Principles of AI Planning

8. Planning as search: relaxation heuristics

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Bernhard Nebel and Robert Mattmüller

November 17th, 2017

Parallel plans

Towards better relaxed plans

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.

Relaxed plan steps

How to decide which operators to apply in forward direction?

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

Definition (plan step)

A plan step is a set of operators $\omega = \{\langle \chi_1, e_1 \rangle, \ldots, \langle \chi_n, e_n \rangle\}$.

In the special case of all operators of ω being relaxed, we further define:

- Plan step ω is applicable in state s iff $s \models \chi_i$ for all $i \in \{1, \ldots, n\}$.
- The result of applying ω to s, in symbols $app_\omega(s)$, is defined as the state s' with $on(s') = on(s) \cup \bigcup_{i=1}^{n} [e_i]_s$.

general semantics for plan steps $\rightsquigarrow$ much later

Applying relaxed plan steps: examples

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

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

Serializations

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

Definition (serialization)

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

Lemma (conservativeness of plan step semantics)

If ω is a plan step applicable in a state s of a relaxed planning task, then each serialization $o_1, \dots, o_n$ of ω is applicable in s and $app_{o_1, \dots, o_n}(s)$ dominates $app_\omega(s)$.

- Does equality hold for all/some serialization(s)?
- What if there are no conditional effects?
- What if we allowed general (unrelaxed) planning tasks?

Parallel plans

Definition (parallel plan)

A parallel plan for a relaxed planning task $\langle A, I, O^+, \gamma\rangle$ is a sequence of plan steps $\omega_1, \ldots, \omega_n$ of operators in O^+ with:

- $s_0 := I$
- For $i = 1, \ldots, n$, step ω_i is applicable in s_{i-1} and $s_i := app_{\omega_i}(s_{i-1})$.
- $s_n \models \gamma$

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

Forward states, plan steps and sets

Idea: In the forward phase of the heuristic computation,

1. apply plan step with all operators applicable initially,
2. apply plan step with all operators applicable then,
3. and so on.

Definition (forward state/plan step/set)

Let $\Pi^+ = \langle A, I, O^+, \gamma \rangle$ be a relaxed planning task.

The *n*-th forward state, in symbols s_n^F ($n \in \mathbb{N}_0$),
the *n*-th forward plan step, in symbols ω_n^F ($n \in \mathbb{N}_1$), and
the *n*-th forward set, in symbols S_n^F ($n \in \mathbb{N}_0$), are defined as:

- $s_0^F := I$
- $\omega_n^F := \{o \in O^+ \mid o \text{ applicable in } s_{n-1}^F\}$ for all $n \in \mathbb{N}_1$
- $s_n^F := app_{\omega_n^F}(s_{n-1}^F)$ for all $n \in \mathbb{N}_1$
- $S_n^F := on(s_n^F)$ for all $n \in \mathbb{N}_0$

The max heuristic $h_{\max}$

Definition (parallel forward distance)

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

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

Definition (max heuristic $h_{\max}$)

Let $\Pi = \langle A, I, O, \gamma \rangle$ be a planning task in positive normal form, and let s be a state of Π.
The max heuristic estimate for s, $h_{\max}(s)$, is the parallel forward distance of the relaxed planning task $\langle A, s, O^+, \gamma \rangle$.

Remark: $h_{\max}$ is safe, goal-aware, admissible and consistent. (Why?)

So far, so good…

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

For this purpose, we use so-called relaxed planning graphs.

Relaxed planning graphs

AND/OR dags

Definition (AND/OR dag)

An AND/OR dag $\langle V, A, type \rangle$ is a directed acyclic graph $\langle V, A \rangle$ with a label function $type : V \to \{\wedge, \vee\}$ partitioning nodes into AND nodes ($type(v) = \wedge$) and OR nodes ($type(v) = \vee$).

Note: AND nodes drawn as squares, OR nodes as circles.

Definition (truth values in AND/OR dags)

Let $G = \langle V, A, type \rangle$ be an AND/OR dag, and let $u \in V$ be a node with successor set $\{v_1, \dots, v_k\} \subseteq V$.
The (truth) value of u, $val(u)$, is inductively defined as:

- If $type(u) = \wedge$, then $val(u) = val(v_1) \wedge \cdots \wedge val(v_k)$.
- If $type(u) = \vee$, then $val(u) = val(v_1) \vee \cdots \vee val(v_k)$.

Relaxed planning graphs

Let Π^+ be a relaxed planning task, and let $k \in \mathbb{N}_0$.

The relaxed planning graph of Π^+ for depth k, in symbols $RPG_k(\Pi^+)$, is an AND/OR dag that encodes

- which propositions can be made true in k plan steps, and
- how they can be made true.

Its construction is a bit involved, so we present it in stages.

Running example

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

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

Running example: forward sets and plan steps

$$I = \{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 1, e \mapsto 0, f \mapsto 0, g \mapsto 0, h \mapsto 0\}$$
$$o_1 = \langle b \vee (c \wedge d), b \wedge ((a \wedge b) \rhd e)\rangle$$
$$o_2 = \langle \top, f\rangle, \quad o_3 = \langle f, g\rangle, \quad o_4 = \langle f, h\rangle$$

$$S_0^F = \{a, c, d\}$$
$$\omega_1^F = \{o_1, o_2\}$$
$$S_1^F = \{a, b, c, d, f\}$$
$$\omega_2^F = \{o_1, o_2, o_3, o_4\}$$
$$S_2^F = \{a, b, c, d, e, f, g, h\}$$
$$\omega_3^F = \omega_2^F$$
$$S_3^F = S_2^F \text{ etc.}$$

Components of relaxed planning graphs

A relaxed planning graph consists of four kinds of components:

- Proposition nodes represent the truth value of propositions after applying a certain number of plan steps.
- Idle arcs represent the fact that state variables, once true, remain true.
- Operator subgraphs represent the possibility and effect of applying a given operator in a given plan step.
- The goal subgraph represents the truth value of the goal condition after k plan steps.

Relaxed planning graph: proposition layers

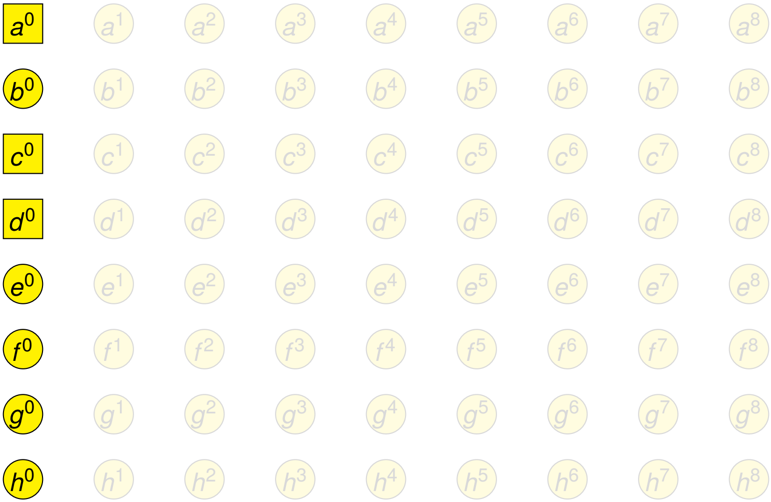
Let $\Pi^+ = \langle A, I, O^+, \gamma \rangle$ be a relaxed planning task, let $k \in \mathbb{N}_0$.

For each $i \in \{0, \ldots, k\}$, $RPG_k(\Pi^+)$ contains one proposition layer which consists of:

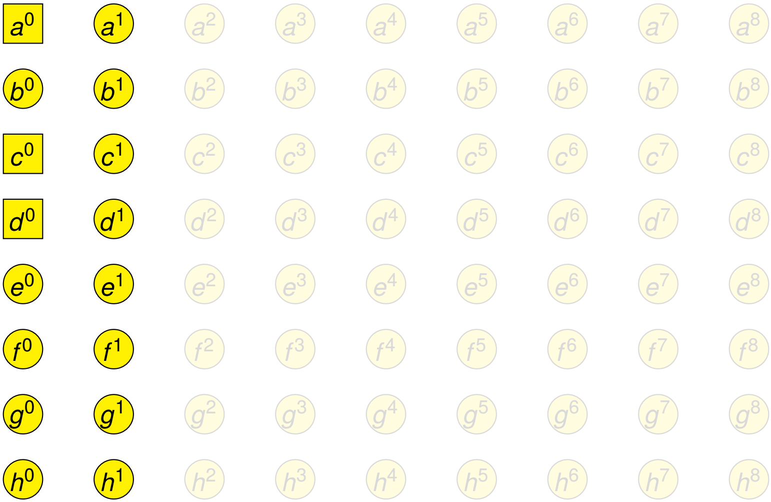
- a proposition node a^i for each state variable $a \in A$.

Node a^i is an AND node if $i = 0$ and $I \models a$.
Otherwise, it is an OR node.

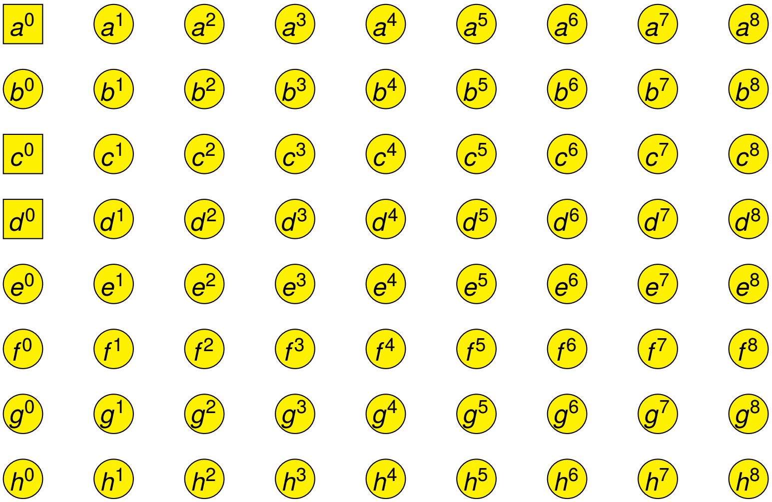
Relaxed planning graph: proposition layers



Relaxed planning graph: proposition layers



Relaxed planning graph: proposition layers

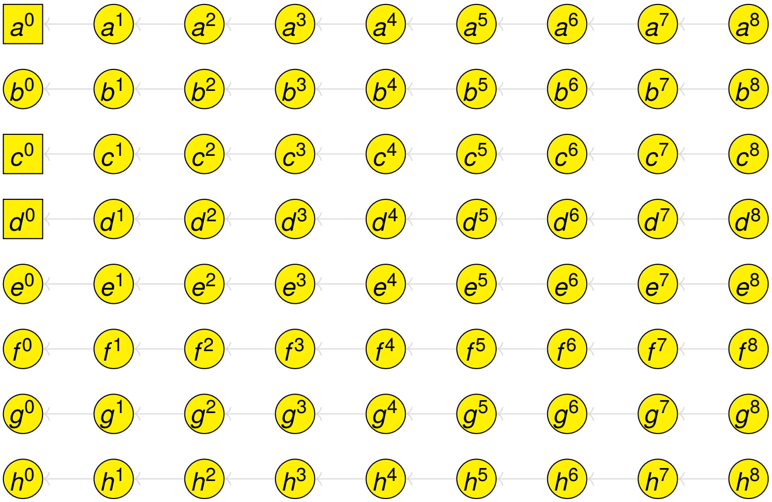


Relaxed planning graph: idle arcs

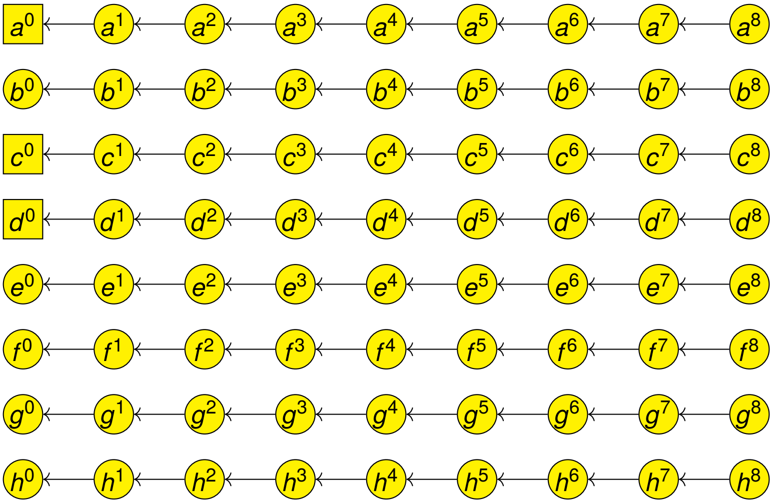
For each proposition node a^i with $i \in \{1, \ldots, k\}$, $RPG_k(\Pi^+)$ contains an arc from a^i to a^{i-1} (idle arcs).

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: operator subgraphs

For each $i \in \{1,\dots,k\}$ and each operator $o^+ = \langle \chi, e^+ \rangle \in O^+$, $RPG_k(\Pi^+)$ contains a subgraph called an operator subgraph with the following parts:

- one formula node n^i_φ for each formula φ which is a subformula of χ or of some effect condition in e^+:
  - If $\varphi = a$ for some atom a, n^i_φ is the proposition node a^{i-1}.
  - If $\varphi = \top$, n^i_φ is a new AND node without outgoing arcs.
  - If $\varphi = \bot$, n^i_φ is a new OR node without outgoing arcs.
  - If $\varphi = (\varphi' \wedge \varphi'')$, n^i_φ is a new AND node with outgoing arcs to $n^i_{\varphi'}$ and $n^i_{\varphi''}$.
  - If $\varphi = (\varphi' \vee \varphi'')$, n^i_φ is a new OR node with outgoing arcs to $n^i_{\varphi'}$ and $n^i_{\varphi''}$.

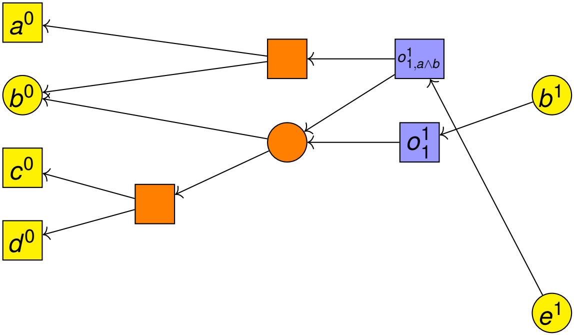
Relaxed planning graph: operator subgraphs

For each $i \in \{1, \ldots, k\}$ and each operator $o^+ = \langle \chi, e^+ \rangle \in O^+$, $RPG_k(\Pi^+)$ contains a subgraph called an operator subgraph with the following parts:

- for each conditional effect $(\chi' \rhd a)$ in e^+, an effect node $o^i_{\chi'}$ (an AND node) with outgoing arcs to the precondition formula node n^i_χ and effect condition formula node $n^i_{\chi'}$, and incoming arc from proposition node a^i
  - unconditional effects a (effects which are not part of a conditional effect) are treated the same, except that there is no arc to an effect condition formula node
  - effects with identical condition (including groups of unconditional effects) share the same effect node
  - the effect node for unconditional effects is denoted by o^i

Relaxed planning graph: operator subgraphs

Operator subgraph for $o_1 = \langle b \vee (c \wedge d), b \wedge ((a \wedge b) \rhd e)\rangle$ for layer $i = 1$.



Relaxed planning graph: goal subgraph

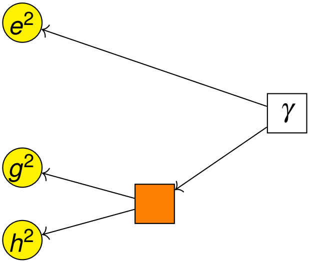
$RPG_k(\Pi^+)$ contains a subgraph called a goal subgraph with the following parts:

- one formula node n^k_φ for each formula φ which is a subformula of γ:
  - If $\varphi = a$ for some atom a, n^k_φ is the proposition node a^i.
  - If $\varphi = \top$, n^k_φ is a new AND node without outgoing arcs.
  - If $\varphi = \bot$, n^k_φ is a new OR node without outgoing arcs.
  - If $\varphi = (\varphi' \wedge \varphi'')$, n^k_φ is a new AND node with outgoing arcs to $n^k_{\varphi'}$ and $n^k_{\varphi''}$.
  - If $\varphi = (\varphi' \vee \varphi'')$, n^k_φ is a new OR node with outgoing arcs to $n^k_{\varphi'}$ and $n^k_{\varphi''}$.

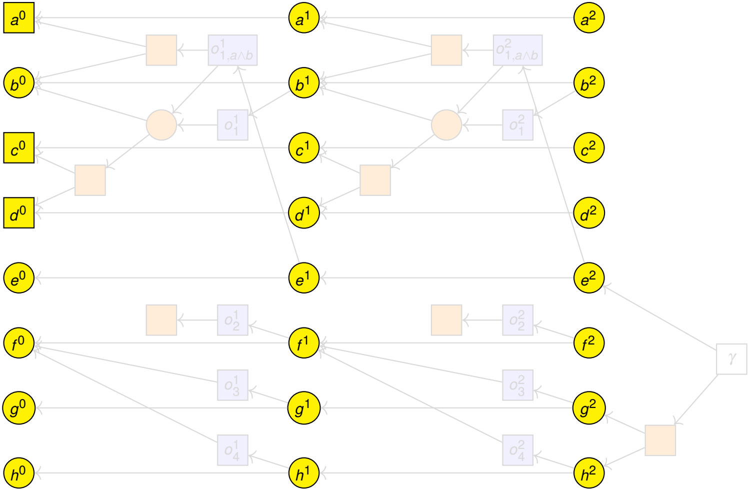
The node n^k_γ is called the goal node.

Relaxed planning graph: goal subgraphs

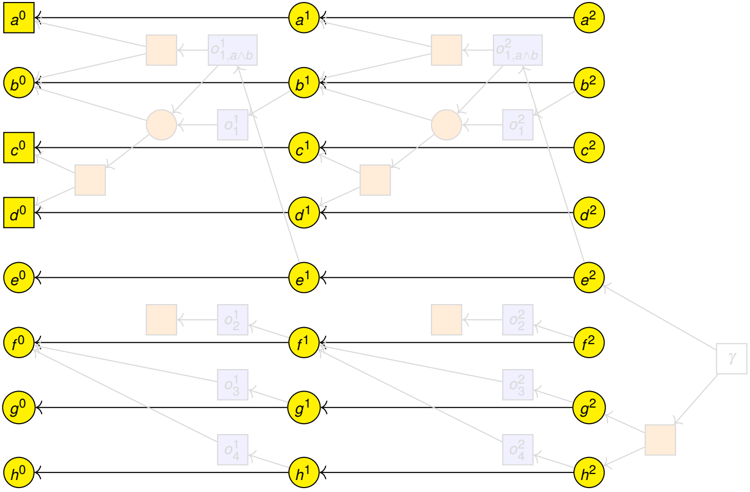
Goal subgraph for $\gamma = e \wedge (g \wedge h)$ and depth $k = 2$:



Relaxed planning graph: complete (depth 2)

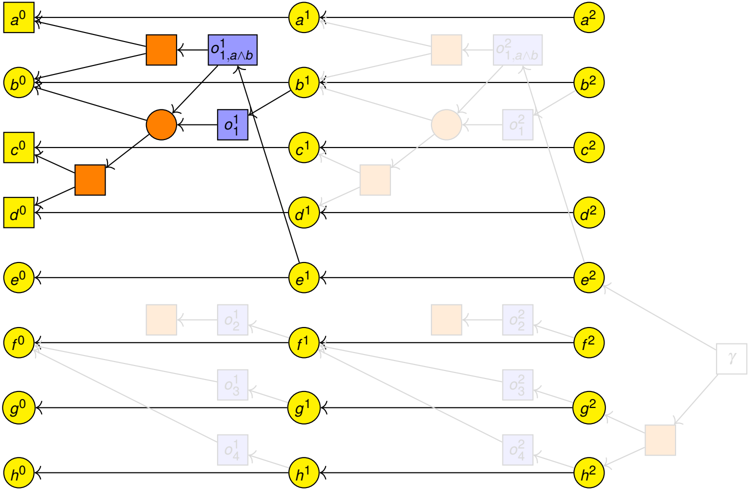


Relaxed planning graph: complete (depth 2)



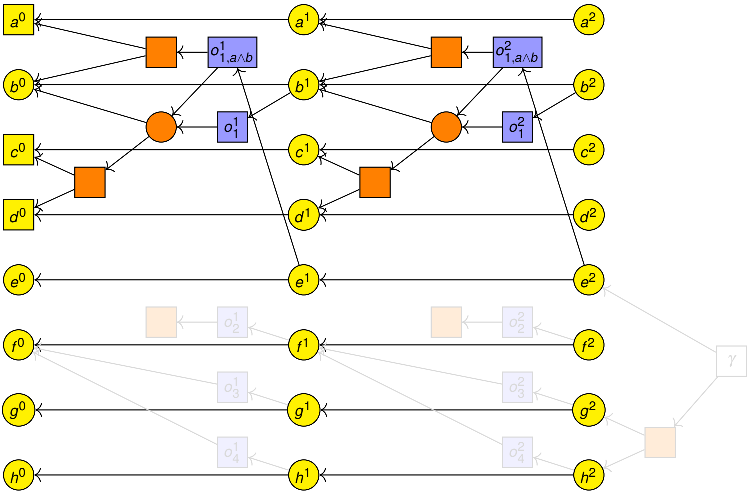
Relaxed planning graph: complete (depth 2)

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



Relaxed planning graph: complete (depth 2)

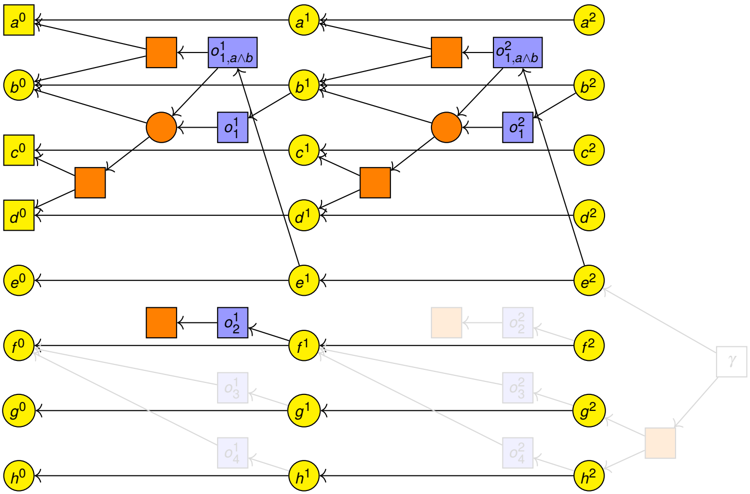
$o_1 = \langle b \vee (c \wedge d), b \wedge ((a \wedge b) \rhd e) \rangle$



UNI FREIBURG

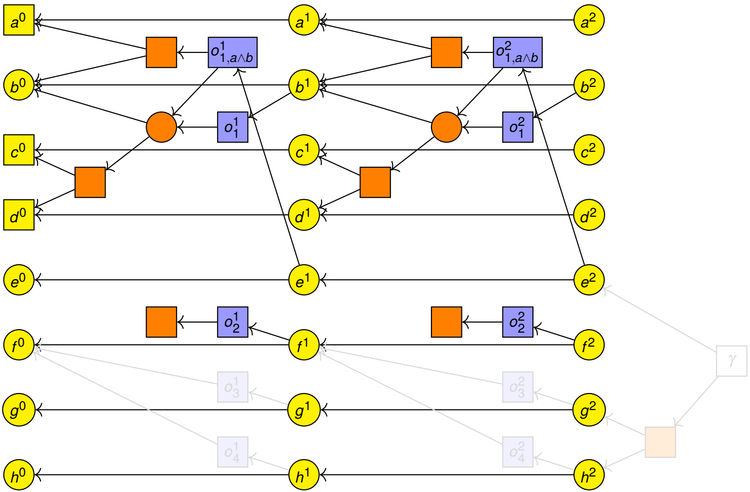
Relaxed planning graph: complete (depth 2)

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



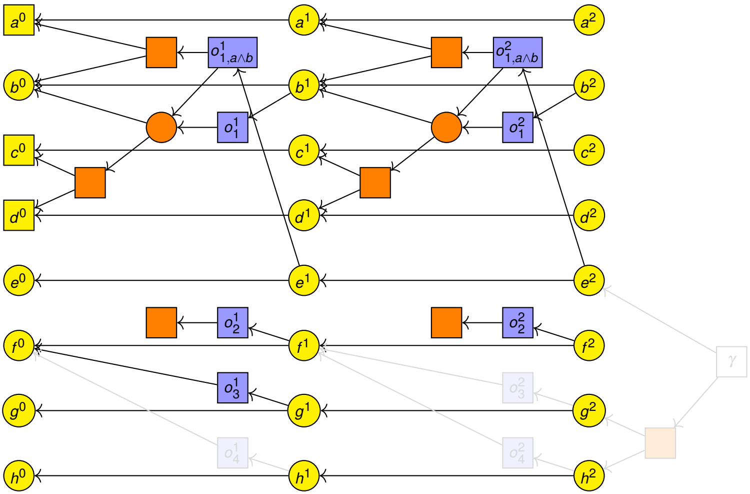
Relaxed planning graph: complete (depth 2)

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

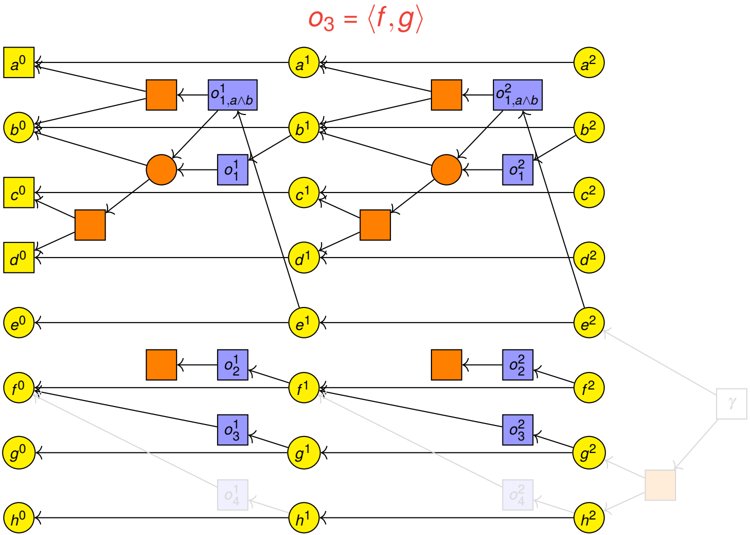


Relaxed planning graph: complete (depth 2)

$o_3 = \langle f, g \rangle$

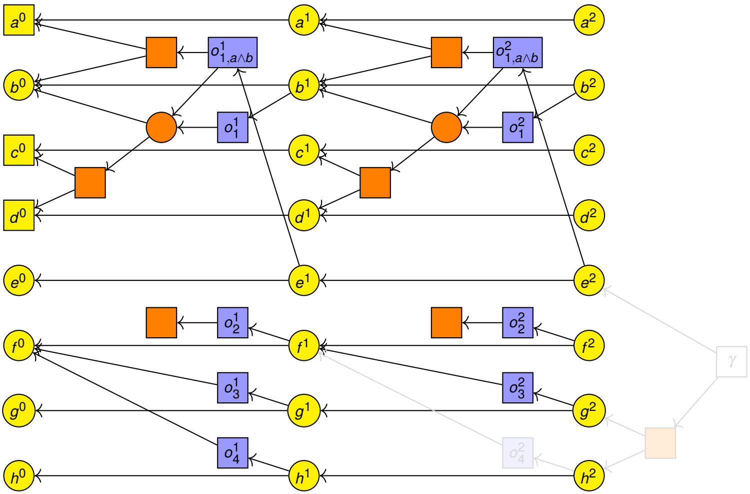


Relaxed planning graph: complete (depth 2)

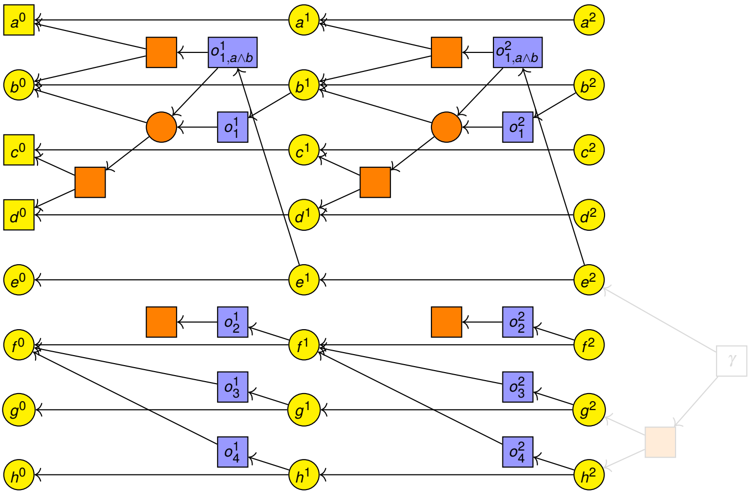


Relaxed planning graph: complete (depth 2)

$o_4 = \langle f, h \rangle$

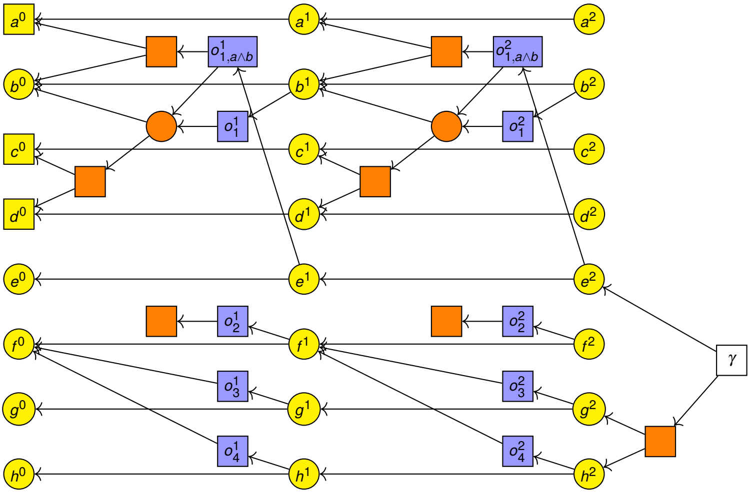


$o_4 = \langle f, h \rangle$

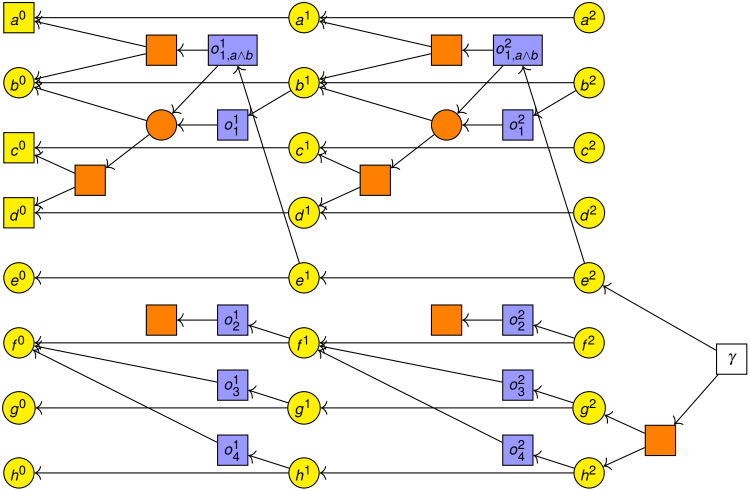


Relaxed planning graph: complete (depth 2)

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



Relaxed planning graph: complete (depth 2)



Parallel plans

Relaxed planning graphs

Introduction

Construction

Truth values

Relaxation heuristics

Summary

Connection to forward sets and plan steps

Theorem (relaxed planning graph truth values)

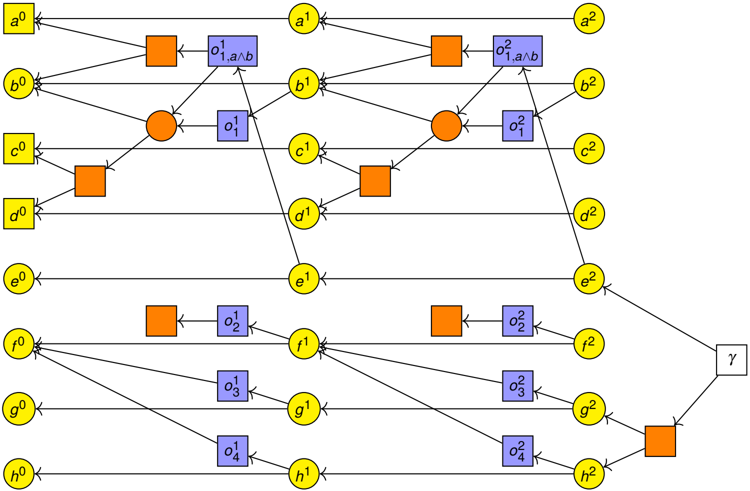
Let $\Pi^+ = \langle A, I, O^+, \gamma \rangle$ be a relaxed planning task. Then the truth values of the nodes of its depth-k relaxed planning graph $RPG_k(\Pi^+)$ relate to the forward sets and forward plan steps of Π^+ as follows:

- *Proposition nodes:*
  For all $a \in A$ and $i \in \{0, \dots, k\}$, $val(a^i) = 1$ iff $a \in S_i^F$.
- *(Unconditional) effect nodes:*
  For all $o \in O^+$ and $i \in \{1, \dots, k\}$, $val(o^i) = 1$ iff $o \in \omega_i^F$.
- *Goal nodes:*
  $val(n_\gamma^k) = 1$ iff the parallel forward distance of Π^+ is at most k.

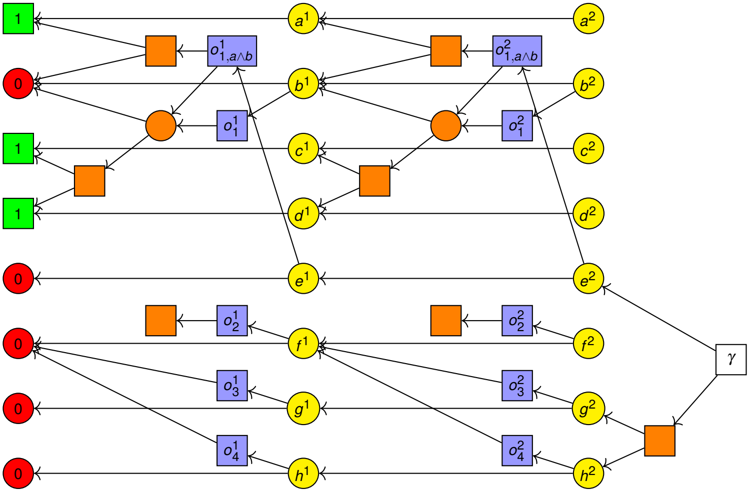
(We omit the straight-forward proof.)

Parallel plans
Relaxed planning graphs
Introduction
Construction
Truth values
Relaxation heuristics
Summary

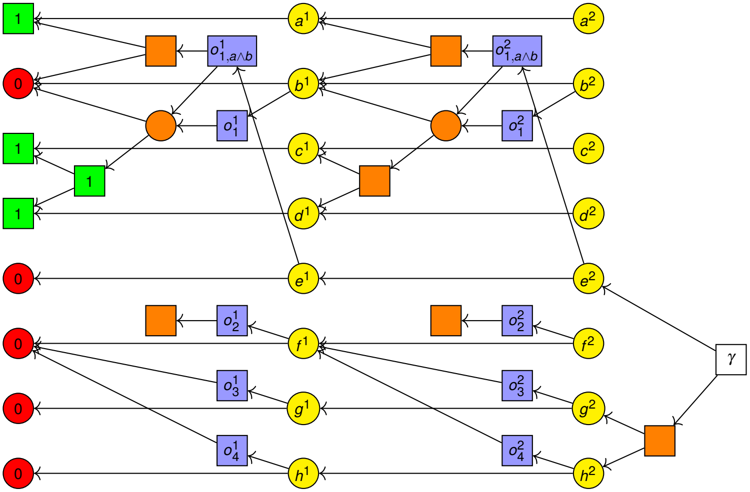
Computing the node truth values



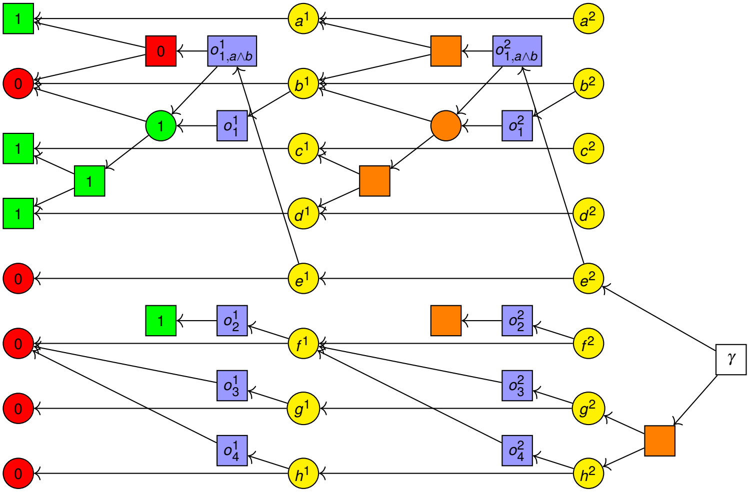
Computing the node truth values



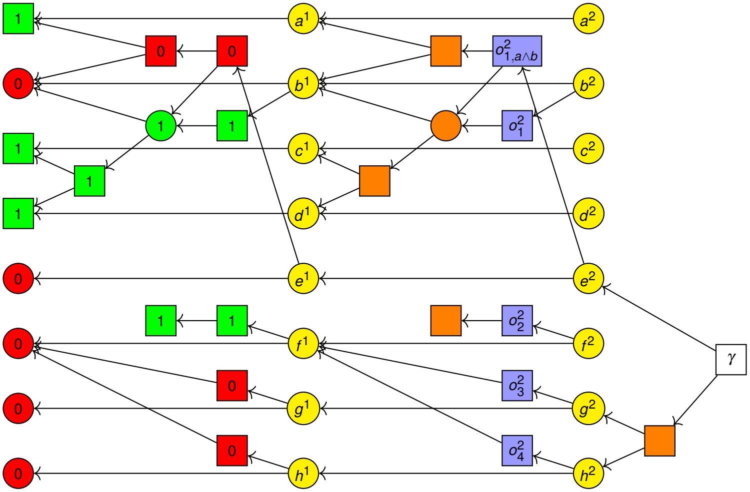
Computing the node truth values



Computing the node truth values



Computing the node truth values



Parallel plans

Relaxed planning graphs

Introduction

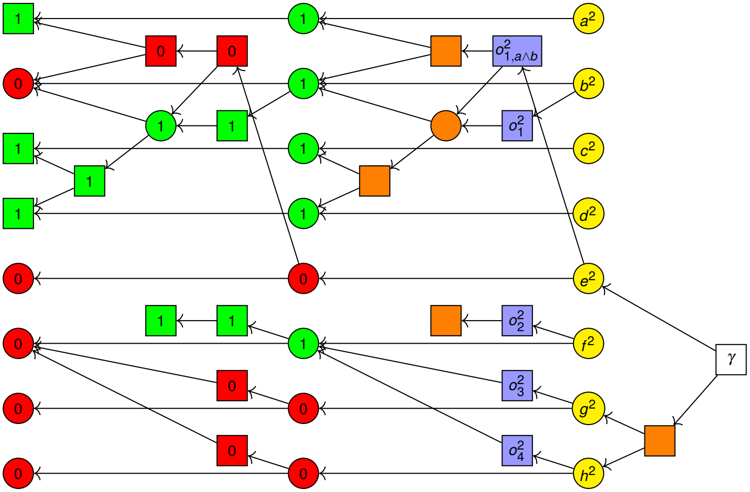
Construction

Truth values

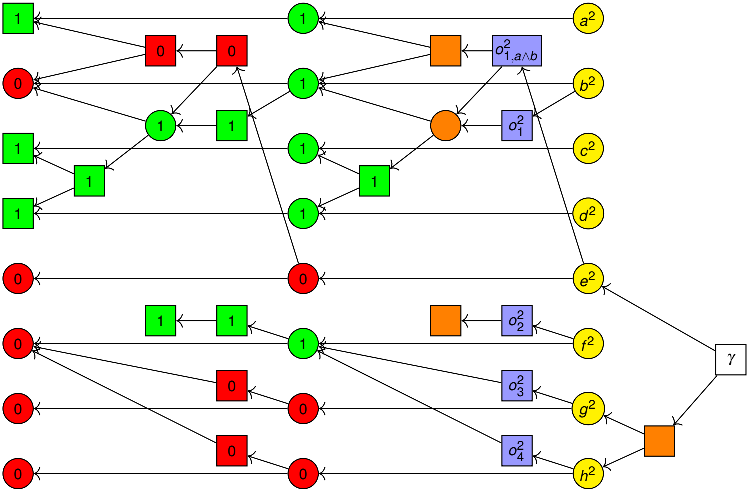
Relaxation heuristics

Summary

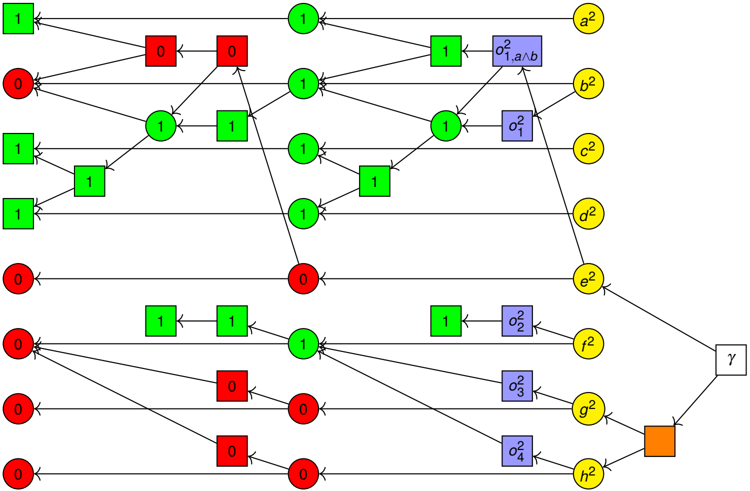
Computing the node truth values



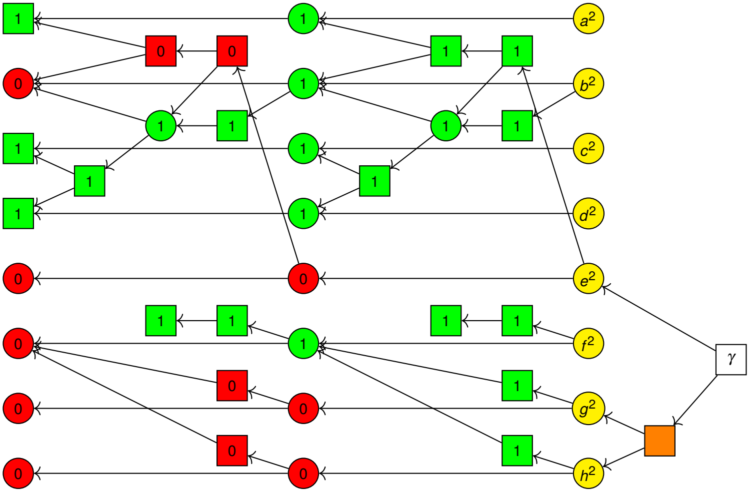
Computing the node truth values



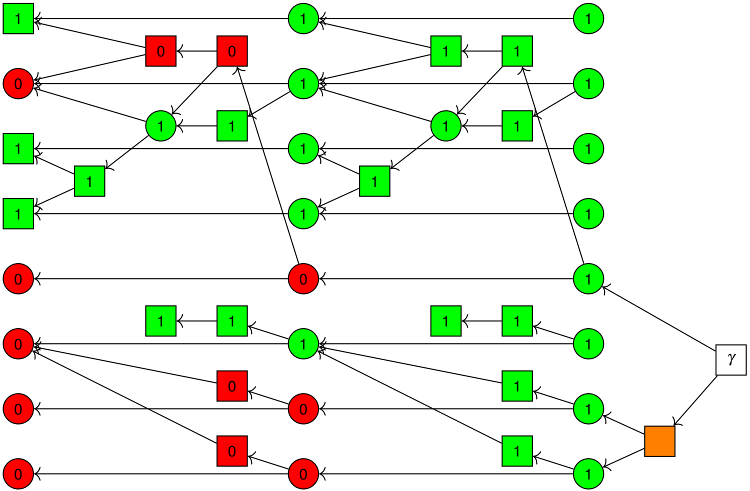
Computing the node truth values



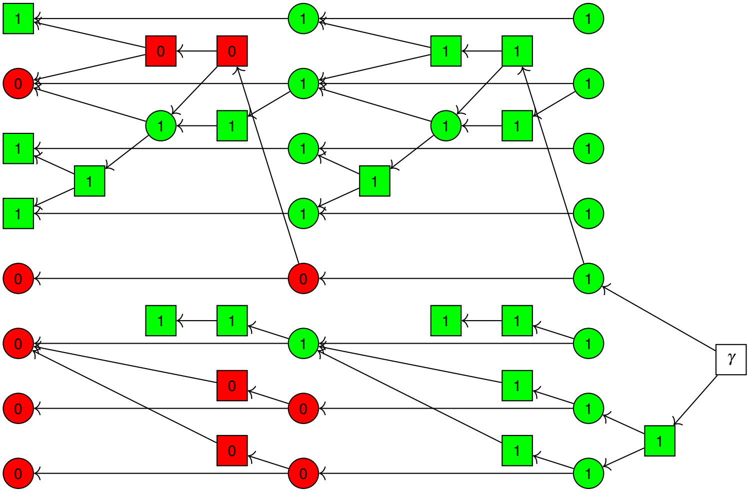
Computing the node truth values



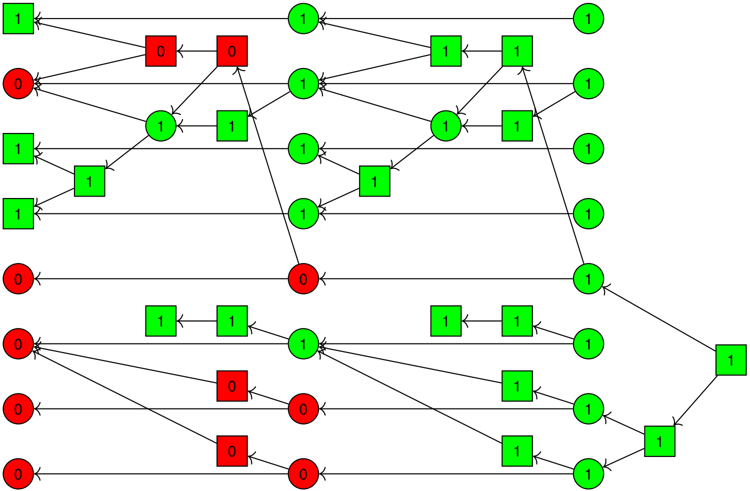
Computing the node truth values



Computing the node truth values



Computing the node truth values



Relaxed planning graphs for STRIPS

Remark: Relaxed planning graphs have historically been defined for STRIPS tasks only. In this case, we can simplify:

- Only one effect node per operator: STRIPS does not have conditional effects.
  - Because each operator has only one effect node, effect nodes are called operator nodes in relaxed planning graphs for STRIPS.
- No goal nodes: The test whether all goals are reached is done by the algorithm that evaluates the AND/OR dag.
- No formula nodes: Operator nodes are directly connected to their preconditions.

$\rightsquigarrow$ Relaxed planning graphs for STRIPS are layered digraphs and only have proposition and operator nodes.

Relaxation heuristics

Computing parallel forward distances from RPGs

Parallel plans
Relaxed planning graphs
Relaxation heuristics
Generic template
h_{max}
h_{add}
h_{sa}
Incremental computation
h_{FF}
Comparison & practice
Summary

So far, relaxed planning graphs offer us a way to compute parallel forward distances:

Parallel forward distances from relaxed planning graphs

def *parallel-forward-distance*(Π^+):
    Let A be the set of state variables of Π^+.
    for $k \in \{0, 1, 2, \dots\}$:
        $rpg := RPG_k(\Pi^+)$
        Evaluate truth values for *rpg*.
        if goal node of *rpg* has value 1:
            return k
        else if $k = |A|$:
            return ∞

Remarks on the algorithm

- The relaxed planning graph for depth $k \geq 1$ can be built incrementally from the one for depth $k-1$:
  - Add new layer k.
  - Move goal subgraph from layer $k-1$ to layer k.
- Similarly, all truth values up to layer $k-1$ can be reused.
- Thus, overall computation with maximal depth m requires time $O(\|RPG_m(\Pi^+)\|) = O((m+1)\cdot\|\Pi^+\|)$.
- This is not a very efficient way of computing parallel forward distances (and wouldn't be used in practice).
- However, it allows computing additional information for the relaxed planning graph nodes along the way, which can be used for heuristic estimates.

Computing heuristics from relaxed planning graphs

def *generic-rpg-heuristic*($\langle A, I, O, \gamma \rangle, s$):

  $\Pi^+ := \langle A, s, O^+, \gamma \rangle$

  for $k \in \{0, 1, 2, \ldots\}$:

    $rpg := RPG_k(\Pi^+)$

    Evaluate truth values for *rpg*.

    if goal node of *rpg* has value 1:

      Annotate true nodes of *rpg*.

      if termination criterion is true:

        return heuristic value from annotations

    else if $k = |A|$:

      return ∞

- $\rightsquigarrow$ generic template for heuristic functions
- $\rightsquigarrow$ to get concrete heuristic: fill in highlighted parts

Concrete examples for the generic heuristic

Many planning heuristics fit the generic template:

- additive heuristic h_{add} (Bonet, Loerincs & Geffner, 1997)
- max heuristic $h_{\max}$ (Bonet & Geffner, 1999)
- FF heuristic h_{FF} (Hoffmann & Nebel, 2001)
- cost-sharing heuristic h_{cs} (Mirkis & Domshlak, 2007)
  - not covered in this course
- set-additive heuristic h_{sa} (Keyder & Geffner, 2008)

Remarks:

- For all these heuristics, equivalent definitions that don't refer to relaxed planning graphs are possible.
- Historically, such equivalent definitions have mostly been used for $h_{\max}$, h_{add} and h_{sa}.
- For those heuristics, the most efficient implementations do not use relaxed planning graphs explicitly.

Forward cost heuristics

- The simplest relaxed planning graph heuristics are forward cost heuristics.
- Examples: h_{max}, h_{add}
- Here, node annotations are cost values (natural numbers).
- The cost of a node estimates how expensive (in terms of required operators) it is to make this node true.

Forward cost heuristics: fitting the template

Forward cost heuristics

Computing annotations:

- Propagate cost values bottom-up using a combination rules for OR nodes and for AND nodes.
- At effect nodes, add 1 after applying combination rule.

Termination criterion:

- stability: terminate if cost for proposition node a^k equals cost for a^{k-1} for all true propositions a in layer k

Heuristic value:

- The heuristic value is the cost of the goal node.

- Different forward cost heuristics only differ in their choice of combination rules.

The max heuristic $h_{\max}$ (again)

Forward cost heuristics: max heuristic $h_{\max}$

Combination rule for AND nodes:

- $cost(u) = \max(\{cost(v_1), \ldots, cost(v_k)\})$
  (with $\max(\emptyset) := 0$)

Combination rule for OR nodes:

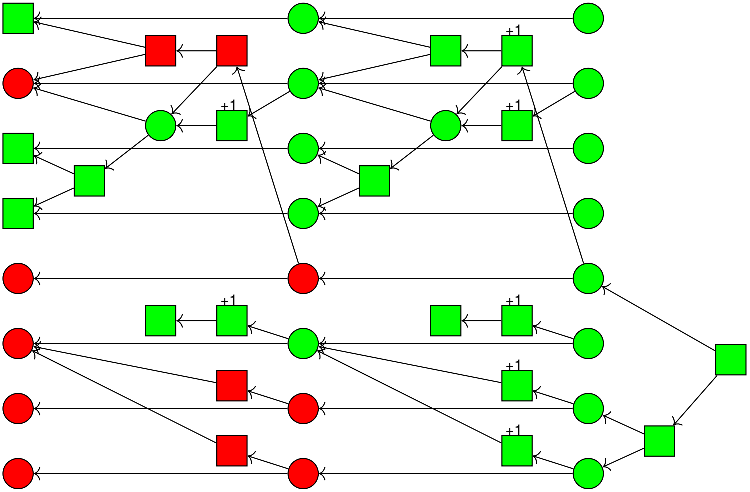
- $cost(u) = \min(\{cost(v_1), \ldots, cost(v_k)\})$

In both cases, $\{v_1, \ldots, v_k\}$ is the set of true successors of u.

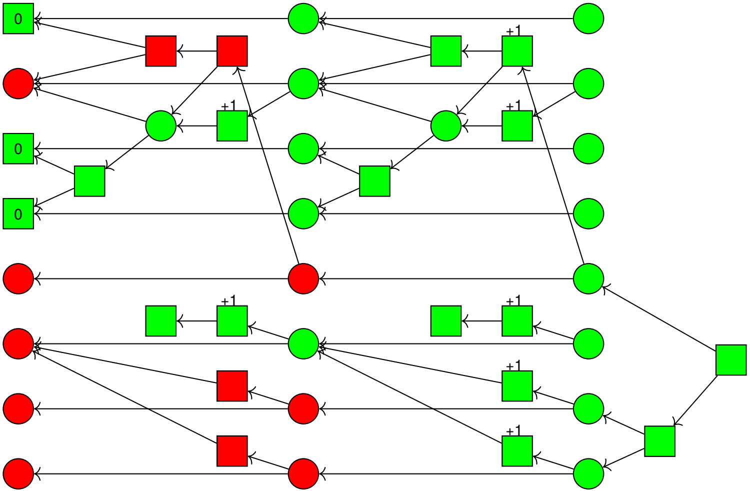
Intuition:

- AND rule: If we have to achieve several conditions, estimate this by the most expensive cost.
- OR rule: If we have a choice how to achieve a condition, pick the cheapest possibility.

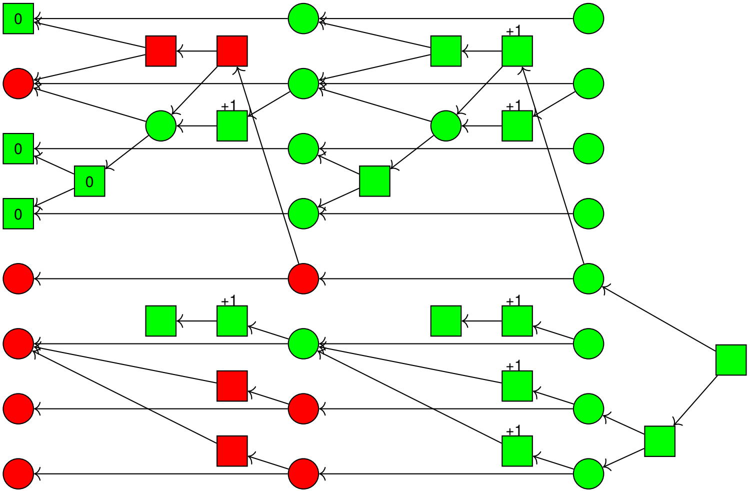
Running example: $h_{\max}$



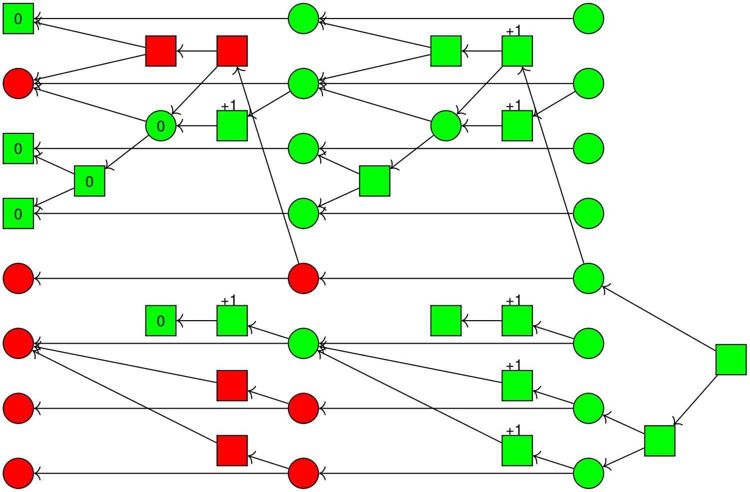
Running example: $h_{\max}$



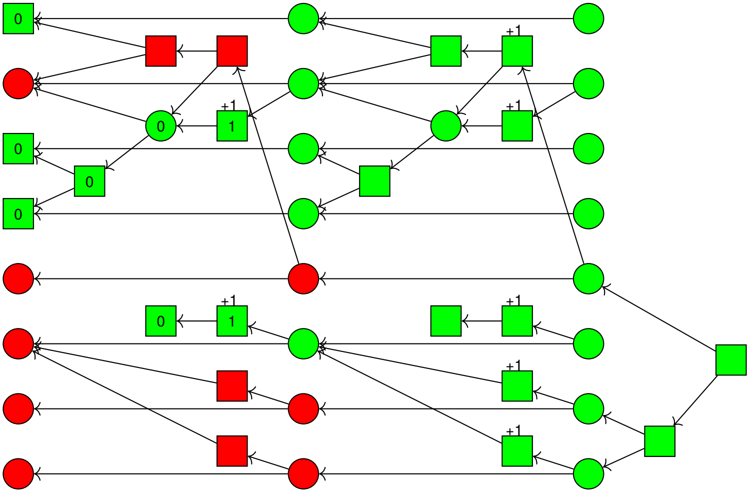
Running example: $h_{\max}$



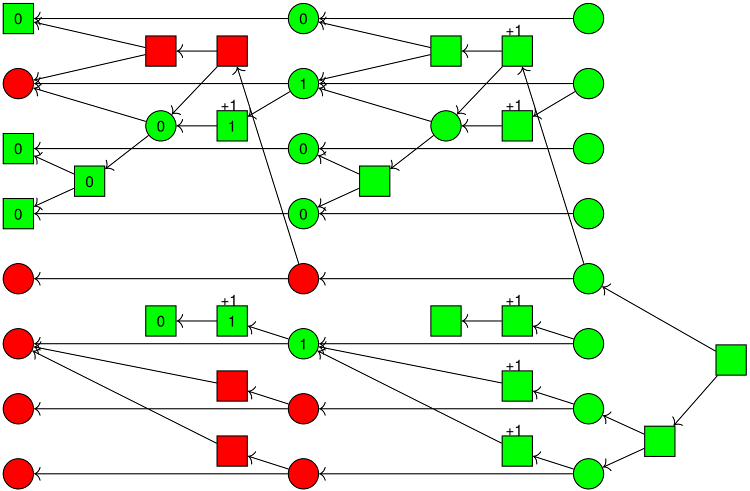
Running example: $h_{\max}$



Running example: $h_{\max}$



Running example: $h_{\max}$



Parallel plans

Relaxed planning graphs

Relaxation heuristics

Generic template

$h_{\max}$

h_{add}

h_{sa}

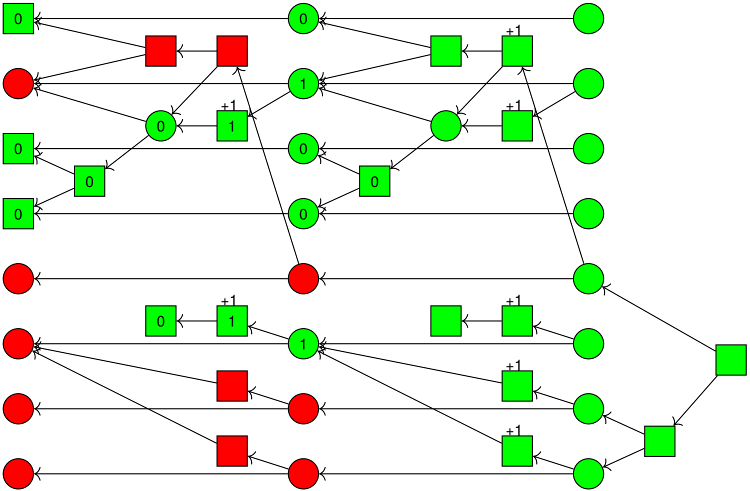
Incremental computation

h_{FF}

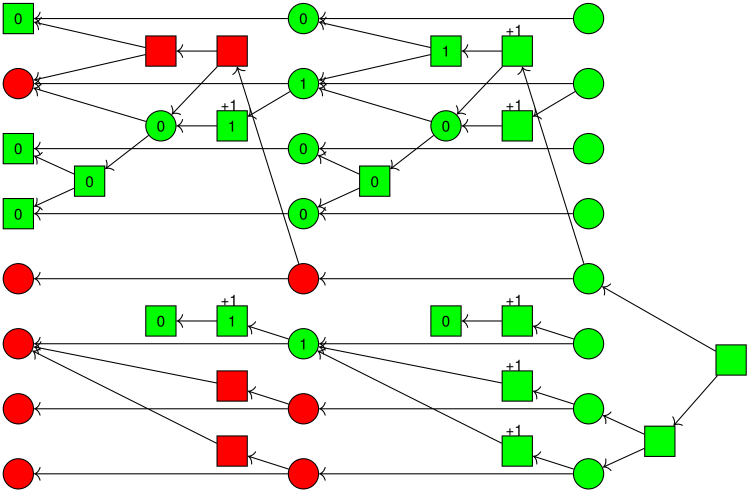
Comparison & practice

Summary

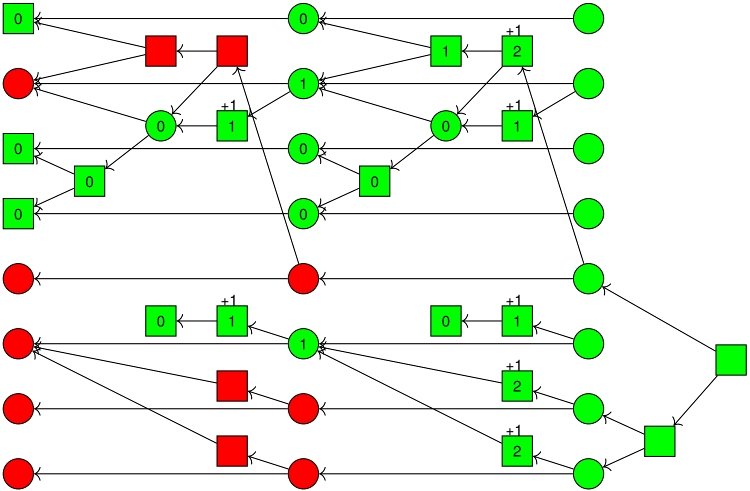
Running example: $h_{\max}$



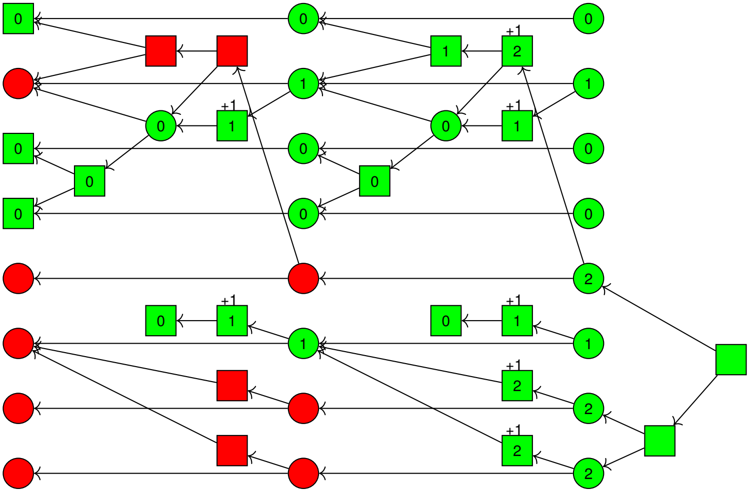
Running example: $h_{\max}$



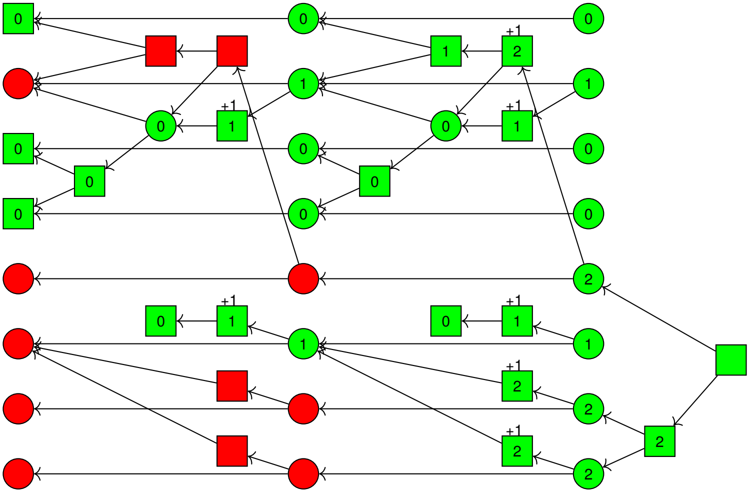
Running example: $h_{\max}$



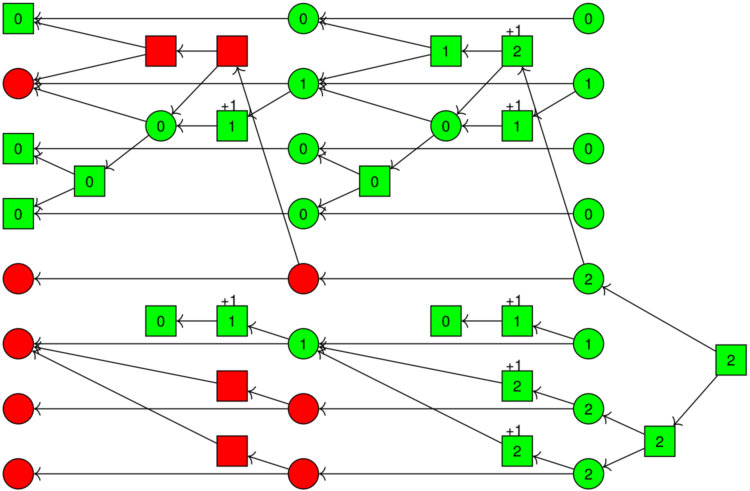
Running example: $h_{\max}$



Running example: $h_{\max}$



Running example: $h_{\max}$



Remarks on h_{max}

- The definition of h_{max} as a forward cost heuristic is equivalent to our earlier definition in this chapter.
- Unlike the earlier definition, it generalizes to an extension where every operator has an associated non-negative cost (rather than all operators having cost 1).
- In the case without costs (and only then), it is easy to prove that the goal node has the same cost in all graphs $RPG_k(\Pi^+)$ where it is true. (Namely, the cost is equal to the lowest value of k for which the goal node is true.)
- We can thus terminate the computation as soon as the goal becomes true, without waiting for stability.
- The same is not true for other forward-propagating heuristics (h_{add}, h_{cs}, h_{sa}).

The additive heuristic

Forward cost heuristics: additive heuristic h_{add}

Combination rule for AND nodes:

- $cost(u) = cost(v_1) + \ldots + cost(v_k)$
  (with $\sum(\emptyset) := 0$)

Combination rule for OR nodes:

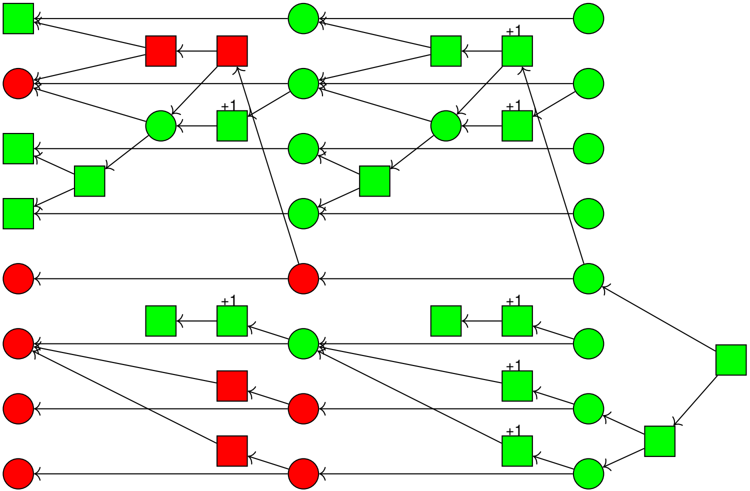
- $cost(u) = \min(\{cost(v_1), \ldots, cost(v_k)\})$

In both cases, $\{v_1, \ldots, v_k\}$ is the set of true successors of u.

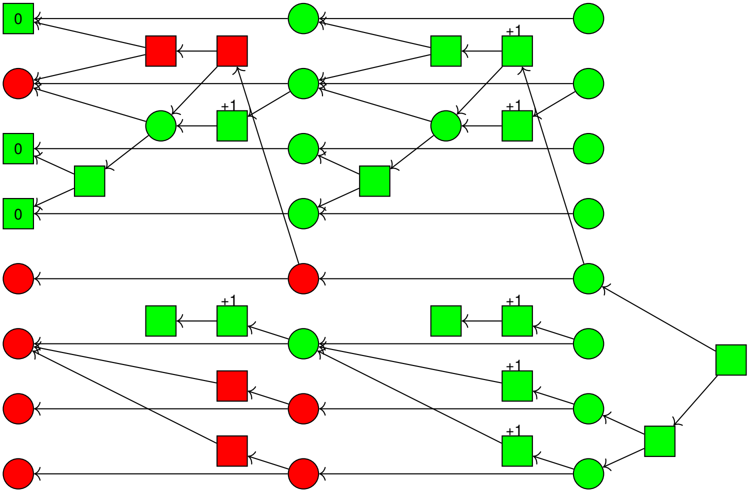
Intuition:

- AND rule: If we have to achieve several conditions, estimate this by the cost of achieving each in isolation.
- OR rule: If we have a choice how to achieve a condition, pick the cheapest possibility.

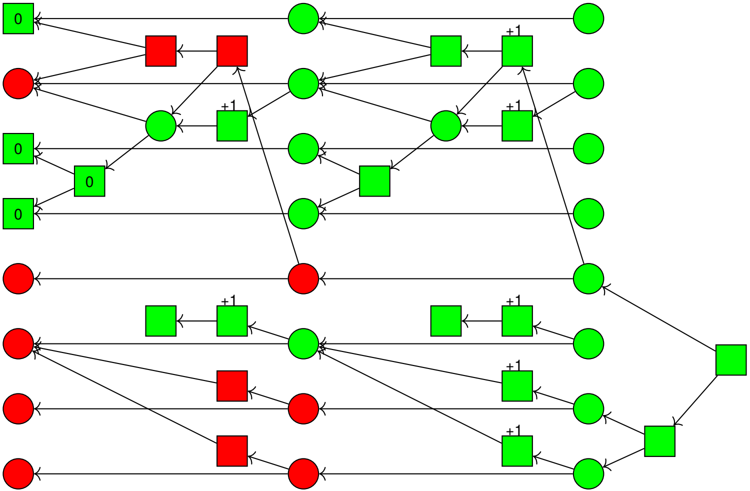
Running example: h_{add}



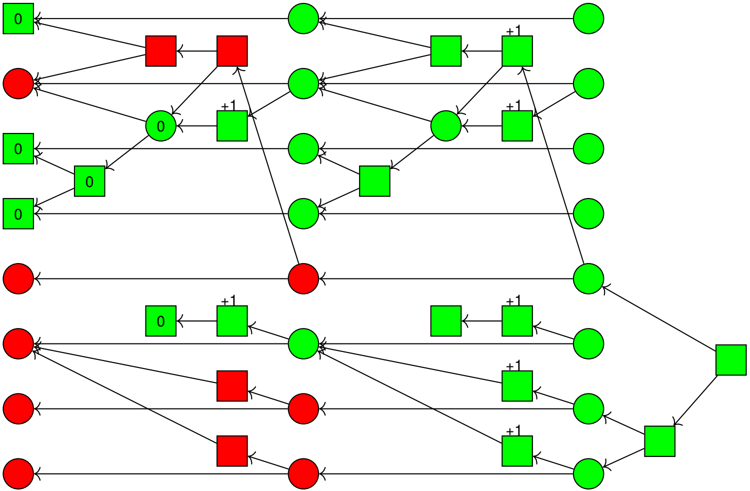
Running example: h_{add}



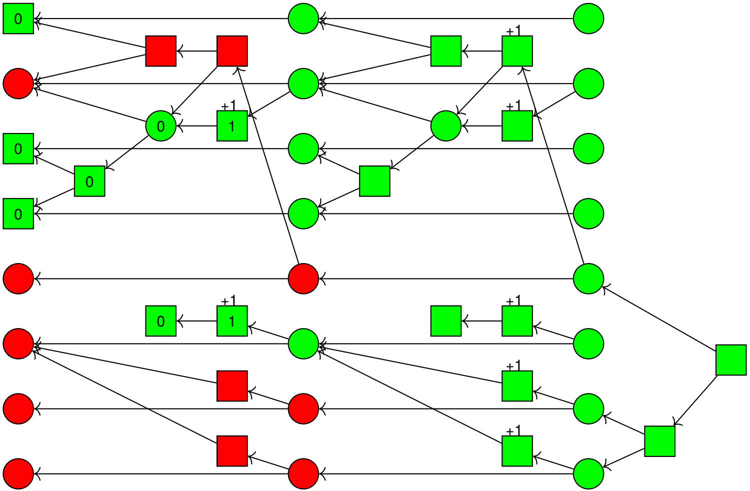
Running example: h_{add}



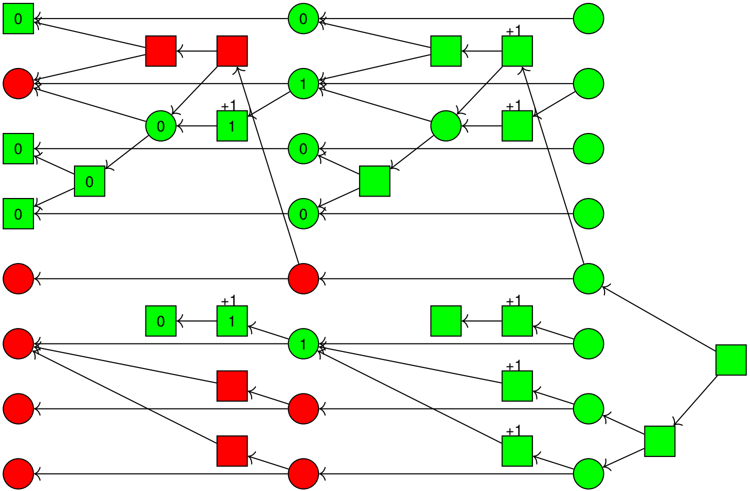
Running example: h_{add}



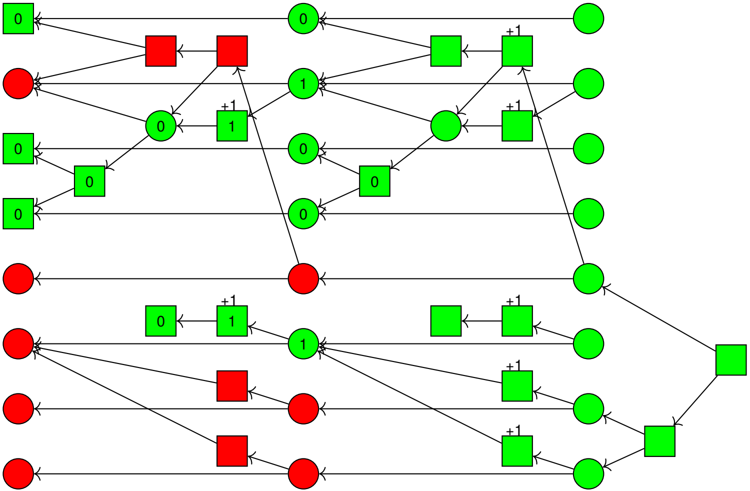
Running example: h_{add}



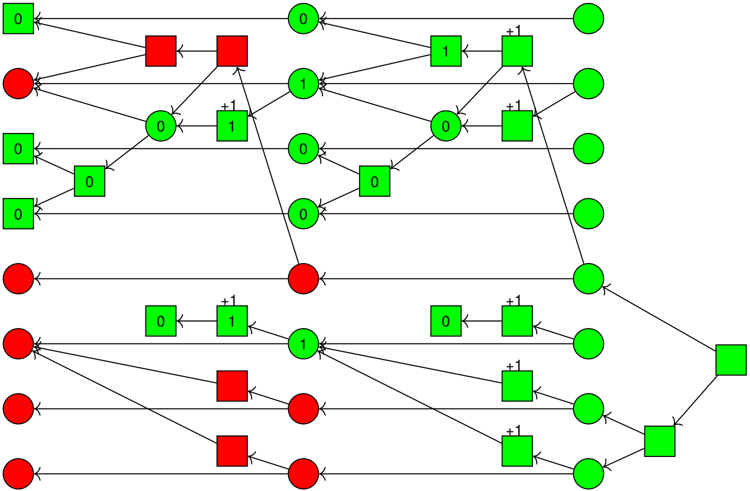
Running example: h_{add}



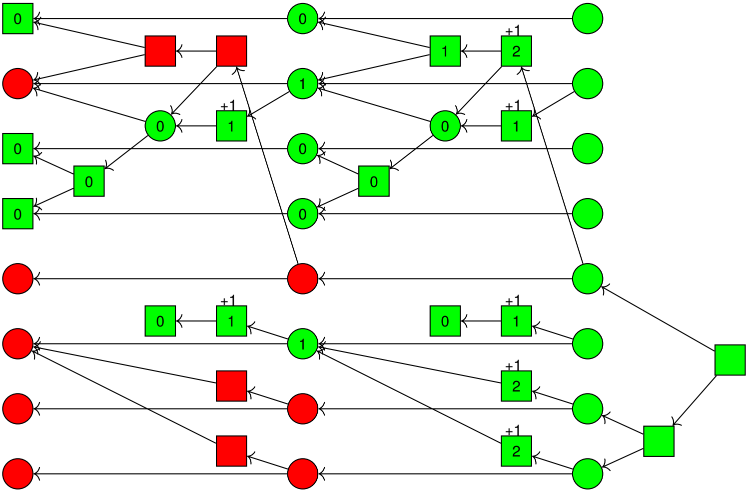
Running example: h_{add}



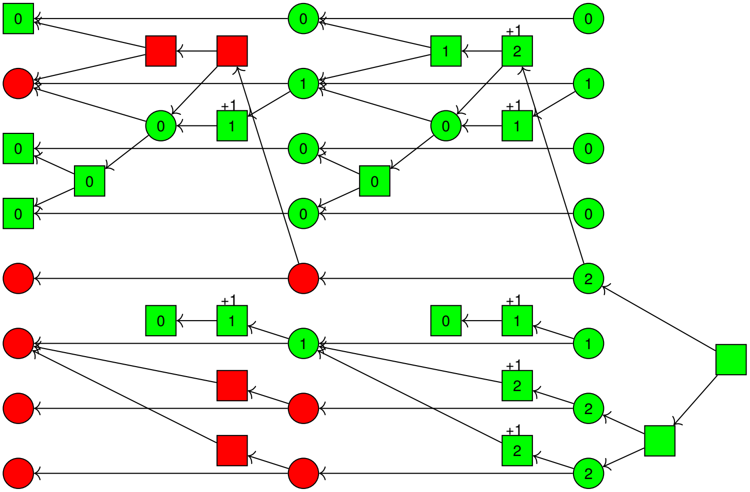
Running example: h_{add}



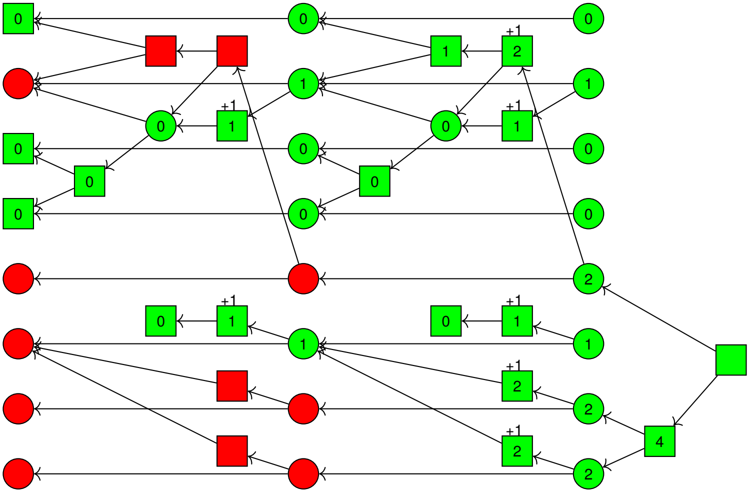
Running example: h_{add}



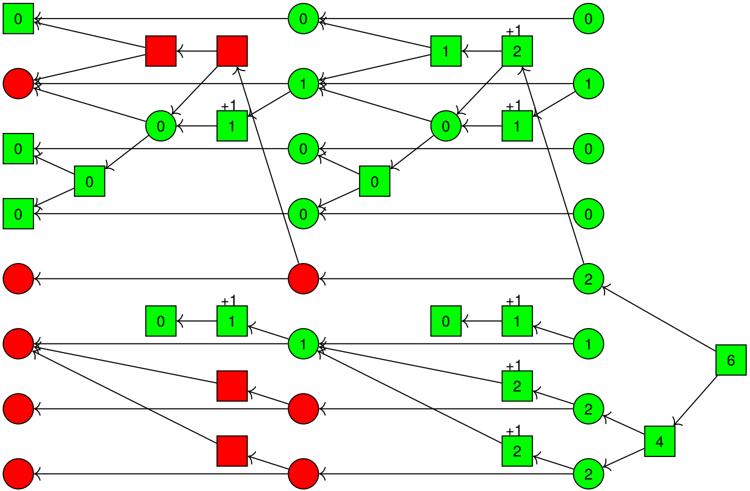
Running example: h_{add}



Running example: h_{add}



Running example: h_{add}



Remarks on h_{add}

- It is important to test for stability in computing h_{add}!
  (The reason for this is that, unlike h_{max}, cost values of true propositions can decrease from layer to layer.)
- Stability is achieved after layer $|A|$ in the worst case.
- h_{add} is safe and goal-aware.
- Unlike h_{max}, h_{add} is a very informative heuristic in many planning domains.
- The price for this is that it is not admissible (and hence also not consistent), so not suitable for optimal planning.
- In fact, it almost always overestimates the h^{+} value because it does not take positive interactions into account.

The set-additive heuristic

- We now discuss a refinement of the additive heuristic called the set-additive heuristic h_{sa}.
- The set-additive heuristic addresses the problem that h_{add} does not take positive interactions into account.
- Like h_{max} and h_{add}, h_{sa} is calculated through forward propagation of node annotations.
- However, the node annotations are not cost values, but sets of operators (kind of).
- The idea is that by taking set unions instead of adding costs, operators needed only once are counted only once.

Disclaimer: There are some quite subtle differences between the h_{sa} heuristic as we describe it here and the "real" heuristic of Keyder & Geffner. We do not want to discuss this in detail, but please note that such differences exist.

Operators needed several times

- The original h_{sa} heuristic as described in the literature is defined for STRIPS tasks and propagates sets of operators.
- This is fine because in relaxed STRIPS tasks, each operator need only be applied once.
- The same is not true in general: in our running example, operator o_1 must be applied twice in the relaxed plan.
- In general, it only makes sense to apply an operator again in a relaxed planning task if a previously unsatisfied effect condition has been made true.
- For this reason, we keep track of operator/effect condition pairs rather than just plain operators.

Set-additive heuristic: fitting the template

The set-additive heuristic h_{sa}

Computing annotations:

- Annotations are sets of operator/effect condition pairs, computed bottom-up.

  Combination rule for AND nodes:
  - $ann(u) = ann(v_1) \cup \cdots \cup ann(v_k)$ (with $\bigcup(\emptyset) := \emptyset$)

  Combination rule for OR nodes:
  - $ann(u) = ann(v_i)$ for some v_i minimizing $|ann(v_i)|$
    In case of several minimizers, use any tie-breaking rule.

  In both cases, $\{v_1, \ldots, v_k\}$ is the set of true successors of u. At effect nodes, add the corresponding operator/effect condition pair to the set after applying combination rule.

  …

Set-additive heuristic: fitting the template (ctd.)

The set-additive heuristic h_{sa} (ctd.)

Computing annotations:

- …(Effect nodes for unconditional effects are represented just by the operator, without a condition.)

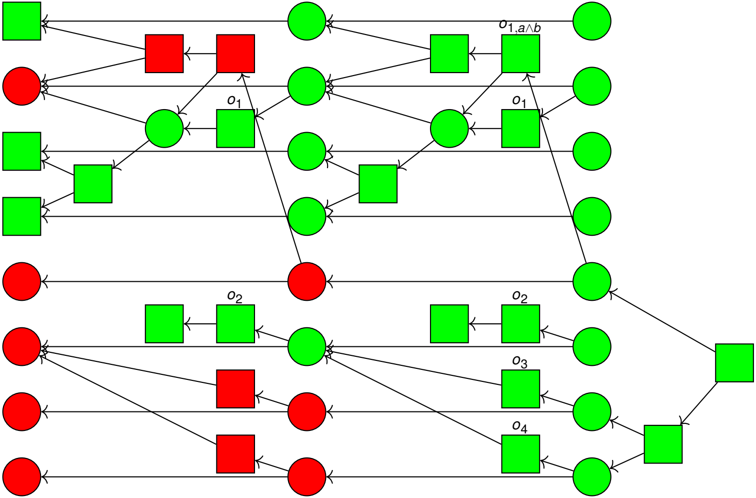
Termination criterion:

- stability: terminate if set for proposition node a^k has same cardinality as for a^{k-1} for all true propositions a in layer k

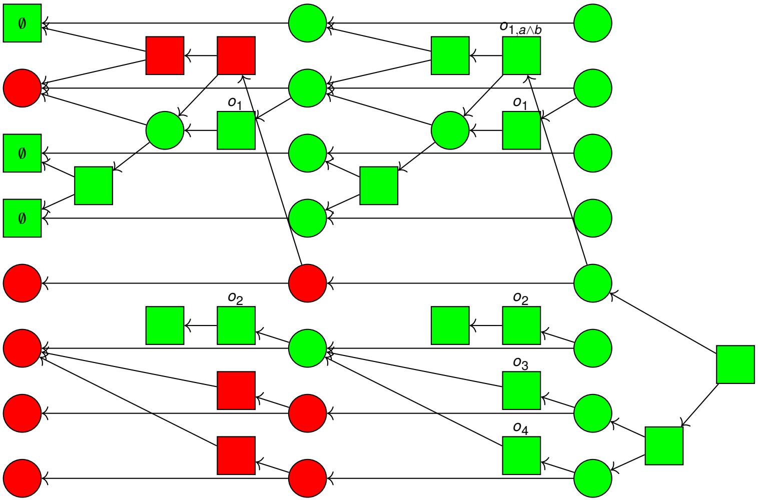
Heuristic value:

- The heuristic value is the set cardinality of the goal node annotation.

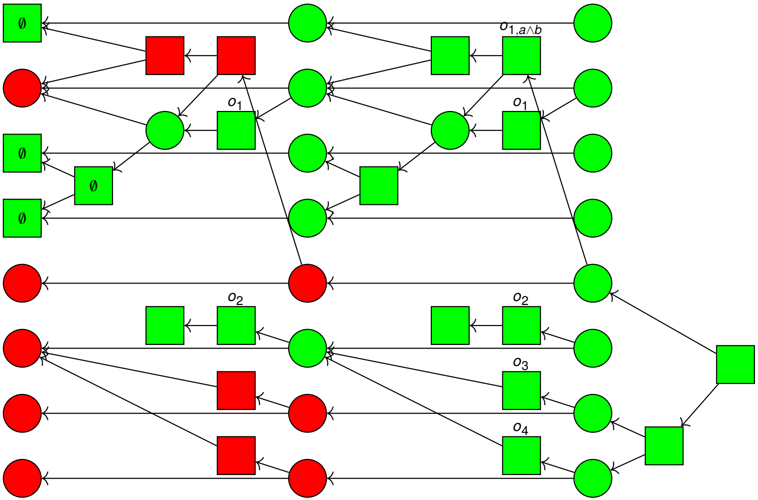
Running example: h_{sa}



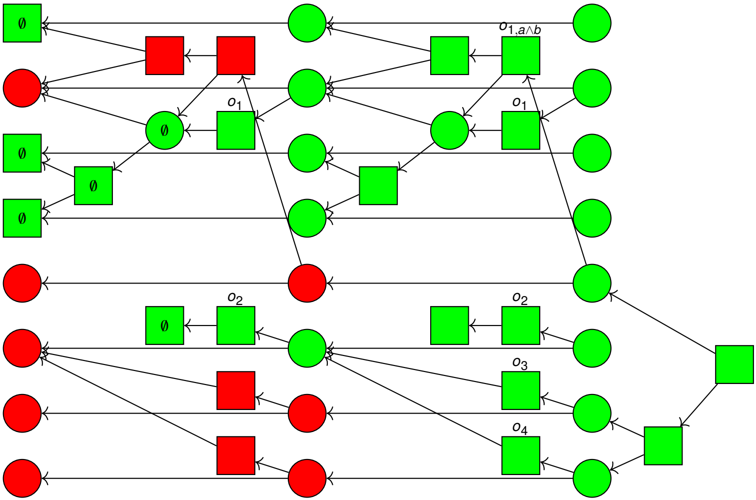
Running example: h_{sa}



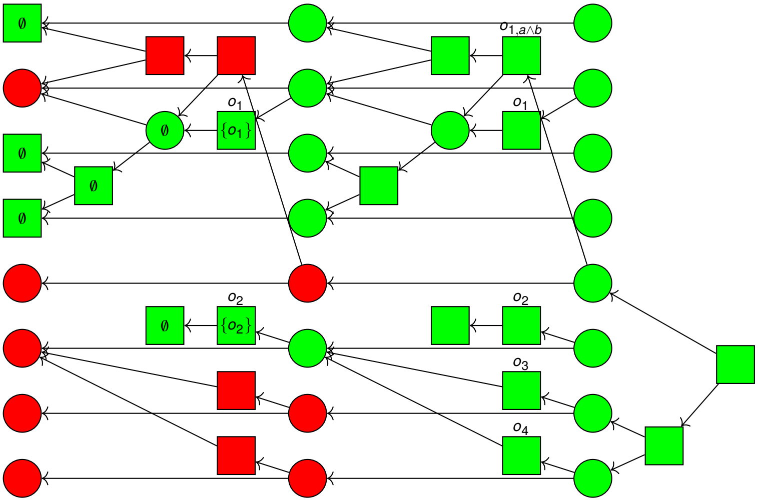
Running example: h_{sa}



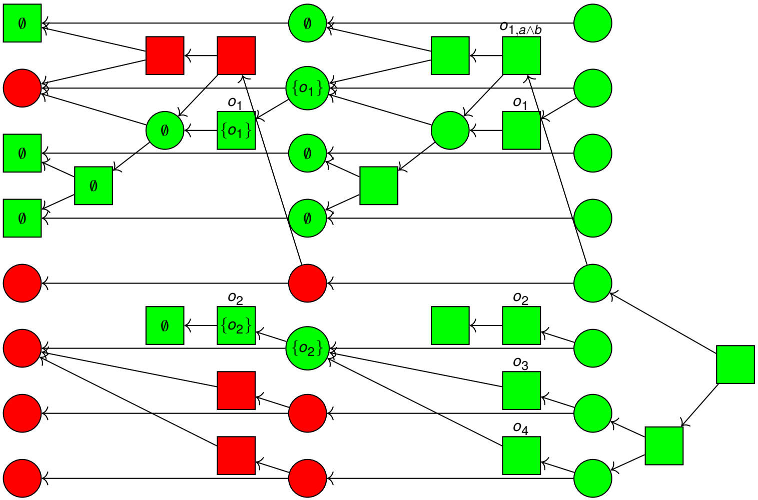
Running example: h_{sa}



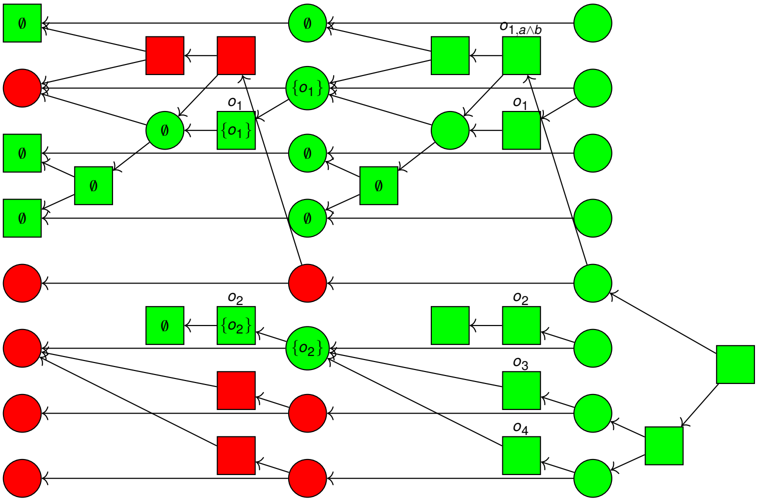
Running example: h_{sa}



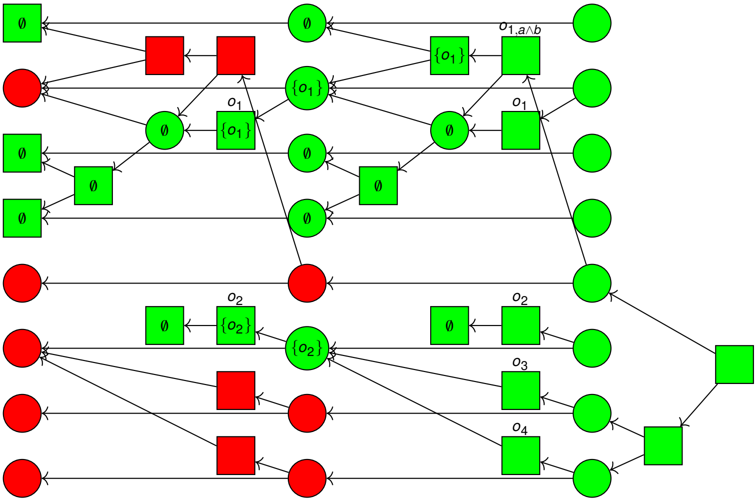
Running example: h_{sa}



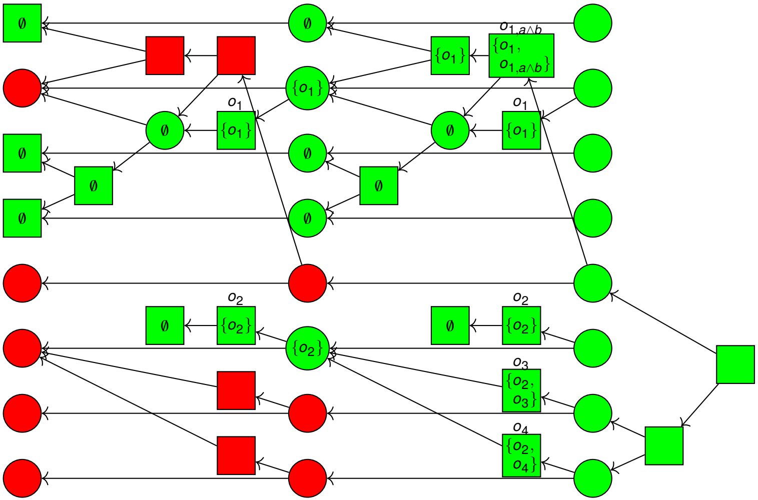
Running example: h_{sa}



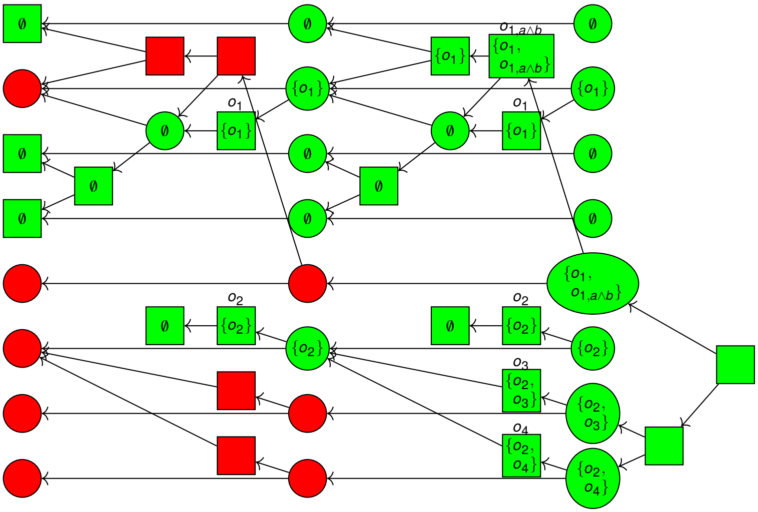
Running example: h_{sa}



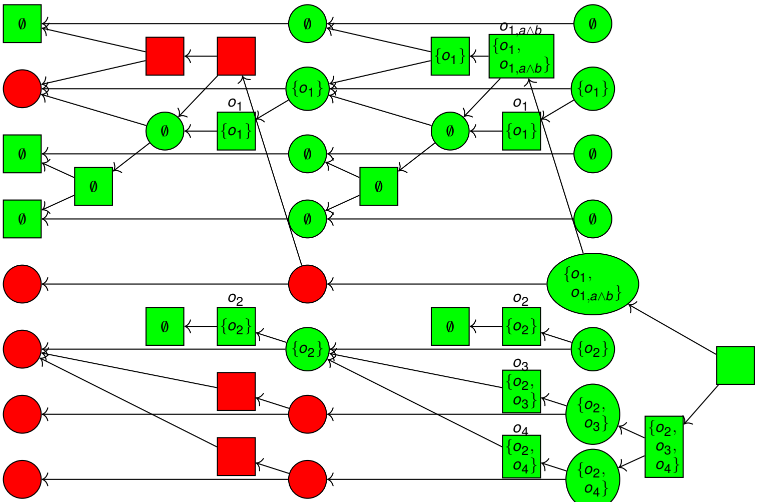
Running example: h_{sa}



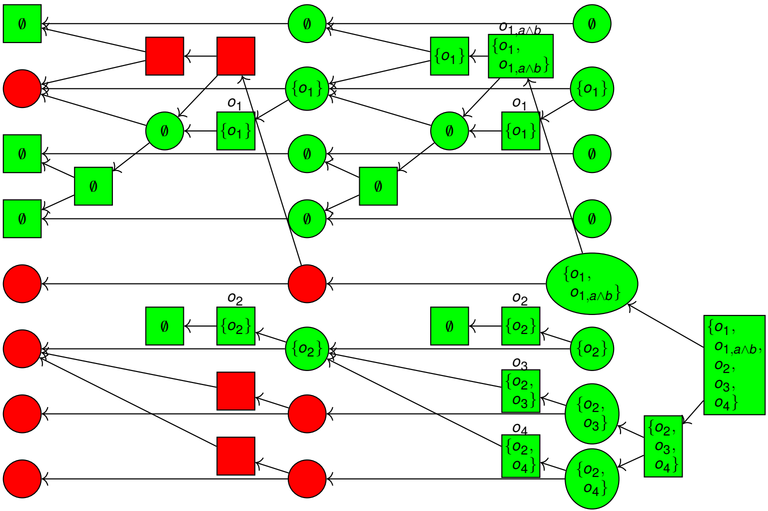
Running example: h_{sa}



Running example: h_{sa}



Running example: h_{sa}



Remarks on h_{sa}

- The same remarks for stability as for h_{add} apply.
- Like h_{add}, h_{sa} is safe and goal-aware, but neither admissible nor consistent.
- h_{sa} is generally better informed than h_{add}, but significantly more expensive to compute.
- The h_{sa} value depends on the tie-breaking rule used, so h_{sa} is not well-defined without specifying the tie-breaking rule.
- The operators contained in the goal node annotation, suitably ordered, define a relaxed plan for the task.
  - Operators mentioned several times in the annotation must be added as many times in the relaxed plan.

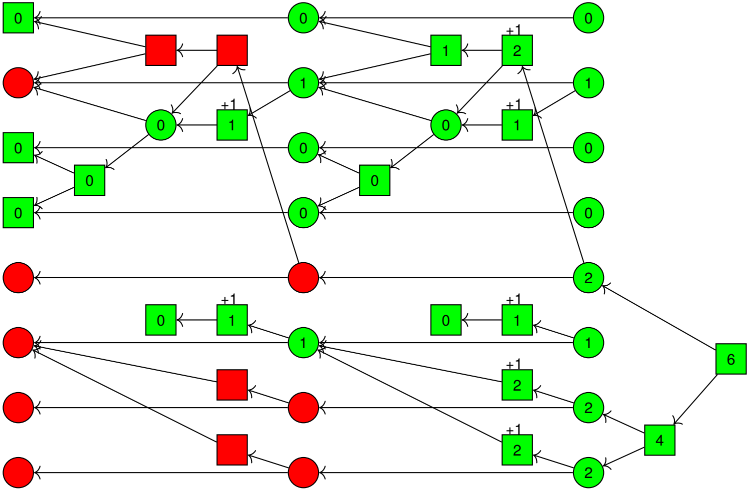
Incremental computation of forward heuristics

One nice property of forward-propagating heuristics is that they allow incremental computation:

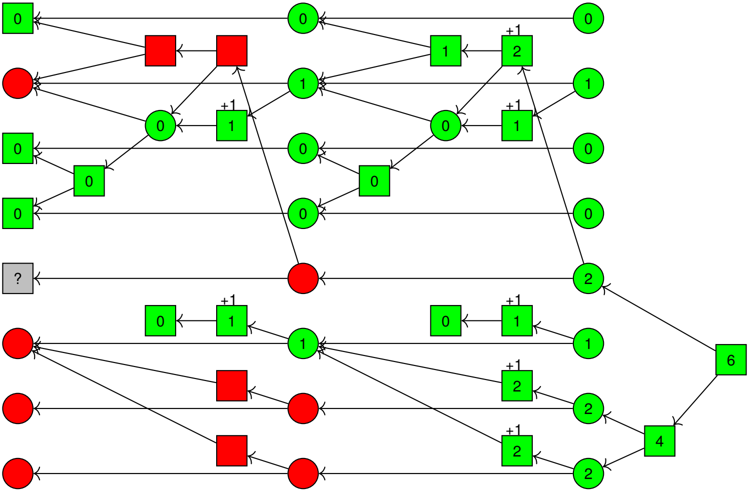
- when evaluating several states in sequence which only differ in a few state variables, can
  - start computation from previous results and
  - keep track only of what needs to be recomputed
- typical use case: depth-first style searches (e. g., IDA*)
- rarely exploited in practice

Incremental computation example: h_{add}

Result for $\{a \mapsto 1, b \mapsto 0, c \mapsto 1, d \mapsto 1, e \mapsto 0, f \mapsto 0, g \mapsto 0, h \mapsto 0\}$

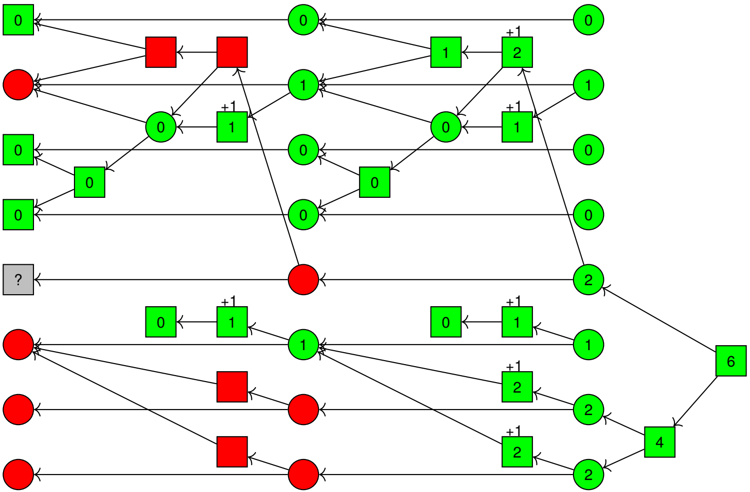


Change value of e to 1.



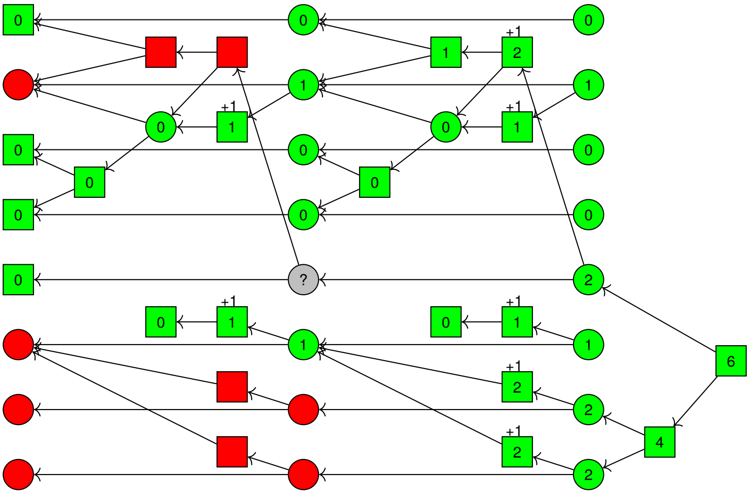
Incremental computation example: h_{add}

Recompute outdated values.



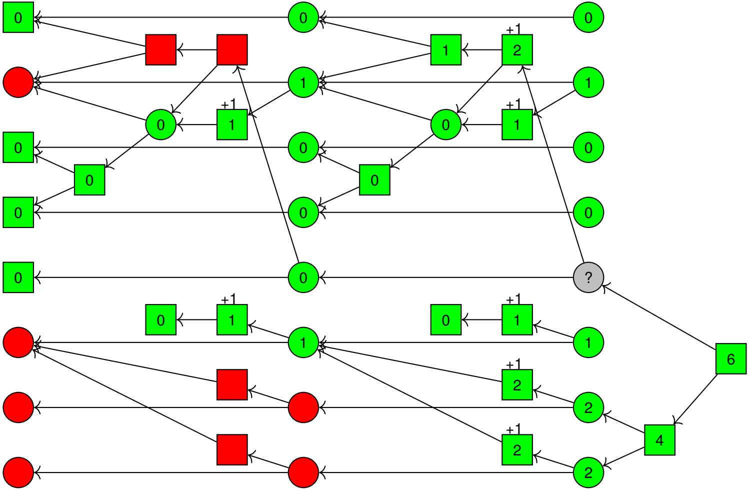
Incremental computation example: h_{add}

Recompute outdated values.



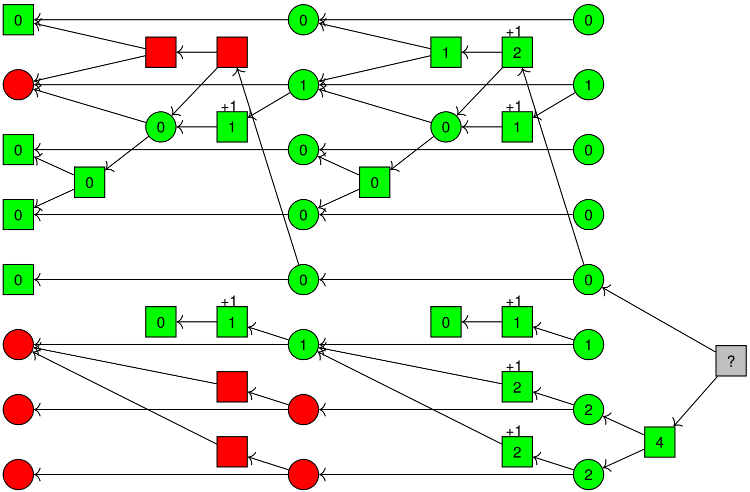
Incremental computation example: h_{add}

Recompute outdated values.



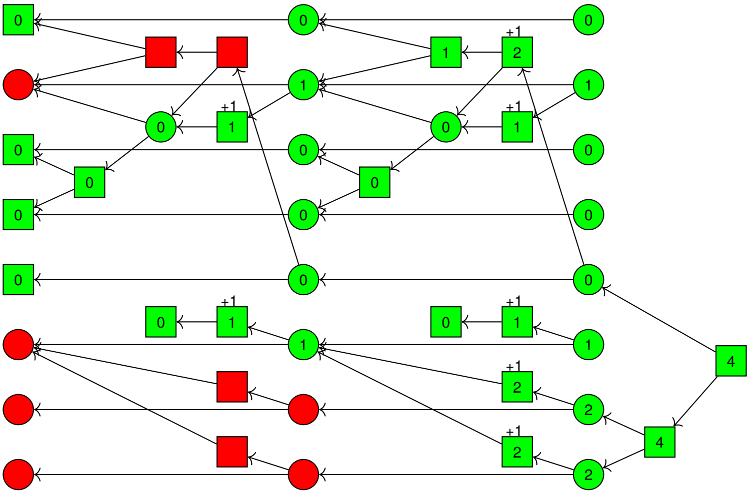
Incremental computation example: h_{add}

Recompute outdated values.



Incremental computation example: h_{add}

Recompute outdated values.



Heuristic estimate h_{FF}

- h_{sa} is more expensive to compute than the other forward propagating heuristics because we must propagate sets.
- It is possible to get the same advantage over h_{add} combined with efficient propagation.
- Key idea of h_{FF}: perform a backward propagation that selects a sufficient subset of nodes to make the goal true (called a solution graph in AND/OR dag literature).
- The resulting heuristic is almost as informative as h_{sa}, yet computable as quickly as h_{add}.

Note: Our presentation inverts the historical order. The set-additive heuristic was defined after the FF heuristic (sacrificing speed for even higher informativeness).

The FF heuristic h_{FF}

Computing annotations:

- Annotations are Boolean values, computed top-down.

  A node is marked when its annotation is set to 1 and unmarked if it is set to 0. Initially, the goal node is marked, and all other nodes are unmarked.

  We say that a true AND node is justified if all its true successors are marked, and that a true OR node is justified if at least one of its true successors is marked.

  ...

FF heuristic: fitting the template (ctd.)

The FF heuristic h_{FF} (ctd.)

Computing annotations:

- …

  Apply these rules until all marked nodes are justified:

  1. Mark all true successors of a marked unjustified AND node.
  2. Mark the true successor of a marked unjustified OR node with only one true successor.
  3. Mark a true successor of a marked unjustified OR node connected via an idle arc.
  4. Mark any true successor of a marked unjustified OR node.

  The rules are given in priority order: earlier rules are preferred if applicable.

FF heuristic: fitting the template (ctd.)

The FF heuristic h_{FF} (ctd.)

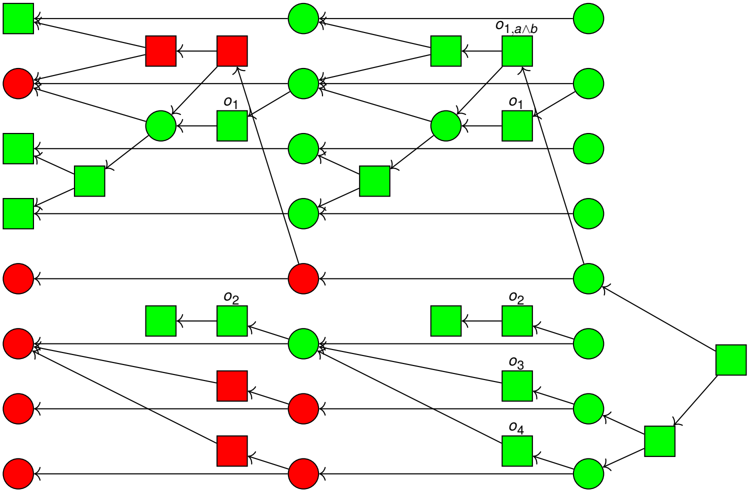
Termination criterion:

- Always terminate at first layer where goal node is true.

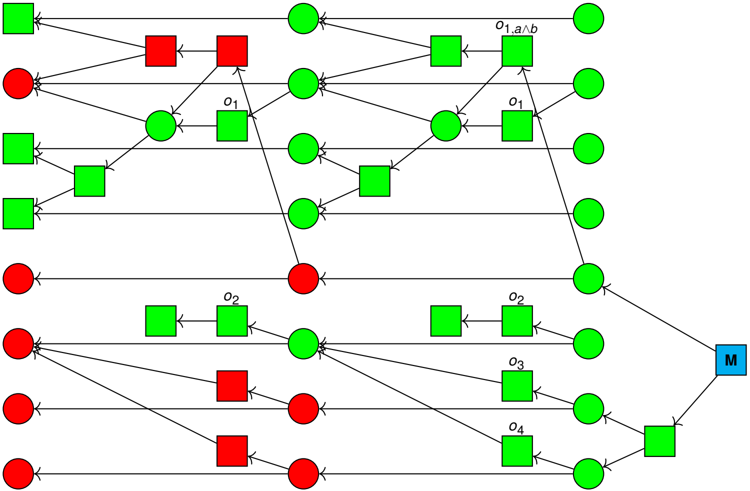
Heuristic value:

- The heuristic value is the number of operator/effect condition pairs for which at least one effect node is marked.

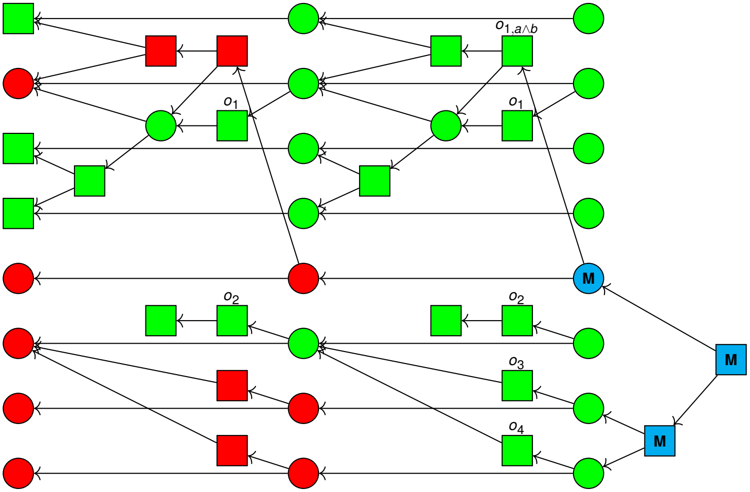
Running example: h_{FF}



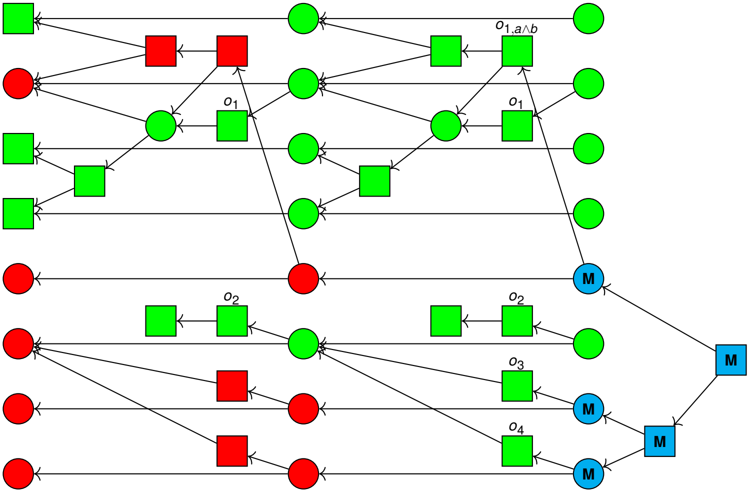
Running example: h_{FF}



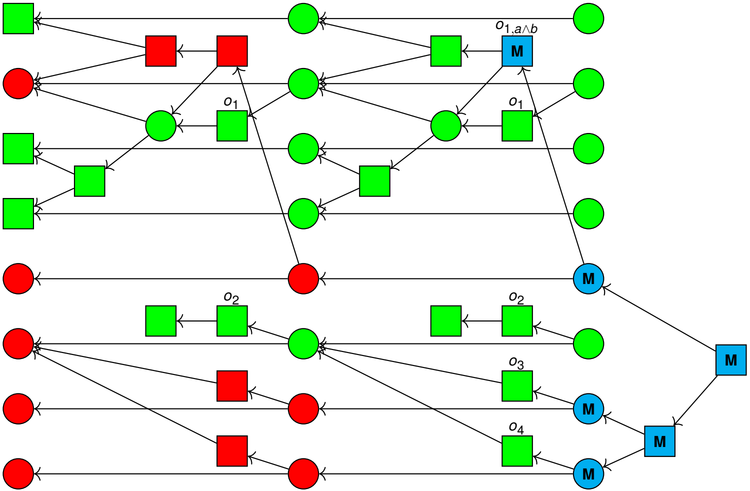
Running example: h_{FF}



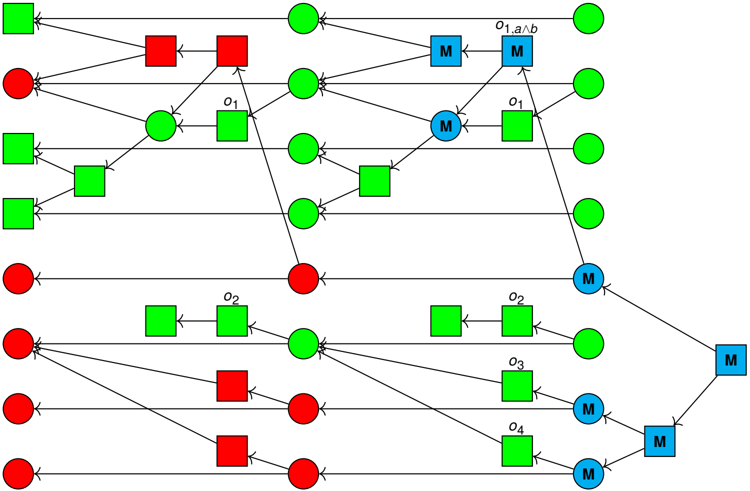
Running example: h_{FF}



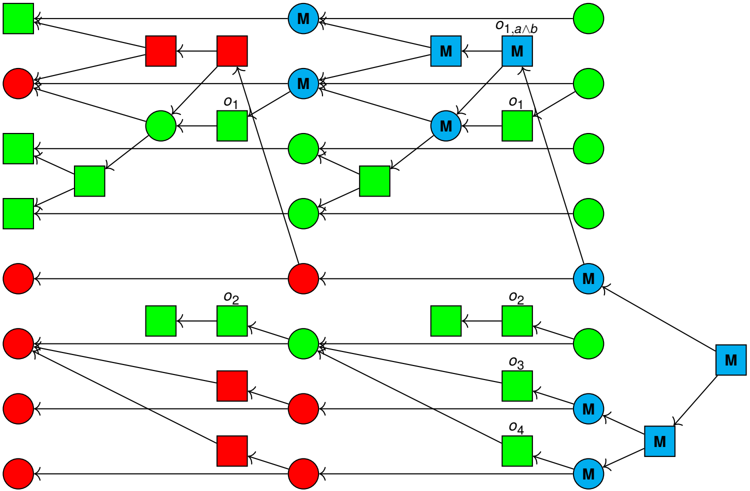
Running example: h_{FF}



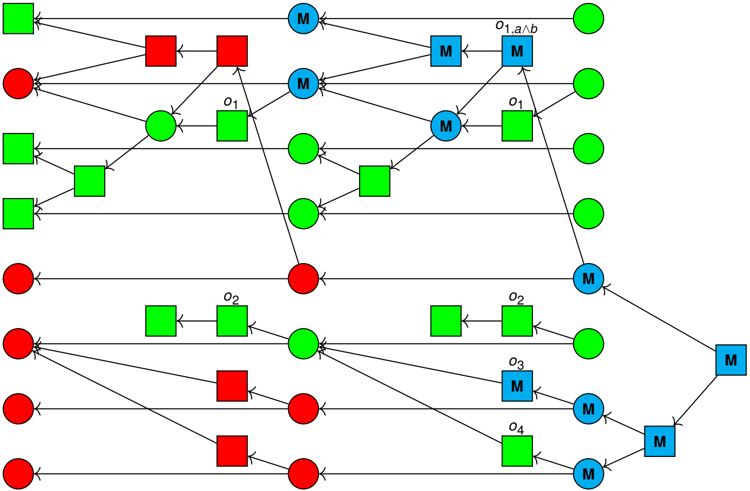
Running example: h_{FF}



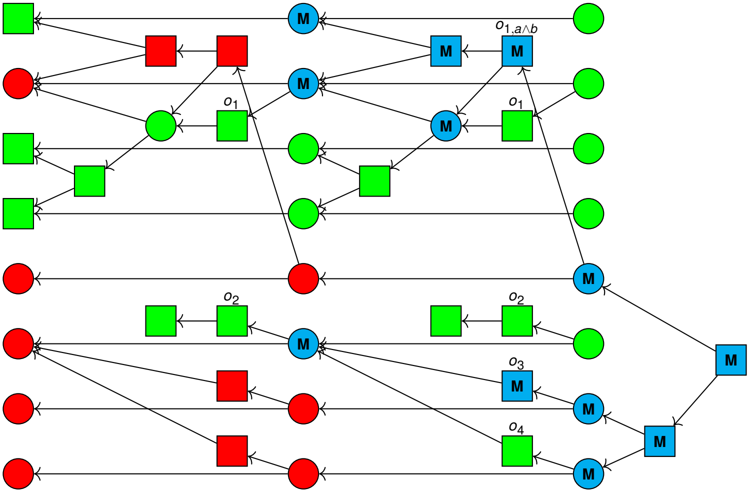
Running example: h_{FF}



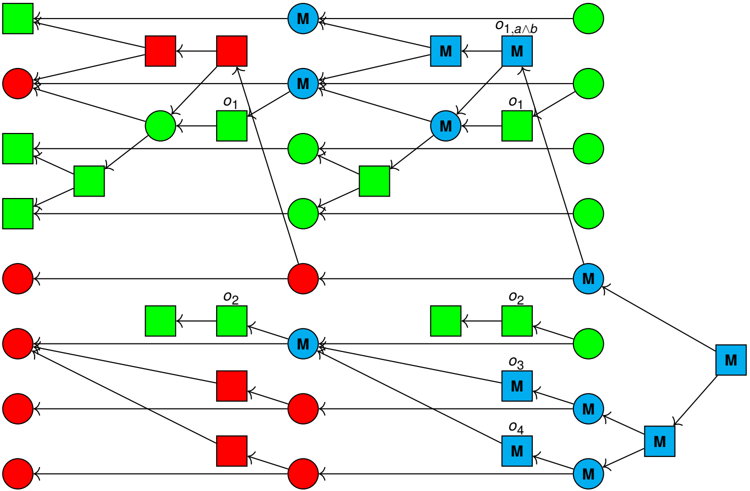
Running example: h_{FF}



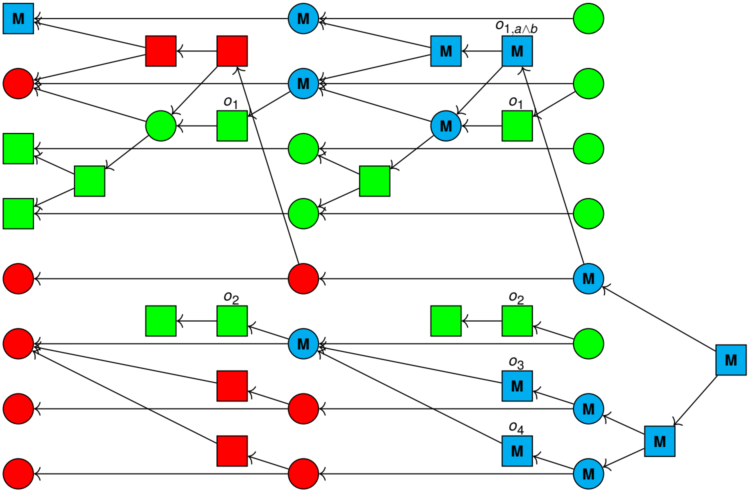
Running example: h_{FF}



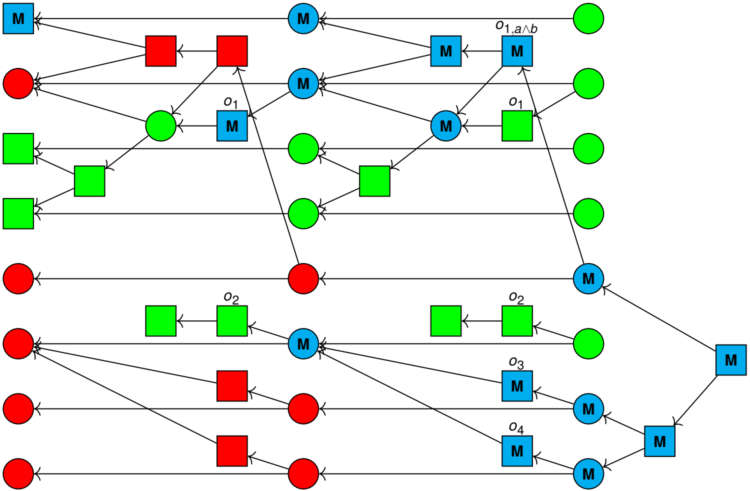
Running example: h_{FF}



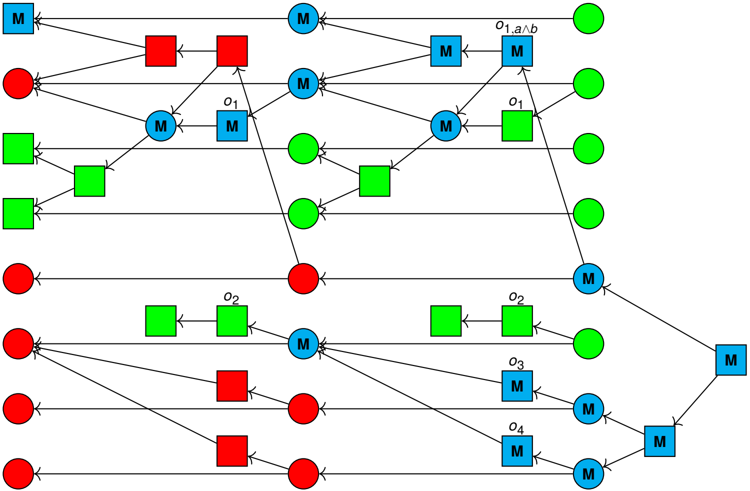
Running example: h_{FF}



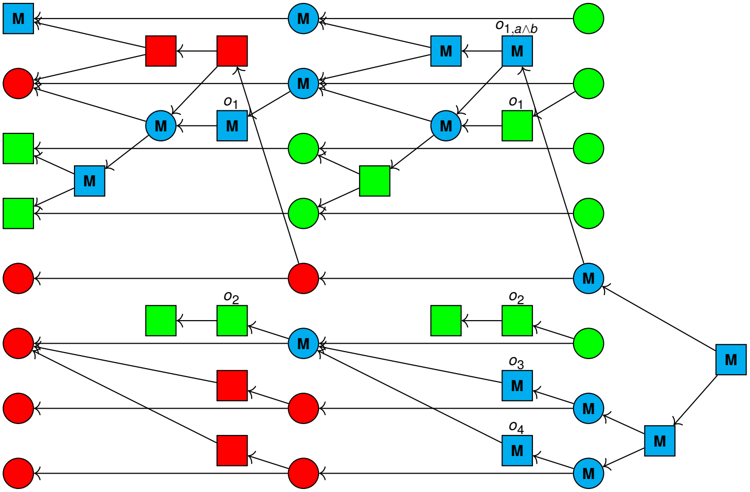
Running example: h_{FF}



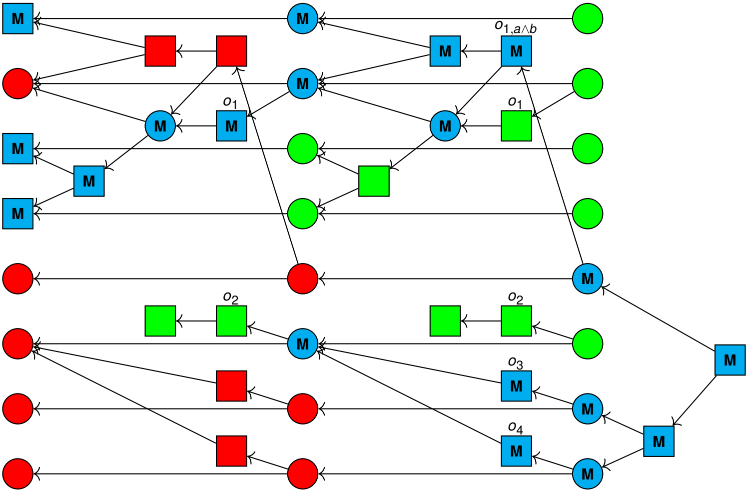
Running example: h_{FF}



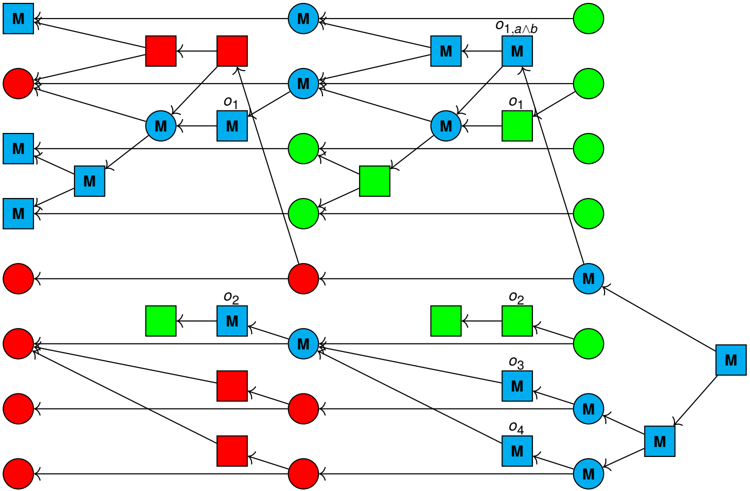
Running example: h_{FF}



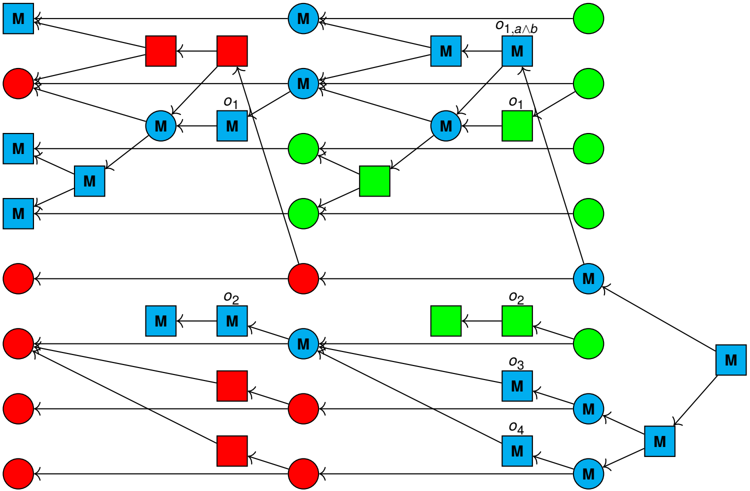
Running example: h_{FF}



Running example: h_{FF}



Running example: h_{FF}



Remarks on h_{FF}

- Like h_{add} and h_{sa}, h_{FF} is safe and goal-aware, but neither admissible nor consistent.
- Its informativeness can be expected to be slightly worse than for h_{sa}, but is usually not far off.
- Unlike h_{sa}, h_{FF} can be computed in linear time.
- Similar to h_{sa}, the operators corresponding to the marked operator/effect condition pairs define a relaxed plan.
- Similar to h_{sa}, the h_{FF} value depends on tie-breaking when the marking rules allow several possible choices, so h_{FF} is not well-defined without specifying the tie-breaking rule.
  - The implementation in FF uses additional rules of thumb to try to reduce the size of the generated relaxed plan.

Comparison of relaxation heuristics

Theorem (relationship between relaxation heuristics)

Let s be a state of planning task $\langle A, I, O, \gamma \rangle$*. Then:*

- $h_{max}(s) \leq h^{+}(s) \leq h^{*}(s)$
- $h_{max}(s) \leq h^{+}(s) \leq h_{sa}(s) \leq h_{add}(s)$
- $h_{max}(s) \leq h^{+}(s) \leq h_{FF}(s) \leq h_{add}(s)$
- h^{*}*,* h_{FF} *and* h_{sa} *are pairwise incomparable*
- h^{*} *and* h_{add} *are incomparable*

Moreover, h^{+}*,* h_{max}*,* h_{add}*,* h_{sa} *and* h_{FF} *assign* ∞ *to the same set of states.*

Note: For inadmissible heuristics, dominance is in general neither desirable nor undesirable. For relaxation heuristics, the objective is usually to get as close to h^{+} as possible.

Relaxation heuristics in practice: HSP

Example (HSP)

HSP (Bonet & Geffner) was one of the four top performers at the 1st International Planning Competition (IPC-1998).

Key ideas:

- hill climbing search using h_{add}
- on plateaus, keep going for a number of iterations, then restart
- use a closed list during exploration of plateaus

Literature: Bonet, Loerincs & Geffner (1997), Bonet & Geffner (2001)

Example (FF)

FF (Hoffmann & Nebel) won the 2nd International Planning Competition (IPC-2000).

Key ideas:

- enforced hill-climbing search using h_{FF}
- helpful action pruning: in each search node, only consider successors from operators that add one of the atoms marked in proposition layer 1
- goal ordering: in certain cases, FF recognizes and exploits that certain subgoals should be solved one after the other

If main search fails, FF performs greedy best-first search using h_{FF} without helpful action pruning or goal ordering.

Relaxation heuristics in practice: Fast Downward

Example (Fast Downward)

Fast Downward (Helmert & Richter) won the satisficing track of the 4th International Planning Competition (IPC-2004).

Key ideas:

- greedy best-first search using h_{FF} and causal graph heuristic (not relaxation-based)
- search enhancements:
  - multi-heuristic best-first search
  - deferred evaluation of heuristic estimates
  - preferred operators (similar to FF's helpful actions)

Literature: Helmert (2006)

Relaxation heuristics in practice: SGPlan

UNI FREIBURG

Example (SGPlan)

SGPlan (Wah, Hsu, Chen & Huang) won the satisficing track of the 5th International Planning Competition (IPC-2006).

Key ideas:

- FF
- problem decomposition techniques
- domain-specific techniques

Literature: Chen, Wah & Hsu (2006)

Relaxation heuristics in practice: LAMA

Example (LAMA)

LAMA (Richter & Westphal) won the satisficing track of the 6th International Planning Competition (IPC-2008).

Key ideas:

- Fast Downward
- landmark pseudo-heuristic instead of causal graph heuristic ("somewhat" relaxation-based)
- anytime variant of Weighted A* instead of greedy best-first search

Literature: Richter, Helmert & Westphal (2008), Richter & Westphal (2010)

Summary

- Relaxed planning graphs are AND/OR dags. They encode which propositions can be made true in Π^+ and how.
  - Closely related to forward sets and forward plan steps, based on the notion of parallel relaxed plans.
  - They can be constructed and evaluated efficiently, in time $O((m+1)\|\Pi^+\|)$ for planning task Π and depth m.
- By annotating RPG nodes with appropriate information, we can compute many useful heuristics.
- Examples: max heuristic $h_{\max}$, additive heuristic h_{add}, set-additive heuristic h_{sa} and FF heuristic h_{FF}
  - Of these, only $h_{\max}$ admissible (but not very accurate).
  - The others are much more informative. The set-additive heuristic is the most sophisticated one.
  - The FF heuristic is often similarly informative. It offers a good trade-off between accuracy and computation time.