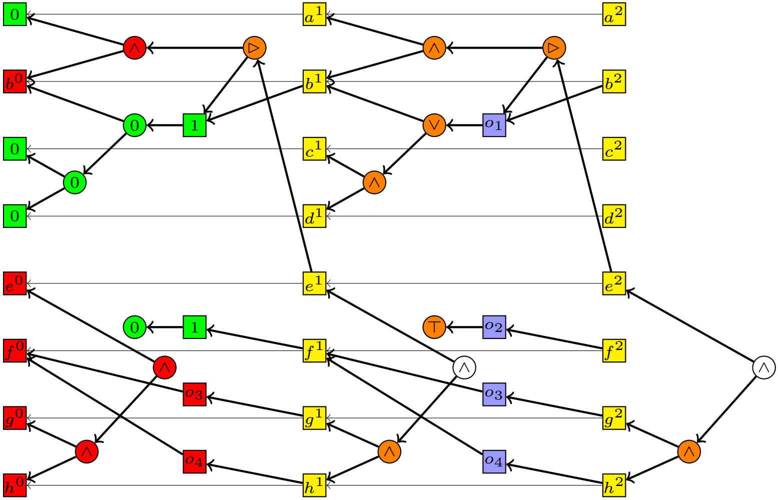
Computing h_{add}



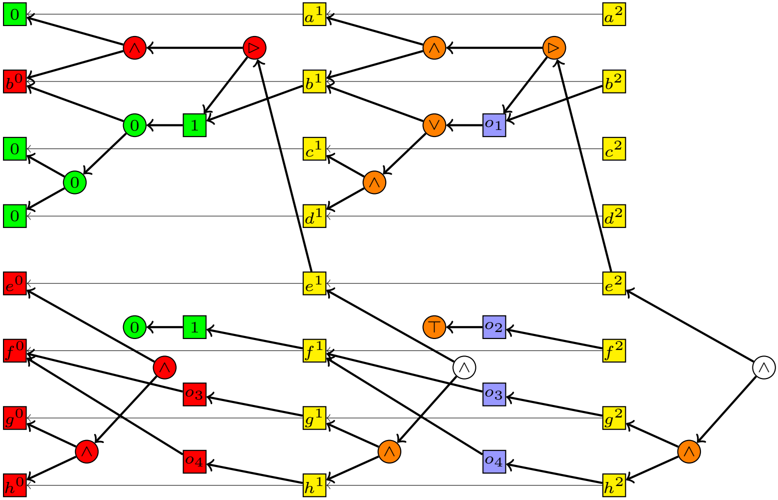
Computing h_{add}



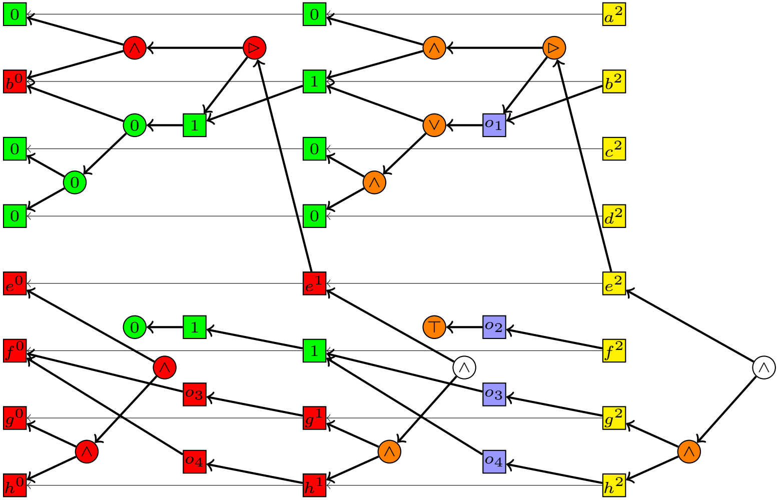
Computing h_{add}



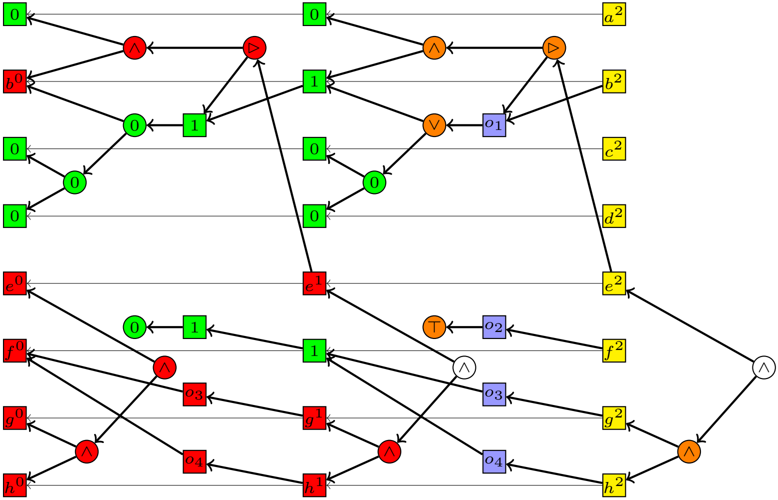
Computing h_{add}



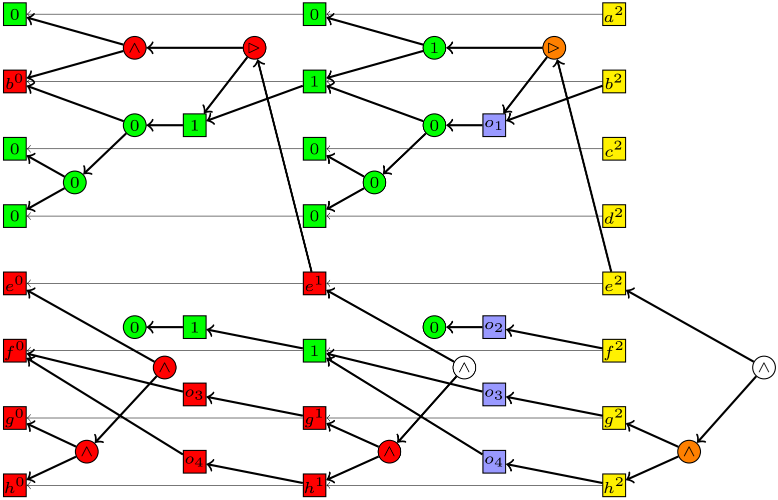
Computing h_{add}



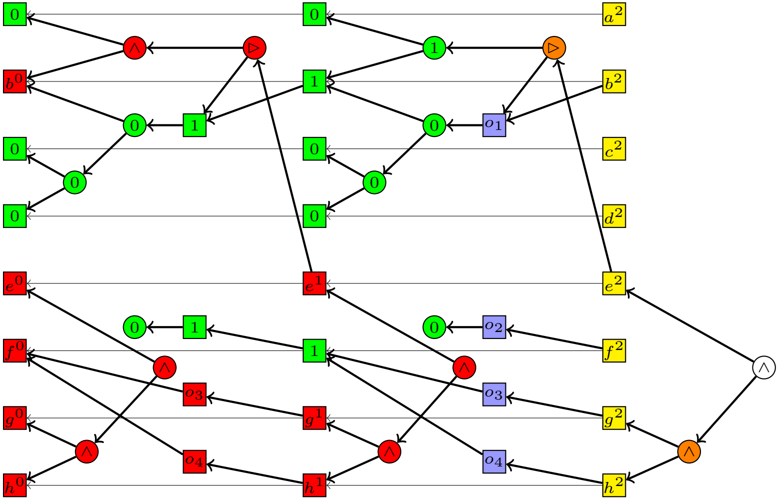
Computing h_{add}



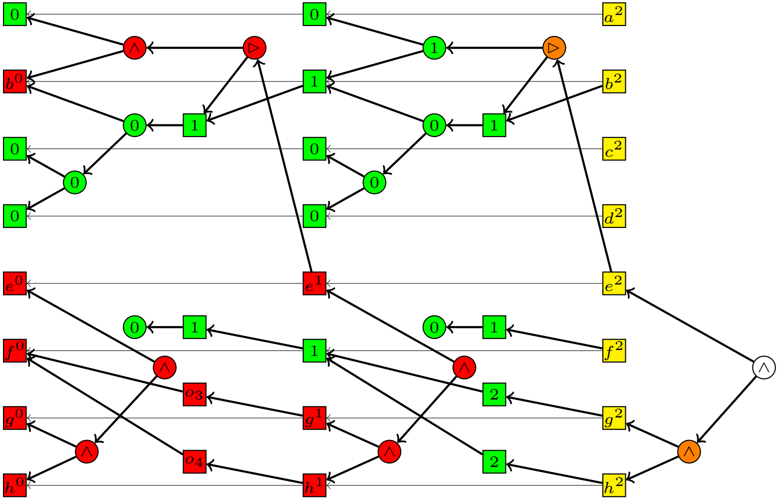
Computing h_{add}



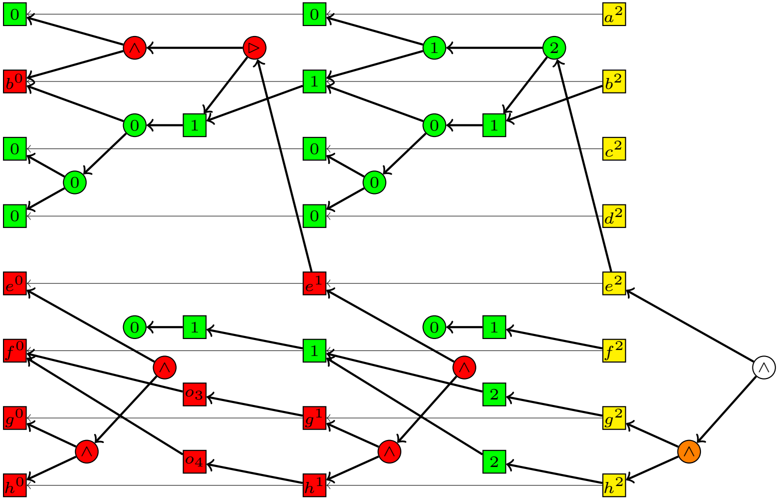
Computing h_{add}



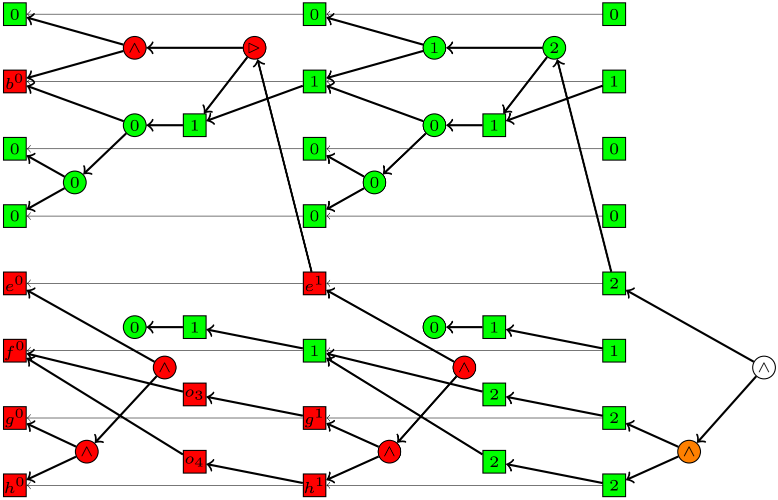
Computing h_{add}



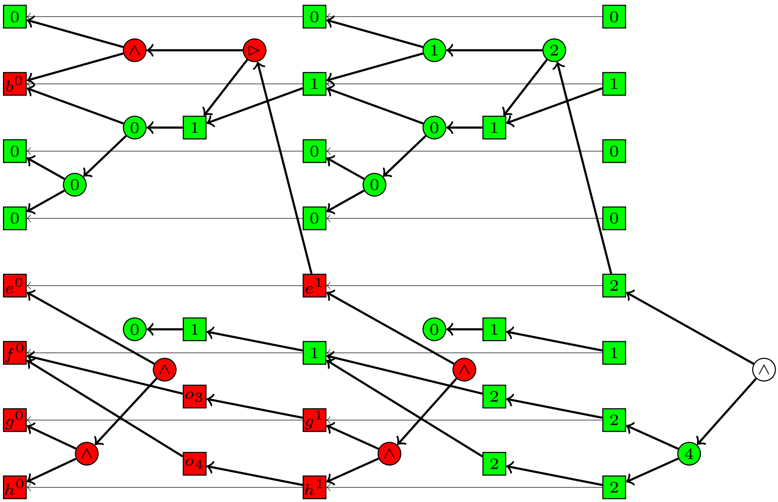
Computing h_{add}



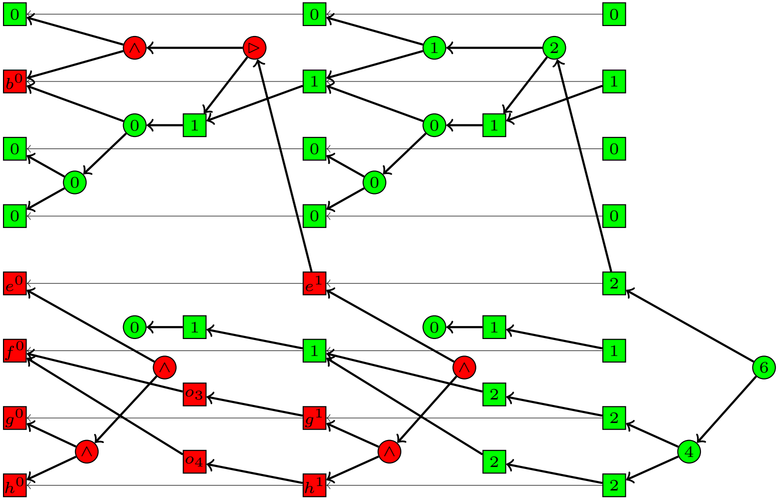
Computing h_{add}



Computing h_{add}



Computing h_{add}



Heuristic estimate h_{FF}

- h_{add} over-estimates because of the independence assumption.
- Positive interactions are ignored.
- Idea: Prune the sub-graph so that it non-redundantly makes the goal node true.
  - Start at the first true goal node.
  - Select both predecessors of a conjunction gate.
  - Select one true predecessor of disjucntion gate.
- Use the number of actions in corresponding parallel plan as the heuristic estimate.
- Of course, one would like to have a minimal sub-graph. However determining the minimal sub-graph is as hard as finding a minimal relaxed plan, i.e., NP-hard!

Heuristic estimate h_{FF}

- h_{add} over-estimates because of the independence assumption.
- Positive interactions are ignored.
- Idea: Prune the sub-graph so that it non-redundantly makes the goal node true.
  - Start at the first true goal node.
  - Select both predecessors of a conjunction gate.
  - Select one true predecessor of disjucntion gate.
- Use the number of actions in corresponding parallel plan as the heuristic estimate.
- Of course, one would like to have a minimal sub-graph. However determining the minimal sub-graph is as hard as finding a minimal relaxed plan, i.e., NP-hard!

Heuristic estimate h_{FF}

- h_{add} over-estimates because of the independence assumption.
- Positive interactions are ignored.
- Idea: Prune the sub-graph so that it non-redundantly makes the goal node true.
  - Start at the first true goal node.
  - Select both predecessors of a conjunction gate.
  - Select one true predecessor of disjucntion gate.
- Use the number of actions in corresponding parallel plan as the heuristic estimate.
- Of course, one would like to have a minimal sub-graph. However determining the minimal sub-graph is as hard as finding a minimal relaxed plan, i.e., NP-hard!

Heuristic estimate h_{FF}

- h_{add} over-estimates because of the independence assumption.
- Positive interactions are ignored.
- Idea: Prune the sub-graph so that it non-redundantly makes the goal node true.
  - Start at the first true goal node.
  - Select both predecessors of a conjunction gate.
  - Select one true predecessor of disjucntion gate.
- Use the number of actions in corresponding parallel plan as the heuristic estimate.
- Of course, one would like to have a minimal sub-graph. However determining the minimal sub-graph is as hard as finding a minimal relaxed plan, i.e., NP-hard!

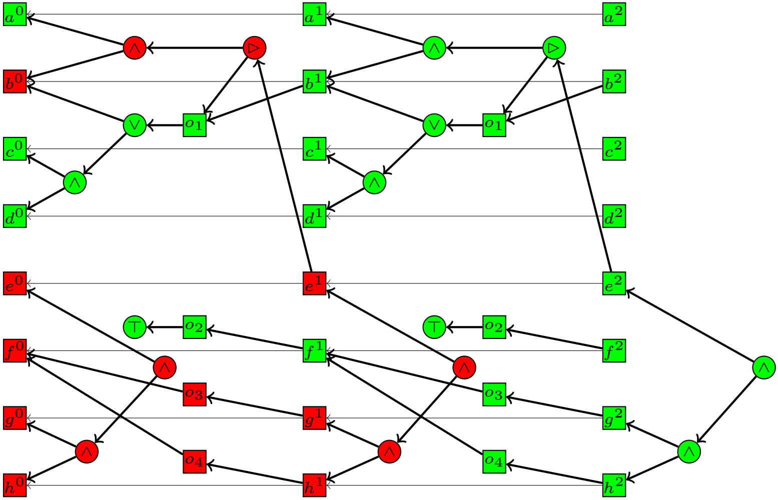
Heuristic estimate h_{FF}

- h_{add} over-estimates because of the independence assumption.
- Positive interactions are ignored.
- Idea: Prune the sub-graph so that it non-redundantly makes the goal node true.
  - Start at the first true goal node.
  - Select both predecessors of a conjunction gate.
  - Select one true predecessor of disjucntion gate.
- Use the number of actions in corresponding parallel plan as the heuristic estimate.
- Of course, one would like to have a minimal sub-graph. However determining the minimal sub-graph is as hard as finding a minimal relaxed plan, i.e., NP-hard!

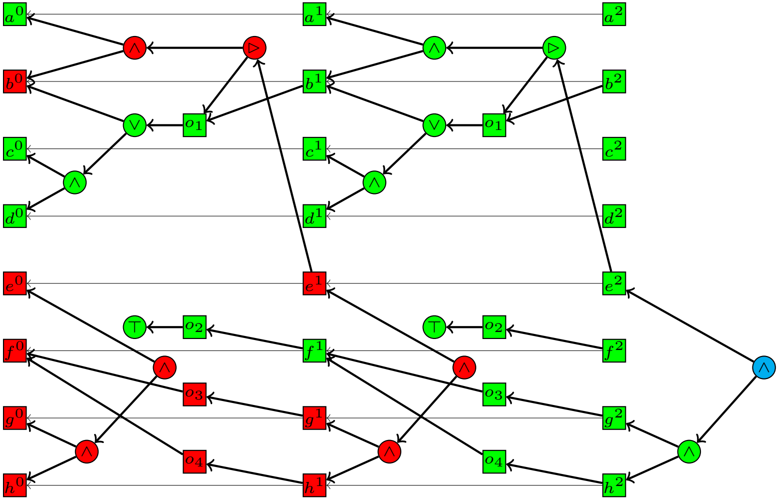
Heuristic estimate h_{FF}

- h_{add} over-estimates because of the independence assumption.
- Positive interactions are ignored.
- Idea: Prune the sub-graph so that it non-redundantly makes the goal node true.
  - Start at the first true goal node.
  - Select both predecessors of a conjunction gate.
  - Select one true predecessor of disjucntion gate.
- Use the number of actions in corresponding parallel plan as the heuristic estimate.
- Of course, one would like to have a minimal sub-graph. However determining the minimal sub-graph is as hard as finding a minimal relaxed plan, i.e., NP-hard!

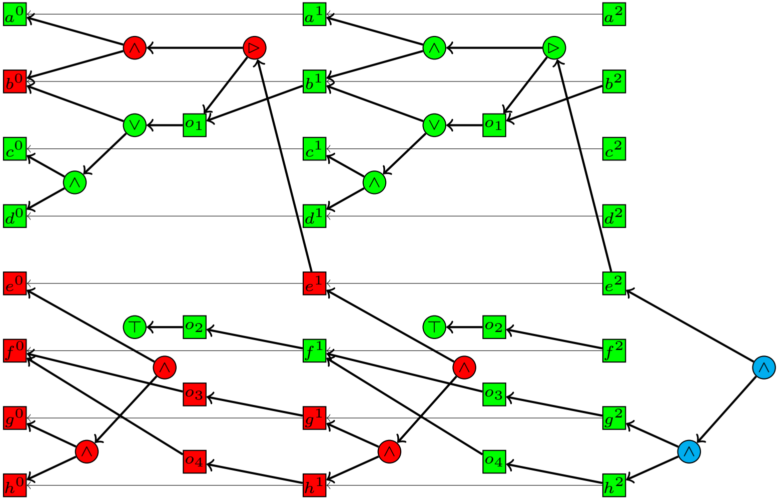
Selection of an h_{FF} sub-graph



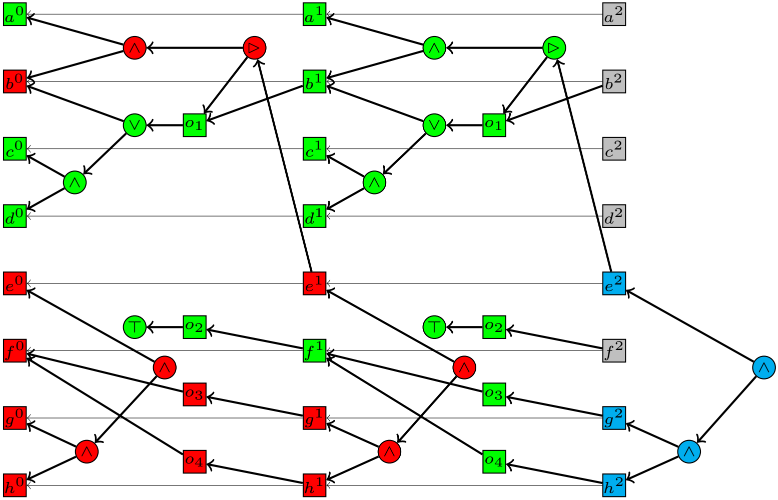
Selection of an h_{FF} sub-graph



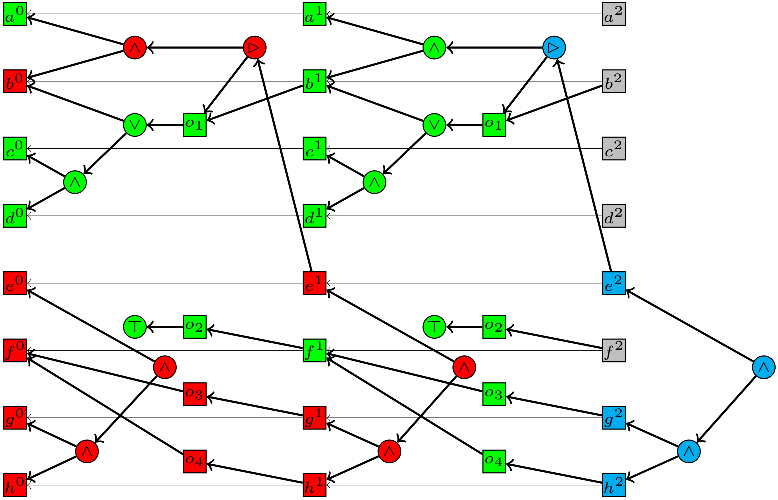
Selection of an h_{FF} sub-graph



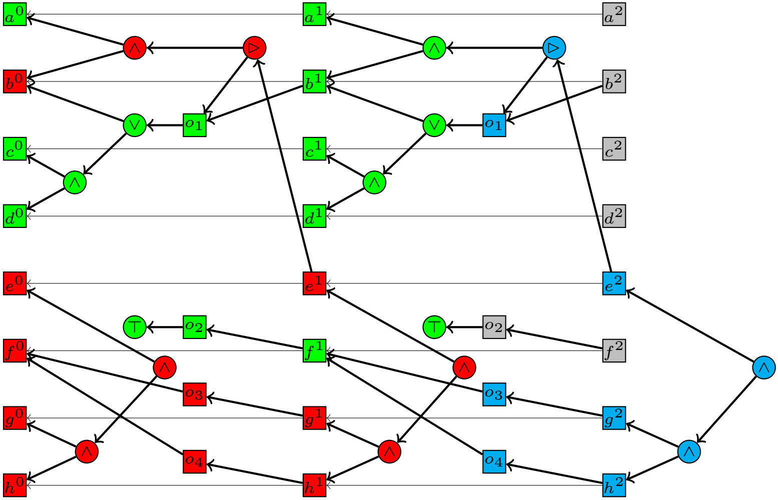
Selection of an h_{FF} sub-graph



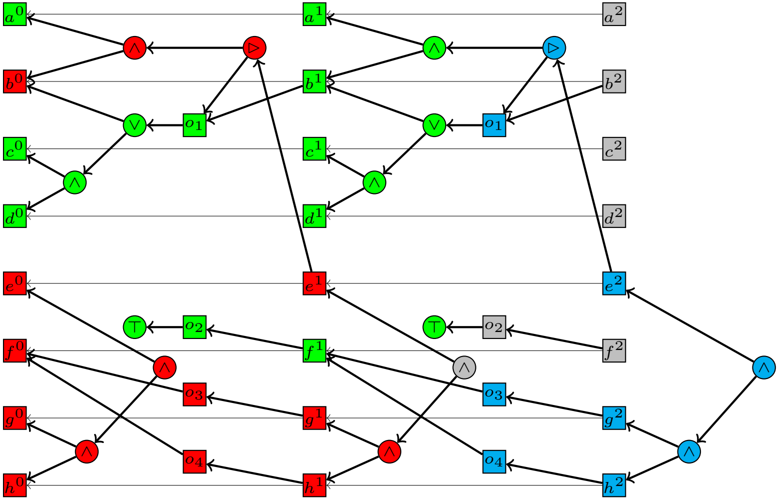
Selection of an h_{FF} sub-graph



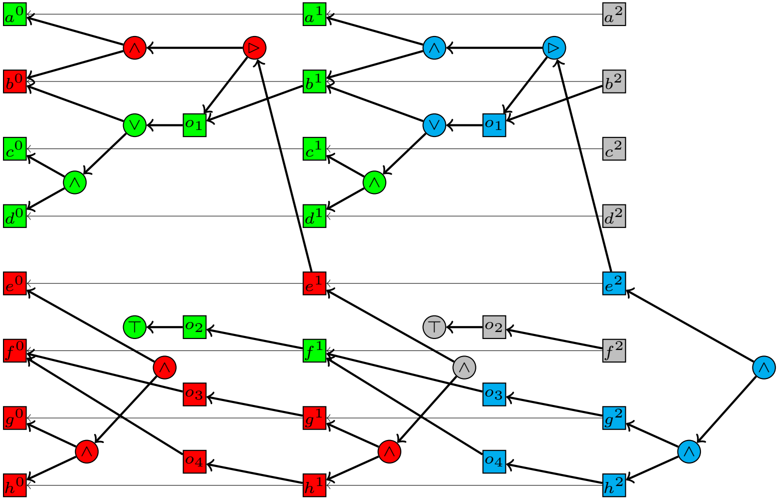
Selection of an h_{FF} sub-graph



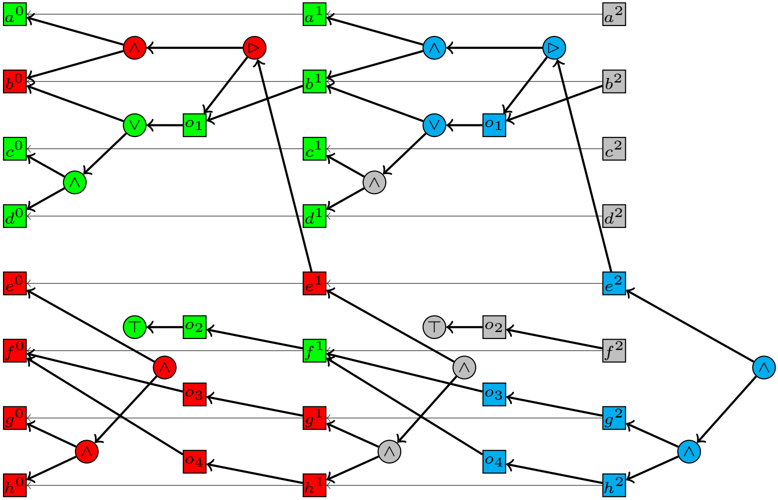
Selection of an h_{FF} sub-graph



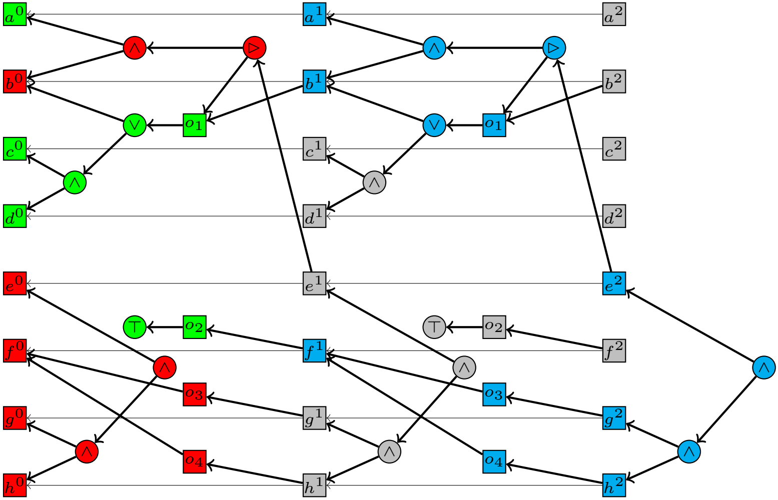
Selection of an h_{FF} sub-graph



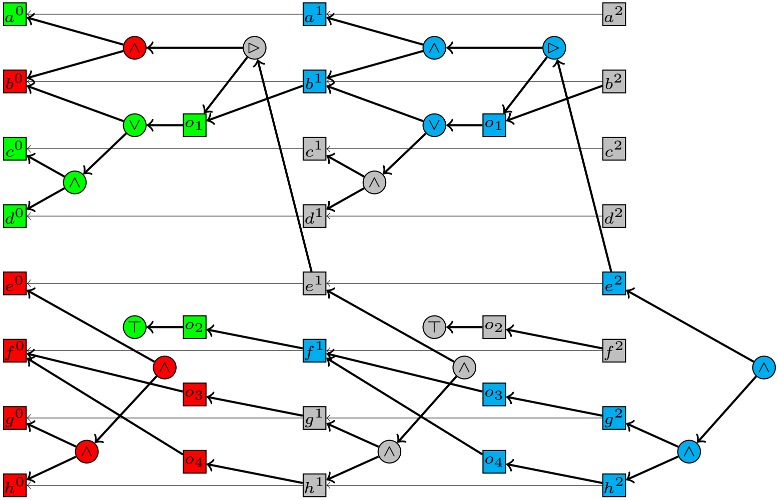
Selection of an h_{FF} sub-graph



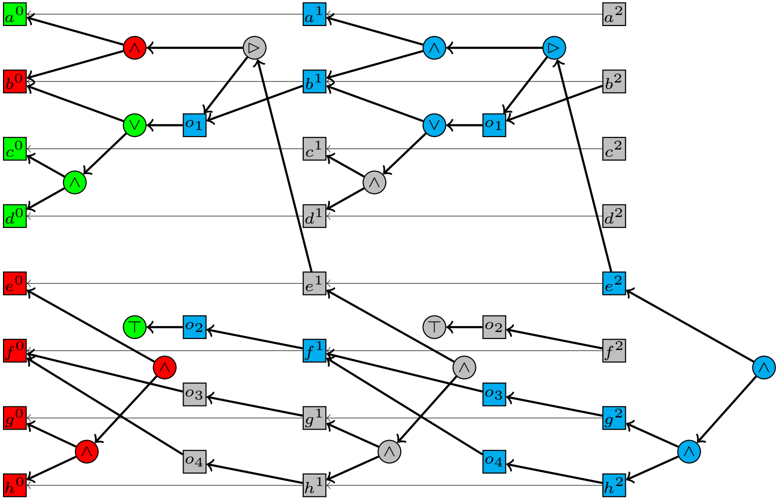
Selection of an h_{FF} sub-graph



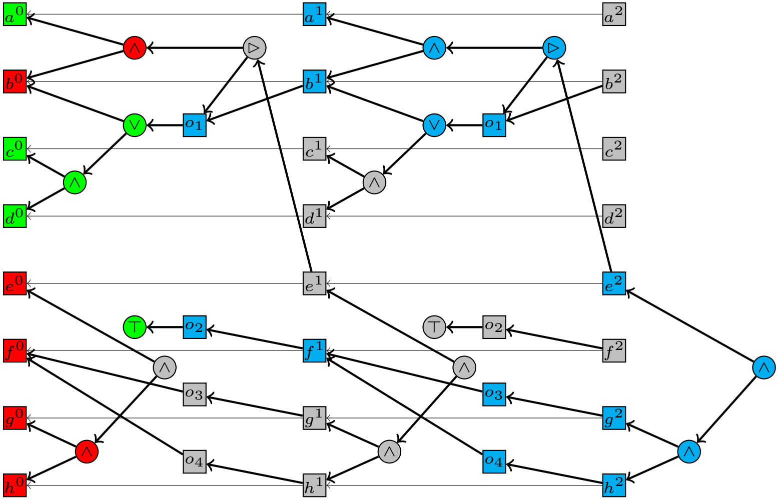
Selection of an h_{FF} sub-graph



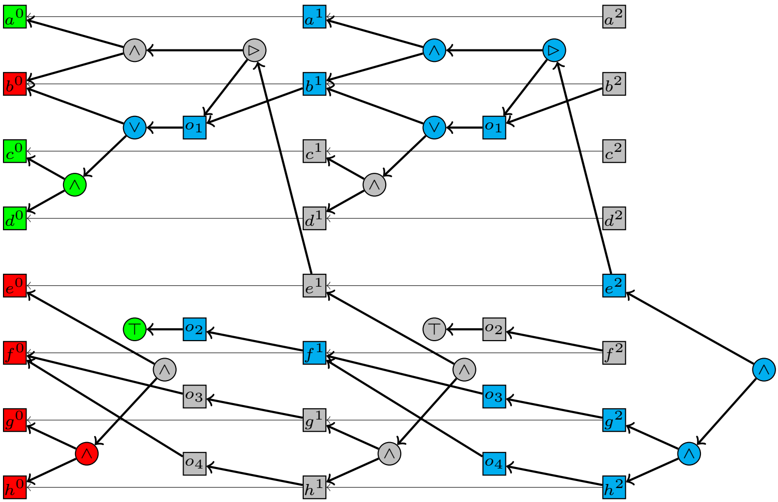
Selection of an h_{FF} sub-graph



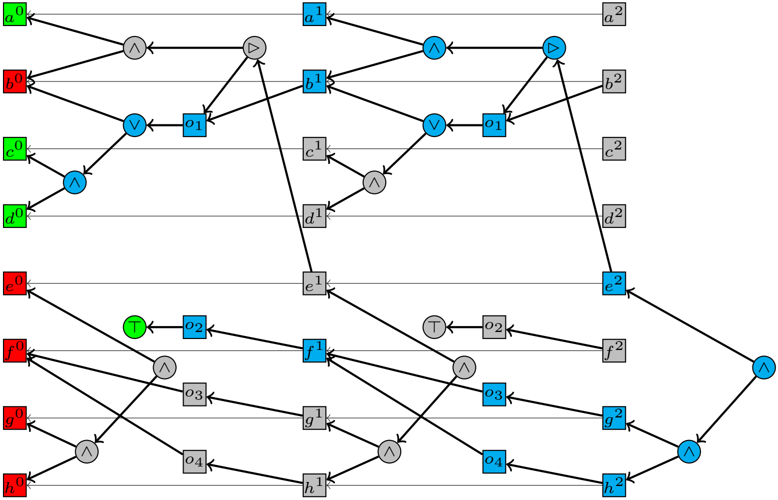
Selection of an h_{FF} sub-graph



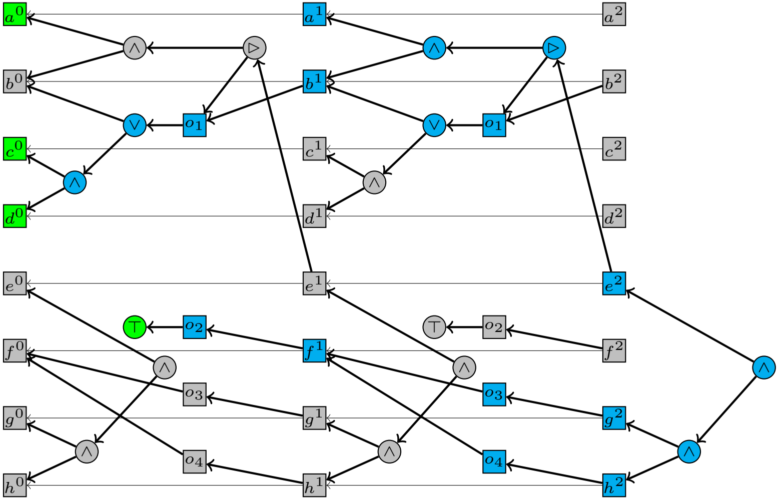
Selection of an h_{FF} sub-graph



Selection of an h_{FF} sub-graph



Selection of an h_{FF} sub-graph



Why is it hard to find a shortest relaxed plan?

The problem is hard, even if our actions do not have preconditions (and are all executed in parallel in one step)!

Problem

The set cover problem ist the following problem:

- *Given a set M, a collection of subsets $C = \{C_1, \dots, C_n\}$, with $C_i \subseteq M$ and a natural number k.*
- *Does there exist a set cover of size k, i.e., a subset of $S = \{S_1, \dots, S_j\} \subseteq C$ with $S_1 \cup \dots \cup S_j = M$ and $j \leq k$?*

Theorem

The set cover problem is NP-complete.

Why is it hard to find a shortest relaxed plan?

The problem is hard, even if our actions do not have preconditions (and are all executed in parallel in one step)!

Problem

The set cover problem ist the following problem:

- *Given a set M, a collection of subsets $C = \{C_1, \ldots, C_n\}$, with $C_i \subseteq M$ and a natural number k.*
- *Does there exist a set cover of size k, i.e., a subset of $S = \{S_1, \ldots, S_j\} \subseteq C$ with $S_1 \cup \ldots \cup S_j = M$ and $j \leq k$?*

Theorem

The set cover problem is NP-complete.

Why is it hard to find a shortest relaxed plan?

The problem is hard, even if our actions do not have preconditions (and are all executed in parallel in one step)!

Problem

The set cover *problem ist the following problem:*

- *Given a set M, a collection of subsets $C = \{C_1, \ldots, C_n\}$, with $C_i \subseteq M$ and a natural number k.*
- *Does there exist a set cover of size k, i.e., a subset of $S = \{S_1, \ldots, S_j\} \subseteq C$ with $S_1 \cup \ldots \cup S_j = M$ and $j \leq k$?*

Theorem

The set cover problem is NP-complete.

The Reduction

- An instance of the set cover problem $\langle M, C, k\rangle$ is given.
- Construct a relaxed planning task $\langle A, I, O^+, G\rangle$:
  - $A = M$,
  - $G = \bigwedge_{a \in A} a$,
  - $I = \{ a \mapsto 0 \mid a \in A \}$,
  - $O^+ = \{ \langle \top, \bigwedge_{a \in C_i} a\rangle \mid C_i \in C \}$
- Now clearly: There exists a plan containing at most k operators *iff* there exists a set cover of size k
- This implies that finding a shortest plan is NP-hard.

The Reduction

- An instance of the set cover problem $\langle M, C, k\rangle$ is given.
- Construct a relaxed planning task $\langle A, I, O^+, G\rangle$:
  - $A = M$,
  - $G = \bigwedge_{a \in A} a$,
  - $I = \{ a \mapsto 0 \mid a \in A \}$,
  - $O^+ = \{ \langle \top, \bigwedge_{a \in C_i} a\rangle \mid C_i \in C \}$
- Now clearly: There exists a plan containing at most k operators *iff* there exists a set cover of size k
- This implies that finding a shortest plan is NP-hard.

The Reduction

- An instance of the set cover problem $\langle M, C, k\rangle$ is given.
- Construct a relaxed planning task $\langle A, I, O^+, G\rangle$:
  - $A = M$,
  - $G = \bigwedge_{a \in A} a$,
  - $I = \{ a \mapsto 0 \mid a \in A \}$,
  - $O^+ = \{ \langle \top, \bigwedge_{a \in C_i} a\rangle \mid C_i \in C \}$
- Now clearly: There exists a plan containing at most k operators *iff* there exists a set cover of size k
- This implies that finding a shortest plan is NP-hard.

The Reduction

- An instance of the set cover problem $\langle M, C, k \rangle$ is given.
- Construct a relaxed planning task $\langle A, I, O^+, G \rangle$:
  - $A = M$,
  - $G = \bigwedge_{a \in A} a$,
  - $I = \{ a \mapsto 0 \mid a \in A \}$,
  - $O^+ = \{ \langle \top, \bigwedge_{a \in C_i} a \rangle \mid C_i \in C \}$
- Now clearly: There exists a plan containing at most k operators *iff* there exists a set cover of size k
- This implies that finding a shortest plan is NP-hard.

The Reduction

- An instance of the set cover problem $\langle M, C, k \rangle$ is given.
- Construct a relaxed planning task $\langle A, I, O^+, G \rangle$:
  - $A = M$,
  - $G = \bigwedge_{a \in A} a$,
  - $I = \{ a \mapsto 0 \mid a \in A \}$,
  - $O^+ = \{ \langle \top, \bigwedge_{a \in C_i} a \rangle \mid C_i \in C \}$
- Now clearly: There exists a plan containing at most k operators *iff* there exists a set cover of size k
- This implies that finding a shortest plan is NP-hard.

The Reduction

- An instance of the set cover problem $\langle M, C, k \rangle$ is given.
- Construct a relaxed planning task $\langle A, I, O^+, G \rangle$:
  - $A = M$,
  - $G = \bigwedge_{a \in A} a$,
  - $I = \{ a \mapsto 0 \mid a \in A \}$,
  - $O^+ = \{ \langle \top, \bigwedge_{a \in C_i} a \rangle \mid C_i \in C \}$
- Now clearly: There exists a plan containing at most k operators *iff* there exists a set cover of size k
- This implies that finding a shortest plan is NP-hard.

The Reduction

- An instance of the set cover problem $\langle M, C, k \rangle$ is given.
- Construct a relaxed planning task $\langle A, I, O^+, G \rangle$:
  - $A = M$,
  - $G = \bigwedge_{a \in A} a$,
  - $I = \{ a \mapsto 0 \mid a \in A \}$,
  - $O^+ = \{ \langle \top, \bigwedge_{a \in C_i} a \rangle \mid C_i \in C \}$
- Now clearly: There exists a plan containing at most k operators *iff* there exists a set cover of size k
- This implies that finding a shortest plan is NP-hard.

The Reduction

- An instance of the set cover problem $\langle M, C, k\rangle$ is given.
- Construct a relaxed planning task $\langle A, I, O^+, G\rangle$:
  - $A = M$,
  - $G = \bigwedge_{a \in A} a$,
  - $I = \{ a \mapsto 0 \mid a \in A \}$,
  - $O^+ = \{ \langle \top, \bigwedge_{a \in C_i} a\rangle \mid C_i \in C \}$
- Now clearly: There exists a plan containing at most k operators *iff* there exists a set cover of size k
- This implies that finding a shortest plan is NP-hard.

Putting h_{FF} to work: FF

The FF planning system works roughly as follows:

- It does enforced hill-climbing using h_{FF}. This is hill-climbing extended by breadth-first search in cases where there are no states with a better heuristic value.
- In addition, FF uses helpful action pruning, i.e., it considers only those actions that are used in the first level of the relaxed planning graph.
- If a hill-climbing step with helpful action pruning fails, then the fall-back is to use all possible actions.
- If no plan is found, FF restarts the search as a greedy best-first search with h_{FF} as the heuristic.

Putting h_{FF} to work: FF

The FF planning system works roughly as follows:

- It does enforced hill-climbing using h_{FF}. This is hill-climbing extended by breadth-first search in cases where there are no states with a better heuristic value.
- In addition, FF uses helpful action pruning, i.e., it considers only those actions that are used in the first level of the relaxed planning graph.
- If a hill-climbing step with helpful action pruning fails, then the fall-back is to use all possible actions.
- If no plan is found, FF restarts the search as a greedy best-first search with h_{FF} as the heuristic.

Putting h_{FF} to work: FF

The FF planning system works roughly as follows:

- It does enforced hill-climbing using h_{FF}. This is hill-climbing extended by breadth-first search in cases where there are no states with a better heuristic value.
- In addition, FF uses helpful action pruning, i.e., it considers only those actions that are used in the first level of the relaxed planning graph.
- If a hill-climbing step with helpful action pruning fails, then the fall-back is to use all possible actions.
- If no plan is found, FF restarts the search as a greedy best-first search with h_{FF} as the heuristic.

Putting h_{FF} to work: FF

The FF planning system works roughly as follows:

- It does enforced hill-climbing using h_{FF}. This is hill-climbing extended by breadth-first search in cases where there are no states with a better heuristic value.
- In addition, FF uses helpful action pruning, i.e., it considers only those actions that are used in the first level of the relaxed planning graph.
- If a hill-climbing step with helpful action pruning fails, then the fall-back is to use all possible actions.
- If no plan is found, FF restarts the search as a greedy best-first search with h_{FF} as the heuristic.