State-Space Planning and Plan-Space Planning

Chapter 10

Outline for the Week

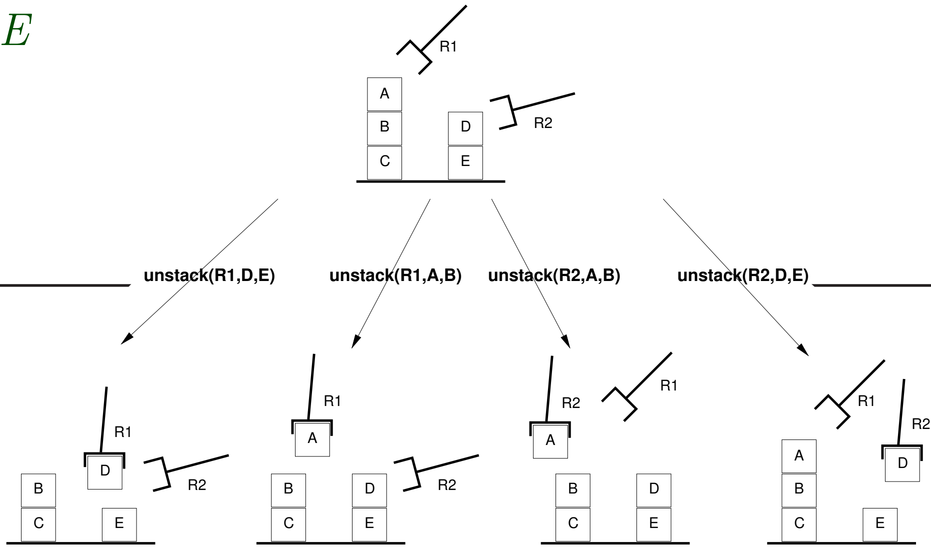
- ♢ State-space planning
- ♢ Progression planning (forward search)
- ♢ Heuristics
- ♢ Regression planning
- ♢ Plan-space planning

State-space planning

- Planning procedures are often search procedures
- They differ by the search space they consider
- State-space planning explores the most obvious search space:
  - Nodes labelled by states of the world
  - Actions define successor states
  - Plans are paths from the initial node to a goal node
- Search space can be explored in many ways:
  - forward, backward
  - using a variety of strategies (breadth-first, depth-first, A*, . . .)
  - using a variety of heuristics
  - The STRIPS representation enables an efficient exploration and domain-independent heuristics

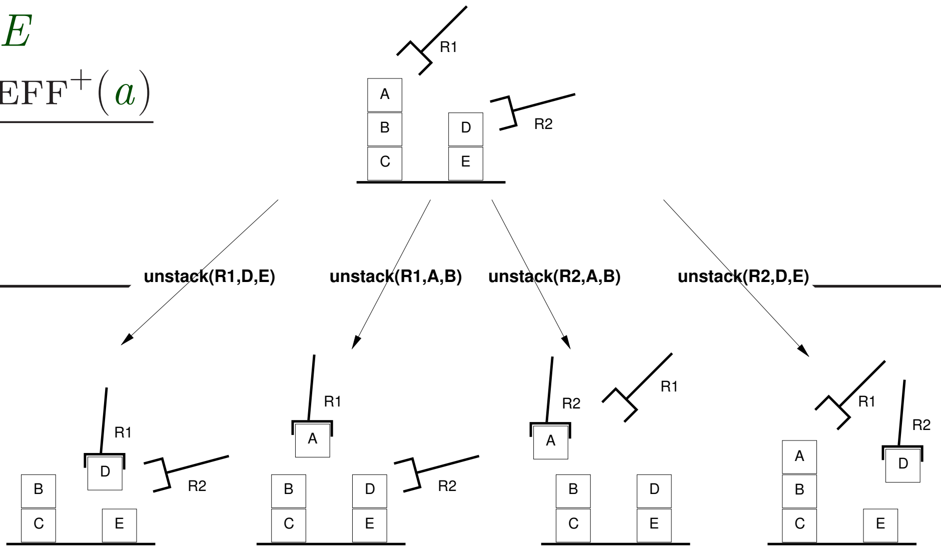
Progression planning (forward search)

```
function FORWARD-SEARCH($O, s_0, g$) returns an action sequence, or failure
    $s \leftarrow s_0$
    $\pi \leftarrow \langle\rangle$
    loop do
        if $s$ satisfies $g$ then return $\pi$
        $E \leftarrow \{a \mid$ $a$ is a ground instance of an operator in $O$
                        such that $a$ is applicable in $s$ $\}$
        if $E = \{\}$ then return failure
        choose an action $a \in E$
        $s \leftarrow \gamma(s, a)$
        $\pi \leftarrow \pi.a$
    end
```



Progression planning (forward search)

```
function FORWARD-SEARCH($O, s_0, g$) returns an action sequence, or failure
    $s \leftarrow s_0$
    $\pi \leftarrow \langle\rangle$
    loop do
        if $\underline{g \subseteq s}$ then return $\pi$
        $E \leftarrow \{a \mid a$ is a ground instance of an operator in $O$
                          such that $\underline{\text{PRE}(a) \subseteq s}\}$
        if $E = \{\}$ then return failure
        choose an action $a \in E$
        $\underline{s \leftarrow (s \setminus \text{EFF}^{-}(a)) \cup \text{EFF}^{+}(a)}$
        $\pi \leftarrow \pi . a$
    end
```



Properties of Forward-Search

Forward-Search can be used in conjunction with any search strategy to implement **choose**, breadth-first search, depth-first search, iterative-deepening, greedy search, A*, IDA*, . . .

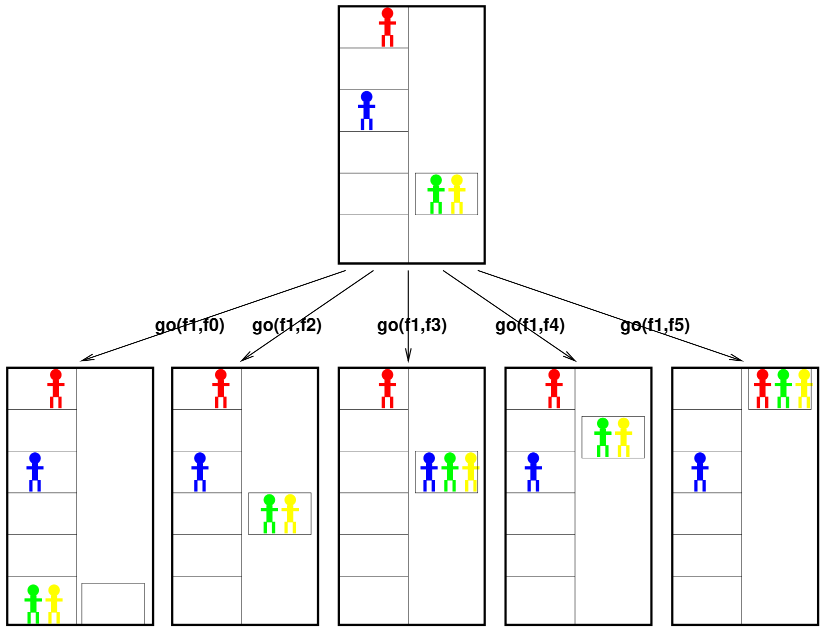
Forward-Search is sound: any plan returned is guaranteed to be a solution to the problem.

Forward-Search is complete: provided the underlying search strategy is complete, it will always return a solution to the problem if there is one.

For instance, when used with breadth-first search it will be complete, when used with depth-first search it will be complete if the state space is finite – in general, we need to detect and forbid loops.

Branching factor in Forward-Search

Forward-Search can have a large branching factor



It wastes a lot of time trying irrelevant actions

How do we cope with this?:

domain-specific: search control rules, heuristics

domain-independent: heuristics extracted from the STRIPS problem description

backward search: from the goal to the initial state

Domain-independent heuristics

Example: count number of unachieved goal propositions; fast, inadmissible and not very informative.

From the search lectures:

- An admissible heuristic is optimistic: it gives a lower bound on the true cost of a solution to the problem
- An admissible heuristic can be obtained by relaxing a problem P into a simpler problem P': the cost of any optimal solution to P' is a lower bound on the cost of the optimal solution to P

We relax STRIPS problem descriptions to obtain generic planning heuristics:

- delete relaxation heuristics
- abstraction heuristics
- landmark heuristics

Delete relaxation

Let P be a planning problem and let P^+ be the relaxed problem obtained by ignoring the negative effects (delete list) of every action

P^+ is called the delete-relaxation of P

P^+ is like P except that: $\text{EFF}^-(a) = \{\,\}$ for all a

A solution for P^+ is called a relaxed plan.

Delete relaxation

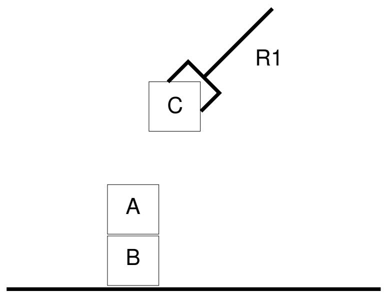
Let P be a planning problem and let P^+ be the relaxed problem obtained by ignoring the negative effects (delete list) of every action

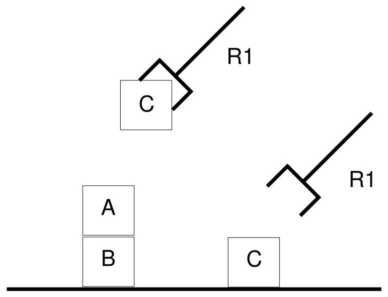
P^+ is called the delete-relaxation of P

P^+ is like P except that: $\text{EFF}^-(a) = \{\,\}$ for all a

A solution for P^+ is called a relaxed plan.

- $s = \{\text{on}(\text{A}, \text{B}), \text{clear}(\text{A}), \text{ontable}(\text{B}), \text{holding}(\text{R1}, \text{C})\}$
- $a = \text{putdown}(\text{R1}, \text{C})$
  precondition $\{\text{holding}(\text{R1}, \text{C})\}$
  effect $\{\text{ontable}(\text{C}), \text{clear}(\text{C}), \text{handempty}(\text{R1}), \neg\text{holding}(\text{R1}, \text{C})\}$
- $\gamma(s, a) = \{\text{on}(\text{A}, \text{B}), \text{clear}(\text{A}), \text{ontable}(\text{B}), \text{holding}(\text{R1}, \text{C}), \text{ontable}(\text{C}), \text{clear}(\text{C}), \text{handempty}(\text{R1})\}$





Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

Real World (before)

Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

Real World (after)

Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

Relaxed World (before)

Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

Relaxed World (after)

Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

Real World (before)

Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

Real World (after)

Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

Relaxed World (before)

Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

Relaxed World (after)

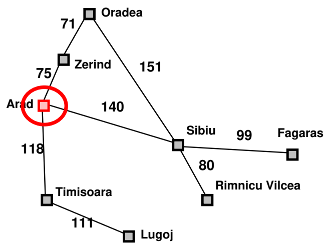
Delete relaxation - intuition

Delete-relaxed planning: once a fact becomes true, it remains true forever

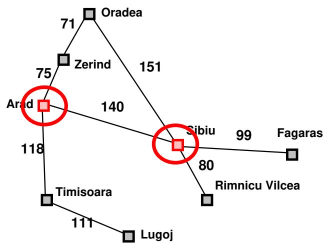
$\xrightarrow{\text{fill}}$

$\xrightarrow{\text{drink}}$

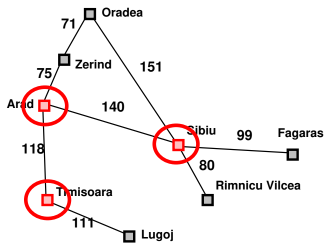
$\xrightarrow{\text{drink} \ldots} \ldots$



$\xrightarrow{\text{go(A,S)}}$



$\xrightarrow{\text{go(A,T)}}$



$\xrightarrow{\text{go(S,F)} \ldots} \ldots$

Once an action is applicable, it remains applicable

An action does not need to be applied more than once in a relaxed plan

Delete relaxation heuristics

The cost h^+ of an optimal solution to P^+ is a lower bound on the cost of an optimal solution of for P, hence an admissible heuristic
$\Rightarrow h^+(s)$: optimal solution of P^+ using state s as initial state.

But, finding an optimal solution for P^+ is NP-hard
(PLANMIN is NP-complete for problems with only positive effects)

$\rightarrow$ Need to further relax the problem to get efficient admissible heuristics
gives the h^{max} heuristic [Bonet & Geffner, 1999]

Finding an arbitrary solution for P^+ (PLANSAT) is polynomial
(PLANSAT is polynomial with only positive effects)

$\rightarrow$ Relaxed plans are used to derive inadmissible heuristics
gives the h^{FF} heuristic [Hoffmann & Nebel, AIJ 2001]

$h^{\mathbf{max}}$ and $h^{\mathbf{sum}}$ heuristics

Relax the problem by ignoring the negative effects $\textsc{eff}^-$

Further relax the problem by ignoring interactions between subgoals
$\Rightarrow$ if $g = \{p_1, p_2, \ldots, p_n\}$, each p_i is a subgoal

Heuristics $h(s, g)$ estimate the minimum cost from s to g. When $g = \{p\}$:

- $h(s, \{p\}) = 0$ if $p \in s$
- $h(s, \{p\}) = \infty$ if $p \notin s$ and $\forall a \in A, p \notin \textsc{eff}^+(a)$
- $h(s, \{p\}) = \min\limits_{\substack{a \in A \\ p \in \textsc{eff}^+(a)}} [h(s, \textsc{pre}(a)) + c(a)]$ otherwise

- admissible h^{max} heuristic: cost to reach a set is the max of costs.
  looks at the critical path: $h(s, g) = \max_{p \in g} h(s, \{p\})$
- non-admissible h^{sum} heuristic: cost to reach a set is the sum of costs.
  assumes subgoal independence: $h(s, g) = \sum_{p \in g} h(s, \{p\})$
- admissible: cost to reach a set is the max of costs to reach each *pair*.
  generalisation h^m heuristic: max of costs to reach each subset of size m

Computing $h^{\max}$

Let n be a node labelled by state s in an A* search. We need to compute $h(s, g)$ to evaluate n and put it in frontier.

It suffices to compute $h(s, \{p\})$ for each proposition p.

```
for each proposition $p$
    if $p \in s$ then $H[p] \leftarrow 0$ else $H[p] \leftarrow \infty$
repeat:
    for each action $a$
        $H[a] \leftarrow \max_{p \in \text{PRE}(a)} H[p]$
        for each proposition $p \in \text{EFF}^+(a)$
            $H[p] \leftarrow \min(H[p], H[a] + c(a))$
until a fixed point is reached
return $h(s, g) = \max_{p \in g} H[p]$
```

Polynomial time algorithm. For h^{sum}, replace $\max$ with $\sum$.

$h^{\textbf{FF}}$ heuristic

Delete-relaxation heuristics so far:

- h^+ too hard to compute
- h^{max} not very informative
- h^{sum} can greatly over-estimate h^*

New heuristic h^{FF}:

- arbitrary solution to P^+
- inadmissible
- compromise between h^{max} and h^{sum}
- makes sense when actions have unit costs
- relaxed reachability $+$ relaxed plan extraction in plan graph without mutex

$h^{\mathbf{FF}}$ heuristic - relaxed reachability

Level 0

ontable(A)

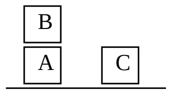
on(B,A)

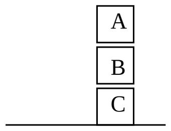
clear(B)

handempty(R)

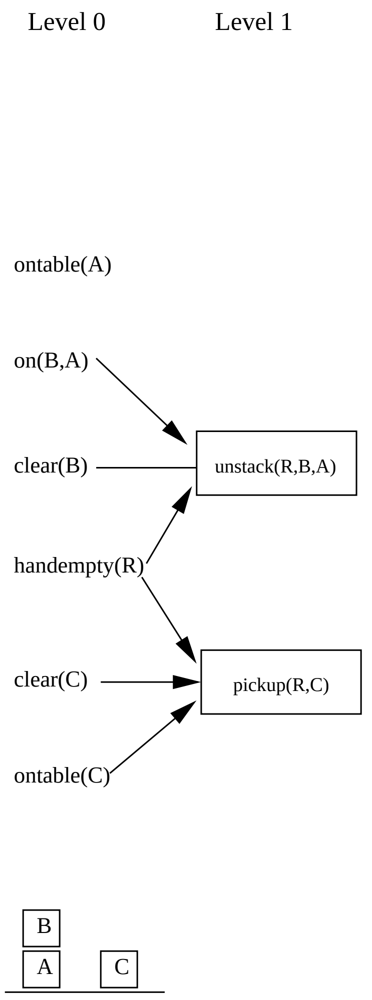
clear(C)

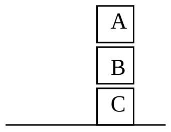
ontable(C)



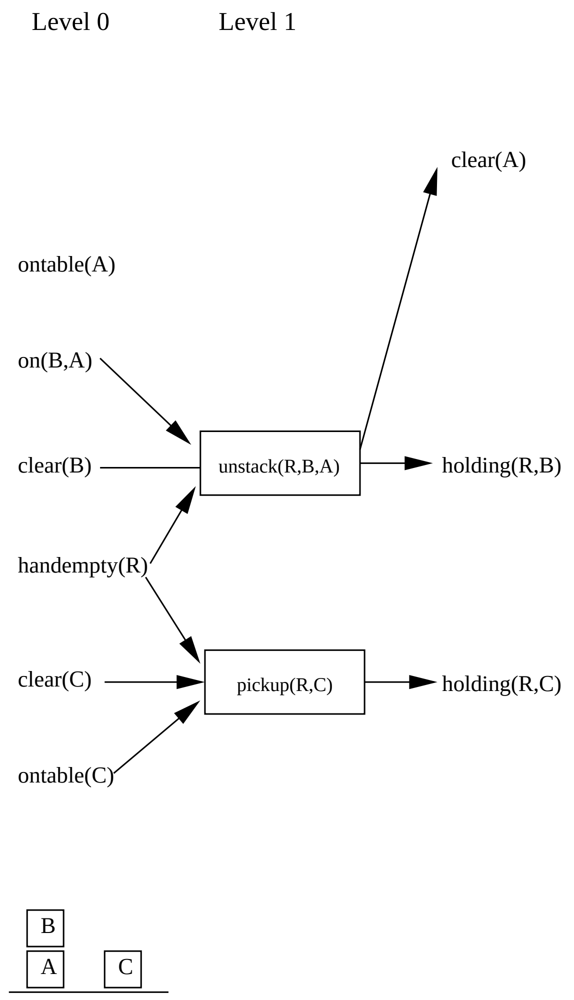


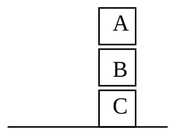
$h^{\mathbf{FF}}$ heuristic - relaxed reachability



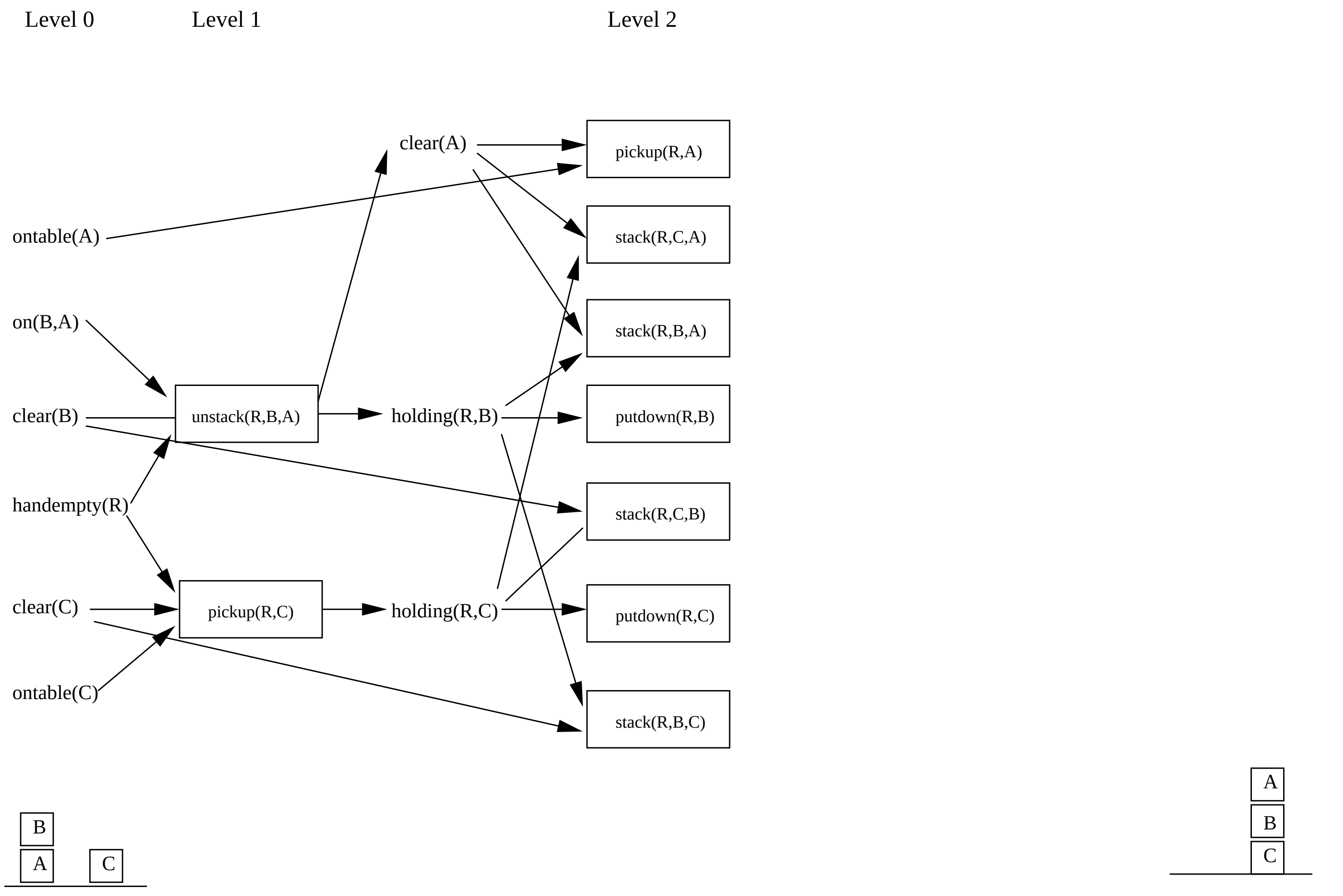


$h^{\mathbf{FF}}$ heuristic - relaxed reachability

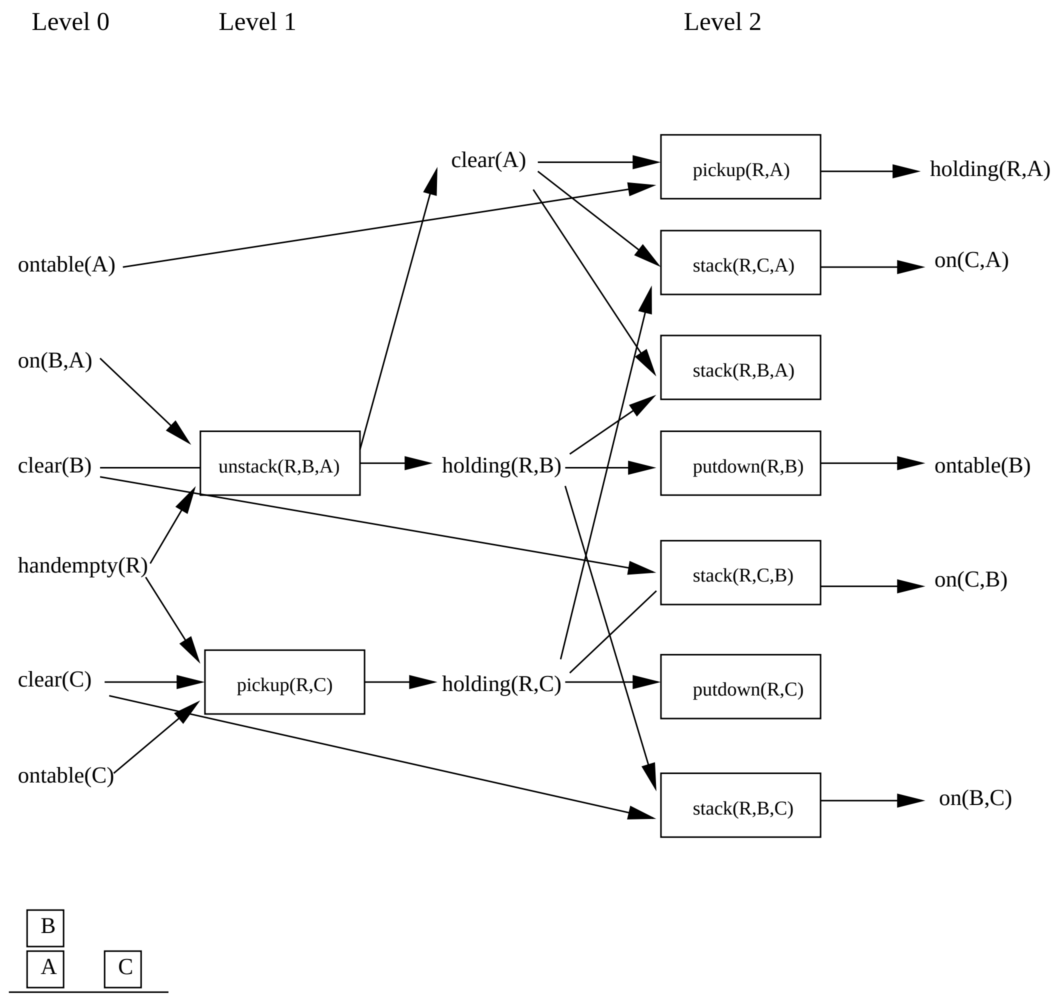


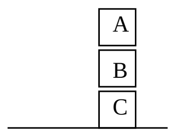


$h^{\mathbf{FF}}$ heuristic - relaxed reachability

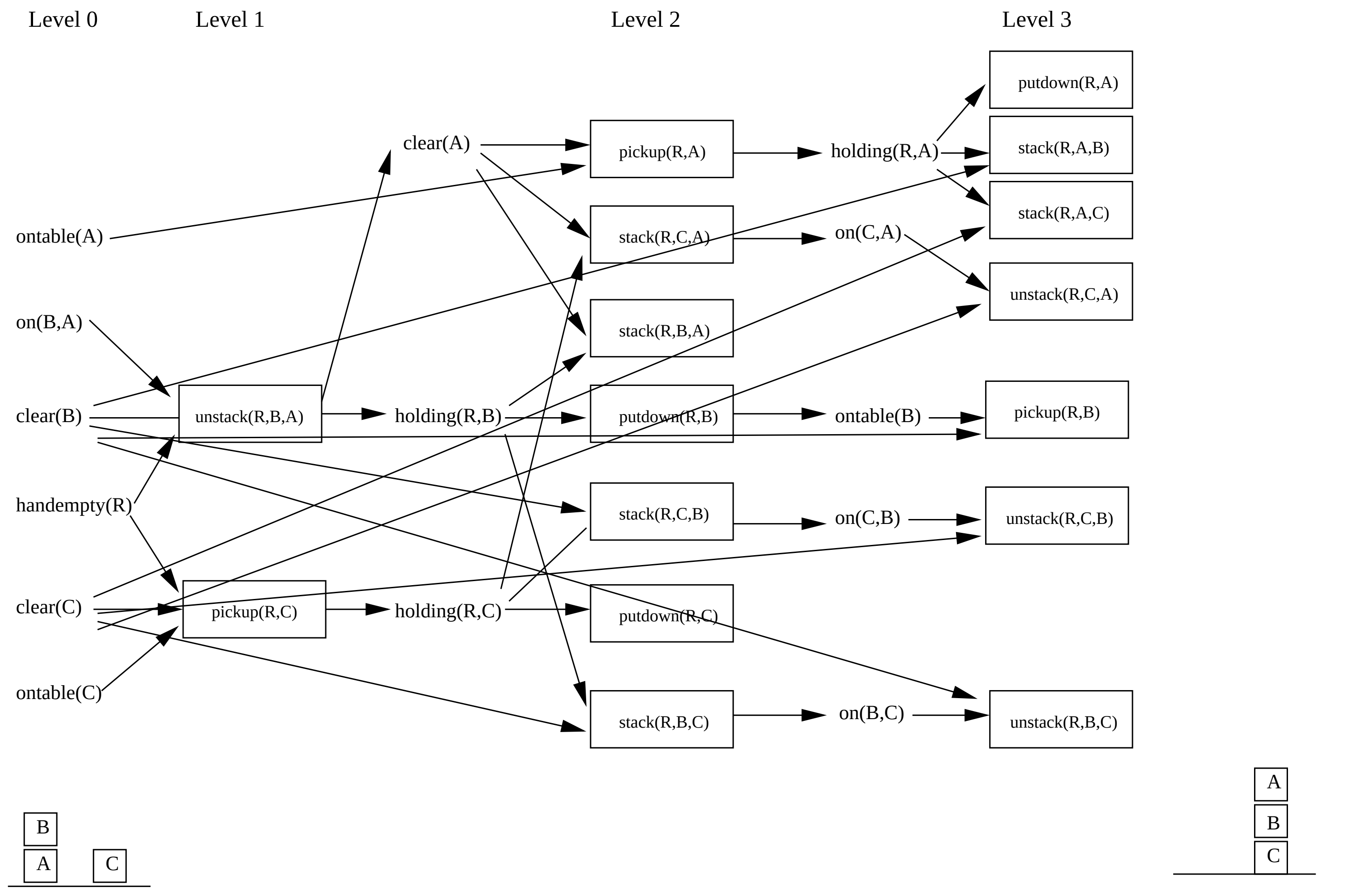


$h^{\mathbf{FF}}$ heuristic - relaxed reachability

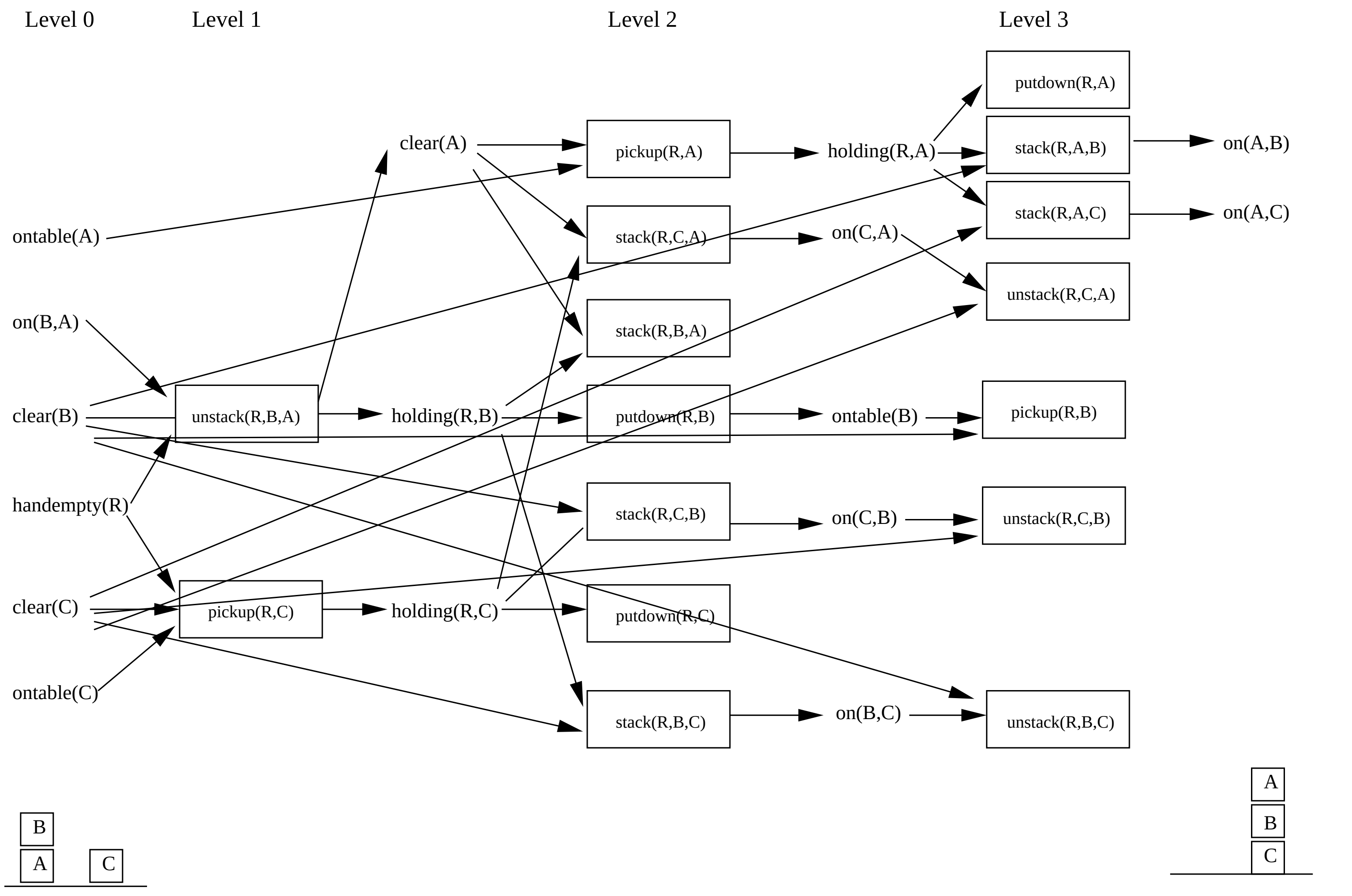




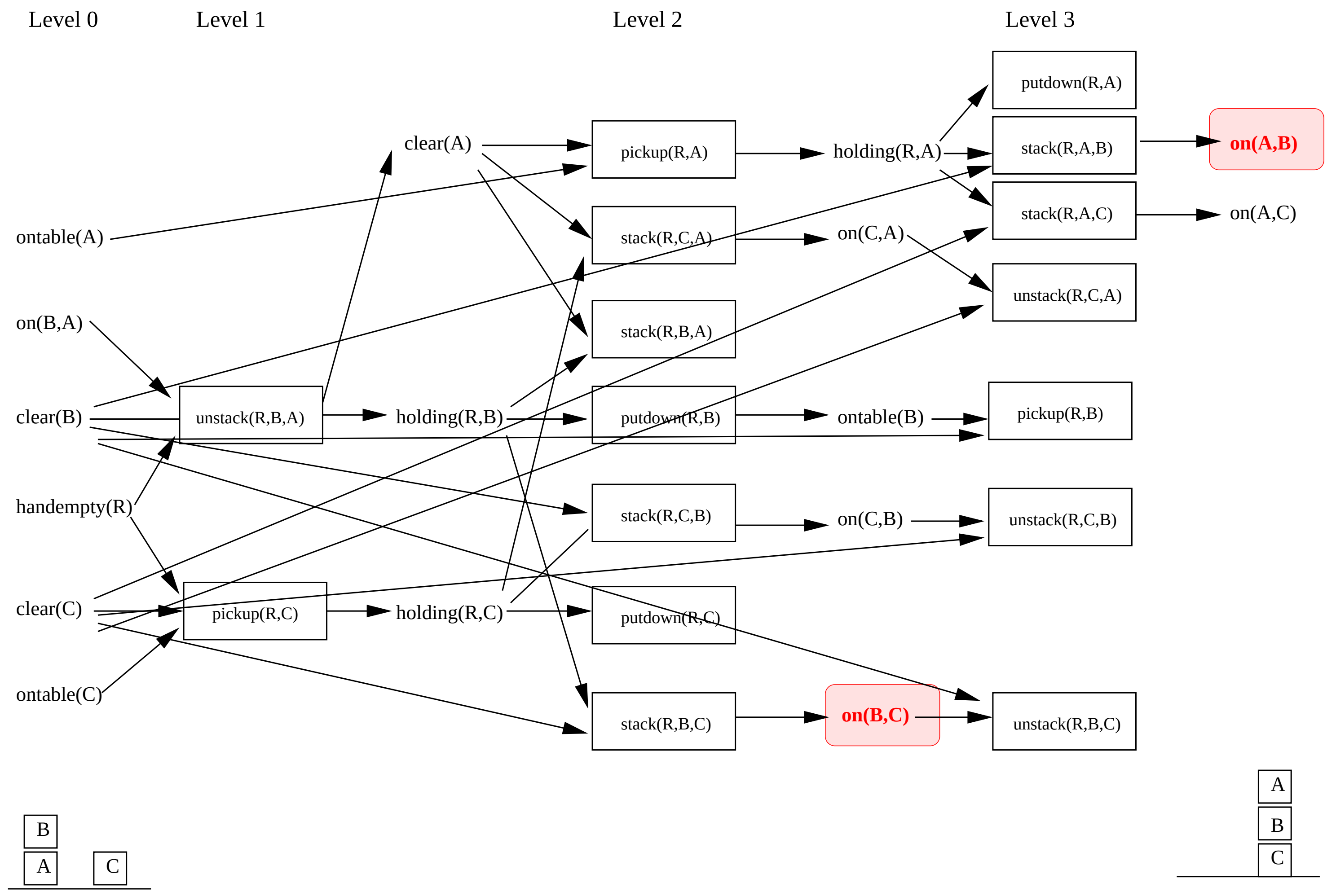
$h^{\mathbf{FF}}$ heuristic - relaxed reachability



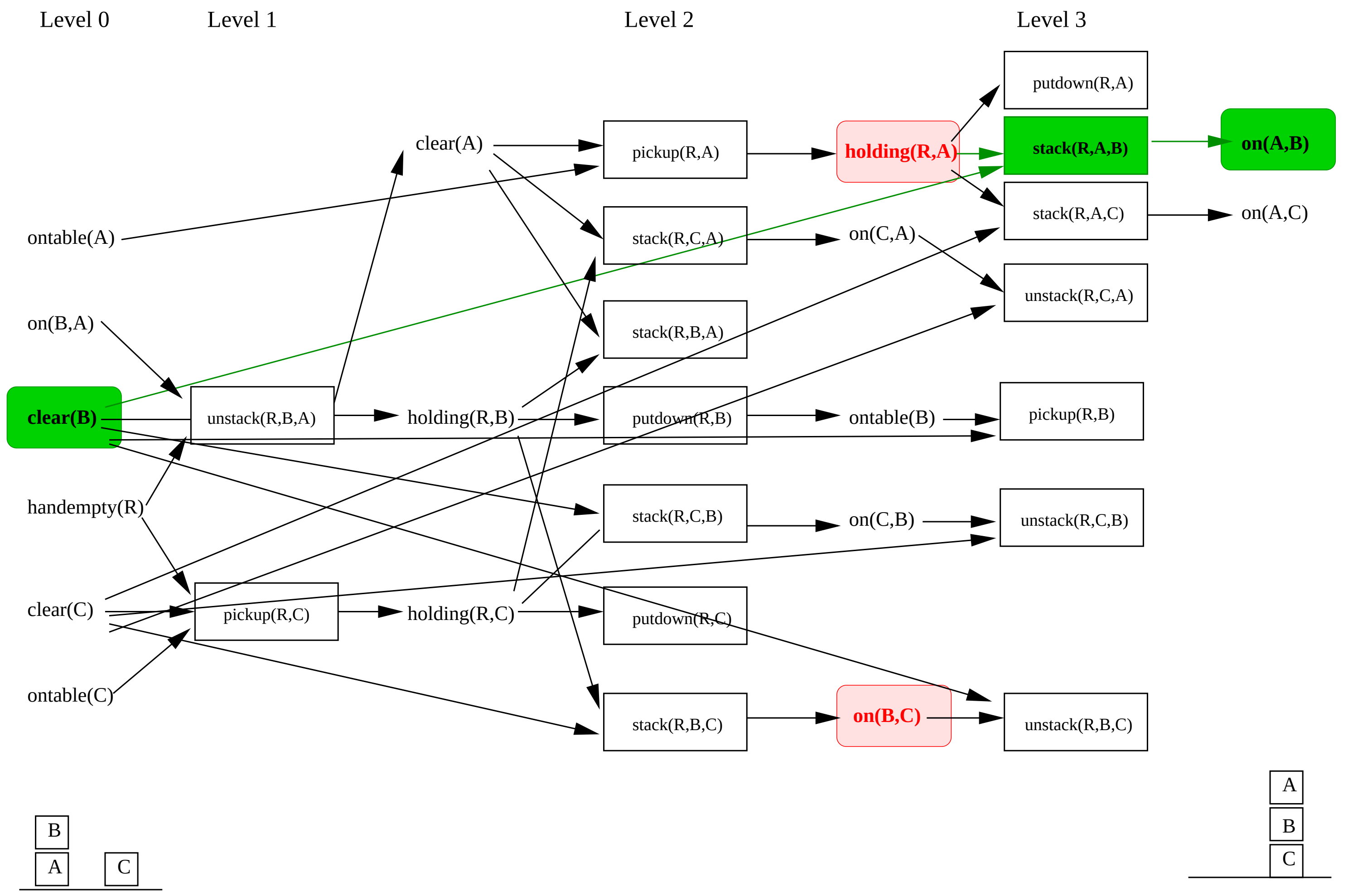
$h^{\mathbf{FF}}$ heuristic - relaxed reachability



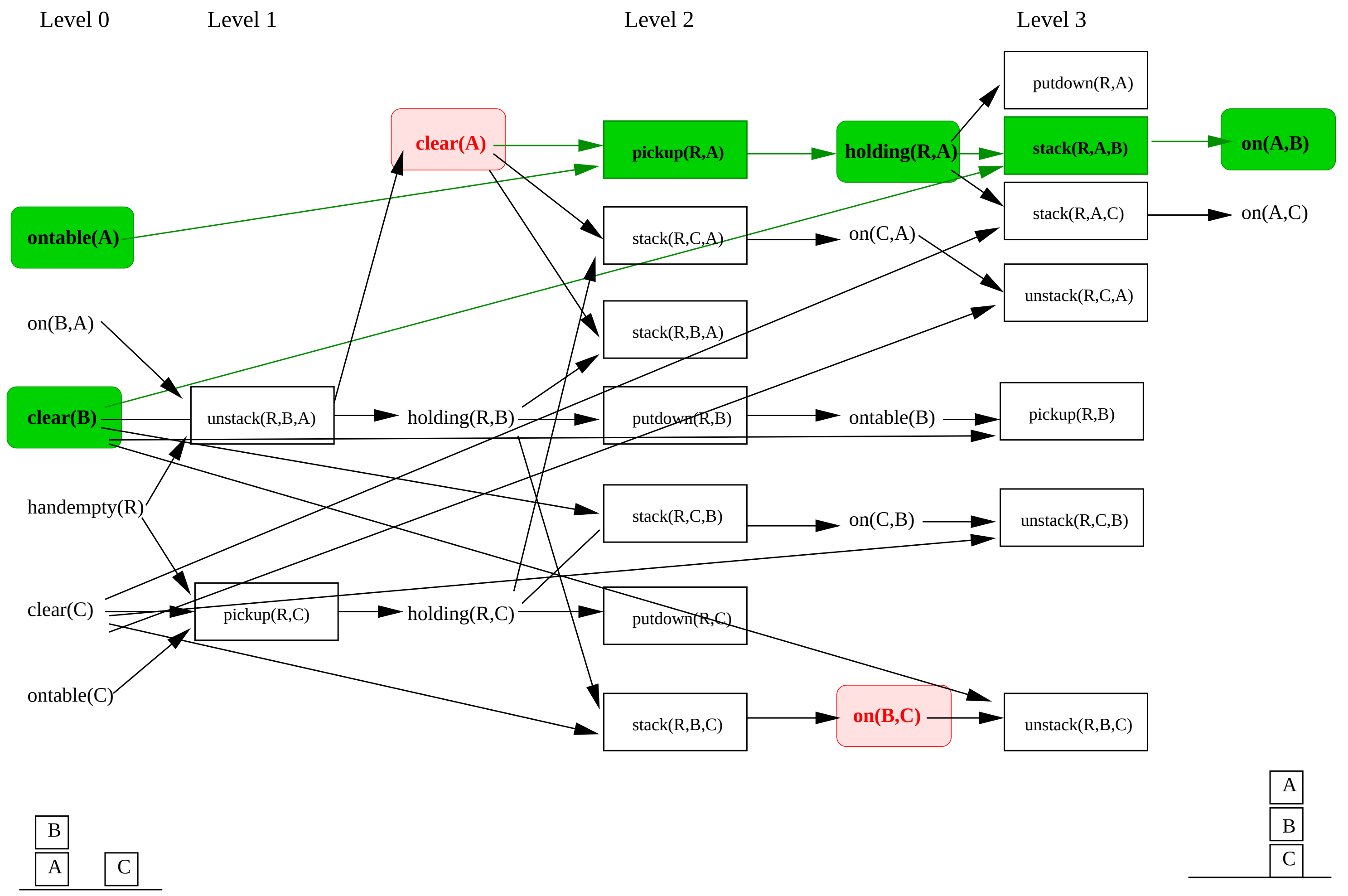
h^{FF} heuristic - plan extraction



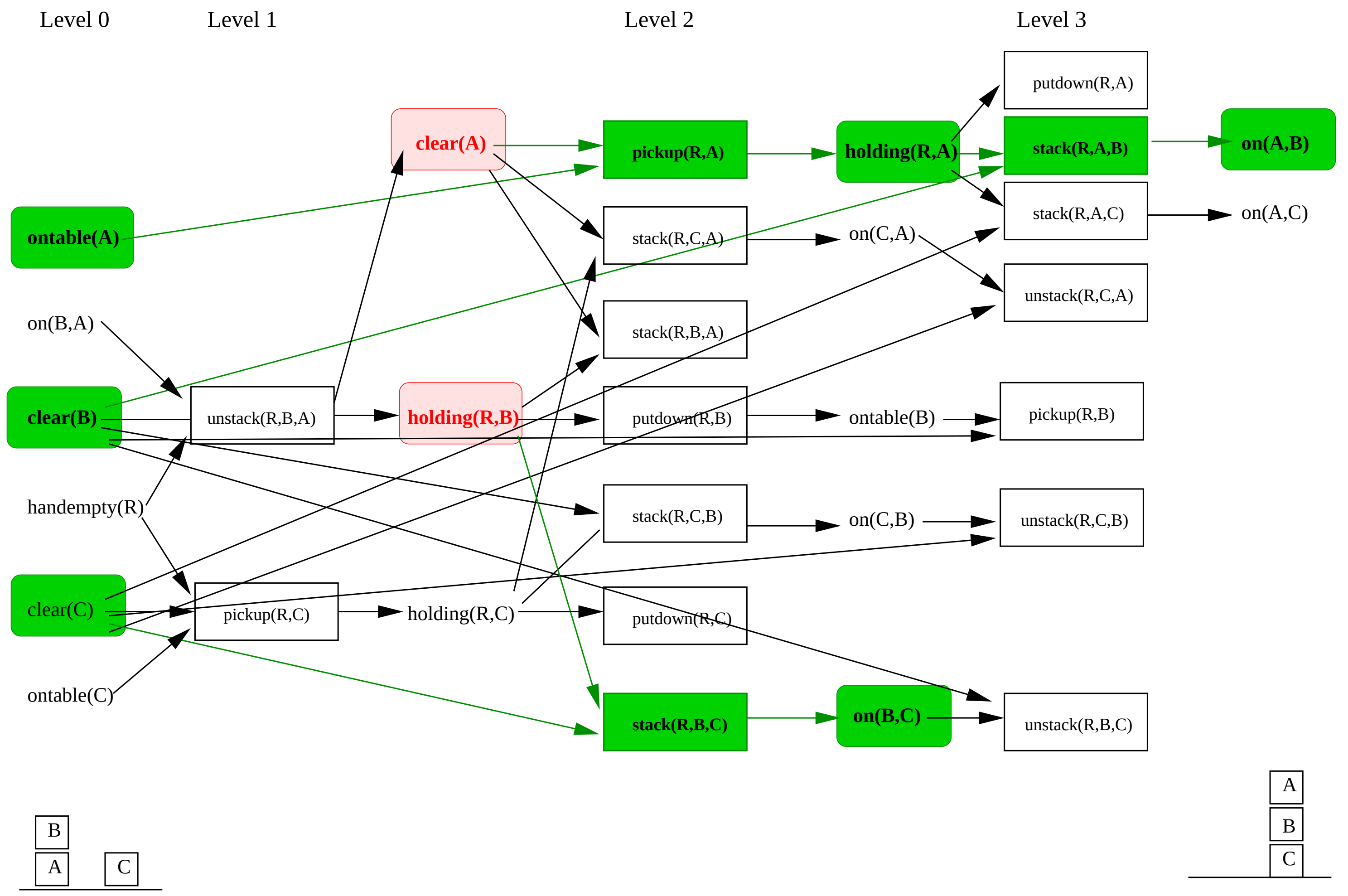
$h^{\mathbf{FF}}$ heuristic - plan extraction



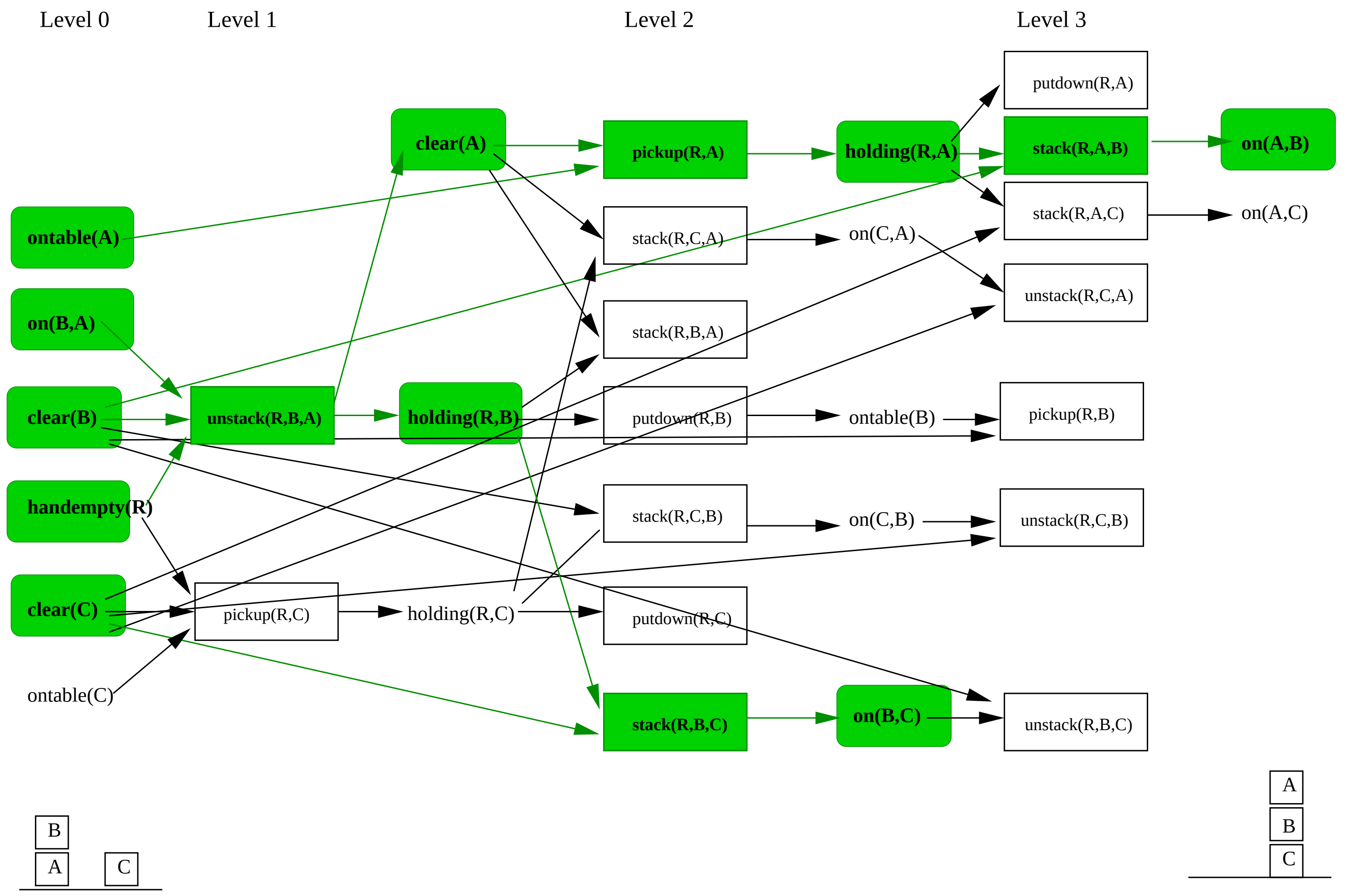
$h^{\mathbf{FF}}$ heuristic - plan extraction



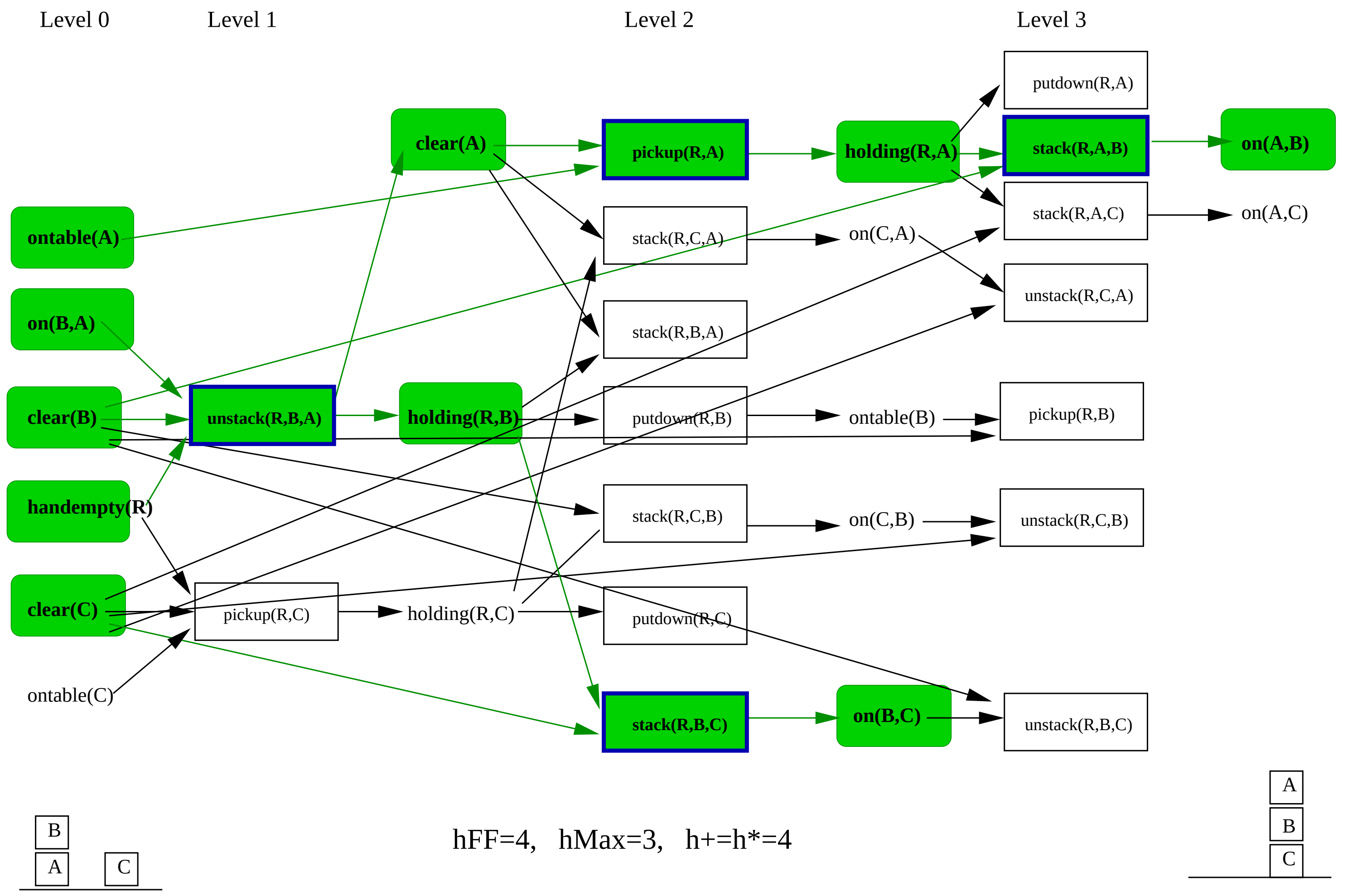
$h^{\mathbf{FF}}$ heuristic - plan extraction



h^{FF} heuristic - plan extraction



h^{FF} heuristic - plan extraction



hFF=4, hMax=3, h+=h*=4

$h^{\textbf{FF}}$ heuristic

Delete-relaxation heuristics so far:

- h^+ too hard to compute
- h^{max} not very informative
- h^{sum} can greatly over-estimate h^*

New heuristic h^{FF}:

- arbitrary solution to P^+
- inadmissible
- compromise between h^{max} and h^{sum}
- makes sense when actions have unit costs
- relaxed reachability + relaxed plan extraction in plan graph without mutex

$\Rightarrow$ h^{max}: first level in a planning graph in which the goal appears

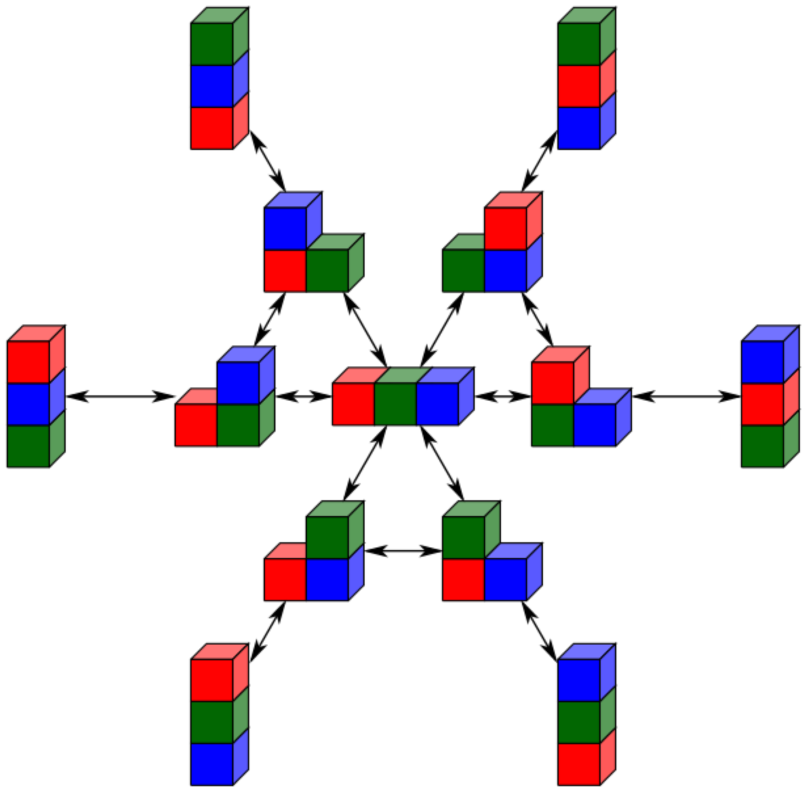
$\Rightarrow$ h^{FF}: is the number of actions in the relaxed plan

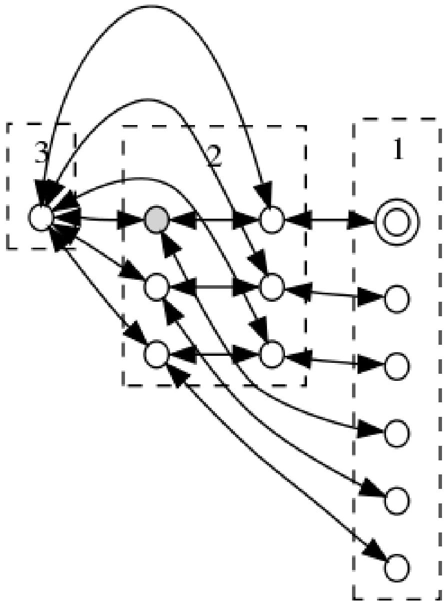
Abstraction heuristics

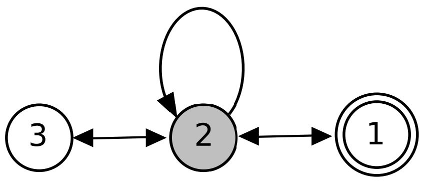
Simplify the problem by ignoring <u>parts</u> of it.

- Drop preconditions from actions
- Consider only a subset of predicates/propositions
- Count objects with a given property, ignoring the identity of objects
  e.g. count clear blocks
- Ignore so much that the abstract problem is small enough to be solved by uninformed search
- Use memory to avoid repeated searches (pattern databases)

Example: counting clear blocks







Formal definition

Problem $P' = (S', A', \gamma', s'_0, S'_G, c')$ is an abstraction of $P = (S, A, \gamma, s_0, S_G, c)$ if there exists an abstraction mapping $\phi : S \mapsto S'$, such that:

- ϕ preserves the initial state:

  $\phi(s_0) = s'_0$

- ϕ preserves goal states:

  if $s \in S_G$ then $\phi(s) \in S'_G$

- ϕ preserves transitions:

  if $\gamma(s, a) = t$ then $\exists a' \in A'\ \gamma'(\phi(s), a') = \phi(t)$ with $c'(a') \leq c(a)$

The abstraction heuristic $h^\phi(s, g)$ induced by ϕ is given by the the cost of the optimal path from $\phi(s)$ to $\phi(g)$ in P'

Theorem: h^ϕ is admissible (and consistent).

With the STRIPS representation, pattern database heuristics are defined by projecting the states on a subset of propositions (the pattern).

Landmark heuristics

Proposition l is a landmark for problem P iff all plans for P make l true.
$\rightarrow$ e.g. clear(B) is a landmark if any block below B is misplaced.

Landmark heuristic: counts the number of yet unachieved landmarks.
$\rightarrow$ generalisation of the number of unachieved goals heuristic
$\rightarrow$ used in the LAMA planner [Richter, AAAI 2008]

Inadmissible (even if action costs are 1) as it assumes landmark independence.
$\rightarrow$ Admissible versions exist.

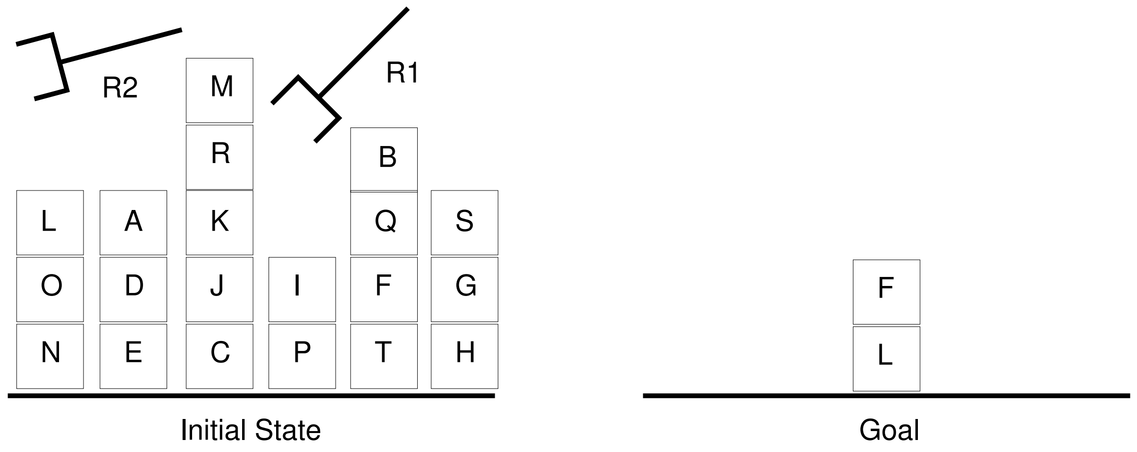
Sufficient condition for proposition l to be a landmark for problem P:
$\Rightarrow$ the delete relaxation P^+ is not solvable when l is removed from the add-list of all actions.

A complete landmark set can be computed in polynomial time for the delete relaxation, once as pre-processing. Gives a set of landmarks for P.

The current best heuristics are landmark heuristics variants

Regression planning (backward search)

For some problems, goal directed search pays



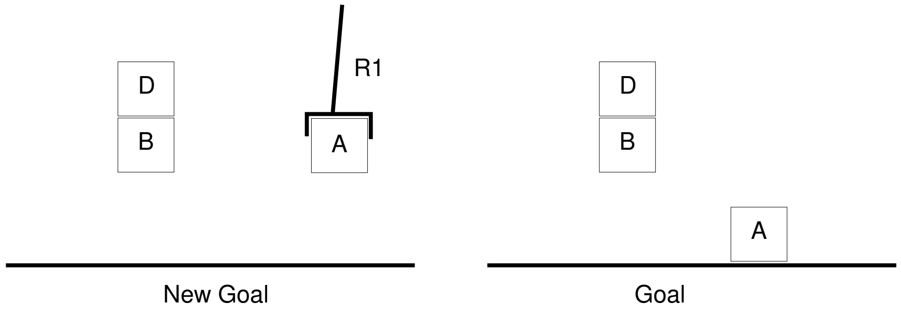
For forward search, we started at the initial state and computed state transitions, leading to a new state $s' = \gamma(s, a)$

For backward search, we start at the goal and compute inverse state transitions a.k.a regression, leading to a new goal $g' = \gamma^{-1}(g, a)$

What do we really mean by $\gamma^{-1}(g, a)$?? First we need to define relevance

Regression - relevance

- An action a is relevant for goal g if:
  - it makes at least one of the propositions in g true: $g \cap \mathrm{EFF}^{+}(a) \neq \{\,\}$
  - it does not make any of the propositions in g false: $g \cap \mathrm{EFF}^{-}(a) = \{\,\}$
- If a is relevant for g then: $\gamma^{-1}(g,a) = (g \setminus \mathrm{EFF}^{+}(a)) \cup \mathrm{PRE}(a)$



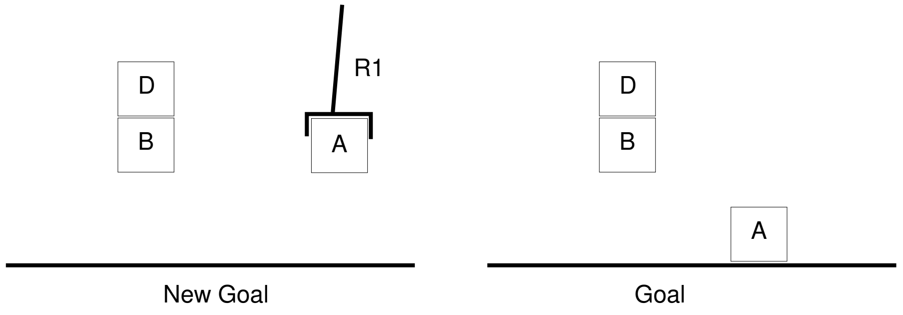
- Example:
  - $g = \{$on(D,B), clear(D), ontable(A), clear$(\mathsf{A})\}$
  - $a =$ putdown$(\mathsf{R1},\mathsf{A})$

operator	putdown(r,x)
precondition	$\{$holding$(r,x)\}$
effect	$\{$ontable(x), clear(x), handempty(r), $\neg$holding$(r,x)\}$

  - $\gamma^{-1}(g,a) = \{$on(D,B), clear(D), holding$(\mathsf{R1},\mathsf{A})\}$

Regression - relevance

- An action a is relevant for goal g if:
  - it makes at least one of the propositions in g true: $g \cap \mathrm{EFF}^{+}(a) \neq \{\,\}$
  - it does not make any of the propositions in g false: $g \cap \mathrm{EFF}^{-}(a) = \{\,\}$
- If a is relevant for g then: $\gamma^{-1}(g, a) = (g \setminus \mathrm{EFF}^{+}(a)) \cup \mathrm{PRE}(a)$



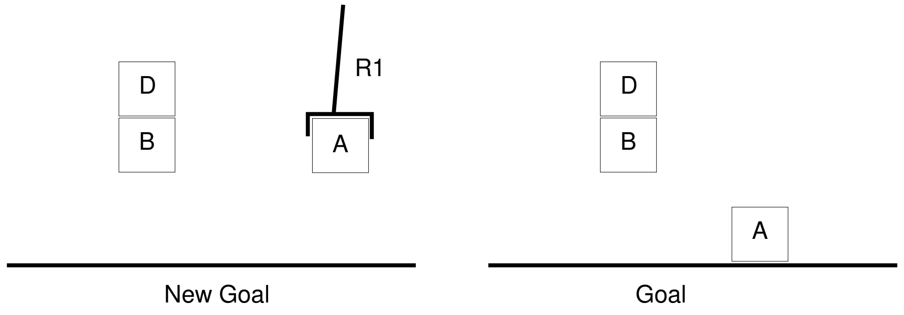
- Example:
  - $g = \{$on(D, B), clear(D), ontable(A), clear(A)$\}$
  - $a =$ putdown(R1, A)

operator	putdown(r, x)
precondition	{holding(R1, A)}
effect	{ontable(A), clear(A), handempty(R1), ¬holding(R1, A)}

  - $\gamma^{-1}(g, a) = \{$on(D, B), clear(D), holding(R1, A)$\}$

Regression - relevance

- An action a is relevant for goal g if:
  - it makes at least one of the propositions in g true: $g \cap \mathrm{EFF}^{+}(a) \neq \{\,\}$
  - it does not make any of the propositions in g false: $g \cap \mathrm{EFF}^{-}(a) = \{\,\}$
- If a is relevant for g then: $\gamma^{-1}(g, a) = (g \setminus \mathrm{EFF}^{+}(a)) \cup \mathrm{PRE}(a)$



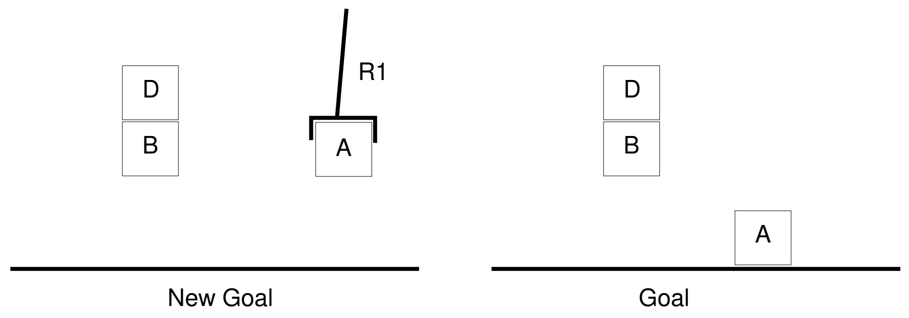
- Example:
  - $g = \{\text{on(D, B), clear(D), }\underline{\text{ontable(A), clear(A)}}\}$
  - $a = \text{putdown(R1, A)}$

operator	putdown(r, x)
precondition	{holding(R1, A)}
effect	{ontable(A), clear(A), handempty(R1), ¬holding(R1, A)}

  - $\gamma^{-1}(g, a) = \{\text{on(D, B), clear(D), holding(R1, A)}\}$

Regression - relevance

- An action a is relevant for goal g if:
  - it makes at least one of the propositions in g true: $g \cap \text{EFF}^{+}(a) \neq \{\,\}$
  - it does not make any of the propositions in g false: $g \cap \text{EFF}^{-}(a) = \{\,\}$
- If a is relevant for g then: $\gamma^{-1}(g,a) = (g \setminus \text{EFF}^{+}(a)) \cup \text{PRE}(a)$



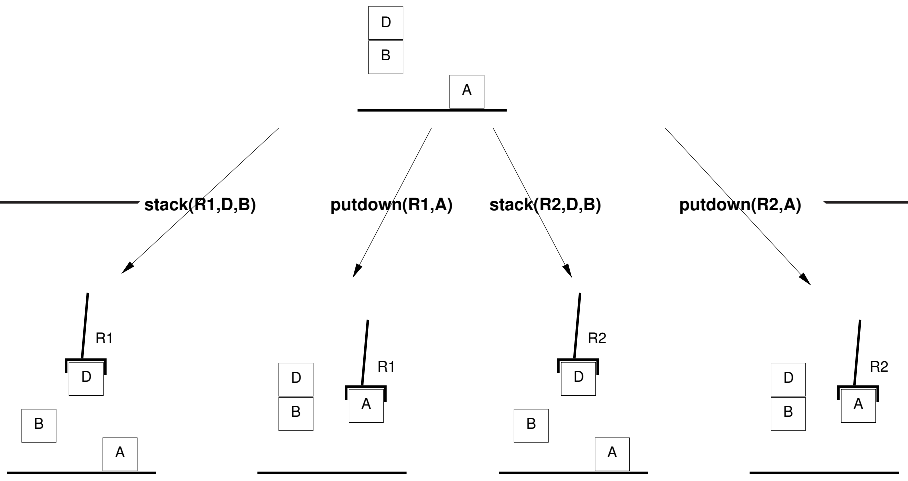
- Example:
  - $g = \{$on(D, B), clear(D), ~~ontable(A), clear(A)~~$\}$
  - $a =$ putdown(R1, A)

operator	putdown(r, x)
precondition	{holding(R1, A)}
effect	{ontable(A), clear(A), handempty(R1), ¬holding(R1, A)}

  - $\gamma^{-1}(g,a) = \{$on(D, B), clear(D), $\underline{\text{holding(R1, A)}}\}$

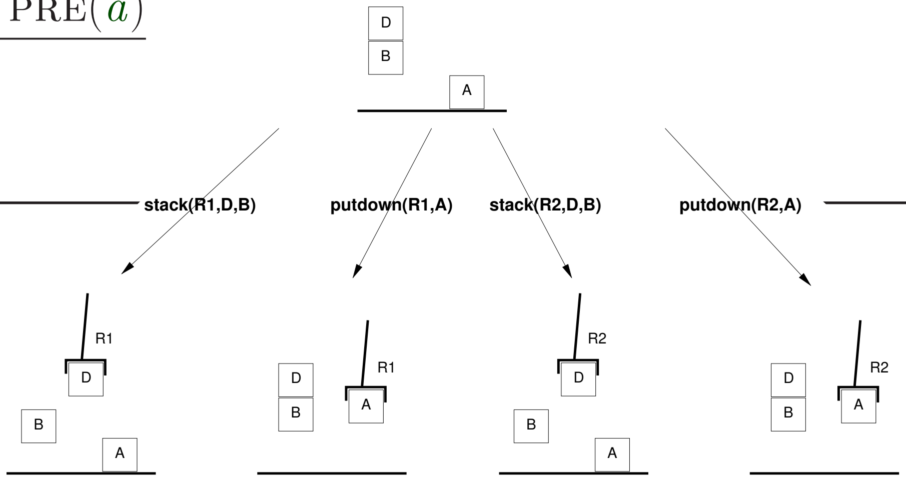
Regression planning (backward search)

```
function BACKWARD-SEARCH($O, s_0, g$) returns an action sequence, or failure
    $\pi \leftarrow \langle\rangle$
    loop do
        if $s_0$ satisfies $g$ then return $\pi$
        $E \leftarrow \{a \mid a$ is a ground instance of an operator in $O$
                          such that $a$ is relevant for $g\}$
        if $E = \{\}$ then return failure
        choose an action $a \in E$
        $g \leftarrow \gamma^{-1}(g, a)$
        $\pi \leftarrow a.\pi$
    end
```



Regression planning (backward search)

```
function BACKWARD-SEARCH($O, s_0, g$) returns an action sequence, or failure
    $\pi \leftarrow \langle\rangle$
    loop do
        if $\underline{g \subseteq s_0}$ then return $\pi$
        $E \leftarrow \{a \mid a$ is a ground instance of an operator in $O$
                    such that $\underline{g \cap \mathrm{EFF}^+(a) \neq \{\} \text{ and } g \cap \mathrm{EFF}^-(a) = \{\}}\}$
        if $E = \{\}$ then return failure
        choose an action $a \in E$
        $\underline{g \leftarrow (g \setminus \mathrm{EFF}^+(a)) \cup \mathrm{PRE}(a)}$
        $\pi \leftarrow a.\pi$
    end
```



Properties of Backward-Search

Backward-Search can be used in conjunction with any search strategy to implement **choose**, breadth-first search, depth-first search, iterative-deepening, greedy search, A*, IDA*, . . .

Backward-Search is sound: any plan returned is guaranteed to be a solution to the problem.

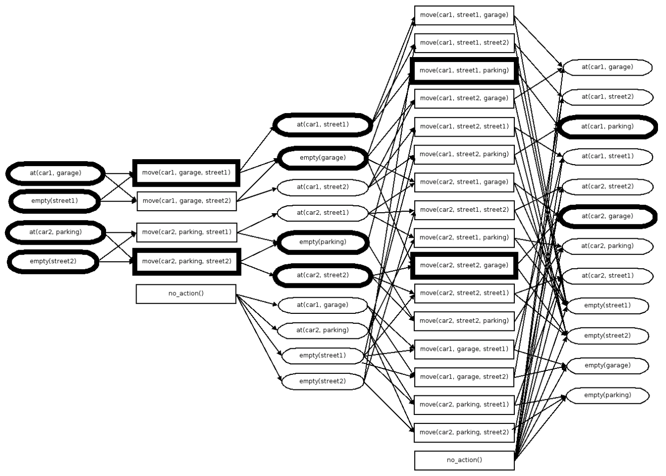
Backward-Search is complete: provided the underlying search strategy is complete, it will always return a solution to the problem if there is one.

For instance, when used with breadth-first search it will be complete, when used with depth-first search it will be complete if the state space is finite – in general, need to detect and forbid loops, by checking that no previous goal is a subset of the current one.

Heuristics: many of the state-space heuristics are symmetric.
Forward: $h(s, g)$, backwards: $h(s_0, c)$ where c is the current goal.

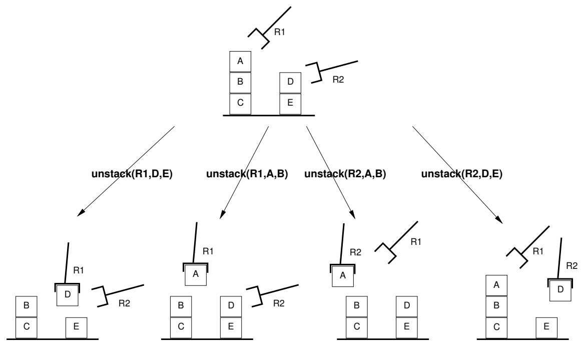
Different Plans, Search Spaces, and Approaches

Parallel plans:
graph-based and sat-based approaches

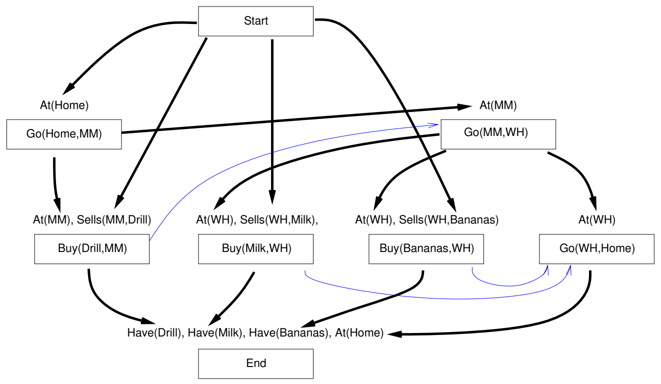


at(car1, garage)
empty(street1)
at(car2, parking)
empty(street2)
move(car1, garage, street1)
move(car1, garage, street2)
move(car2, parking, street1)
move(car2, parking, street2)
no_action()
at(car1, street1)
empty(garage)
at(car1, street2)
at(car2, street1)
empty(parking)
at(car2, street2)
at(car1, garage)
at(car2, parking)
empty(street1)
empty(street2)
move(car1, street1, garage)
move(car1, street1, street2)
move(car1, street1, parking)
move(car1, street2, garage)
move(car1, street2, street1)
move(car1, street2, parking)
move(car2, street1, garage)
move(car2, street1, street2)
move(car2, street1, parking)
move(car2, street2, garage)
move(car2, street2, street1)
move(car2, street2, parking)
move(car1, garage, street1)
move(car1, garage, street2)
move(car2, parking, street1)
move(car2, parking, street2)
no_action()
at(car1, garage)
at(car1, street2)
at(car1, parking)
at(car1, street1)
at(car2, street2)
at(car2, garage)
at(car2, parking)
at(car2, street1)
empty(street1)
empty(street2)
empty(garage)
empty(parking)

Totally-ordered plans:
state space search approaches



Partially-ordered plans:
plan space approaches



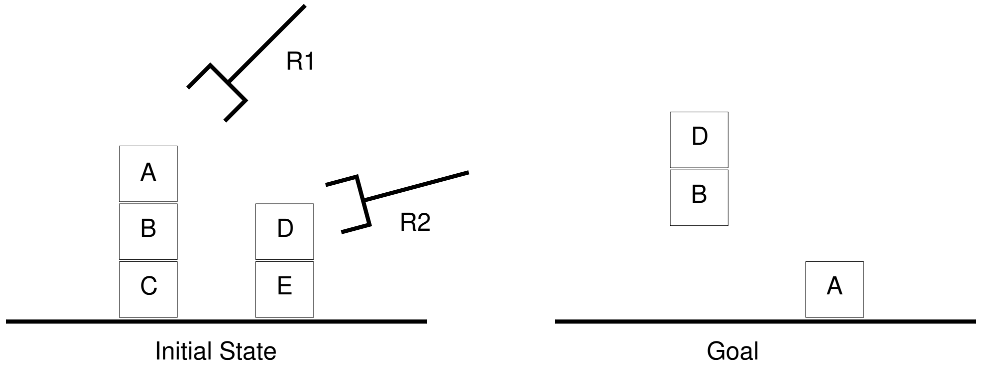
Outline

- ♢ Motivation
- ♢ Partial plans
- ♢ Flaws
- ♢ Plan-space planning algorithm
- ♢ Example

Motivation

State-space search produces inflexible plans.

Part of the ordering in an action sequence is not related to causality:

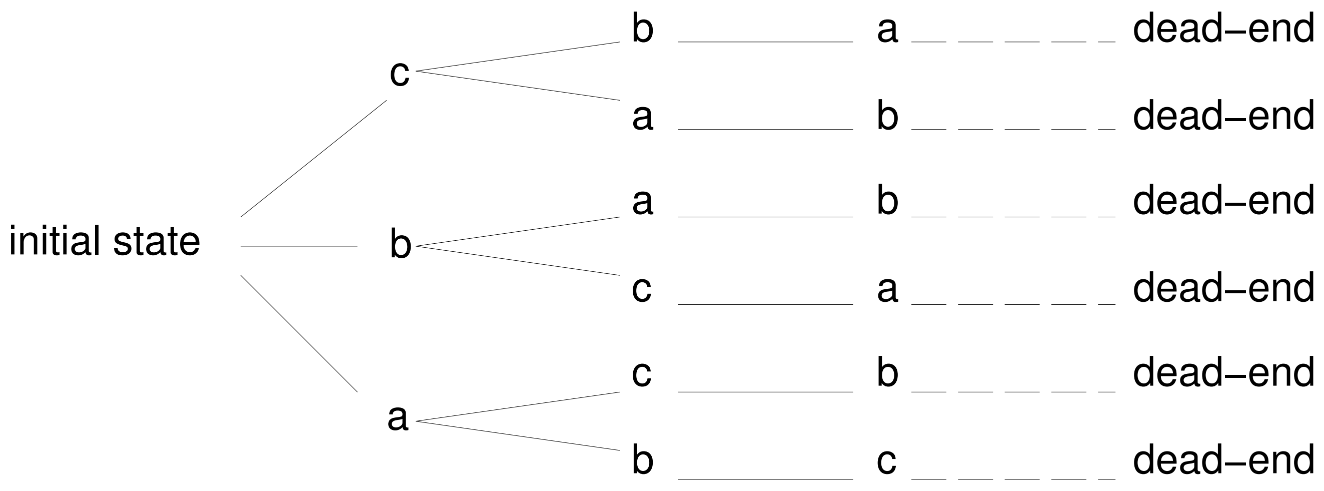


sequence:
$\langle \text{unstack}(\text{R1}, \text{A}, \text{B}), \text{unstack}(\text{R2}, \text{D}, \text{E}), \text{putdown}(\text{R1}, \text{A}), \text{stack}(\text{R2}, \text{D}, \text{B}) \rangle$

partially ordered plan only needs: $\text{unstack}(\text{R1}, \text{A}, \text{B}) < \text{putdown}(\text{R1}, \text{A})$, $\text{unstack}(\text{R2}, \text{D}, \text{E}) < \text{stack}(\text{R2}, \text{D}, \text{B})$, and $\text{unstack}(\text{R1}, \text{A}, \text{B}) < \text{stack}(\text{R2}, \text{D}, \text{B})$

Motivation

State-space search wastes time examining many different orderings of the same set of actions:



Not ordering actions unecessarily can speed up planning

Motivation

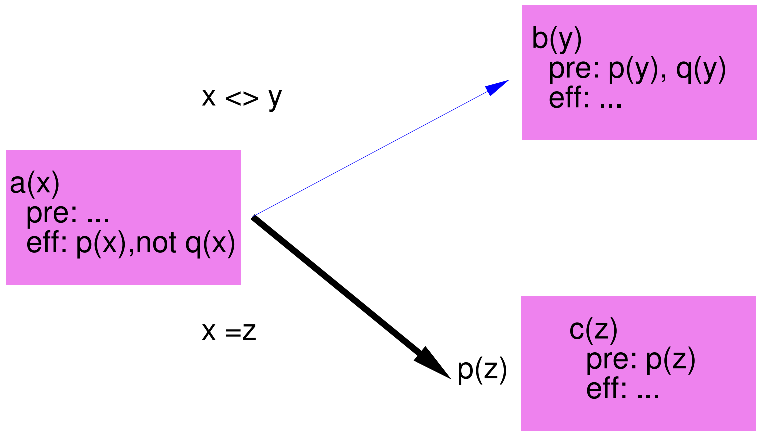
Plan space search:

- no notion of states, just partial plans
- adopts a least-commitment strategy: don't commit to orderings, instantiations, etc, unless necessary
- produces a partially ordered plan: represents all sequences of actions compatible with the partial ordering
- benefits: speed-ups (in principle), flexible execution, easier replanning

Plan space search: basic idea

Plan-space search builds a partial plan:

- multiset O of operators $\{o_1, \ldots, o_n\}$
- set $<$ of ordering constraints $o_i < o_j$ (with transitivity built in)
- set B of binding constraints $x = y$, $x \neq y$, $x \in D$, $x \notin D$, substitutions
- set L of causal links $o_i \xrightarrow{p} o_j$ stating that (effect p) of o_i establishes precondition p of o_j, with $o_i < o_j$ and binding constraints in B for parameters of o_i and o_j appearing in p



Plan-space search: basic idea

Nodes are partial plans

- initial node is $(O : \{\text{start}, \text{end}\}, <: \{\text{start} < \text{end}\}, B : \{\,\}, L : \{\,\})$ with $\text{EFF}(\text{start}) = s_0$ and $\text{PRE}(\text{end}) = g$.

Successors are determined by plan refinment operations

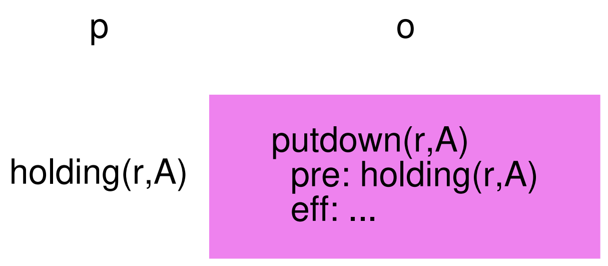
- each operation add elements to O, $<$, B, L to resolve a flaw in the plan

Search through the plan space until a partial plan is found which has no flaw:

- no open precondition: all preconditions of all operators in O are established by causal links in L
- no threat (each linearisation is safe): for every causal link $o_i \xrightarrow{p} o_j$, every o_k with $\text{EFF}^-(o_k)$ unifable with p is such that $o_k < o_i$ or $o_j < o_k$
- $<$ and B are consistent

How to resolve an open precondition flaw?

Flaw: an operator o in the plan has a precondition p which is not established

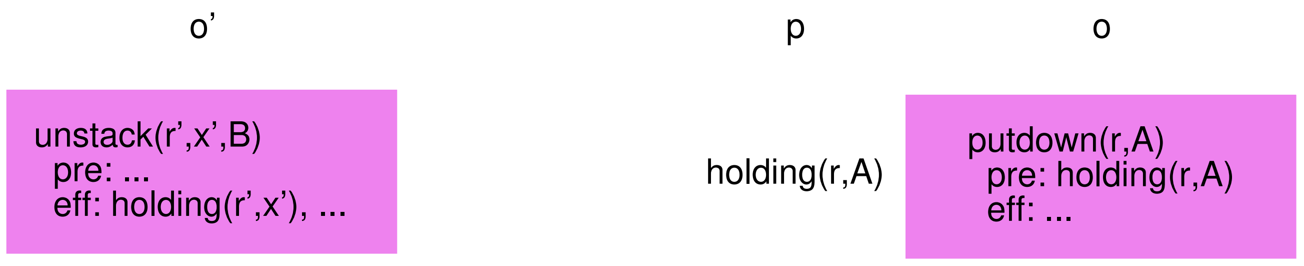


How to resolve an open precondition flaw?

Flaw: an operator o in the plan has a precondition p which is not established

Resolving the flaw:

1. find an operator o' (either already in the plan or insert it) which can be used to establish p, i.e. o' can be ordered before o and one of its effects can unify with p

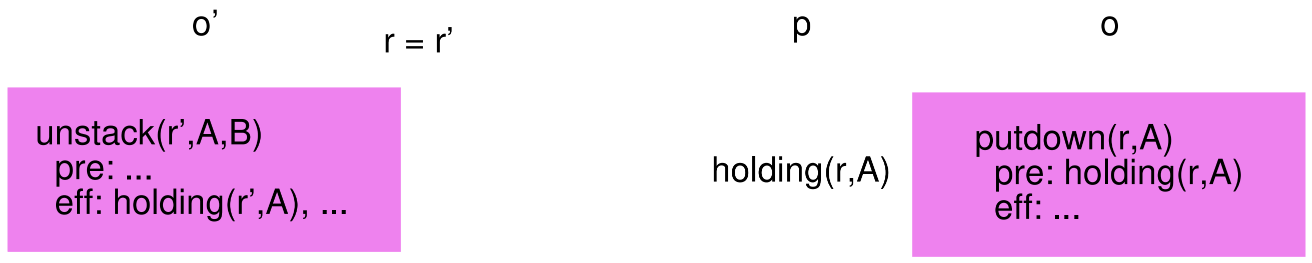


How to resolve an open precondition flaw?

Flaw: an operator o in the plan has a precondition p which is not established

Resolving the flaw:

1. find an operator o' (either already in the plan or insert it) which can be used to establish p, i.e. o' can be ordered before o and one of its effects can unify with p
2. add to B binding constraints to unify the effect of o' with p

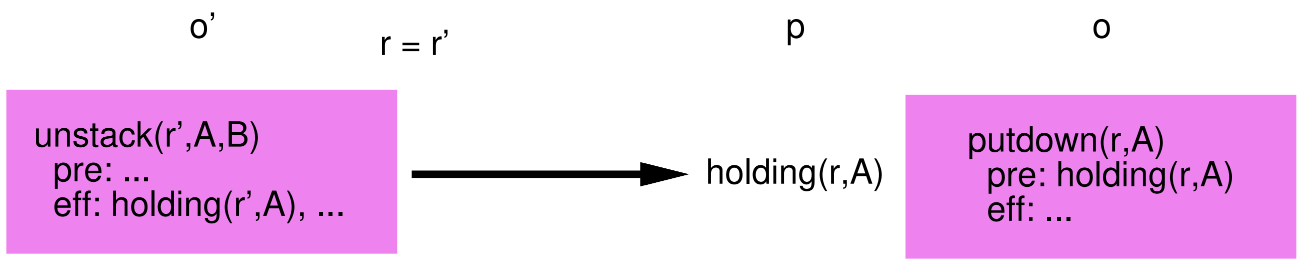


How to resolve an open precondition flaw?

Flaw: an operator o in the plan has a precondition p which is not established

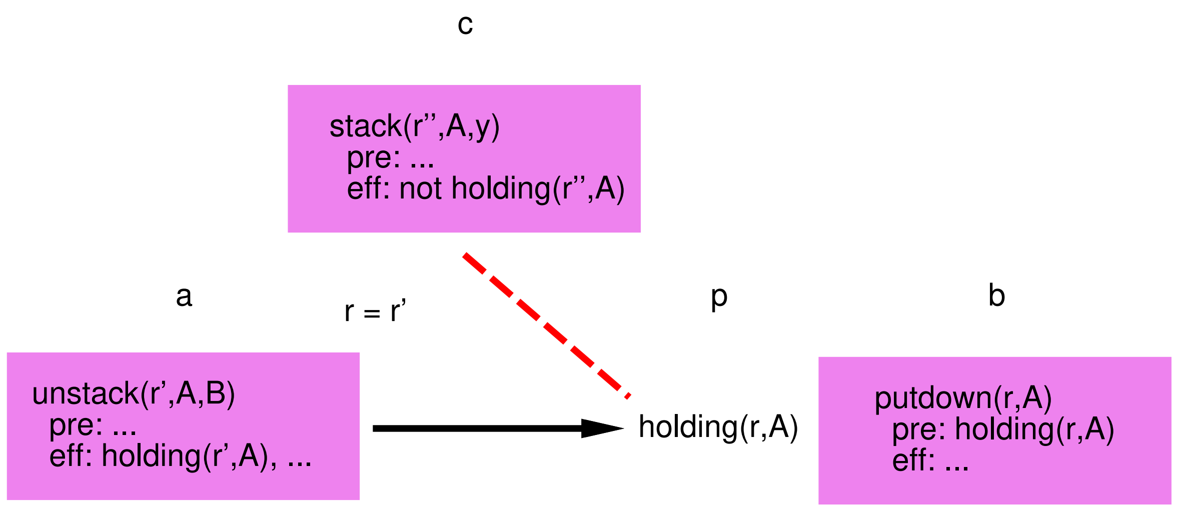
Resolving the flaw:

1. find an operator o' (either already in the plan or insert it) which can be used to establish p, i.e. o' can be ordered before o and one of its effects can unify with p
2. add to B binding constraints to unify the effect of o' with p
3. add to L the causal link $o' \xrightarrow{p} o$ (and the ordering constraint $o' < o$).



How to resolve a threat flaw?

Flaw: An operator a establishes a condition p for operator b, but another operator c is capable of deleting p before b gets to use it

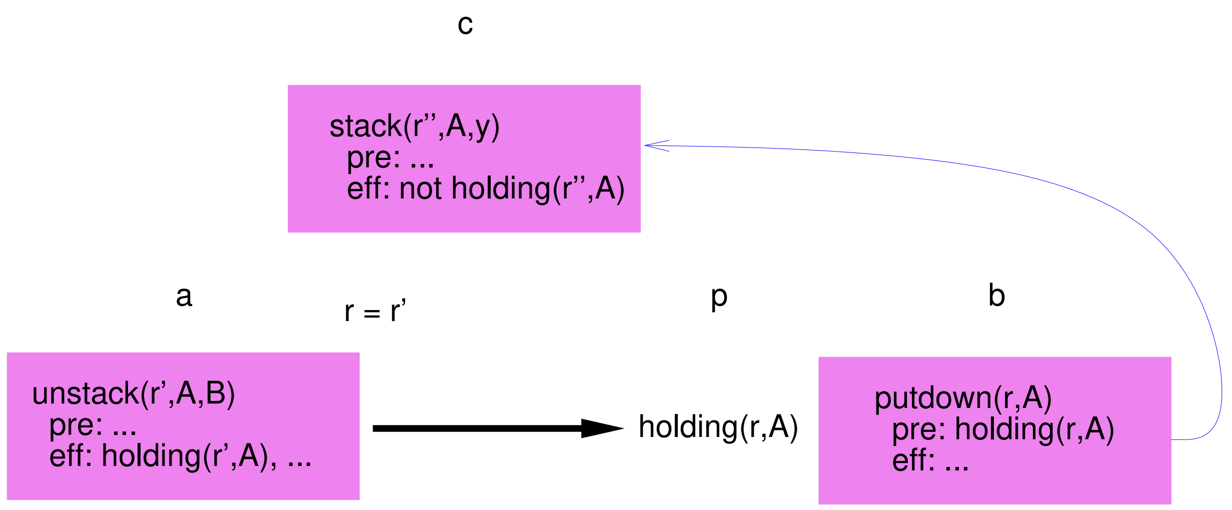


How to resolve a threat flaw?

Flaw: An operator a establishes a condition p for operator b, but another operator c is capable of deleting p before b gets to use it

Resolving the flaw - 3 possibilities:

1. order c after b

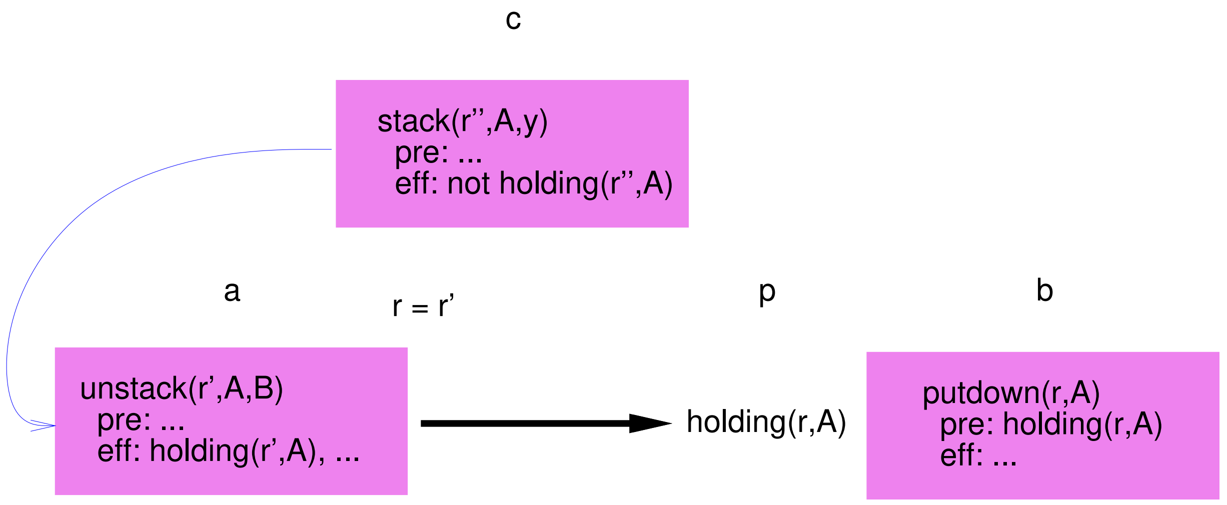


How to resolve a threat flaw?

Flaw: An operator a establishes a condition p for operator b, but another operator c is capable of deleting p before b gets to use it

Resolving the flaw - 3 possibilities:

1. order c after b
2. order c before a

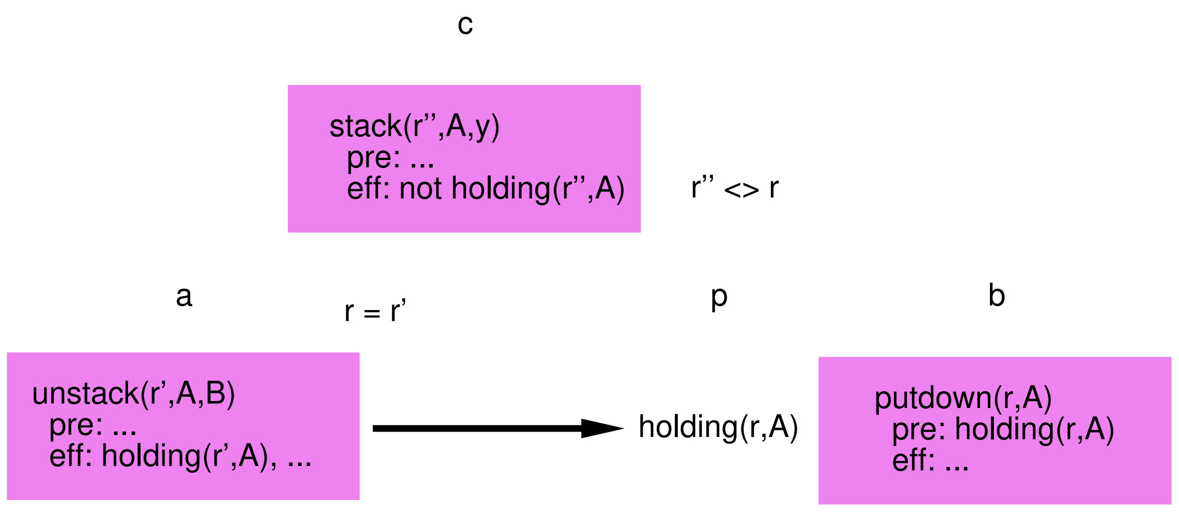


How to resolve a threat flaw?

Flaw: An operator a establishes a condition p for operator b, but another operator c is capable of deleting p before b gets to use it

Resolving the flaw - 3 possibilities:

1. order c after b
2. order c before a
3. add a binding constraint preventing c to delete p



Plan-space planning algorithm

```
function PLAN-SPACE-PLANNING($\pi$) returns a plan, or failure
    $F \leftarrow$ OPEN-PRECONDITIONS($\pi$) $\cup$ THREATS($\pi$)
    if $F = \{\}$ then return $\pi$
    select a flaw $f \in F$
    $R \leftarrow$ RESOLVE($f, \pi$)
    if $R = \{\}$ then return failure
    choose a resolver $r \in R$
    $\pi' \leftarrow$ REFINE($r, \pi$)
    return PLAN-SPACE-PLANNING($\pi'$)
```

PLAN-SPACE-PLANNING is sound and complete

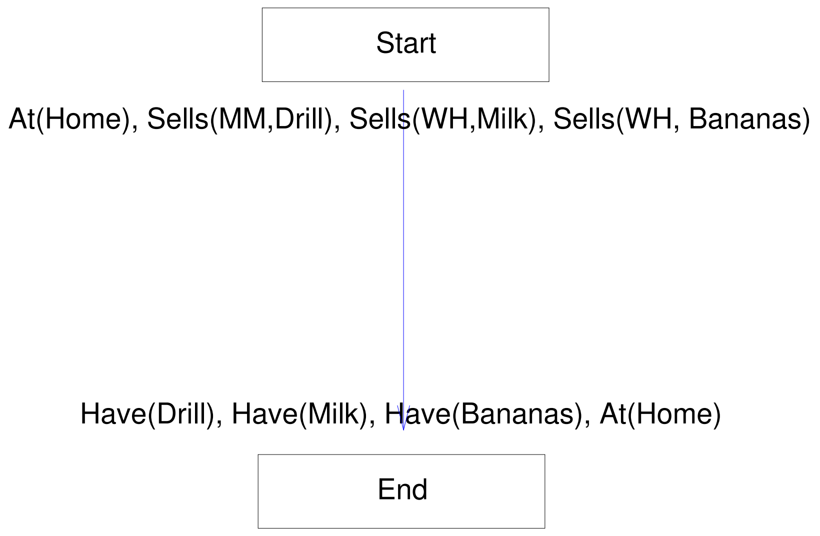
Grounded variant: no binding constraints needed

Example

- operator Start
  - precondition: $\{\,\}$
  - effect: $\{$At(Home), Sells(MM,Drill), Sells(WH,Milk), Sells(WH,Bananas)$\}$
- operator End
  - precondition: $\{$At(Home), Have(Drill), Have(Milk), Have(Bananas)$\}$
  - effect: $\{\,\}$
- operator Go(l, l')
  - precondition: $\{$At$(l)\}$
  - effect: $\{$At$(l'), \neg$At$(l)\}$
- operator Buy(i, s)
  - precondition: $\{$At(s), Sells$(s, i)\}$
  - effect: $\{$Have$(i)\}$

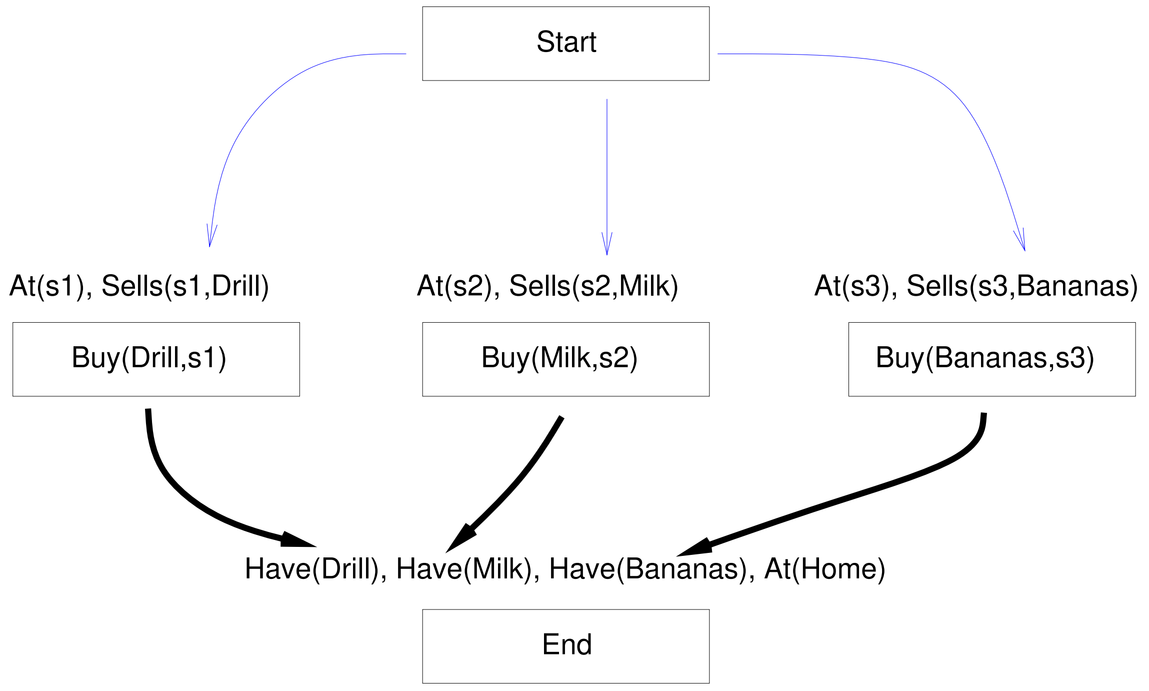
Example

Initial Plan


Start
At(Home), Sells(MM,Drill), Sells(WH,Milk), Sells(WH, Bananas)
Have(Drill), Have(Milk), Have(Bananas), At(Home)
End

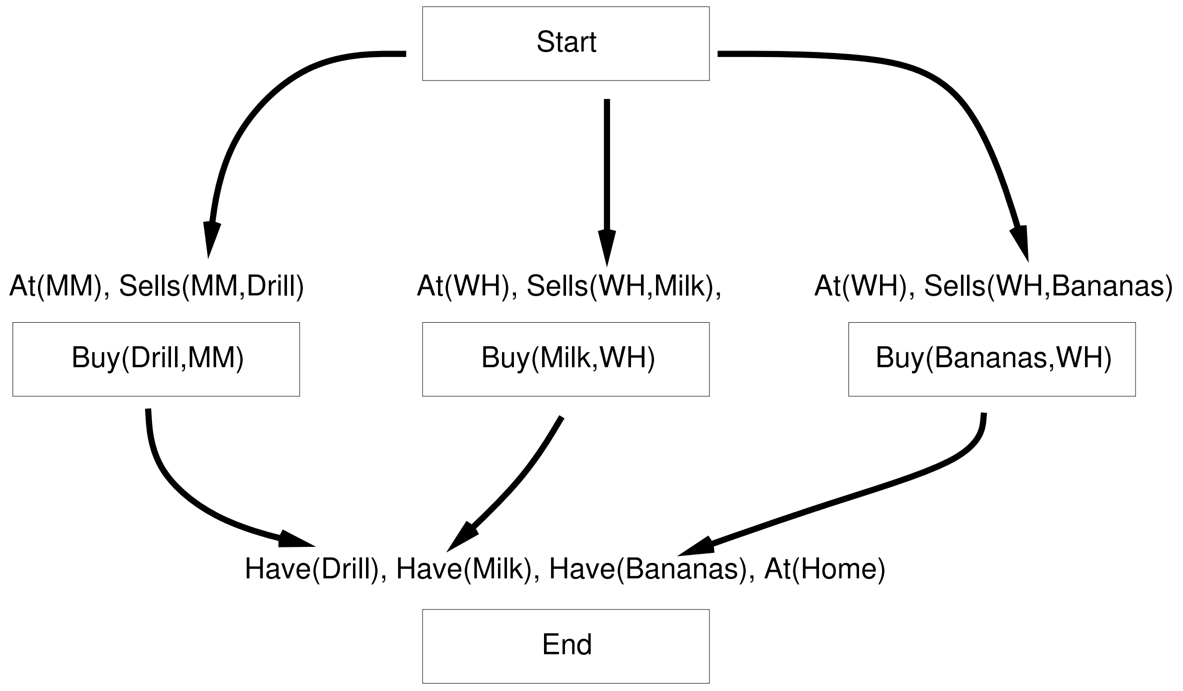
Example

The only possible ways to establish the "Have" preconditions



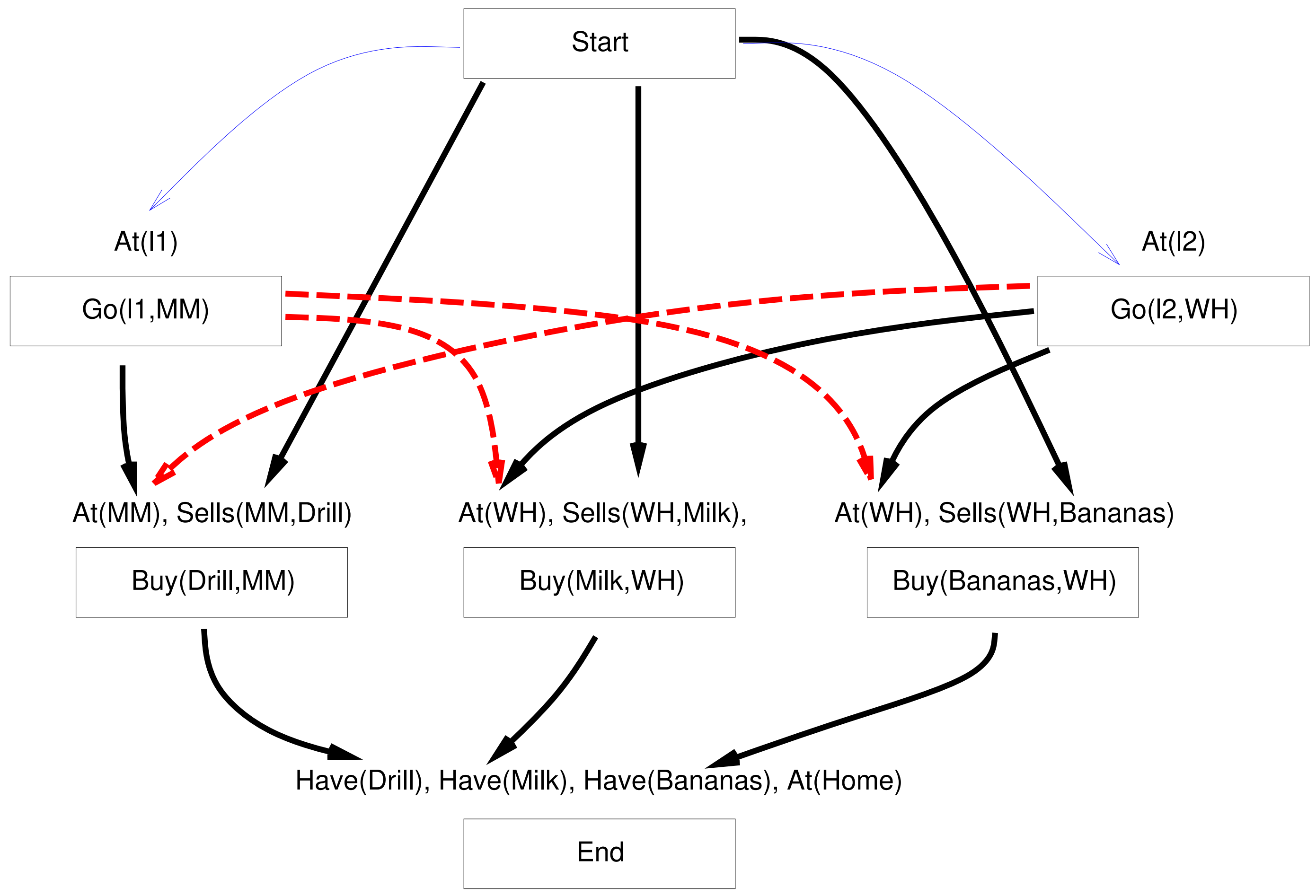
Example

The only possible ways to establish the “Sells” preconditions



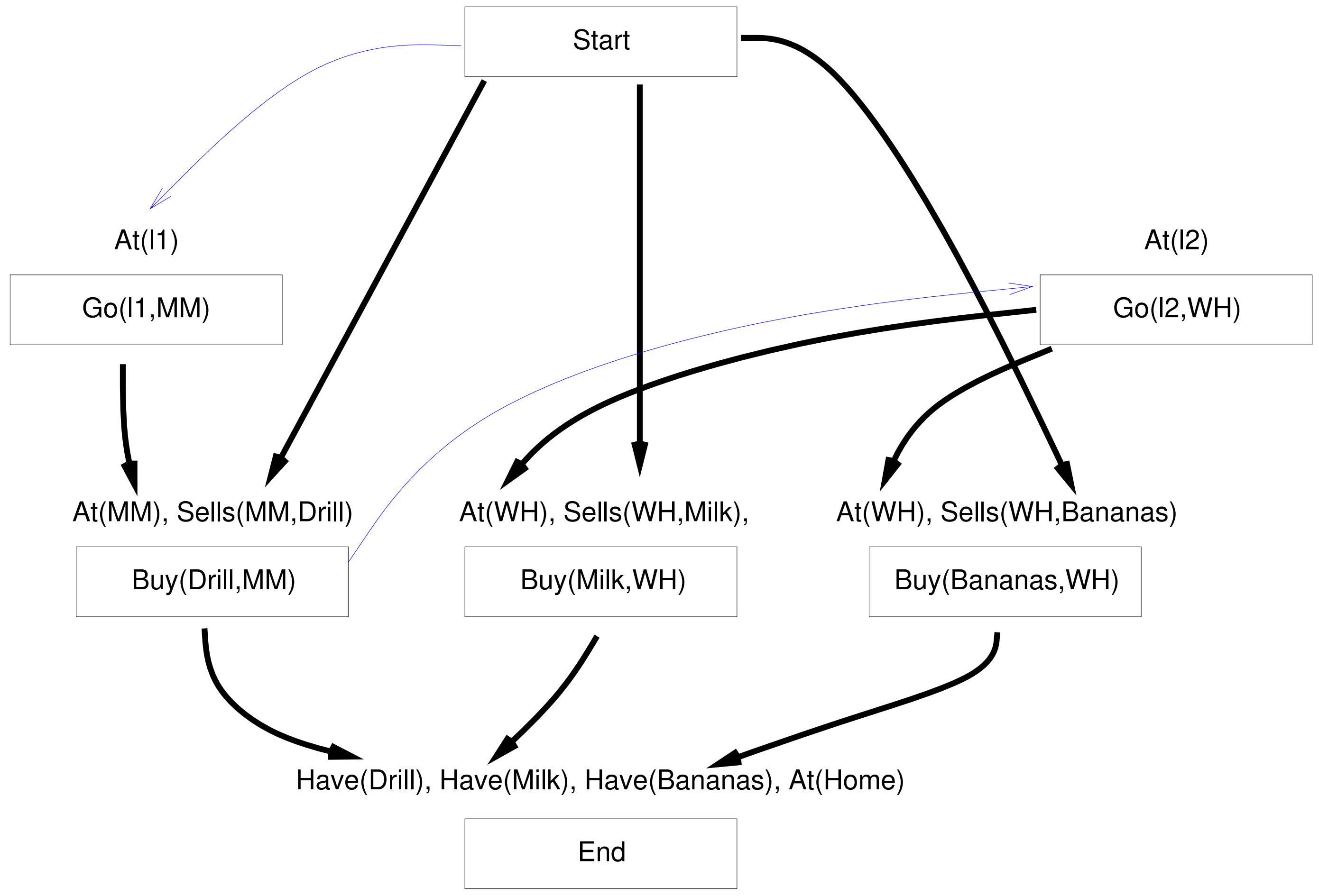
Example

The only ways to establish At(MM) and At(WH).
Note the threats



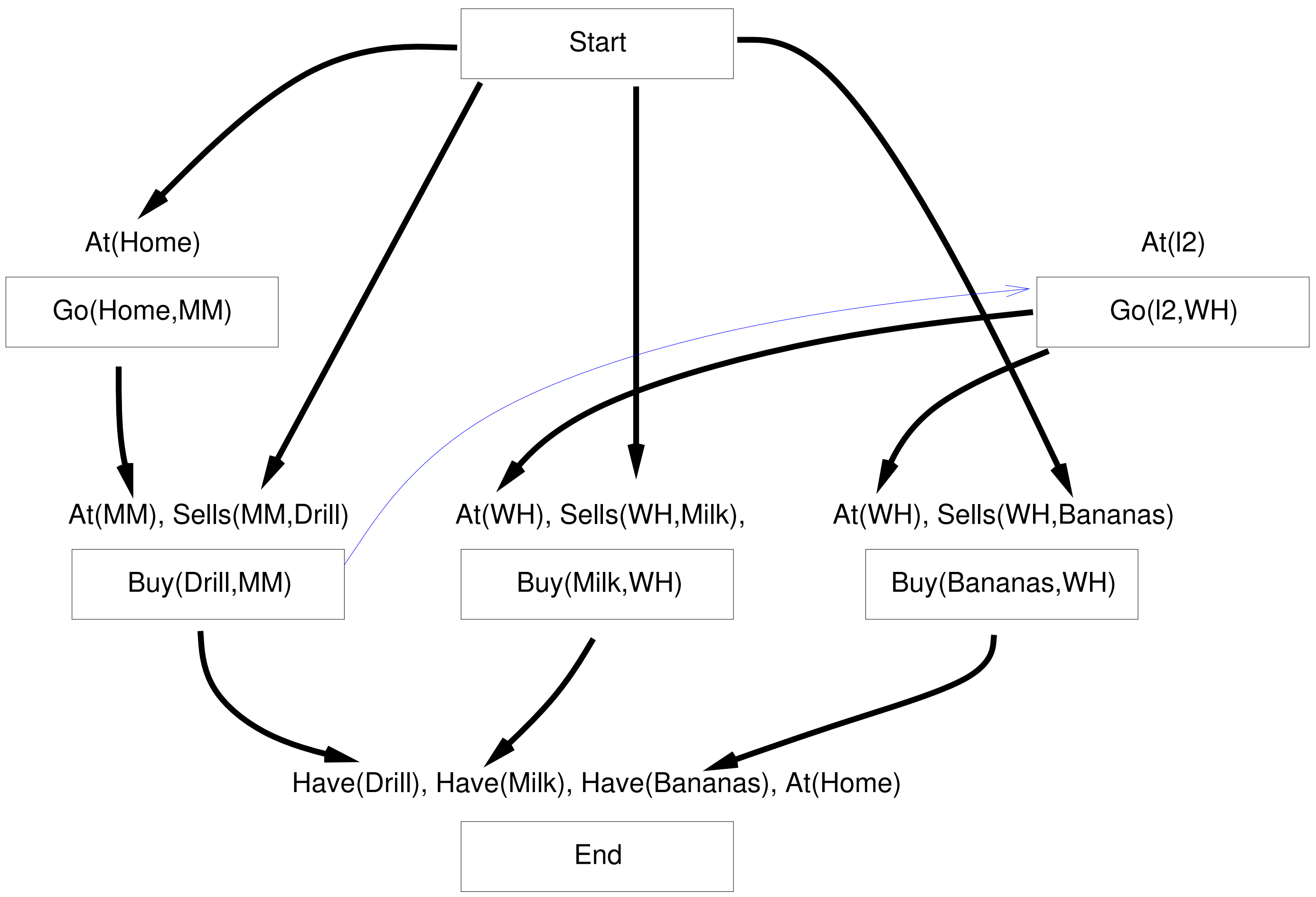
Example

To resolve the 3rd threat, order Go$(l2, \text{WH})$ after Buy(Drill).
This resolves all three threats.



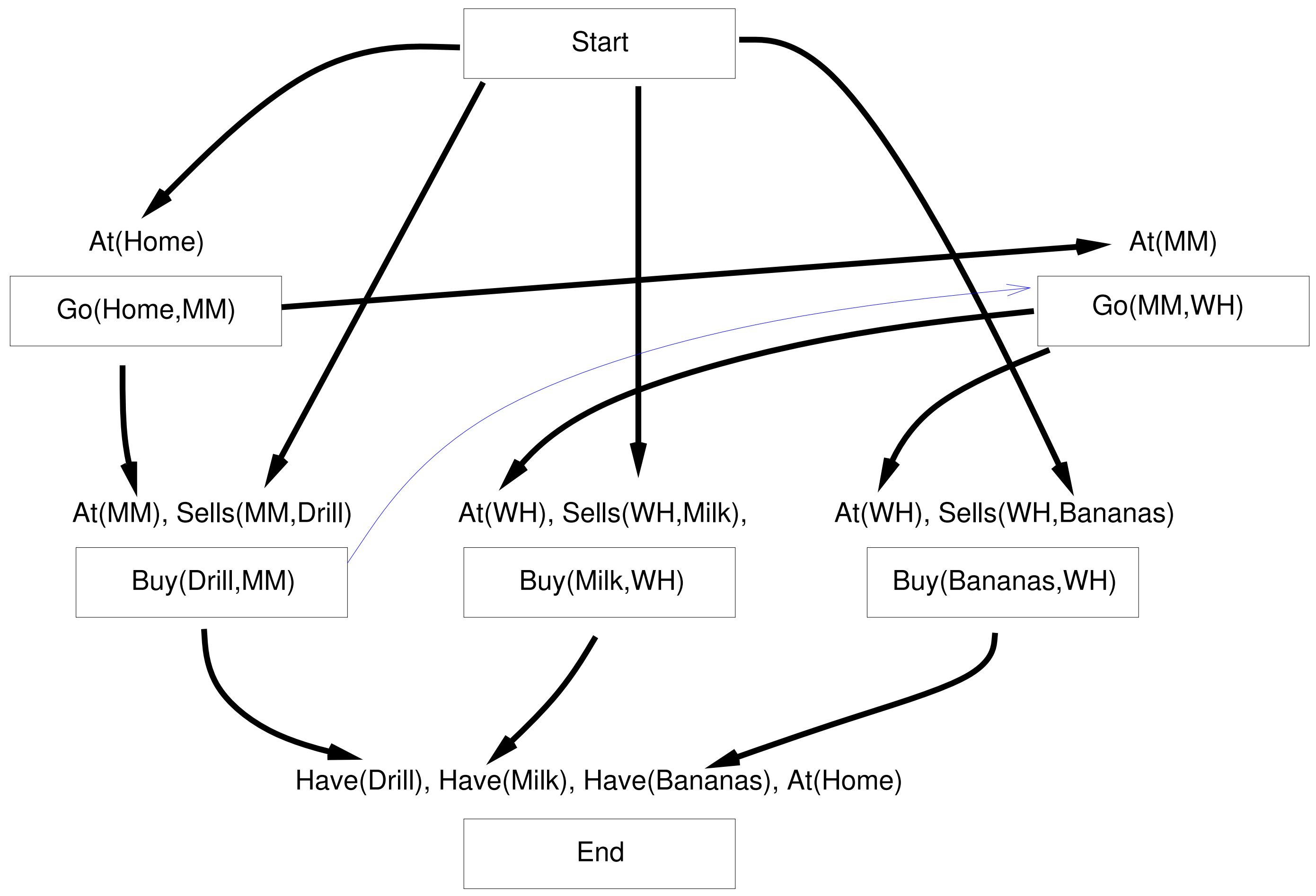
Example

Add binding constraint $l1 =$ Home and causal link to establish At$(l1)$



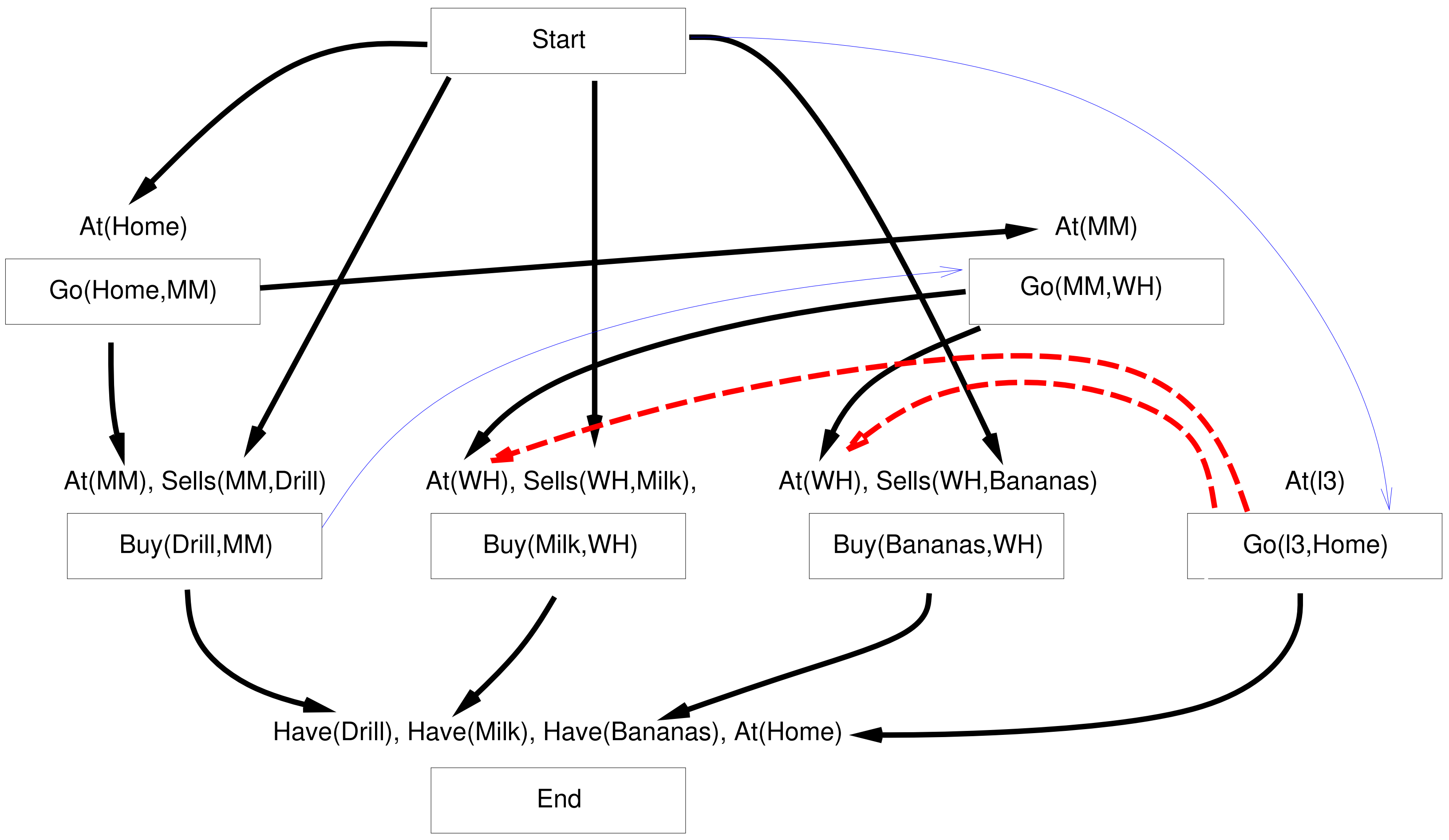
Example

Add binding constraint $l2 =$ MM and causal link to establish At$(l2)$



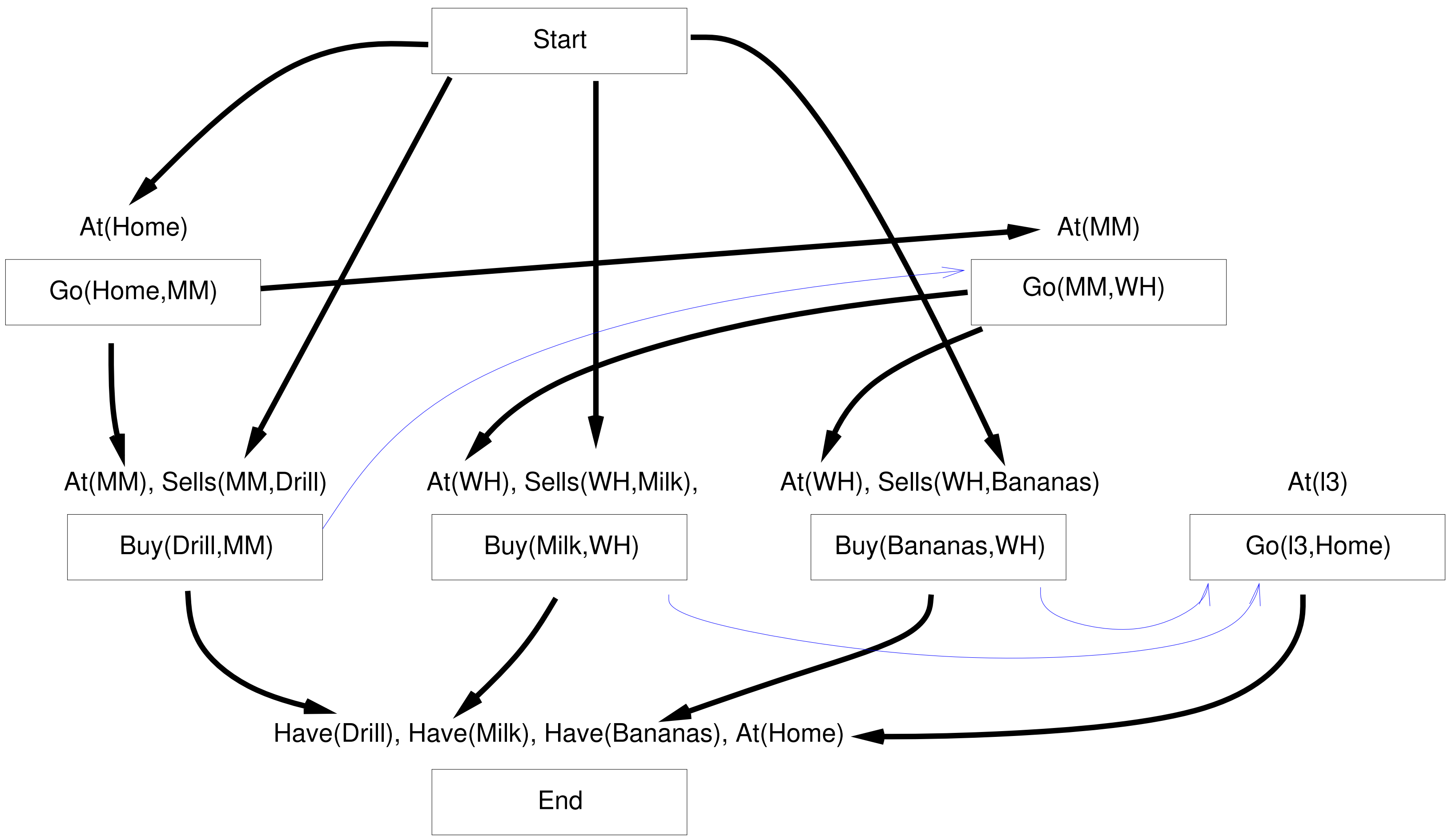
Example

Establish At(Home) for end.
Note the threats.



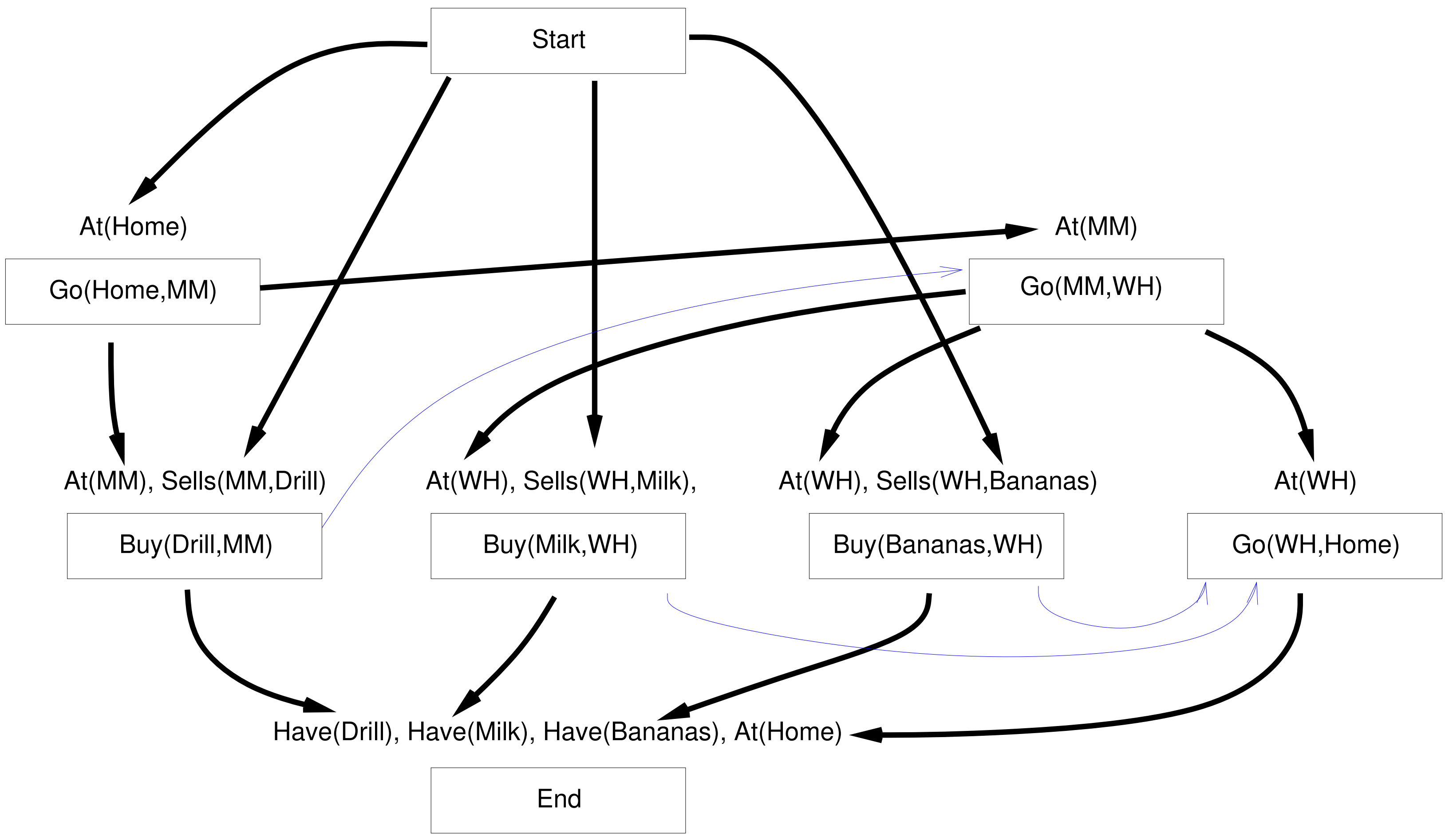
Example

Order Go(Home) after Buy(Milk) and Buy(Banana) to remove the threats.



Example

Add binding constraint $l3 =$ WH and causal link to establish At$(l3)$.
The plan is flawless.



Summary

Graph-based planning produces a polynomial-size graph that gives us a necessary condition for the existence of a parallel plan of a given length. If one really exist, it can be extracted by backward search through the graph.

SAT planning uses a SAT solver to solve the bounded (parallel) plan generation problem. This is efficient when optimal parallel plans are short. Logical planning formalisms must deal with the frame problem.

State-space planning produces totally-ordered plans by a forward or backward search in the state space. This requires domain-independent heuristics or domain-specific control rules to be efficient

Plan-space planning produces partially-ordered plans. This approach does not commit to orderings or bindings unless necessary. It searches the space of partial plans, refining the plan at each step to remove flaws.

Current planning research extends these methods to handle time, uncertainty, multiple agents, discrete/continuous systems, and real world applications.