Lecture *Hierarchical Planning*

Chapter:
Solving (Non-Hierarchical) Planning Problems via Search

Dr. Pascal Bercher

Institute of Artificial Intelligence,
Ulm University, Germany

Winter Term 2018/2019

(Compiled on: February 19, 2019)

ulm university universität uulm

Overview:

How to Solve Planning Problems?

Solving techniques:

How to Solve Planning Problems?

Solving techniques:

- Via reduction, i.e., compilation to other problems like:

How to Solve Planning Problems?

Solving techniques:

- Via reduction, i.e., compilation to other problems like:
  - SAT, i.e., Satisfiability.

How to Solve Planning Problems?

Solving techniques:

- Via reduction, i.e., compilation to other problems like:
  - SAT, i.e., Satisfiability.
  - ASP, i.e., Answer Set Programming (not covered).

How to Solve Planning Problems?

Solving techniques:

- Via reduction, i.e., compilation to other problems like:
  - SAT, i.e., Satisfiability.
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).

How to Solve Planning Problems?

Solving techniques:

- Via reduction, i.e., compilation to other problems like:
  - SAT, i.e., Satisfiability.
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).
- Search:

How to Solve Planning Problems?

Solving techniques:

- Via reduction, i.e., compilation to other problems like:
  - SAT, i.e., Satisfiability.
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).
- Search:
  - Progression search.

How to Solve Planning Problems?

Solving techniques:

- Via reduction, i.e., compilation to other problems like:
  - SAT, i.e., Satisfiability.
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).
- Search:
  - Progression search.
  - Regression search, e.g., via POCL planning.

How to Solve Planning Problems?

Solving techniques:

- Via reduction, i.e., compilation to other problems like:
  - SAT, i.e., Satisfiability.
  - ASP, i.e., Answer Set Programming (not covered).
  - Many more (what ever problem (class) fits to the current problem).
- Search:
  - Progression search.
  - Regression search, e.g., via POCL planning.
  - Local search (not covered).

Search-based Planning

This chapter covers *planning as heuristic search*:

Search-based Planning

This chapter covers *planning as heuristic search*:

- Forward progression search in the space of world states: *Classical Planning*.

Search-based Planning

This chapter covers *planning as heuristic search*:

- Forward progression search in the space of world states: *Classical Planning*.
- (Regression-like) search in the space of partial plans: *Partial-Order Causal Link (POCL) planning*.

Search-based Planning

This chapter covers *planning as heuristic search*:

- Forward progression search in the space of world states: *Classical Planning*.
- (Regression-like) search in the space of partial plans: *Partial-Order Causal Link (POCL) planning*.

→ Both will be extended for hierarchical planning.

Search-based Planning

This chapter covers *planning as heuristic search*:

- Forward progression search in the space of world states: *Classical Planning*.
- (Regression-like) search in the space of partial plans: *Partial-Order Causal Link (POCL) planning*.

$\rightarrow$ Both will be extended for hierarchical planning.

- The (relaxed) planning graph as a basis for several heuristics used for planning as heuristic search – both in non-hierarchical and in hierarchical planning.

Introduction

- Classical planning, i.e., forward progression search, is conceptually extremely simple:

Introduction

- Classical planning, i.e., forward progression search, is conceptually extremely simple:
  - Start with the initial state.

Introduction

- Classical planning, i.e., forward progression search, is conceptually extremely simple:
  - Start with the initial state.
  - Apply all applicable actions to that state, generating a set of successor states.

Introduction

- Classical planning, i.e., forward progression search, is conceptually extremely simple:
  - Start with the initial state.
  - Apply all applicable actions to that state, generating a set of successor states.
  - Select the most-promising state and repeat until solution is found.

Introduction

- Classical planning, i.e., forward progression search, is conceptually extremely simple:
  - Start with the initial state.
  - Apply all applicable actions to that state, generating a set of successor states.
  - Select the most-promising state and repeat until solution is found.
  - → This is essentially exactly *standard search* (cf. second lecture).

Introduction

- Classical planning, i.e., forward progression search, is conceptually extremely simple:
  - Start with the initial state.
  - Apply all applicable actions to that state, generating a set of successor states.
  - Select the most-promising state and repeat until solution is found.
  - $\rightarrow$ This is essentially exactly *standard search* (cf. second lecture).
- For classical problems, this approach is currently state of the art (in combination with informed heuristics).

Introduction

- Classical planning, i.e., forward progression search, is conceptually extremely simple:
  - Start with the initial state.
  - Apply all applicable actions to that state, generating a set of successor states.
  - Select the most-promising state and repeat until solution is found.
  - → This is essentially exactly *standard search* (cf. second lecture).
- For classical problems, this approach is currently state of the art (in combination with informed heuristics).
- This algorithm will be extended for hierarchical planning.

Pseudo Code

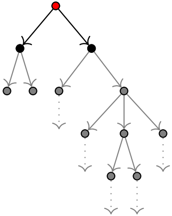
Algorithm: Classical Planning

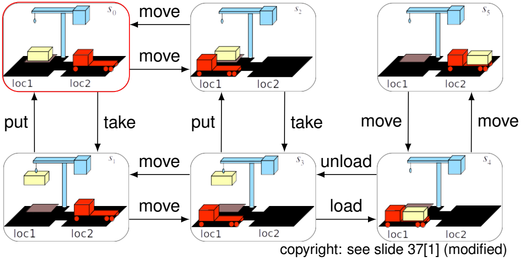
Input: A STRIPS planning problem $\langle V, A, s_I, g \rangle$

Output: A solution $\bar{a}$ or ***fail*** if none exists

$fringe \leftarrow \{(s_I, \varepsilon)\}$
while $fringe \neq \emptyset$ **do**
    $(s, \bar{a}) \leftarrow nodeSelectAndRemove(fringe)$
    if $s \supseteq g$ **then return** $\bar{a}$
    for $a \in A$ **do**
        if $pre(a) \subseteq s$ **then**
            $s' = (s \setminus del(a)) \cup add(a)$
            $fringe \leftarrow fringe \cup \{(s', \bar{a} \circ a)\}$
return ***fail***

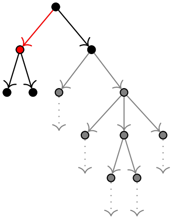
Example

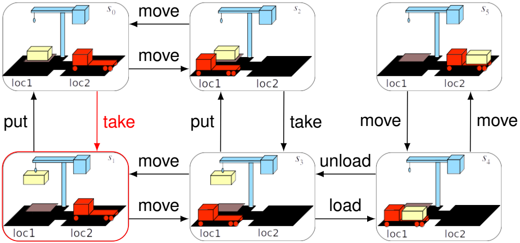




$$s_I = \left\{ \begin{array}{l} TruckAtLoc2, \\ CrateAtLoc1 \end{array} \right\}$$

Example





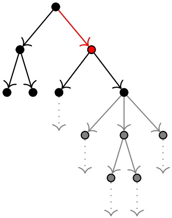
copyright: see slide 37[1] (modified)

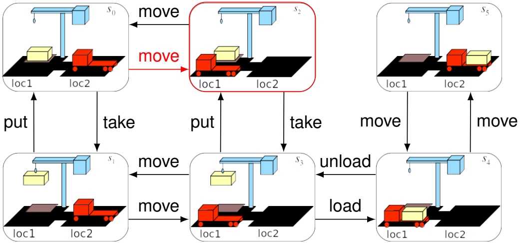
$s_I = \left\{ \begin{array}{l} TruckAtLoc2, \\ CrateAtLoc1 \end{array} \right\}$

CrateAtLoc1 → *take* → HoldCrate, ¬CrateAtLoc1

$\left\{ \begin{array}{l} TruckAtLoc2, \\ HoldCrate \end{array} \right\}$

Example





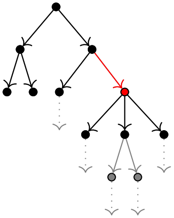
copyright: see slide 37[1] (modified)

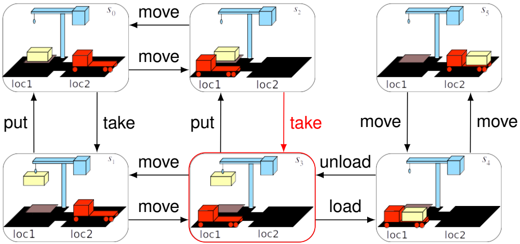
$s_I = \left\{ \begin{array}{l} \mathit{TruckAtLoc2}, \\ \mathit{CrateAtLoc1} \end{array} \right\}$

TruckAtLoc2 — *moveLeft* — TruckAtLoc1, ¬TruckAtLoc2

$\left\{ \begin{array}{l} \mathit{TruckAtLoc1}, \\ \mathit{CrateAtLoc1} \end{array} \right\}$

Example





copyright: see slide 37[1] (modified)

$s_I = \{TruckAtLoc2, CrateAtLoc1\}$

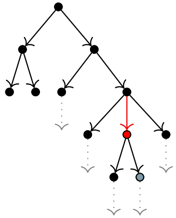
TruckAtLoc2 — *moveLeft* — TruckAtLoc1, ¬TruckAtLoc2

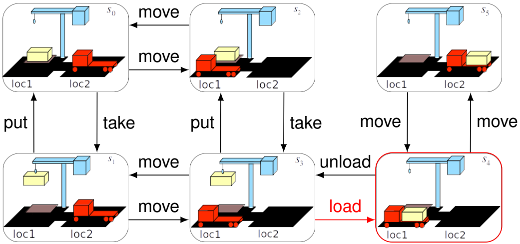
$\{TruckAtLoc1, CrateAtLoc1\}$

CrateAtLoc1 — *take* — HoldCrate, ¬CrateAtLoc1

$\{TruckAtLoc1, HoldCrate\}$

Example





copyright: see slide 37[1] (modified)

$s_I = \left\{ \begin{array}{l} TruckAtLoc2, \\ CrateAtLoc1 \end{array} \right\}$

TruckAtLoc2 — *moveLeft* — TruckAtLoc1, ¬TruckAtLoc2

$\left\{ \begin{array}{l} TruckAtLoc1, \\ CrateAtLoc1 \end{array} \right\}$

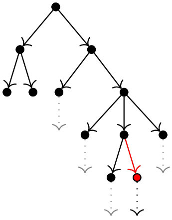
CrateAtLoc1 — *take* — HoldCrate, ¬CrateAtLoc1

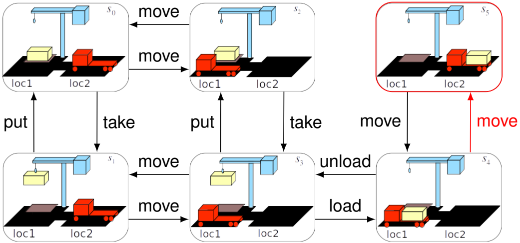
$\left\{ \begin{array}{l} TruckAtLoc1, \\ HoldCrate \end{array} \right\}$

HoldCrate, TruckAtLoc1 — *load* — CrateInTruck, ¬HoldCrate

$\left\{ \begin{array}{l} TruckAtLoc1, \\ CrateInTruck \end{array} \right\}$

Example





copyright: see slide 37[1] (modified)

$s_I = \left\{ \begin{array}{l} \mathit{TruckAtLoc2}, \\ \mathit{CrateAtLoc1} \end{array} \right\}$ — TruckAtLoc2 → *moveLeft* → TruckAtLoc1, ¬TruckAtLoc2 — $\left\{ \begin{array}{l} \mathit{TruckAtLoc1}, \\ \mathit{CrateAtLoc1} \end{array} \right\}$ — CrateAtLoc1 → *take* → HoldCrate, ¬CrateAtLoc1 — $\left\{ \begin{array}{l} \mathit{TruckAtLoc1}, \\ \mathit{HoldCrate} \end{array} \right\}$

HoldCrate, TruckAtLoc1 → *load* → CrateInTruck, ¬HoldCrate — $\left\{ \begin{array}{l} \mathit{TruckAtLoc1}, \\ \mathit{CrateInTruck} \end{array} \right\}$ — TruckAtLoc1 → *moveRight* → TruckAtLoc2, ¬TruckAtLoc1 — $\left\{ \begin{array}{l} \mathit{TruckAtLoc2}, \\ \mathit{CrateInTruck} \end{array} \right\} \supseteq g = \left\{ \begin{array}{l} \mathit{TruckAtLoc2}, \\ \mathit{CrateInTruck} \end{array} \right\}$

Notes

- Only for simplicity, we stored the action sequences directly in the search nodes. Ordinarily, they are inferred from the search space (just as in search).

Notes

- Only for simplicity, we stored the action sequences directly in the search nodes. Ordinarily, they are inferred from the search space (just as in search).
- For good/low runtimes, there exist various techniques that ensure an efficient implementation:

Notes

- Only for simplicity, we stored the action sequences directly in the search nodes. Ordinarily, they are inferred from the search space (just as in search).
- For good/low runtimes, there exist various techniques that ensure an efficient implementation:
  - Use efficient data structures (e.g., bit vectors rather than sets for state representation).

Notes

- Only for simplicity, we stored the action sequences directly in the search nodes. Ordinarily, they are inferred from the search space (just as in search).
- For good/low runtimes, there exist various techniques that ensure an efficient implementation:
  - Use efficient data structures (e.g., bit vectors rather than sets for state representation).
  - Only apply actions that change the current state.

Notes

- Only for simplicity, we stored the action sequences directly in the search nodes. Ordinarily, they are inferred from the search space (just as in search).
- For good/low runtimes, there exist various techniques that ensure an efficient implementation:
  - Use efficient data structures (e.g., bit vectors rather than sets for state representation).
  - Only apply actions that change the current state.
  - Test action applicability efficiently, e.g., relying on decision trees. Cf. *Successor Generators* in the work by Malte Helmert. "The Fast Downward Planning System". In: *Journal of Artificial Intelligence Research (JAIR)* 26 (2006), pp. 191–246

Properties

Theorem

Classical Planning is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*().

Properties

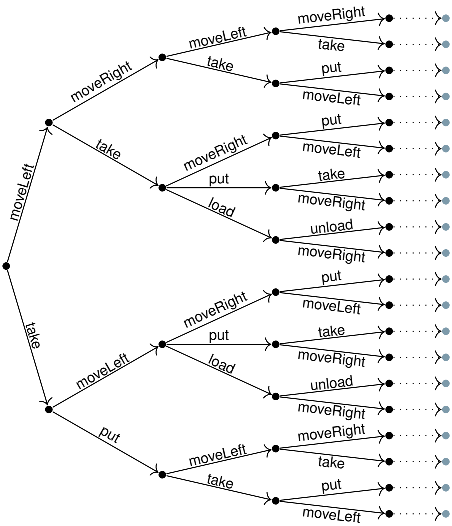
Theorem

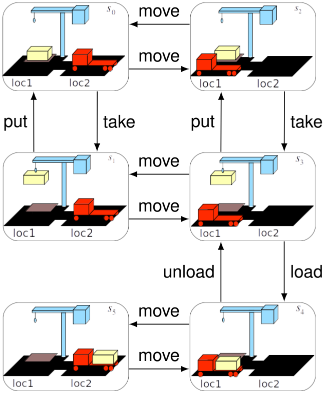
Classical Planning is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*().

Proof:
Follows from the properties of the underlying search algorithm.

Search-Guidance in Classical Planning





Search-Guidance in Classical Planning, cont'd

Problems with the Search-Guidance:

- High branching factor: usually, many actions are applicable in the current state – resulting in a large search fringes.
- Which state to explore next is decided by heuristics (see later in this chapter).

Motivation

- The core idea behind POCL planning is *Least Commitment*. Only make decisions that are actually required:

Motivation

- The core idea behind POCL planning is *Least Commitment*. Only make decisions that are actually required:
  - Only introduce an ordering if required,

Motivation

- The core idea behind POCL planning is *Least Commitment*.
  Only make decisions that are actually required:
  - Only introduce an ordering if required,
  - only insert required variable constraints (not yet covered here), and

Motivation

- The core idea behind POCL planning is *Least Commitment*.
  Only make decisions that are actually required:
  - Only introduce an ordering if required,
  - only insert required variable constraints (not yet covered here), and
  - only insert actions if they contribute towards some precondition.

Motivation

- The core idea behind POCL planning is *Least Commitment*.
  Only make decisions that are actually required:
  - Only introduce an ordering if required,
  - only insert required variable constraints (not yet covered here), and
  - only insert actions if they contribute towards some precondition.
- Consequently, search nodes are partially ordered plans which can represent an exponential number of classical solutions in one node.
  $\rightarrow$ Prevents early commitment on when actions are applied.

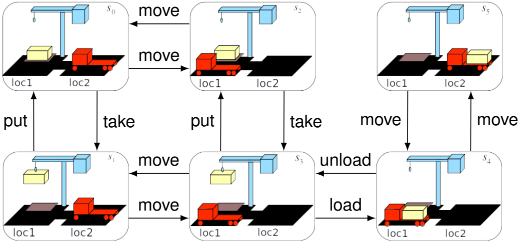
Motivation

- The core idea behind POCL planning is *Least Commitment*.
  Only make decisions that are actually required:
  - Only introduce an ordering if required,
  - only insert required variable constraints (not yet covered here), and
  - only insert actions if they contribute towards some precondition.
- Consequently, search nodes are partially ordered plans which can represent an exponential number of classical solutions in one node.
  $\rightarrow$ Prevents early commitment on when actions are applied.
- In contrast to classical planning, POCL planning searches in a regression-like fashion.

Motivation

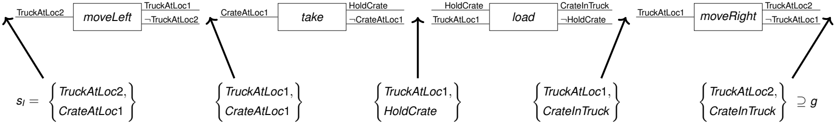
- The core idea behind POCL planning is *Least Commitment*.
  Only make decisions that are actually required:
  - Only introduce an ordering if required,
  - only insert required variable constraints (not yet covered here), and
  - only insert actions if they contribute towards some precondition.
- Consequently, search nodes are partially ordered plans which can represent an exponential number of classical solutions in one node.
  $\rightarrow$ Prevents early commitment on when actions are applied.
- In contrast to classical planning, POCL planning searches in a regression-like fashion.
- This algorithm will (also) be extended to a plan space-based algorithm for hierarchical planning.

Example

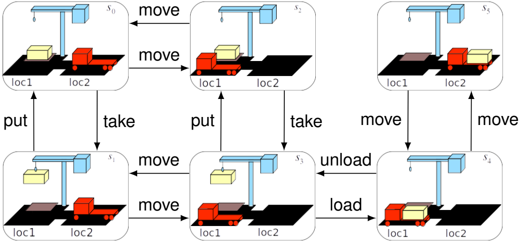


copyright: see slide 37[1] (modified)

Classical (totally ordered) Solution:

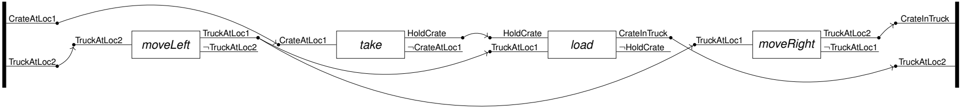


Example

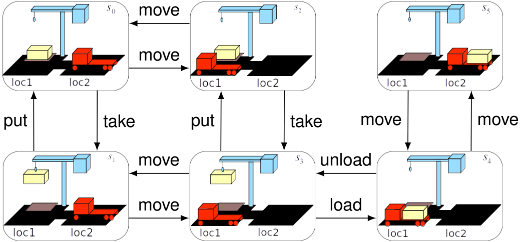


copyright: see slide 37[1] (modified)

Totally ordered POCL solution:

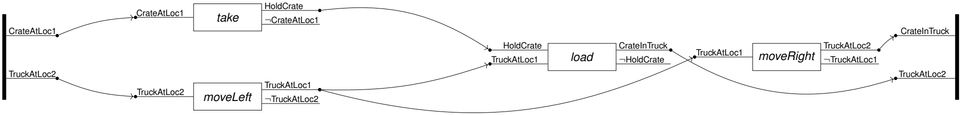


Example



copyright: see slide 37[1] (modified)

Partially ordered POCL solution:



Classical Planning vs. POCL Planning

- Reminder classical planning:

Classical Planning vs. POCL Planning

- Reminder classical planning:
  - Nodes of the search tree/graph contain states.

Classical Planning vs. POCL Planning

- Reminder classical planning:
  - Nodes of the search tree/graph contain states.
  - Edges are actions.

Classical Planning vs. POCL Planning

- Reminder classical planning:
  - Nodes of the search tree/graph contain states.
  - Edges are actions.
  - Plans are extracted from the traversal of the initial state to a goal state.

Classical Planning vs. POCL Planning

- Reminder classical planning:
  - Nodes of the search tree/graph contain states.
  - Edges are actions.
  - Plans are extracted from the traversal of the initial state to a goal state.
  - Solutions are totally ordered action sequences.

Classical Planning vs. POCL Planning

- Reminder classical planning:
  - Nodes of the search tree/graph contain states.
  - Edges are actions.
  - Plans are extracted from the traversal of the initial state to a goal state.
  - Solutions are totally ordered action sequences.
- POCL planning:

Classical Planning vs. POCL Planning

- Reminder classical planning:
  - Nodes of the search tree/graph contain states.
  - Edges are actions.
  - Plans are extracted from the traversal of the initial state to a goal state.
  - Solutions are totally ordered action sequences.
- POCL planning:
  - Nodes of the search tree/graph are partial plans.

Classical Planning vs. POCL Planning

- Reminder classical planning:
  - Nodes of the search tree/graph contain states.
  - Edges are actions.
  - Plans are extracted from the traversal of the initial state to a goal state.
  - Solutions are totally ordered action sequences.
- POCL planning:
  - Nodes of the search tree/graph are partial plans.
  - Edges are plan refinements.

Classical Planning vs. POCL Planning

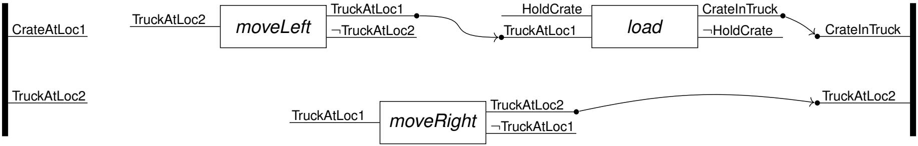
- Reminder classical planning:
  - Nodes of the search tree/graph contain states.
  - Edges are actions.
  - Plans are extracted from the traversal of the initial state to a goal state.
  - Solutions are totally ordered action sequences.
- POCL planning:
  - Nodes of the search tree/graph are partial plans.
  - Edges are plan refinements.
  - Plans are nodes/partial plans with certain properties
    (i.e., they fulfill the solution criteria).

Classical Planning vs. POCL Planning

- Reminder classical planning:
  - Nodes of the search tree/graph contain states.
  - Edges are actions.
  - Plans are extracted from the traversal of the initial state to a goal state.
  - Solutions are totally ordered action sequences.
- POCL planning:
  - Nodes of the search tree/graph are partial plans.
  - Edges are plan refinements.
  - Plans are nodes/partial plans with certain properties (i.e., they fulfill the solution criteria).
  - Solutions are only partially ordered.

Plan Refinements

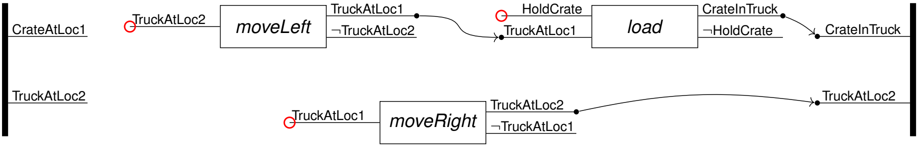
Let a search node contain the following partial plan:



Reminder: Which flaws does this partial plan possess?

Plan Refinements

Let a search node contain the following partial plan:

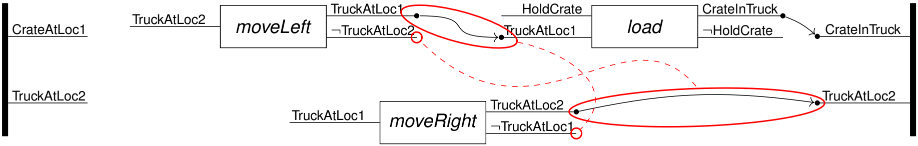


Reminder: Which flaws does this partial plan possess?

- Three open preconditions.

Plan Refinements

Let a search node contain the following partial plan:

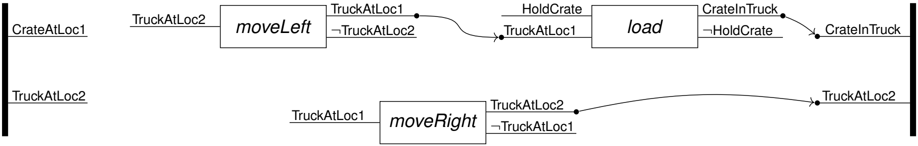


Reminder: Which flaws does this partial plan possess?

- Three open preconditions.
- Two causal threats.

Plan Refinements

Let a search node contain the following partial plan:

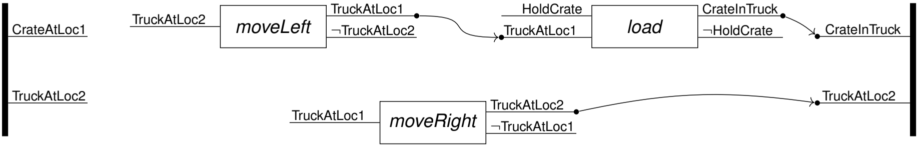


Which refinements exist to fix these flaws?

- Open preconditions:

Plan Refinements

Let a search node contain the following partial plan:

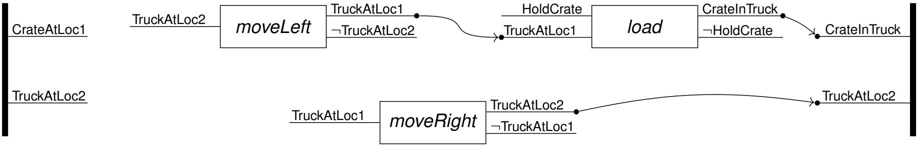


Which refinements exist to fix these flaws?

- Open preconditions:
  - Insert causal links (re-using actions).

Plan Refinements

Let a search node contain the following partial plan:

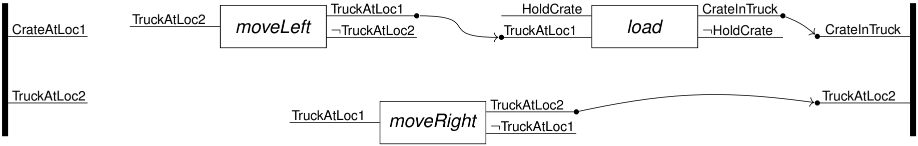


Which refinements exist to fix these flaws?

- Open preconditions:
  - Insert causal links (re-using actions).
  - Insert new actions plus causal links.

Plan Refinements

Let a search node contain the following partial plan:

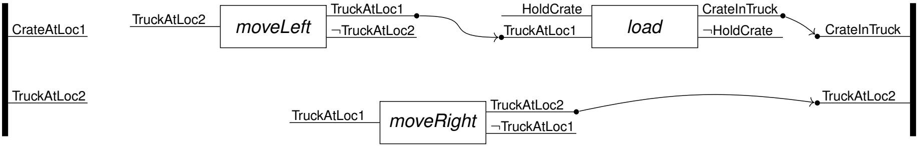


Which refinements exist to fix these flaws?

- Open preconditions:
  - Insert causal links (re-using actions).
  - Insert new actions plus causal links.
- Causal threats:

Plan Refinements

Let a search node contain the following partial plan:

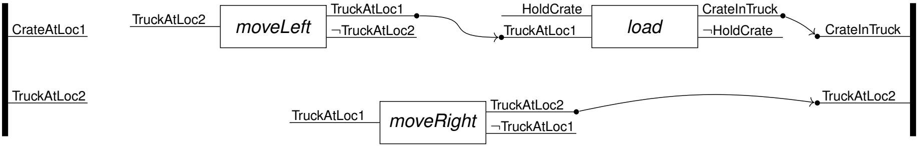


Which refinements exist to fix these flaws?

- Open preconditions:
  - Insert causal links (re-using actions).
  - Insert new actions plus causal links.
- Causal threats:
  - Insert ordering constraints.

Plan Refinements

Let a search node contain the following partial plan:



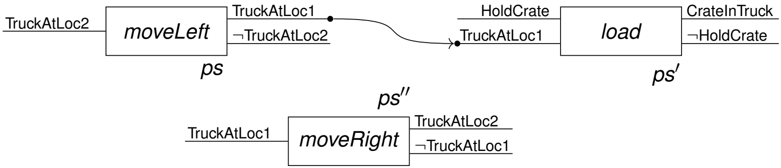
Which refinements exist to fix these flaws?

- Open preconditions:
  - Insert causal links (re-using actions).
  - Insert new actions plus causal links.
- Causal threats:
  - Insert ordering constraints.

→ POCL planning refines search nodes in a *flaw*-directed way: First pick a flaw, then apply all possible modifications.

Resolving Causal Threats

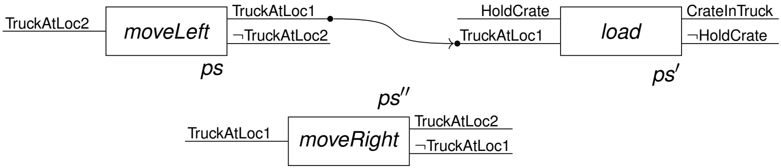
Let $(PS, \prec, CL)$ be a partial plan, $ps, ps' \in PS$ plan steps, and $ps \xrightarrow{\text{TruckAtLoc1}} ps'$ the causal link threatened by $ps'' \in PS$.



Which plan refinements resolve that causal threat?

Resolving Causal Threats

Let $(PS, \prec, CL)$ be a partial plan, $ps, ps' \in PS$ plan steps, and $ps \xrightarrow{\text{TruckAtLoc1}} ps'$ the causal link threatened by $ps'' \in PS$.

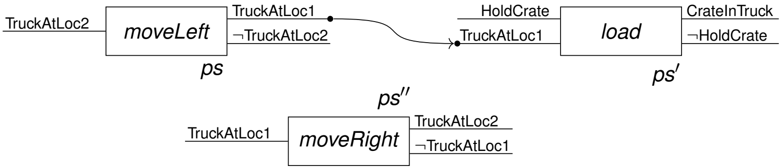


Which plan refinements resolve that causal threat?

- Promotion: order ps'' step *before ps*

Resolving Causal Threats

Let $(PS, \prec, CL)$ be a partial plan, $ps, ps' \in PS$ plan steps, and $ps \xrightarrow{\text{TruckAtLoc1}} ps'$ the causal link threatened by $ps'' \in PS$.

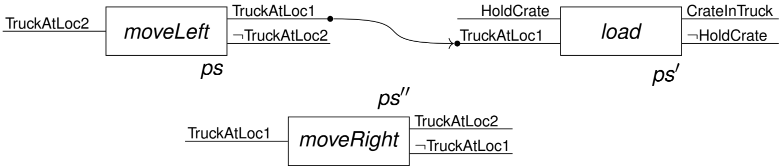


Which plan refinements resolve that causal threat?

- Promotion: order ps'' step *before ps*
- Demotion: order ps'' *behind* ps'

Resolving Causal Threats

Let $(PS, \prec, CL)$ be a partial plan, $ps, ps' \in PS$ plan steps, and $ps \xrightarrow{\text{TruckAtLoc1}} ps'$ the causal link threatened by $ps'' \in PS$.



Which plan refinements resolve that causal threat?

- Promotion: order ps'' step *before* ps
- Demotion: order ps'' *behind* ps'

Note:

In case of lifting, we also get another refinement.

Algorithm, Pseudocode

Algorithm: POCL Planning

Input: A POCL planning problem $\langle V, A, P_I \rangle$
Output: A solution plan P or ***fail*** if none exists

$fringe = \{P_I\}$
while $fringe \neq \emptyset$ **do**
  $P := nodeSelectAndRemove(fringe)$
  $F := flawDetection(P)$
  if $F = \emptyset$ **then return** P
  $f := flawSelection(F)$
  $fringe := \{applyModification(m, f) \mid m$ is a modification for f in $P\}$
return *fail*

Note:
POCL planning was originally an alternative algorithm for classical problems, i.e., no initial partial plan was given.

Algorithm, Choice Points

This algorithm has two choice points:

Algorithm, Choice Points

This algorithm has two choice points:

- *Node selection:*

Algorithm, Choice Points

This algorithm has two choice points:

- *Node selection:*
  - This is a *backtrack point*, i.e., the choice can be wrong. We need to consider *all* possibilities.

Algorithm, Choice Points

This algorithm has two choice points:

- *Node selection:*
  - This is a *backtrack point*, i.e., the choice can be wrong. We need to consider *all* possibilities.
  - How to select a node? Using standard search techniques (cf. lecture on search), which may rely on heuristics.

Algorithm, Choice Points

This algorithm has two choice points:

- *Node selection:*
  - This is a *backtrack point*, i.e., the choice can be wrong. We need to consider *all* possibilities.
  - How to select a node? Using standard search techniques (cf. lecture on search), which may rely on heuristics.
- *Flaw selection:*

Algorithm, Choice Points

This algorithm has two choice points:

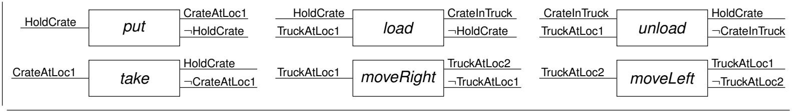
- *Node selection:*
  - This is a *backtrack point*, i.e., the choice can be wrong. We need to consider *all* possibilities.
  - How to select a node? Using standard search techniques (cf. lecture on search), which may rely on heuristics.
- *Flaw selection:*
  - This is *not* a backtrack point, i.e., the choice can *not* be wrong. *Every* flaw needs to be resolved, so the order does not matter.

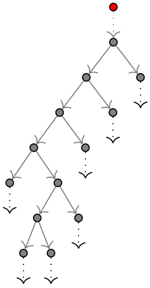
Algorithm, Choice Points

This algorithm has two choice points:

- *Node selection:*
  - This is a *backtrack point*, i.e., the choice can be wrong. We need to consider *all* possibilities.
  - How to select a node? Using standard search techniques (cf. lecture on search), which may rely on heuristics.
- *Flaw selection:*
  - This is *not* a backtrack point, i.e., the choice can *not* be wrong. *Every* flaw needs to be resolved, so the order does not matter.
  - How to select a flaw? There are various possibilities, we only cover a few.

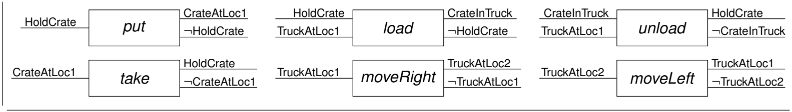
Example

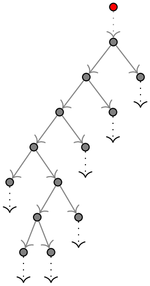




Flaws	**Modifications**
open prec.: CrateInTruck of *goal*	insert *load*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

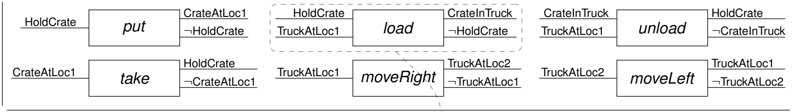
Example

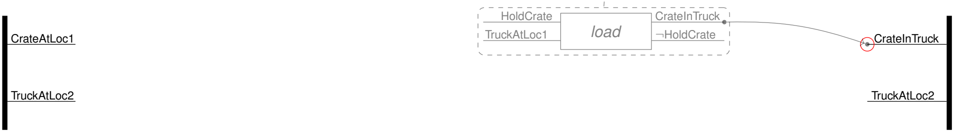


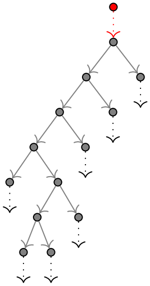


Flaws	Modifications
open prec.:* CrateInTruck of *goal	insert *load*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

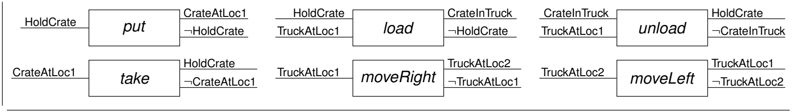


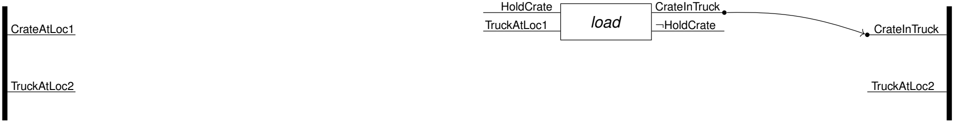


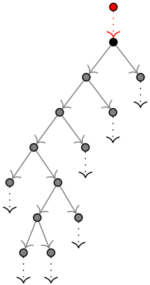


Flaws	Modifications
open prec.:* CrateInTruck of *goal	**insert *load***
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

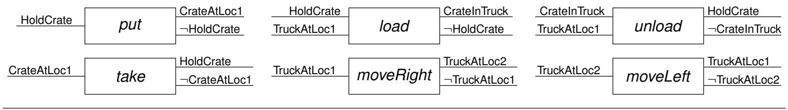


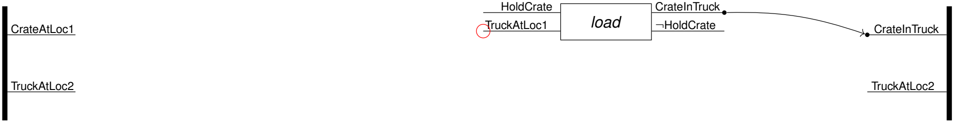


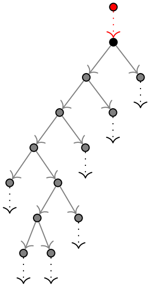


Flaws	Modifications
open prec.: HoldCrate of *load*	insert *take* insert *unload*
open prec.: TruckAtLoc1 of *load*	insert *moveLeft*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

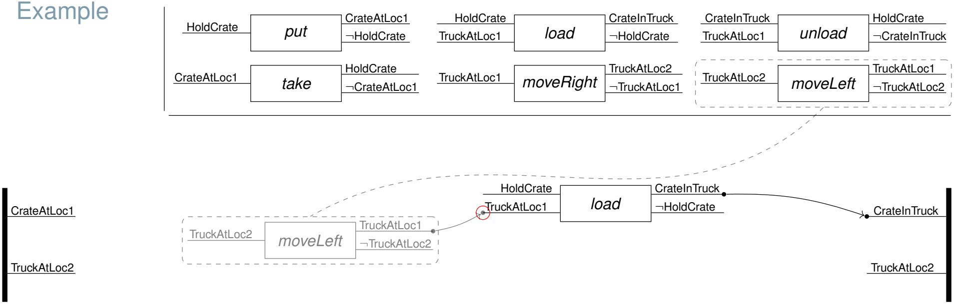


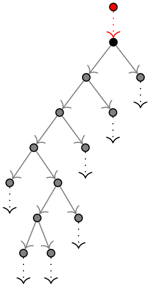




Flaws	Modifications
open prec.: HoldCrate of *load*	insert *take* insert *unload*
open prec.:* TruckAtLoc1 of *load	insert *moveLeft*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

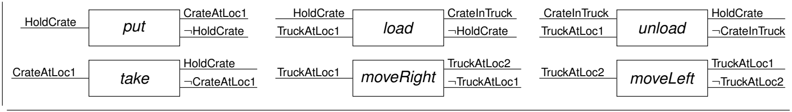
Example

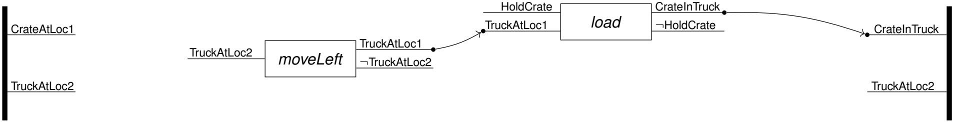


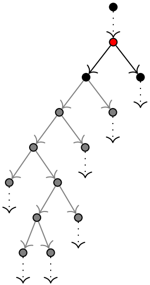


Flaws	Modifications
open prec.: HoldCrate of *load*	insert *take* insert *unload*
open prec.:* TruckAtLoc1 of *load	**insert *moveLeft***
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

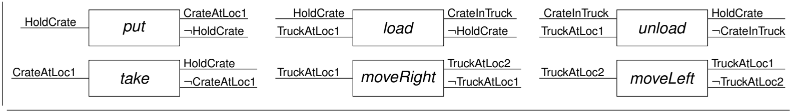


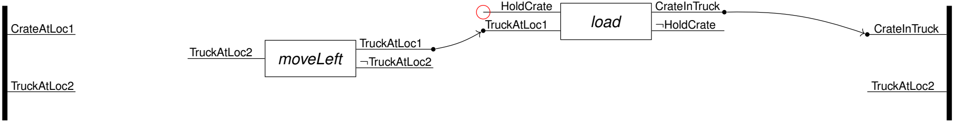


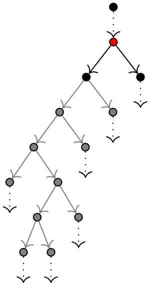


Flaws	Modifications
open prec.: HoldCrate of *load*	insert *take* insert *unload*
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

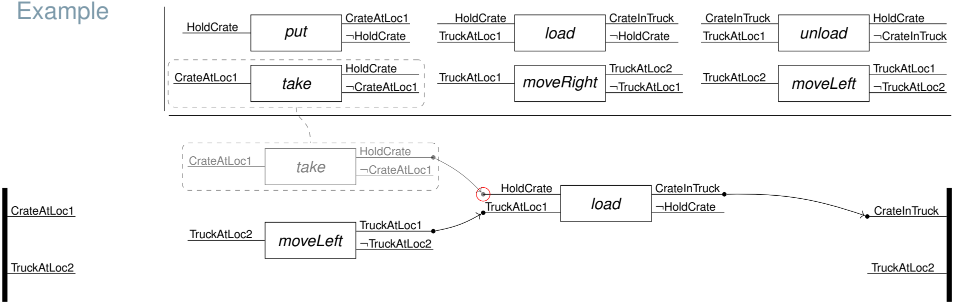


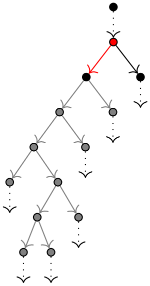




Flaws	**Modifications**
open prec.:* HoldCrate of *load	insert *take* insert *unload*
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

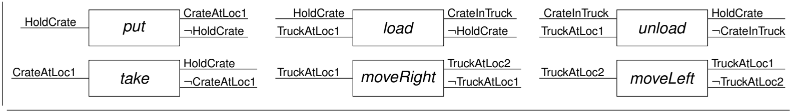
Example

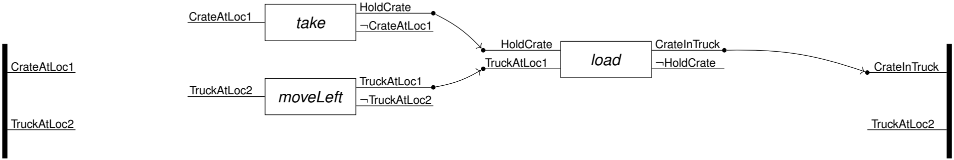


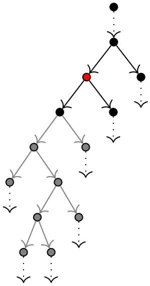


Flaws	Modifications
open prec.:* HoldCrate of *load	**insert *take*** insert *unload*
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

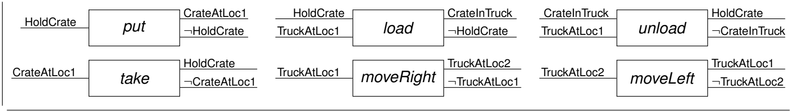


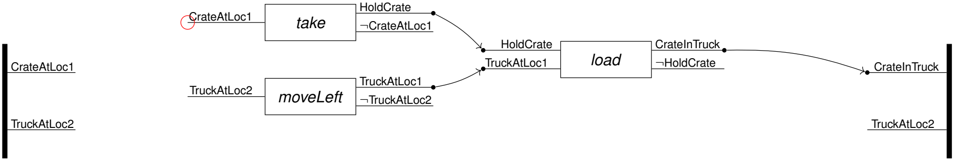


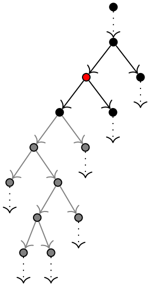


Flaws	**Modifications**
open prec.: CrateAtLoc1 of *take*	insert causal link from *init* insert *put*
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

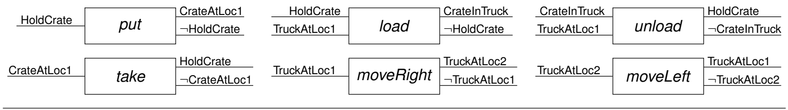


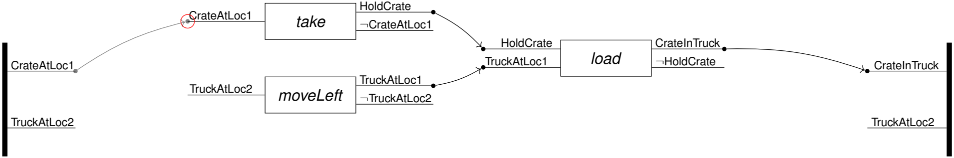


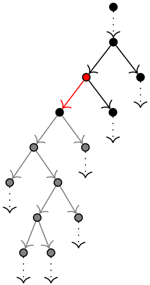


Flaws	**Modifications**
open prec.:* CrateAtLoc1 of *take	insert causal link from *init* insert *put*
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

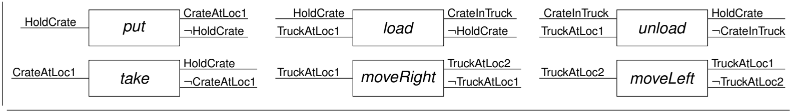


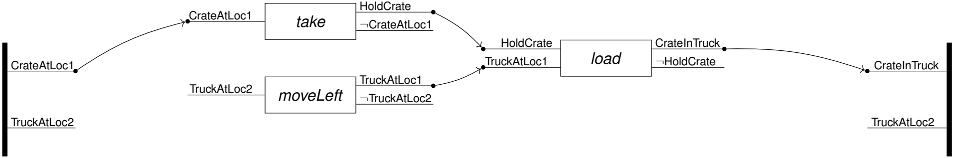


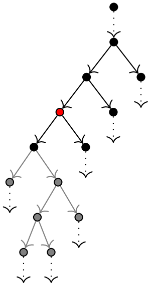


Flaws	Modifications
open prec.:* CrateAtLoc1 of *take	**insert causal link from *init*** insert *put*
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

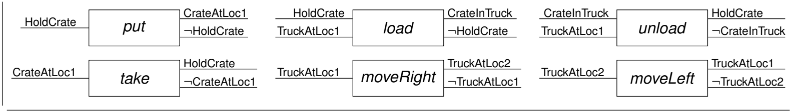


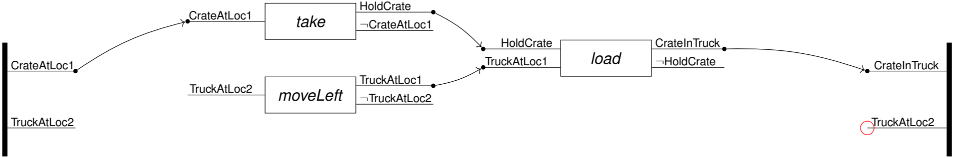


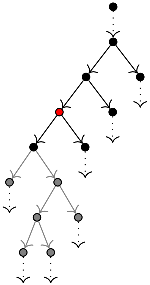


Flaws	**Modifications**
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc2 of *goal*	insert *moveRight* insert causal link from *init*

Example

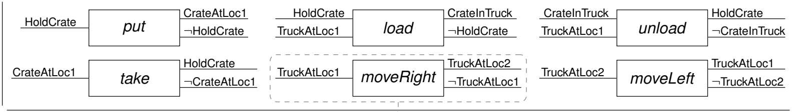


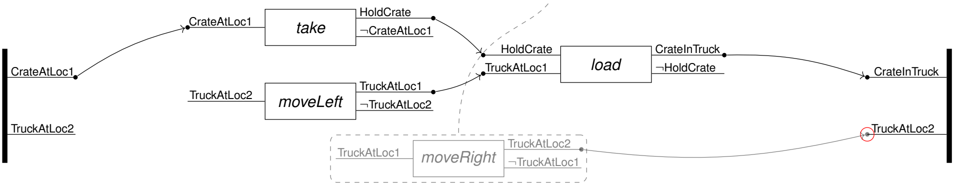


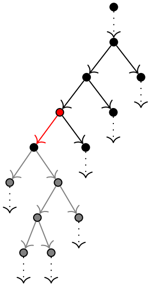


Flaws	Modifications
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.:* TruckAtLoc2 of *goal	insert *moveRight* insert causal link from *init*

Example

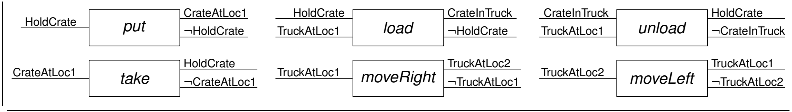


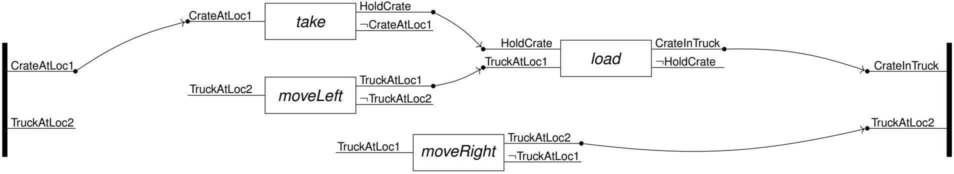


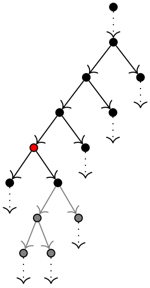


Flaws	**Modifications**
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.:* TruckAtLoc2 of *goal	**insert *moveRight*** insert causal link from *init*

Example

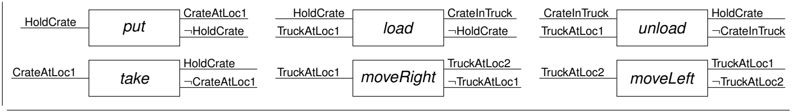


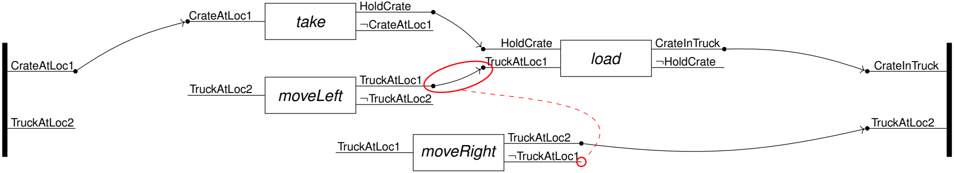


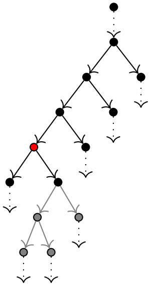


Flaws	**Modifications**
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight* insert causal link from *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*
causal threat: moveLeft $\xrightarrow{\text{TruckAtLoc1}}$ *load* by *moveRight*	promote *moveRight* before *moveLeft* demote *moveRight* after *load*
causal threat: moveRight $\xrightarrow{\text{TruckAtLoc2}}$ *goal* by *moveLeft*	promote *moveLeft* before *moveRight*

Example

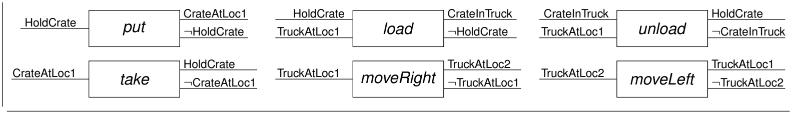


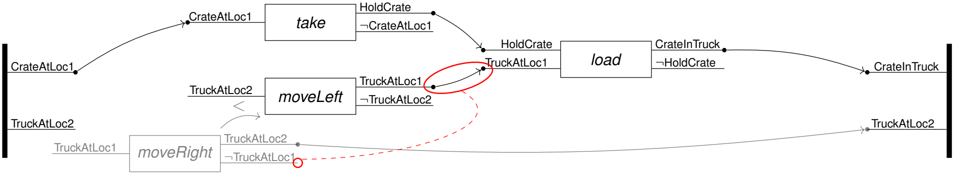


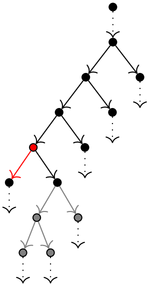


Flaws	**Modifications**
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight* insert causal link from *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*
causal threat:* moveLeft $\xrightarrow{\text{TruckAtLoc1}}$ load by *moveRight	promote *moveRight* before *moveLeft* demote *moveRight* after *load*
causal threat: moveRight $\xrightarrow{\text{TruckAtLoc2}}$ *goal* by *moveLeft*	promote *moveLeft* before *moveRight*

Example

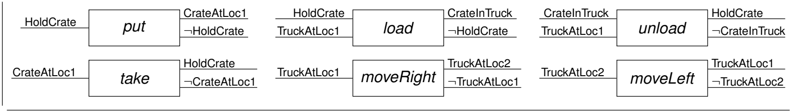


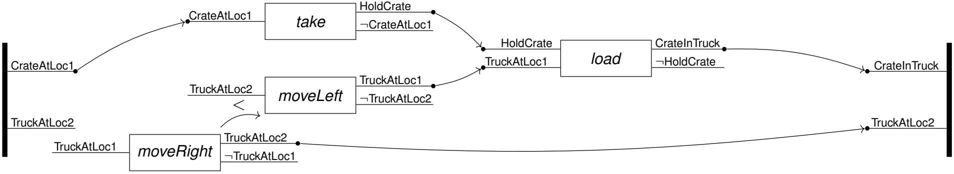


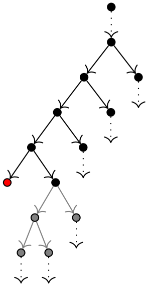


Flaws	Modifications
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight* insert causal link from *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*
causal threat:* moveLeft $\xrightarrow{\text{TruckAtLoc1}}$ load by *moveRight	**promote *moveRight* before *moveLeft*** demote *moveRight* after *load*
causal threat: moveRight $\xrightarrow{\text{TruckAtLoc2}}$ *goal* by *moveLeft*	promote *moveLeft* before *moveRight*

Example

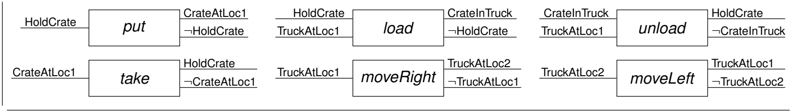


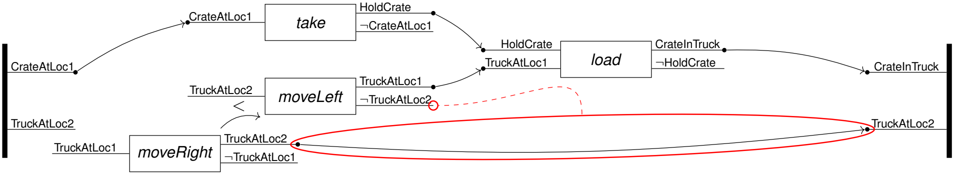


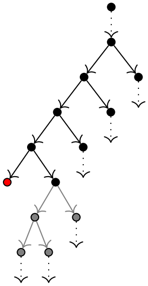


Flaws	**Modifications**
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight* insert causal link from *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*
causal threat: moveRight $\xrightarrow{\text{TruckAtLoc2}}$ *goal* by *moveLeft*	—

Example



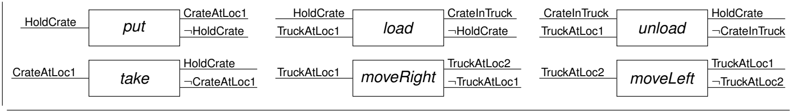


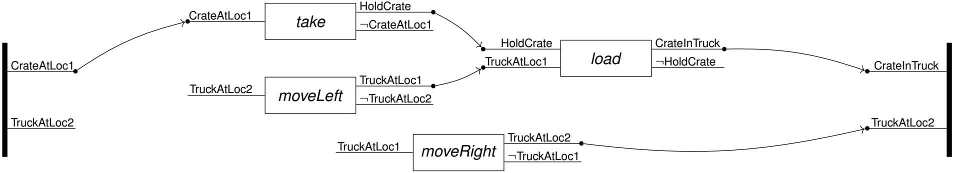


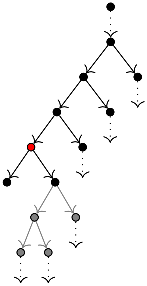
Flaws	Modifications
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight* insert causal link from *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*
causal threat:* moveRight $\xrightarrow{\text{TruckAtLoc2}}$ goal by *moveLeft	—

Discard partial plan due to unresolvable flaw!

Example

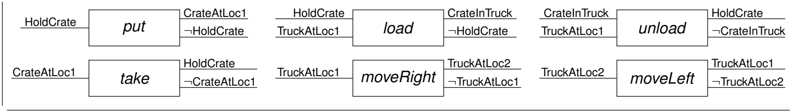


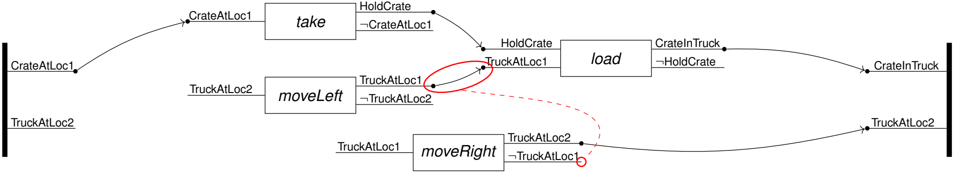


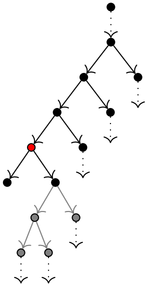


Flaws	Modifications
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight* insert causal link from *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*
causal threat: moveLeft $\xrightarrow{\text{TruckAtLoc1}}$ *load* by *moveRight*	promote *moveRight* before *moveLeft* demote *moveRight* after *load*
causal threat: moveRight $\xrightarrow{\text{TruckAtLoc2}}$ *goal* by *moveLeft*	promote *moveLeft* before *moveRight*

Example

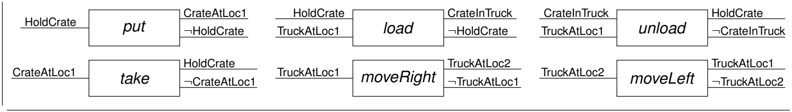


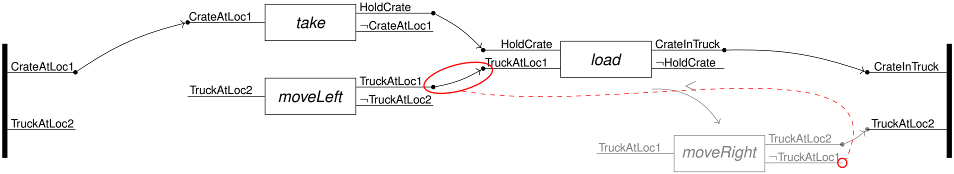


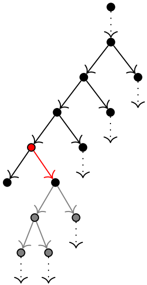


Flaws	Modifications
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight* insert causal link from *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*
causal threat:* moveLeft** $\xrightarrow{\text{TruckAtLoc1}}$ **load by *moveRight	promote *moveRight* before *moveLeft* demote *moveRight* after *load*
causal threat: moveRight $\xrightarrow{\text{TruckAtLoc2}}$ *goal* by *moveLeft*	promote *moveLeft* before *moveRight*

Example

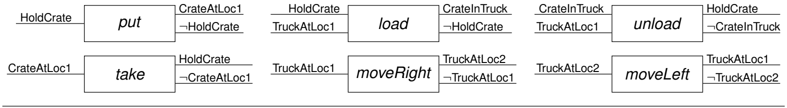


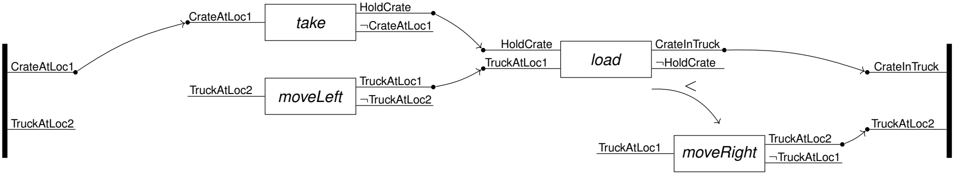


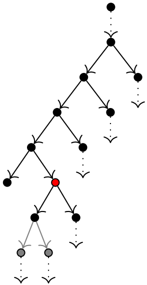


Flaws	Modifications
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight* insert causal link from *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*
causal threat:* moveLeft $\xrightarrow{\text{TruckAtLoc1}}$ load by *moveRight	promote *moveRight* before *moveLeft* **demote *moveRight* after *load***
causal threat: moveRight $\xrightarrow{\text{TruckAtLoc2}}$ *goal* by *moveLeft*	promote *moveLeft* before *moveRight*

Example

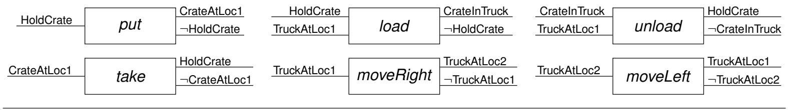


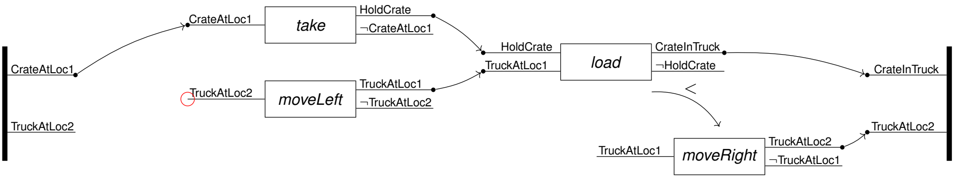


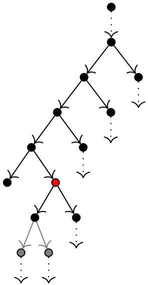


Flaws	**Modifications**
open prec.: TruckAtLoc2 of *moveLeft*	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*

Example

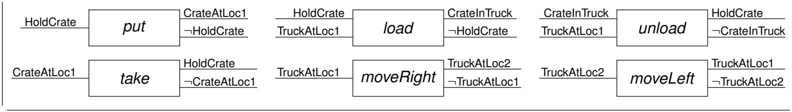


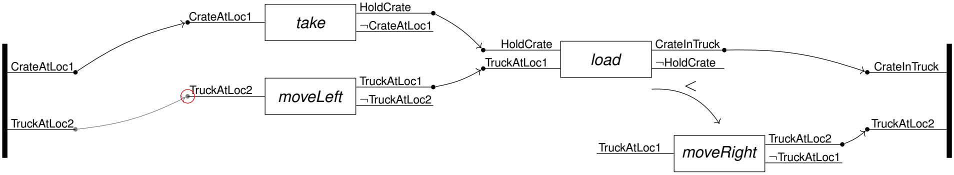


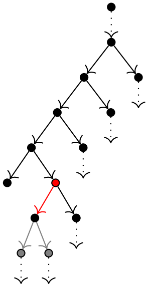


Flaws	**Modifications**
open prec.:* TruckAtLoc2 of *moveLeft	insert causal link from *init* insert *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*

Example

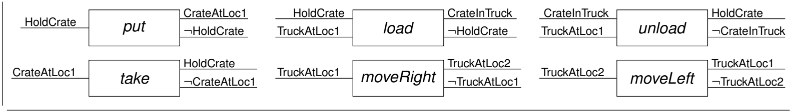


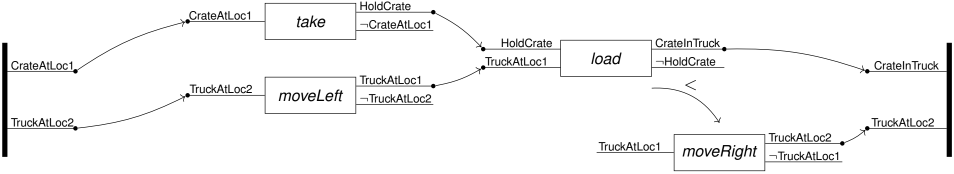


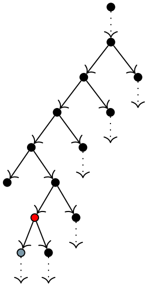


Flaws	**Modifications**
open prec.:* TruckAtLoc2 of *moveLeft	**insert causal link from *init*** insert *moveRight*
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*

Example

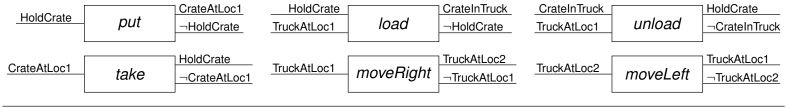


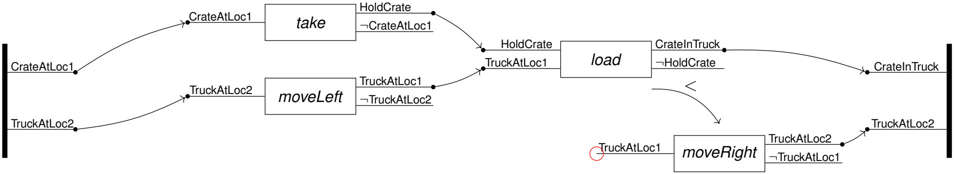


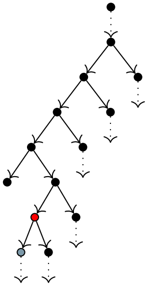


Flaws	**Modifications**
open prec.: TruckAtLoc1 of *moveRight*	insert causal link from *moveLeft* insert *moveLeft*

Example

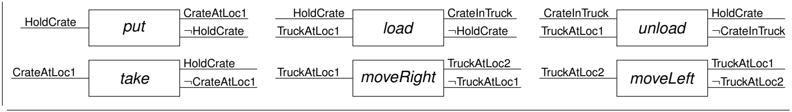


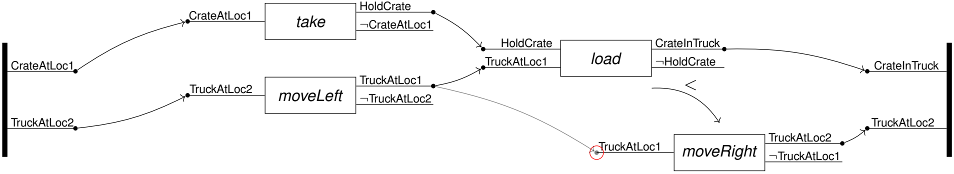


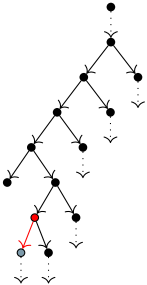


Flaws	**Modifications**
open prec.:* TruckAtLoc1 of *moveRight	insert causal link from *moveLeft* insert *moveLeft*

Example

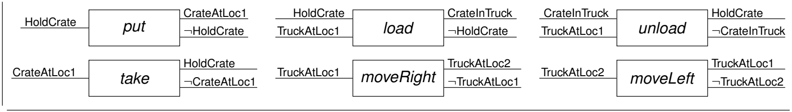


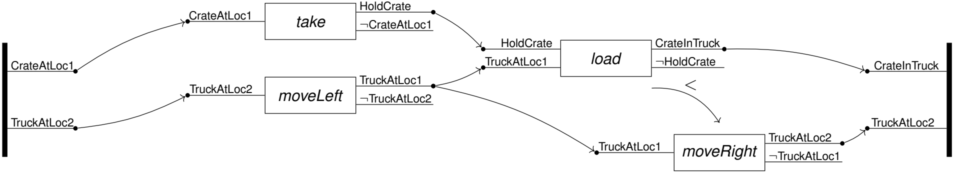


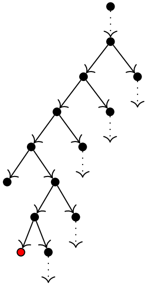


Flaws	**Modifications**
open prec.:* TruckAtLoc1 of *moveRight	**insert causal link from *moveLeft***
	insert *moveLeft*

Example







Flaws	Modifications

Since there is no flaw: Return solution plan!

Flaw Selection Strategies

Which flaw to select?

- *For completeness:*
- *For efficiency:*

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:*

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Causal Threats First (CTF):*
  - Always select a causal threat flaw.

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Causal Threats First (CTF):*
  - Always select a causal threat flaw.
  - → Gives a preference to causal threats: Only deal with link or action insertions after the partial plan has no “internal” issues.

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Causal Threats First (CTF):*
  - Always select a causal threat flaw.
  - → Gives a preference to causal threats: Only deal with link or action insertions after the partial plan has no "internal" issues.
  - → This strategy was part of the well-known POP algorithm by Russell and Norvig's text book *Artificial Intelligence – A Modern Approach* and of the well-known POCL planners SNLP and UCPOP. Here, the *algorithm* resolved all threats before any other flaw (open condition) was selected.

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Least-Cost Flaw-Repair (LCFR):*
  - Always select a flaw that this “cheap” to repair, i.e., for which there are the fewest modifications.

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Least-Cost Flaw-Repair (LCFR):*
  - Always select a flaw that this “cheap” to repair, i.e., for which there are the fewest modifications.

  → This strategy *locally* minimizes the branching factor of the search space.

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Least-Cost Flaw-Repair (LCFR):*
  - Always select a flaw that this “cheap” to repair, i.e., for which there are the fewest modifications.
  - → This strategy *locally* minimizes the branching factor of the search space.
  - → Nice special case: Fix flaws with just one modification! (This choice can *never* be wrong!)

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Left-Most Open Condition First (LMOCF):*
  - Always select a precondition that is closest to the initial state.

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Left-Most Open Condition First (LMOCF):*
  - Always select a precondition that is closest to the initial state.
  - → This strategy first creates one long chain of actions that is rooted in the initial state, then completes it starting from left to right.

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

- *Left-Most Open Condition First (LMOCF):*
  - Always select a precondition that is closest to the initial state.
  - → This strategy first creates one long chain of actions that is rooted in the initial state, then completes it starting from left to right.
  - → Search nodes have only one linearization until the chain finally roots in the initial state.

Flaw Selection Strategies

Which flaw to select?

- *For completeness:* Does not matter!
- *For efficiency:* Strategy as huge impact!

Some flaw selection strategies:

Flaw selection strategies can be combined/concatenated!

For instance, $\langle CTF, LMOCF, LCFR \rangle$ will:

- First eliminate all causal threats,
- among all non-threat flaws select a left-most open condition,
- and among them some flaw with the fewest modifications.

Flaw Selection Strategies, Literature

There are *many* flaw selection strategies known to the literature. Some pointers:

- Håkan L. S. Younes and Reid G. Simmons. "VHPOP: Versatile heuristic partial order planner". In: *Journal of Artificial Intelligence Research (JAIR)* 20 (2003), pp. 405–430
- Martha E. Pollack et al. "Flaw Selection Strategies For Partial-Order Planning". In: *Journal of Artificial Intelligence Research (JAIR)* 6 (1997), pp. 223–262
- Mike Williamson and Steve Hanks. "Flaw Selection Strategies for Value-Directed Planning". In: *Proc. of the 3rd Int. Conf. on Artificial Intelligence Planning Systems (AIPS 1996)*. AAAI Press, 1996, pp. 237–244

Properties

Theorem

POCL Planning is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*()).

Further, POCL planning does not provide the strongest form of completeness. Why?

Properties

Theorem

POCL Planning is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*()).

Further, POCL planning does not provide the strongest form of completeness. Why?

Proof:
Follows from:

- The properties of the underlying search algorithm.

Properties

Theorem

POCL Planning is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*()).

Further, POCL planning does not provide the strongest form of completeness. Why?

Proof:
Follows from:

- The properties of the underlying search algorithm.
- The fact that for each flaw *all* modifications that could possibly resolve that flaw are branched into the search space.

Properties

Theorem

POCL Planning is sound and complete.

The completeness, however, depends on the deployed search strategy, i.e., the implementation of *nodeSelectAndRemove*()).

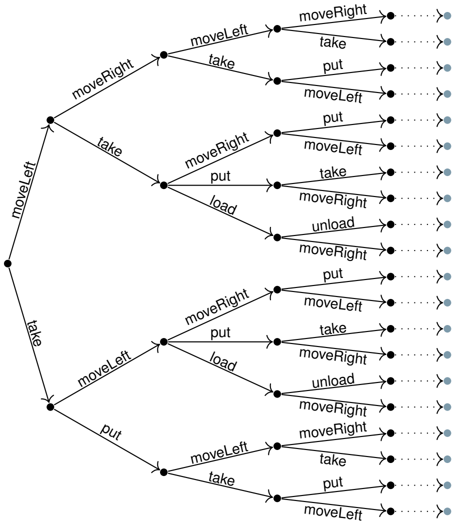
Further, POCL planning does not provide the strongest form of completeness. Why?

Proof:

Follows from:

- The properties of the underlying search algorithm.
- The fact that for each flaw *all* modifications that could possibly resolve that flaw are branched into the search space.
- The strongest form of completeness does not hold, since only causally relevant actions can be added in POCL planning.

Reminder: Search-Guidance in Classical Planning



Main issue in classical planning:

High branching factor, which usually allows many actions to be applicable in the current state – resulting in a large search fringes. (Dealt with by heuristics).

Search-Guidance in POCL Planning

For mainly two reasons the branching factor in POCL planning is usually very small:

Search-Guidance in POCL Planning

For mainly two reasons the branching factor in POCL planning is usually very small:

- Due to the regression-like search procedure and the fact that only causally relevant actions are selected.

Search-Guidance in POCL Planning

For mainly two reasons the branching factor in POCL planning is usually very small:

- Due to the regression-like search procedure and the fact that only causally relevant actions are selected.
- The explicit *flaw selection step* allows to select flaws which produce a small branching factor.

Search-Guidance in POCL Planning

For mainly two reasons the branching factor in POCL planning is usually very small:

- Due to the regression-like search procedure and the fact that only causally relevant actions are selected.
- The explicit *flaw selection step* allows to select flaws which produce a small branching factor.

Problems with the Search-Guidance:

- Despite smaller branching factor, we still need to decide on which partial plans to work next $\rightarrow$ use heuristics!

Search-Guidance in POCL Planning

For mainly two reasons the branching factor in POCL planning is usually very small:

- Due to the regression-like search procedure and the fact that only causally relevant actions are selected.
- The explicit *flaw selection step* allows to select flaws which produce a small branching factor.

Problems with the Search-Guidance:

- Despite smaller branching factor, we still need to decide on which partial plans to work next $\rightarrow$ use heuristics!
- Heuristic design is more complicated here, because there are more constraints to respect (the partial plan rather than just a state).

Introduction

- POCL planning is often referred to as *refinement planning*.
- Informally, *refinement* in the context of POCL planning means that a partially developed plan gets more specialized via adding constraints (such as causal links, actions, ordering constraints).

Introduction

- POCL planning is often referred to as *refinement planning*.
- Informally, *refinement* in the context of POCL planning means that a partially developed plan gets more specialized via adding constraints (such as causal links, actions, ordering constraints).
- More generally, *refinement search* is a theoretical concept, where each search node is interpreted as the set of solution candidates that it induces, i.e., that can be reached from it.

  Example: In POCL planning that's the set of totally ordered action sequences that can be derived from the current partial plan.

Introduction

- POCL planning is often referred to as *refinement planning*.
- Informally, *refinement* in the context of POCL planning means that a partially developed plan gets more specialized via adding constraints (such as causal links, actions, ordering constraints).
- More generally, *refinement search* is a theoretical concept, where each search node is interpreted as the set of solution candidates that it induces, i.e., that can be reached from it.

  Example: In POCL planning that's the set of totally ordered action sequences that can be derived from the current partial plan.
- *Refinement operators* (the modifications) restrict these sets.

Introduction

- POCL planning is often referred to as *refinement planning*.
- Informally, *refinement* in the context of POCL planning means that a partially developed plan gets more specialized via adding constraints (such as causal links, actions, ordering constraints).
- More generally, *refinement search* is a theoretical concept, where each search node is interpreted as the set of solution candidates that it induces, i.e., that can be reached from it.

  Example: In POCL planning that's the set of totally ordered action sequences that can be derived from the current partial plan.
- *Refinement operators* (the modifications) restrict these sets.
- It allows to compare different planning algorithms and to define certain properties.

Formal Definitions

- Let n be search node. Then $\langle\langle n \rangle\rangle$, the *candidates set* is the set of action sequences that can be derived from n via the available refinement operators.

Formal Definitions

- Let n be search node. Then $\langle\langle n \rangle\rangle$, the *candidates set* is the set of action sequences that can be derived from n via the available refinement operators.
- A *refinement operator R* generates, for a search node n, a set of successor nodes $n_1, \ldots, n_m$, such that all resulting candidate sets are proper subsets of the parent search node.
  That is: for all $1 \leq i \leq m$ holds $\langle\langle n_i \rangle\rangle \subseteq \langle\langle n \rangle\rangle$

Formal Definitions

- Let n be search node. Then $\langle\langle n \rangle\rangle$, the *candidates set* is the set of action sequences that can be derived from n via the available refinement operators.
- A *refinement operator R* generates, for a search node n, a set of successor nodes $n_1, \ldots, n_m$, such that all resulting candidate sets are proper subsets of the parent search node.
  That is: for all $1 \leq i \leq m$ holds $\langle\langle n_i \rangle\rangle \subseteq \langle\langle n \rangle\rangle$

  Example In POCL planning "Resolve causal threat t" would be a refinement operator with n_1 being the search node resulting from promotion and n_2 the one resulting from demotion.

Formal Definitions

- Let n be search node. Then $\langle\langle n \rangle\rangle$, the *candidates set* is the set of action sequences that can be derived from n via the available refinement operators.
- A *refinement operator R* generates, for a search node n, a set of successor nodes $n_1, \ldots, n_m$, such that all resulting candidate sets are proper subsets of the parent search node.
  That is: for all $1 \leq i \leq m$ holds $\langle\langle n_i \rangle\rangle \subseteq \langle\langle n \rangle\rangle$

  Example In POCL planning "Resolve causal threat t" would be a refinement operator with n_1 being the search node resulting from promotion and n_2 the one resulting from demotion.
- A refinement operator R is called *complete* if every solution in $\langle\langle n \rangle\rangle$ is contained in at least one of its children candidate sets $\langle\langle n_i \rangle\rangle$.

Formal Definitions

- Let n be search node. Then $\langle\langle n \rangle\rangle$, the *candidates set* is the set of action sequences that can be derived from n via the available refinement operators.
- A *refinement operator R* generates, for a search node n, a set of successor nodes $n_1, \ldots, n_m$, such that all resulting candidate sets are proper subsets of the parent search node.
  That is: for all $1 \leq i \leq m$ holds $\langle\langle n_i \rangle\rangle \subseteq \langle\langle n \rangle\rangle$

  Example In POCL planning "Resolve causal threat t" would be a refinement operator with n_1 being the search node resulting from promotion and n_2 the one resulting from demotion.
- A refinement operator R is called *complete* if every solution in $\langle\langle n \rangle\rangle$ is contained in at least one of its children candidate sets $\langle\langle n_i \rangle\rangle$.
- A refinement operator R is called *systematic* if for all $i \neq j$ holds $\langle\langle n_i \rangle\rangle \cap \langle\langle n_j \rangle\rangle = \emptyset$.

Formal Definitions, cont'd

The concept of *planning as refinement search* was formally introduced by:

- Subbarao Kambhampati et al. "Planning as Refinement Search: A Unified Framework for Evaluating Design Tradeoffs in Partial-Order Planning". In: *Artificial Intelligence* 76.1-2 (1995), pp. 167–238
- Subbarao Kambhampati. "Refinement Planning as a Unifying Framework for Plan Synthesis". In: *AI Magazine* 18.2 (1997), pp. 67–98

(The definitions provided here base upon the '95 article.)

Systematicity in Refinement Planning

Systematicity:

Systematicity in Refinement Planning

Systematicity:

- Informally, *systematic search* means that each plan is found at most once (no redundancy).

Systematicity in Refinement Planning

Systematicity:

- Informally, *systematic search* means that each plan is found at most once (no redundancy).
- Formally, a search algorithm is called *systematic* if all refinement operators are systematic.

Systematicity in Refinement Planning

Systematicity:

- Informally, *systematic search* means that each plan is found at most once (no redundancy).
- Formally, a search algorithm is called *systematic* if all refinement operators are systematic.
- Alternative definition: A search algorithm is called *systematic* if for all search nodes n and n' in different branches of the search tree $\langle\langle n \rangle\rangle \cap \langle\langle n' \rangle\rangle = \emptyset$ holds.

Systematicity in Refinement Planning

Systematicity:

- Informally, *systematic search* means that each plan is found at most once (no redundancy).
- Formally, a search algorithm is called *systematic* if all refinement operators are systematic.
- Alternative definition: A search algorithm is called *systematic* if for all search nodes n and n' in different branches of the search tree $\langle\langle n \rangle\rangle \cap \langle\langle n' \rangle\rangle = \emptyset$ holds.
- Further reading: Subbarao Kambhampati. "On the Utility of Systematicity: Understanding Tradeoffs between Redundancy and Commitment in Partial-Order Planning". In: *Proc. of the 13th Int. Joint Conf. on Artificial Intelligence (IJCAI 1993)*. Morgan Kaufmann, 1993, pp. 1380–1385

Example

Is POCL planning systematic?

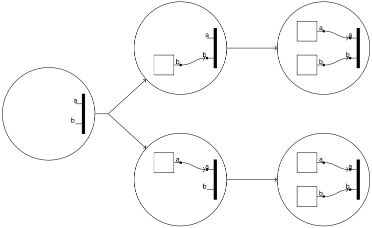
Example

Is POCL planning systematic? No!

Consider a planning problem with $g = \{a, b\}$ and two actions:

$$A = (\emptyset, \{a\}, \emptyset) \quad B = (\emptyset, \{b\}, \emptyset)$$

The following search space proves that it's not systematic:



... right?

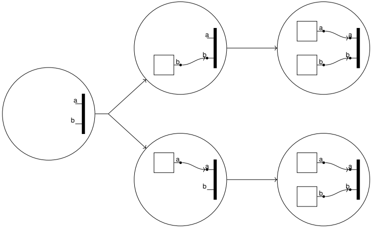
Example

Is POCL planning systematic? No!

Consider a planning problem with $g = \{a, b\}$ and two actions:

$$A = (\emptyset, \{a\}, \emptyset) \quad B = (\emptyset, \{b\}, \emptyset)$$

The following search space proves that it's not systematic:



... right? No! The above tree is *not* a (single) POCL search tree: The flaw selection is missing.

A Correct Example

POCL planning ist not systematic.

Consider a planning problem with $g = \{a, b\}$ and three actions:

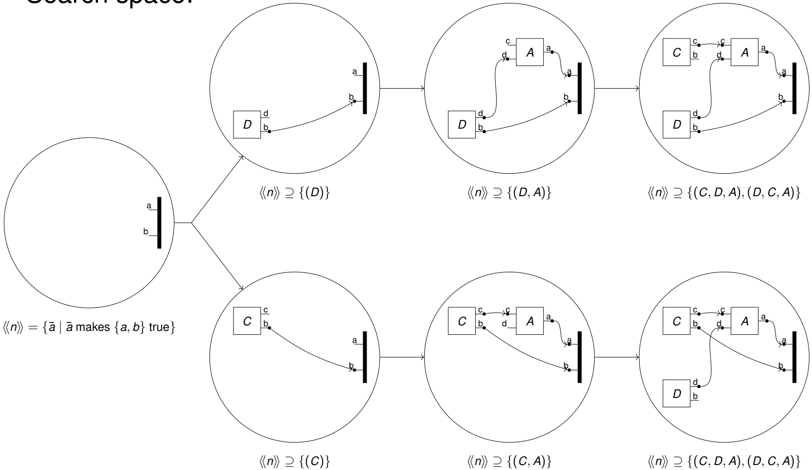
- $A = (\{c, d\}, \{a\}, \emptyset)$
- $CB = (\emptyset, \{c, b\}, \emptyset)$
- $DB = (\emptyset, \{d, b\}, \emptyset)$

With first resolving the goal precondition *b*, the same set of action sequences (*DB*, *CB*, *A* and *CB*, *DB*, *A*) can be derived in two separate branches.

A Correct Example, cont'd

POCL planning is not systematic.

Search space:



Positive Causal Threats

We now extend the set of flaws by a *positive causal threat*.

Definition (Positive Causal Threat)

Let $(PS, \prec, CL)$ be a partial plan. A *positive causal threat* consists of the plan steps $ps, ps' \in PS$, a causal link $ps \xrightarrow{v} ps'$, and the *threatening plan step* $ps'' \in PS$ if and only if

Positive Causal Threats

We now extend the set of flaws by a *positive causal threat*.

Definition (Positive Causal Threat)

Let $(PS, \prec, CL)$ be a partial plan. A *positive causal threat* consists of the plan steps $ps, ps' \in PS$, a causal link $ps \xrightarrow{v} ps'$, and the *threatening plan step* $ps'' \in PS$ if and only if

- $v \in add(ps'')$ (in contrast to $v \in del(ps'')$ for standard threats)

Positive Causal Threats

We now extend the set of flaws by a *positive causal threat*.

Definition (Positive Causal Threat)

Let $(PS, \prec, CL)$ be a partial plan. A *positive causal threat* consists of the plan steps $ps, ps' \in PS$, a causal link $ps \xrightarrow{v} ps'$, and the *threatening plan step* $ps'' \in PS$ if and only if

- $v \in add(ps'')$ (in contrast to $v \in del(ps'')$ for standard threats)
- The ordering constraints allow ps'' to be ordered between ps and ps', i.e., $(\prec \cup \{(ps, ps''), (ps', ps'')\})^*$ is a strict partial order. (* denotes the transitive closure.)

Positive Causal Threats

We now extend the set of flaws by a *positive causal threat*.

Definition (Positive Causal Threat)

Let $(PS, \prec, CL)$ be a partial plan. A *positive causal threat* consists of the plan steps $ps, ps' \in PS$, a causal link $ps \xrightarrow{v} ps'$, and the *threatening plan step* $ps'' \in PS$ if and only if

- $v \in add(ps'')$ (in contrast to $v \in del(ps'')$ for standard threats)
- The ordering constraints allow ps'' to be ordered between ps and ps', i.e., $(\prec \cup \{(ps, ps''), (ps', ps'')\})^*$ is a strict partial order. (* denotes the transitive closure.)

The modifications to resolve this flaw are analogous to standard causal threats.

Positive Causal Threats, cont'd

Theorem

POCL Planning with positive causal threats is systematic.

Proof:

See David McAllester and David Rosenblitt. "Systematic Nonlinear Planning". In: *Proc. of the 9th National Conf. on Artificial Intelligence (AAAI 1991)*. AAAI Press, 1991, pp. 634–639

Positive Causal Threats, cont'd

Theorem

POCL Planning with positive causal threats is systematic.

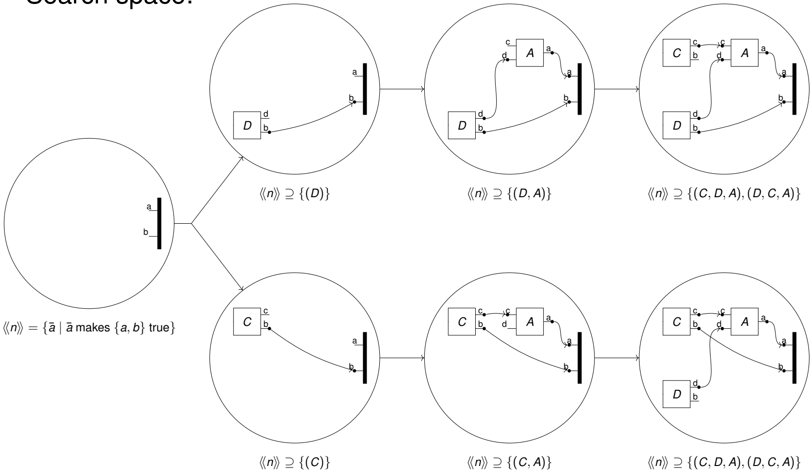
Proof:
See David McAllester and David Rosenblitt. "Systematic Nonlinear Planning". In: *Proc. of the 9th National Conf. on Artificial Intelligence (AAAI 1991)*. AAAI Press, 1991, pp. 634–639

Note that POCL planning is ordinarily done *without* positive causal threats, because it is then usually more efficient (despite being non-systematic).

Influence of Positive Causal Threats, Example

POCL planning is not systematic.

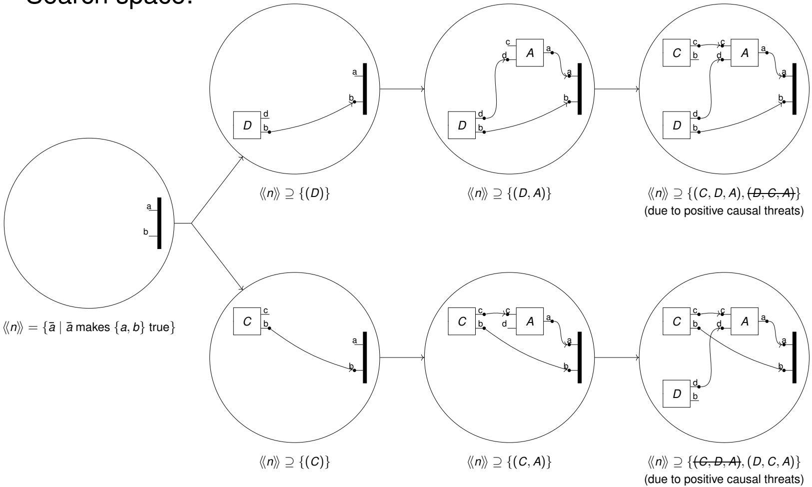
Search space:



Influence of Positive Causal Threats, Example

POCL planning is not systematic.

Search space:



Summary

- Progression (forward) search in the space of states is also often referred to as *classical planning* (note the difference to classical planning *problems*!).

Summary

- Progression (forward) search in the space of states is also often referred to as *classical planning* (note the difference to classical planning *problems*!).
- In addition to many heuristics there exist various techniques to improve its performance (pruning, action selection, symmetry elimination, etc.) – they are not part of this lecture.

Summary

- Progression (forward) search in the space of states is also often referred to as *classical planning* (note the difference to classical planning *problems*!).
- In addition to many heuristics there exist various techniques to improve its performance (pruning, action selection, symmetry elimination, etc.) – they are not part of this lecture.
- Partial-Order Causal-Link (POCL) planning is an alternative approach for solving classical (or POCL) problems.

Summary

- Progression (forward) search in the space of states is also often referred to as *classical planning* (note the difference to classical planning *problems*!).
- In addition to many heuristics there exist various techniques to improve its performance (pruning, action selection, symmetry elimination, etc.) – they are not part of this lecture.
- Partial-Order Causal-Link (POCL) planning is an alternative approach for solving classical (or POCL) problems.
- POCL planning searches in the space of partial plans – in a regression-like fashion.

Summary

- Progression (forward) search in the space of states is also often referred to as *classical planning* (note the difference to classical planning *problems*!).
- In addition to many heuristics there exist various techniques to improve its performance (pruning, action selection, symmetry elimination, etc.) – they are not part of this lecture.
- Partial-Order Causal-Link (POCL) planning is an alternative approach for solving classical (or POCL) problems.
- POCL planning searches in the space of partial plans – in a regression-like fashion.
- In contrast to classical planning, search is a *two-stage* process: In addition to the search node selection, we also select a flaw to work on.

Summary

- Progression (forward) search in the space of states is also often referred to as *classical planning* (note the difference to classical planning *problems*!).
- In addition to many heuristics there exist various techniques to improve its performance (pruning, action selection, symmetry elimination, etc.) – they are not part of this lecture.
- Partial-Order Causal-Link (POCL) planning is an alternative approach for solving classical (or POCL) problems.
- POCL planning searches in the space of partial plans – in a regression-like fashion.
- In contrast to classical planning, search is a *two-stage* process: In addition to the search node selection, we also select a flaw to work on.
- Refinement search is an algorithm-independent concept to be able to compare different algorithms (e.g., their systematicity).

Remarks on the Pros and Cons of Classical vs. POCL Planning

Classical Planning:

- pro: State-based search makes development of heuristics relatively “easy”.

POCL Planning:

- con: Development of heuristic tricky.

Remarks on the Pros and Cons of Classical vs. POCL Planning

Classical Planning:

- pro State-based search makes development of heuristics relatively "easy".
- pro Duplicate check is trivial.

POCL Planning:

- con Development of heuristic tricky.
- pro Search is (nearly) systematic – no duplicate check required.

Remarks on the Pros and Cons of Classical vs. POCL Planning

Classical Planning:

- pro State-based search makes development of heuristics relatively “easy”.
- pro Duplicate check is trivial.
- pro The required search algorithm and data structures are conceptionally very simple.

POCL Planning:

- con Development of heuristic tricky.
- pro Search is (nearly) systematic – no duplicate check required.
- con Implementation of the algorithm and data structures is quite complicated (with many reasoning routines and special cases).

Remarks on the Pros and Cons of Classical vs. POCL Planning

Classical Planning:

- pro State-based search makes development of heuristics relatively "easy".
- pro Duplicate check is trivial.
- pro The required search algorithm and data structures are conceptionally very simple.
- pro Can be implemented to act extremely fast, making it superior if there are not too many totally ordered solutions.

POCL Planning:

- con Development of heuristic tricky.
- pro Search is (nearly) systematic – no duplicate check required.
- con Implementation of the algorithm and data structures is quite complicated (with many reasoning routines and special cases).
- con The reasoning process per node is slower

Remarks on the Pros and Cons of Classical vs. POCL Planning

Classical Planning:

- pro State-based search makes development of heuristics relatively “easy”.
- pro Duplicate check is trivial.
- pro The required search algorithm and data structures are conceptionally very simple.
- pro Can be implemented to act extremely fast, making it superior if there are not too many totally ordered solutions.
- con The branching factor is usually high, because, normally, many actions are applicable to the current state.

POCL Planning:

- con Development of heuristic tricky.
- pro Search is (nearly) systematic – no duplicate check required.
- con Implementation of the algorithm and data structures is quite complicated (with many reasoning routines and special cases).
- con The reasoning process per node is slower
- pro The branching factor is usually very small due to flaw selection.

Remarks on the Pros and Cons of Classical vs. POCL Planning

Classical Planning:

- pro State-based search makes development of heuristics relatively “easy”.
- pro Duplicate check is trivial.
- pro The required search algorithm and data structures are conceptionally very simple.
- pro Can be implemented to act extremely fast, making it superior if there are not too many totally ordered solutions.
- con The branching factor is usually high, because, normally, many actions are applicable to the current state.

POCL Planning:

- con Development of heuristic tricky.
- pro Search is (nearly) systematic – no duplicate check required.
- con Implementation of the algorithm and data structures is quite complicated (with many reasoning routines and special cases).
- con The reasoning process per node is slower
- pro The branching factor is usually very small due to flaw selection.
- con The flaw-based procedure might also increase the search space: For each precondition (i.e., link insertion), a new search node is created. In classical planning, this is just one action application.

Remarks on the Pros and Cons of Classical vs. POCL Planning

Classical Planning:

- pro State-based search makes development of heuristics relatively “easy”.
- pro Duplicate check is trivial.
- pro The required search algorithm and data structures are conceptionally very simple.
- pro Can be implemented to act extremely fast, making it superior if there are not too many totally ordered solutions.
- con The branching factor is usually high, because, normally, many actions are applicable to the current state.

POCL Planning:

- con Development of heuristic tricky.
- pro Search is (nearly) systematic – no duplicate check required.
- con Implementation of the algorithm and data structures is quite complicated (with many reasoning routines and special cases).
- con The reasoning process per node is slower
- pro The branching factor is usually very small due to flaw selection.
- con The flaw-based procedure might also increase the search space: For each precondition (i.e., link insertion), a new search node is created. In classical planning, this is just one action application.
- pro Search nodes can represent an exponential number of classical plans (in one node), making the required search space potentially much smaller.

Copyright Notes and Licenses