Course "Automated Planning: Theory and Practice"
Chapter 15: Pattern Databases

Teacher: Marco Roveri - marco.roveri@unitn.it

M.S. Course: Artificial Intelligence Systems (LM)
A.A.: 2025-2026
Where: DISI, University of Trento
URL: https://shorturl.at/A81hf

Last updated: Monday 10th November, 2025

Terms of Use and Copyright

The material (text, figures) in these slides is authored by Jonas Kvarnström and Marco Roveri.

Pattern Databases: Introduction

- First a main idea behind pattern databases:
  - Let's care about few facts and ignore all others - everywhere
    - Goals, preconditions, effects, states

A form of relaxation...

With some special features!

©https://w-eu.ohmygoodness.com/reserved/cards/202004280553460.monkeys-dont-omg.jpg

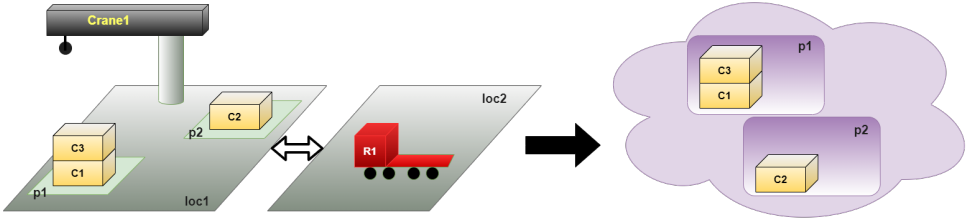
Pattern Databases: Example

- Dock Worker Robots
  - Could choose to care about container locations
    - `(in cont pile)`, `(top container pile)`, `(on container container)`, ...
  - Ignore robot locations, crane locations, location adjacency, ...

Odinary state in problem P,
All facts included!

States are "grouped together"

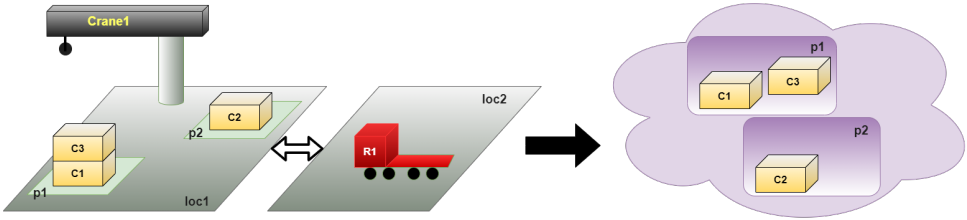
Abstract state in P', representing many states in P with different robot locations, ...



Pattern Databases: Planning in Patterns

- In P' we (pretend that we) can use `Crane1` at `p1` to:
  - pick up `c3` (should be possible)
  - place something on `R1` (too far away, but we don't care)
  - place five containers on a single truck
- But we can't:
  - pick up `c1` (we do care about pile ordering)
  - immediately place `c1` below `c2`, ...

New paths
to the goal!



Achievable States

- Consider the physically achievable states in the BW4:

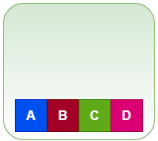
Gound Atoms

- All ground atoms (facts) in this problem instance:

```
(on A A)      (on A B)      (on A C)      (on A D)
(on B A)      (on B B)      (on B C)      (on B D)
(on C A)      (on C B)      (on C C)      (on C D)
(on D A)      (on D B)      (on D C)      (on D D)
(ontable A)   (ontable B)   (ontable C)   (ontable D)
(clear A)     (clear B)     (clear C)     (clear D)
(holding A)   (holding B)   (holding C)   (holding D)
(handempty)
```

A Pattern

- Example: only consider 5 ground facts related to block A
  - "Pattern" := $p = \{(\texttt{on A B}), (\texttt{on A D}), (\texttt{clear A}), (\texttt{ontable A})\}$
- Initial state:

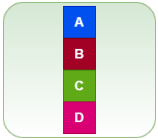



```
(ontable A)
(ontable B)
(ontable C)
(ontable D)
(clear A)
(clear B)
(clear C)
(clear D)
(handempty)
```

```
(ontable A)
(ontable B)
(ontable C)
(ontable D)
(clear A)
(clear B)
(clear C)
(clear D)
(handempty)
```

An "abstract state"

- Goal:




```
(clear A)
(on A B)
(on B C)
(on C D)
(ontable D)
(handempty)
```

```
(clear A)
(on A B)
(on B C)
(on C D)
(ontable D)
(handempty)
```

An "abstract goal"

Transformed Actions

- "Pattern" := $p =$ { (on A B),(on A D),(clear A),(ontable A)}
- Example action: (unstack A B)
  - Before transformation

```
:precondition  (and (clear A) (on A B))
:effect        (and (not (clear A)) (not (on A B))
                 (not (handempty)) (holding A) (clear B))
```

  - After transformation

```
:precondition  (and (clear A) (on A B))
:effect        (and (not (clear A)) (not (on A B)))
```

- Example action: (unstack C D)
  - Before transformation

```
:precondition  (and (clear C) (on C D))
:effect        (and (not (clear C)) (not (on C D))
                 (not (handempty)) (holding C) (clear D))
```

  - After transformation

```
:precondition  (and )
:effect        (and )
```

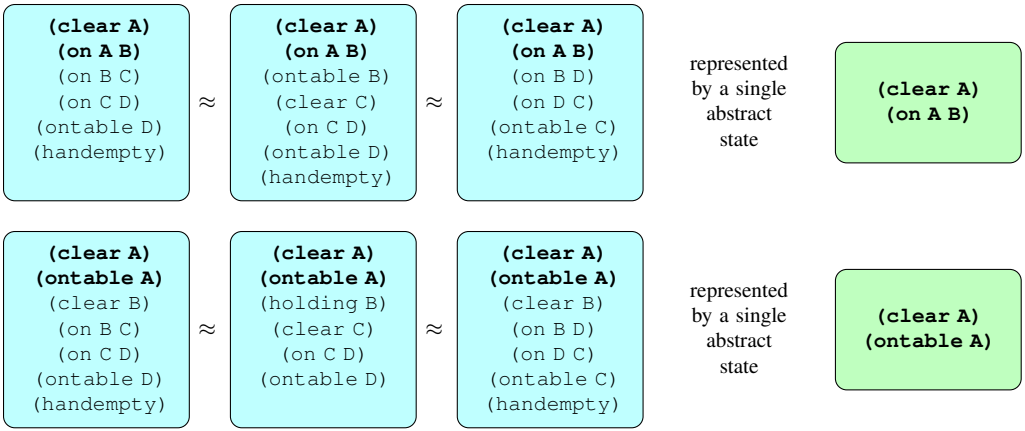
Loses some preconditions and effects!

Let's call this action resulting from pattern application action $a \cap p$ not a set, but "a restricted to p"!

Loses all preconditions and effects! $\Longrightarrow$ never used!

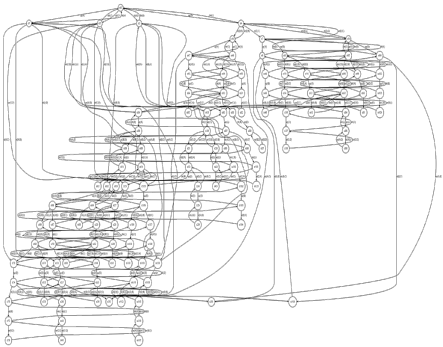
Patterns, Abstract States

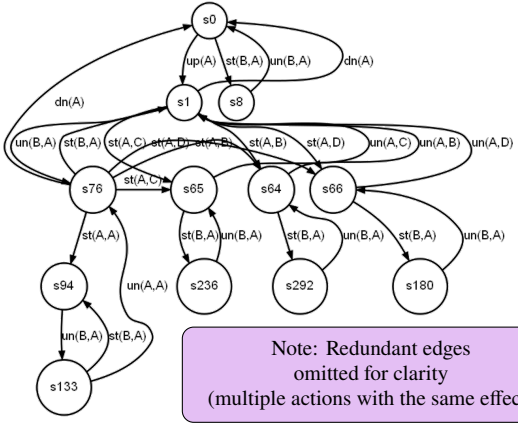
- Given a pattern (set of ground facts) p
  - A state s is represented by the abstract state $s \cap p$



Smaller State Transition Graph

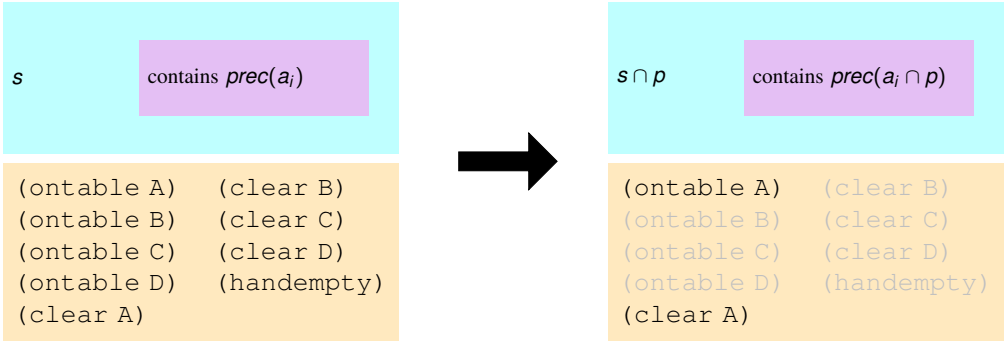
- Reachable state transition graph for the given pattern:
  - Current state: Everything on the table, hand empty, all blocks clear
    - Abstract state: $s_0 = \{$`(ontable A), (clear A)`$\}$





Relaxation

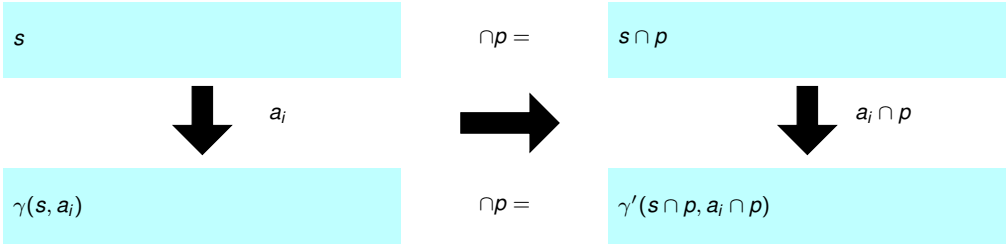
- Why is this a relaxation?
  - Step 1: In s we can execute a_i $\Longrightarrow$ in $s \cap p$ we can execute $a_i \cap p$



Relaxation (cont.)

- If γ' is the state transition function for transformed action/states, then:

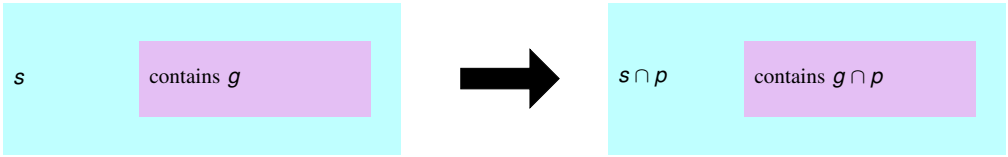
$$\gamma'(s \cap p, a_i \cap p) = \gamma(s, a_i) \cap p$$



$\Longrightarrow$ Executable action sequences are preserved!

Relaxation (cont.)

- If $g \subseteq s$, then $g \cap p \subseteq s \cap p$

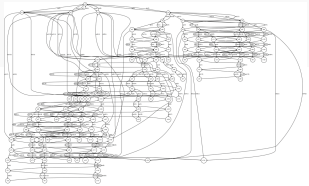


$\Longrightarrow$ Solutions are preserved (but new solutions may arise!)

PDB Heuristic: State Variables

- For PDB heuristics, a state variable representation is useful
  - Typically:
    - Reduces the number of facts
    - Provides more information about which states are actually reachable!
  - Model problems using the state variable representation, or let planners convert automatically from predicate representation

PDB Heuristic: State Variables (cont.)



- Repetition: Blocks world with 4 blocks
  - 536870912 states (reachable and unreachable) in the standard predicate representation
- But in all 125 states reachable from "all-on-table" (all "normal" states):
  - Block A satisfies exactly one of the following:

`(holding A)`	- held in the gripper
`(clear A)`	- at the top of the tower
`(on B A)`	- below B
`(on C A)`	- below C
`(on D A)`	- below D

  - $\Longrightarrow$ remove those facts, introduce state variables

`aboveA` $\in$ `{gripper, nothing, B, C, D}`

PDB Heuristic: State Variables (cont.)

- Example, continued
  - 536870912 states (reachable and unreachable) in the standard predicate representation
  - 20000 states (reachable and unreachable) in state variable representation

```
aboveA  ∈  {nothing, B, C, D, gripper}
aboveB  ∈  {nothing, A, C, D, gripper}
aboveC  ∈  {nothing, A, B, D, gripper}
aboveD  ∈  {nothing, A, B, C, gripper}
posA    ∈  {on-table, other}
posB    ∈  {on-table, other}
posC    ∈  {on-table, other}
posD    ∈  {on-table, other}
hand    ∈  {empty, full}
```

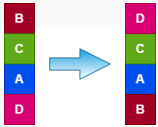
The state variable translation is not part of the PDB heuristic!

Using state variables is useful because PDBs work better with fewer "irrelevant states" in the state space

Allows structure to remain in the abstract search space:
Preserves the fact that A can't be under B and under C)

Also useful when choosing facts: Ignore where A is, care about where B is

PDB Heuristic: Rewriting the Problem



- Rewriting works as before
  - Suppose the pattern is `{ aboveB, aboveD, posB, posD }`
  - Rewrite the goal
    - Original: `{aboveB=A, aboveA=C, aboveD=nothing, hand=empty }`
    - Abstract: `{aboveB=A, aboveD=nothing }`
  - Rewrite actions, removing some preconds / effects
    - `(unstack A D)` no longer requires `aboveA=nothing`
    - `(unstack B C)` still requires `aboveA=nothing`
  - ...

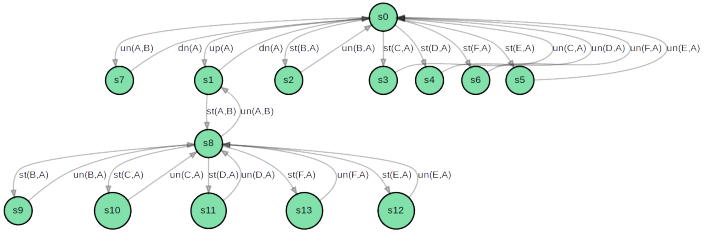
```
aboveA ∈ {nothing, B, C, D, gripper}
aboveB ∈ {nothing, A, C, D, gripper}
aboveC ∈ {nothing, A, B, D, gripper}
aboveD ∈ {nothing, A, B, C, gripper}
posA   ∈ {on-table, other}
posB   ∈ {on-table, other}
posC   ∈ {on-table, other}
posD   ∈ {on-table, other}
hand   ∈ {empty, full}
```

PDB Heuristic: State Space Size

- Abstract states reachable from "all on table", by pattern p ...

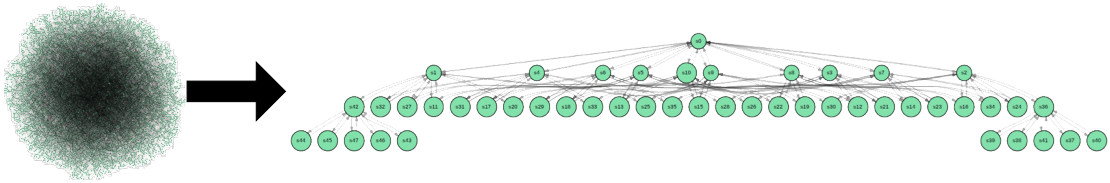
Blocks	All vars	p={aboveA}	p={aboveA,posA}	p={aboveA,aboveB}
4	125	5	10	24
5	866	6	12	35
6	7057	7	14	48
7	65990	8	16	63
8	695417	9	18	80
9	8145730	10	20	99

Immediately
`(unstack A B)`
Don't care if
`aboveB=A`



PDB Heuristic: State Space Size (cont.)

- For 6 blocks, originally 7057 reachable states
  - Pattern {aboveA, aboveB} $\Longrightarrow$ 48 reachable states

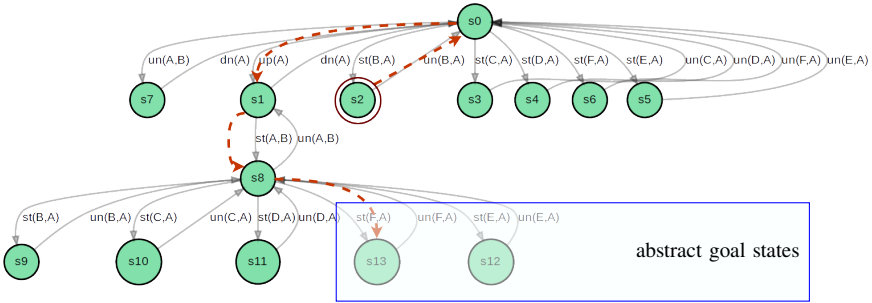


PDB Computation: Main Idea

```
aboveA = B,
posA = on-table,
aboveB = nothing,...
```

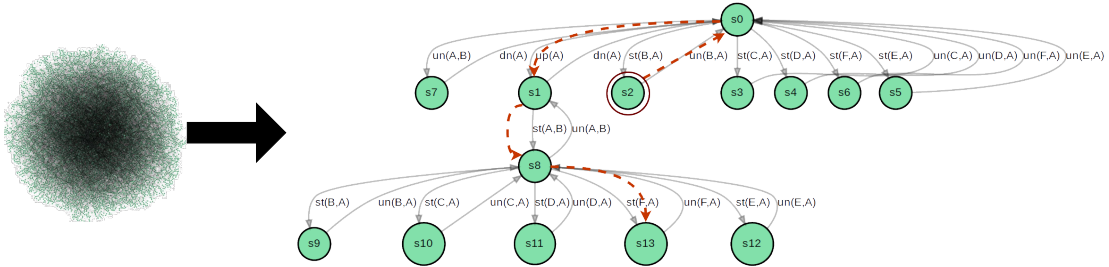
```
aboveA = B,
posA = on-table
```

- To calculate $h_{pdb}(s)$ for a new state s:
  - Example: BW6, pattern `{aboveA,posA}`
  - Convert s to *abstract* state
  - Find optimal path to abstract goal state – in a much smaller seach space
    - Current abstract state is s_2



Tweak 1: Caching

- Because we keep few state variables:
  - *Many* real states map to the same abstract state
  - $\Longrightarrow$ An abstract state may be encountered many times during search
  - $\Longrightarrow$ Cache calculated costs



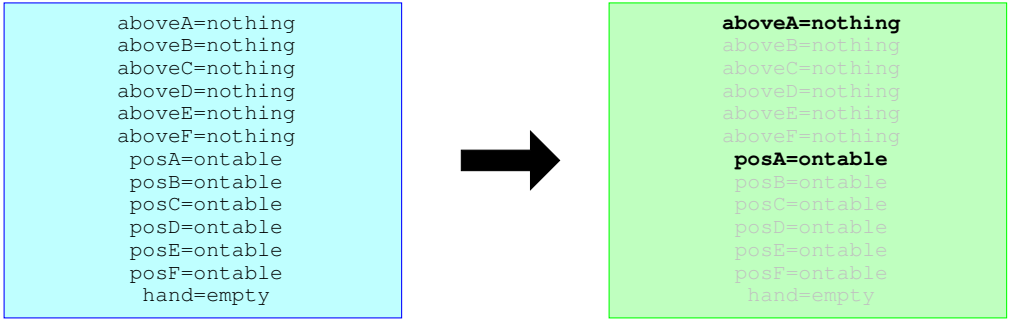
Tweak 2: Databases!

- Dijkstra efficiently finds optimal paths from all abstract states
  - $\Longrightarrow$ Precalculate all heuristic values for each pattern
  - Store in a look-up table – a database

Second main idea!

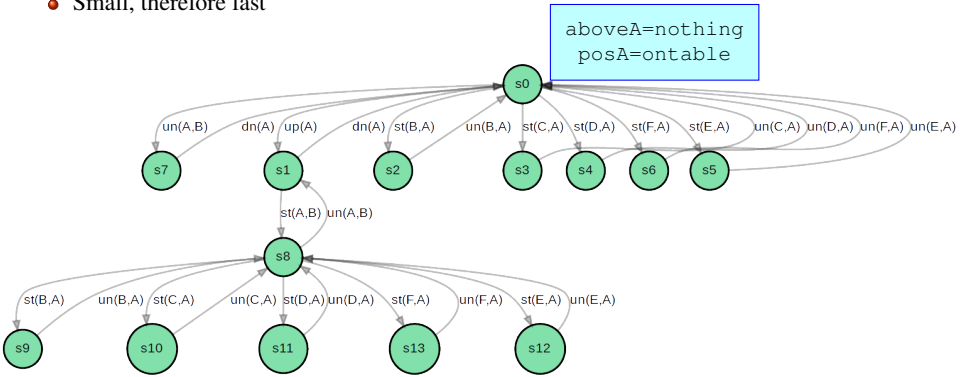
Preprocessing

- Preprocessing step 1 (BW 6, pattern `{aboveA, posA}`
  - Create the initial abstract state...



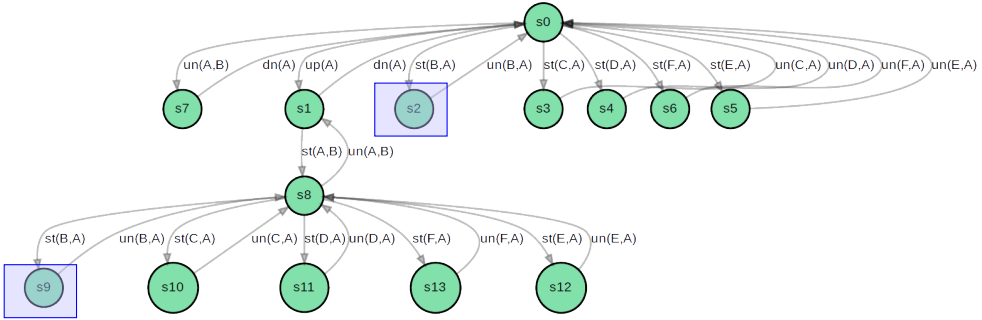
Preprocessing (cont.)

- Preprocessing step 2
  - Find all reachable abstract states
    - Start in the abstract initial state (s_0 below)
    - Apply all applicable actions to all states you find (DFS, BFS, ...)
    - Small, therefore fast



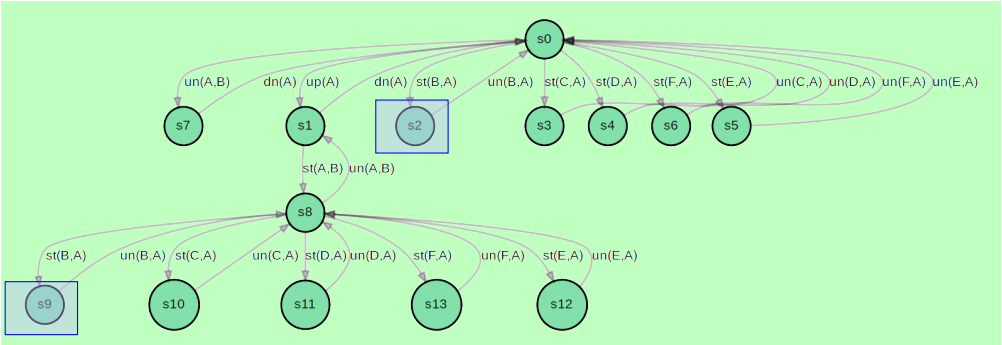
Preprocessing (cont.)

- Preprocessing step 3: Which abstract states satisfy the abstract goal?
  - Real goal: `{aboveA=B,aboveB=C,aboveC=D,aboveD=E,aboveE=F}`
  - Abstract goal: `{aboveA=B}`
    - Satisfied in two of the reachable states



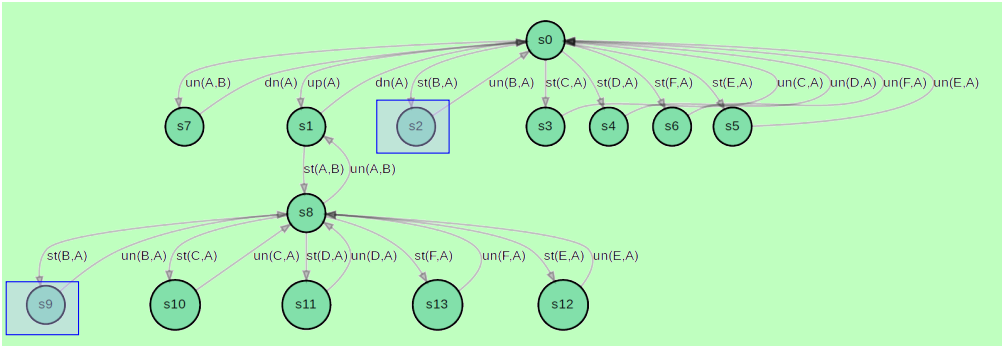
Preprocessing (cont.)

- Preprocessing step 4: Compute the database
  - For every abstract state reachable from the abstract initial state,
  - find a cheapest path to any abstract goal state



Preprocessing (cont.)

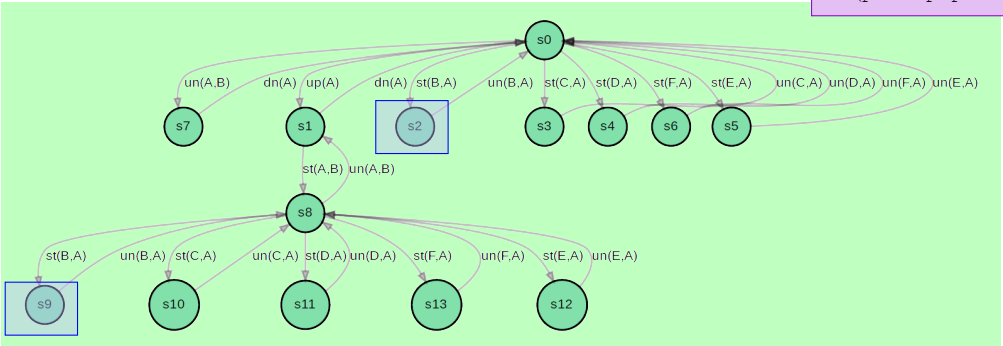
- More efficien computation:
  - Start with cost=0 for all abstract goal states
  - Search backward from these states – single call to Dijkstra
    - Queue initialized with multiple nodes/states of cost 0 (s_2, s_9)
    - Edges followed backward (in explicitly computed graph, no γ^{-1} necessary)



Preprocessing (cont.)

- Example: Priority queue contains $< s_2/cost = 0, s_9/cost = 0 >$
  - Pick s_2
    - A "successor" is s_0, reached backwards by `(stack B A)` of cost 2;
    - s_0 inserted in the queue with cost 2
  - Priority queue contains $< s_9/cost = 0, s_0/cost = 2 >$, pick s_9
    - A successor is s_8 ...

Assuming cost(`stack/unstack`)=2, cost(`pickup/putdown`)=1



PDB Heuristics: Databases

- Database:
  - Stores the cost found by Dijkstra for every abstract state s
    - Cost is optimal within the relaxed problem
    - Cost is admissible for the "real" problem

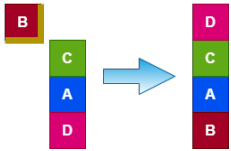
Abstract state	Cost

PDB Heuristics in Forward Search

- Step 1: Automatically generate a pattern
  - A selection of state variables to consider
  - Choosing a good pattern is a difficult problem!
    - Different approaches exist ...
- Step 2: Calculate the pattern database
  - As already discussed

PDB Heuristics in Forward Search (cont.)

- Step 3: Forward search in the original problem
  - For each new successor state s_1, calculate heuristic value $h_{pdb}(s_1)$
    - Example: $s_1 = \{$ `aboveD=A, aboveA=C, aboveC=nothing, aboveB=gripper, posA=other, posB=other, posC=other, posD=ontable, hand=full` $\}$
  - Convert this to an abstract state
    - Example: $s'_1 = \{$ `aboveD=A, aboveB=gripper, posB=other, posD=ontable` $\}$
  - Use the database to quickly look up $h_{pdb}(s'_1)$ = the cost of reaching the nearest abstract goal from s'_1




```
abstract state                                                    ⟹ cost
aboveB = gripper, aboveD = A, posB = other, posD = on-table       ⟹ n1
aboveB = gripper, aboveD = A, posB = other, posD = other          ⟹ n2
...                                                               ⟹
```

Computation time

- How close to $h^*(n)$ can an admissible PDB-based heuristic be?
  - Wrong question!
    - A pattern can include all state variables $\Longrightarrow$ equal to $h^*(n)$

Then computing the database takes too much time...

How much time?

Dijkstra's algorith: O(|E|+|V|log(|V|))

|V| = number of reachable abstract states

# of reachable abstract: Exponential in # of selected state variables

Exponential in the number of state variables in the pattern...

Possible state variables: Linear in problem size (length of PDDL file, ...)
Select all $\Longrightarrow$ PDB computation exponential in problem size

Computation time (cont.)

We want: Polynomial in the problem size (length of PDDL file, ...)

Counteract this:

Exponential in the number of state variables in the pattern ...

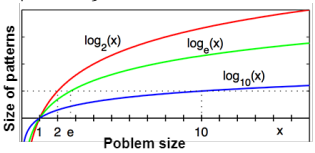
Using this:

Select a logarithmic number of state variables – growing as log(problem size)

Computation time: $O(2^{\log(problemsize)})$, which is polynomial in problem size

Asymptotic Accuracy: Single PDB Heuristic

- Assume (as before) an *infinite* set of poblems $\{P_i | i \geq 0\}$
  - Assume a strategy for selecting a single pattern for each problem
    - So that pattern size grows at most as $O(log(k))$ for problem size k



  - What is the best accuracy guarantee any strategy can give?
    - $h(n) \leq$ cost of reaching the *most expensive subgoal* of size $O(\log(k))$

Subgoal size is not constant but grows with problem size k – good!

But still, $\log(k)$ grows much slower than k ...

  - $\Longrightarrow$ For any given single pattern, asymptotic accuracy is 0 in many domains
    - As before, *practical results* are usually better!

PDB Heuristics: Gripper Example

- A common restricted gripper domain:
  - One robot with two grippers
  - Two rooms
  - All n balls originally in the first room
  - Objective: All balls in the second room

Compact state variable representation:

`loc(ball`$_k$`) ∈ { room1, room2, gripper1, gripper2}`

`loc(robot) ∈ {room1, room2}`

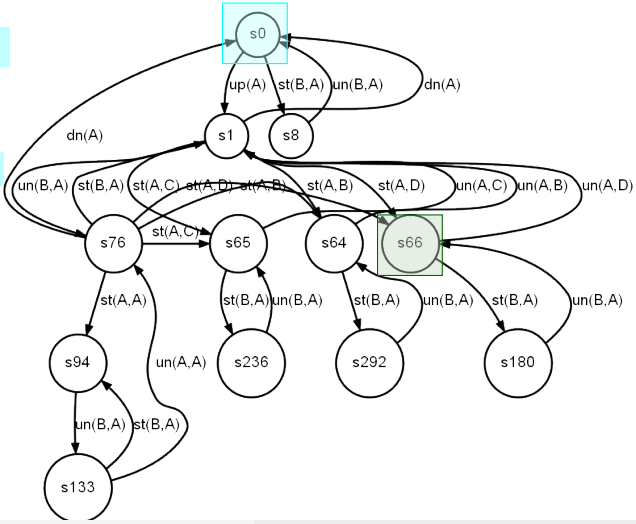
$2 * 4^n$ states, some unreachable

Some possible patterns for $k \geq 1$ balls:

`{loc(ball`$_1$`)}`	$\Longrightarrow$	4 abstract states
`{loc(ball`$_1$`),loc(robot)}`	$\Longrightarrow$	8 abstract states
`{loc(ball`$_i$`)` $\mid i \leq k$`}`	$\Longrightarrow$	4^k abstract states
`{loc(ball`$_i$`)` $\mid i \leq \log(k)$`}`	$\Longrightarrow$	$4^{\log(k)}$ abstract states

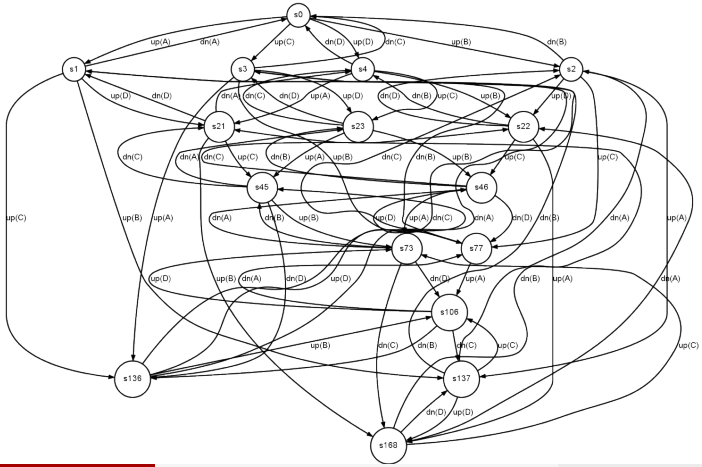
BW4: Subproblem 2

- Subproblem 2: Some facts related to B
  - Current state: Everything on the table, hand empty, all blocks clear
    - Abstract state: `{(ontable B), (clear B)}`
  - Goal state: A on B on C on D
    - Abstract goal: `{(on B C)}`
  - Find a path, compute its cost



BW4: Subproblem 3

- Subproblem 3: Only consider `(holding ?x)` facts...
  - Not handempty, so can hold many blocks



Improving PDBs

- For each pattern, compute a separate pattern database
  - Each such cost is an admissible heuristic

What then? Do we sum the costs?

Possibly in satisficing planning – but would generally not be admissible...

Pattern 1, `aboveB`: I need `(stack A B)`, cost 2

Pattern 2, `aboveC`: I need `(stack B C)`, cost 2

Sum: 2+2=4

Could have used `(buildTower A B C)`, cost 3
Only detectable if we consider the whole problem

But the maximum of two admissible heuristics is also admissible!
And polynomial time: k patterns require k computations

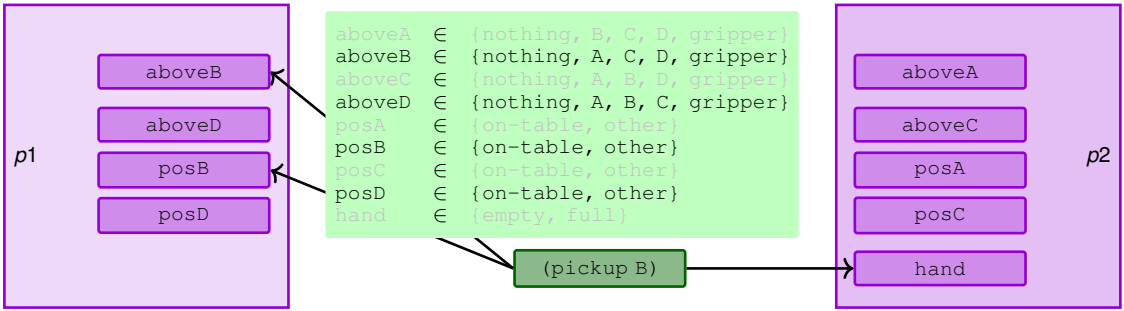
- What is the new level of accuracy?
  - Still 0... asymptotically | worst case
  - But this can still work well in practice!

Additive PDB Heuristics

- To create admissible heuristics by summing PDB heuristics:
  - Each fact should be in at most one pattern
  - Each action should affect facts in at most one pattern
- $\Longrightarrow$ Additive pattern database heuristics

Additive PDB Heuristics (cont.)

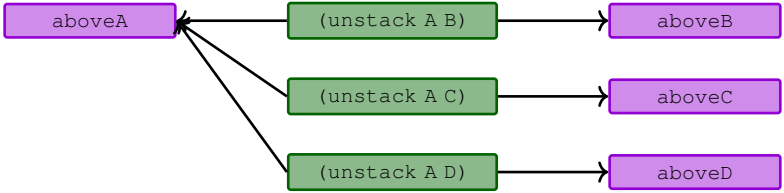
- BW: Is $p1 = \{\text{facts in even rows}\}$, $p2 = \{\text{facts in odd rows}\}$ *additive*?
  - No! `(pickup B)` affects `{aboveB, posB}` in $p1$, `{hand}` in $p2$



One potential problem:
Both patterns could use `(pickup B)` in their optimal solutions
$\Longrightarrow$ sum counts this twice! This is what we're trying to avoid...

Additive PDB Heuristics (cont.)

- BW: Is $p1 = \{\texttt{aboveA}\}$, $p2 = \{\texttt{aboveB}\}$ additive?
  - No! `(unstack A B)` affects `{aboveB}` in $p1$, `{aboveA}` in $p2$
  - True for all combination of `aboveX`!

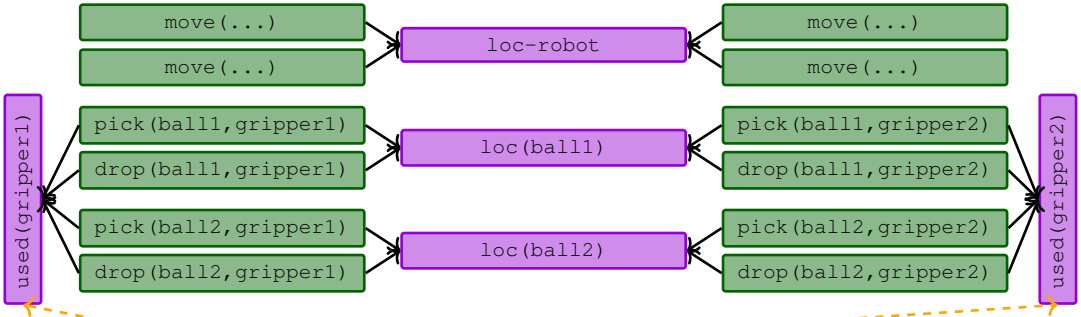


- An additive PDB heur. could use one of these:
  - $p1 = \{\texttt{aboveA}\}$
  - $p1 = \{\texttt{aboveA}, \texttt{aboveC}, \texttt{aboveD}\}$
  - ...
- Can't have two separate patterns $p1$, $p2$ both of which include an `aboveX`
  - Those `aboveX` will be directly connected by some unstack action

This formulation of the Blocks World is "connected in the wrong way" for this approach to work well

Additive PDB Heuristics (cont.)

- "Separating" patterns in the Gripper domain:

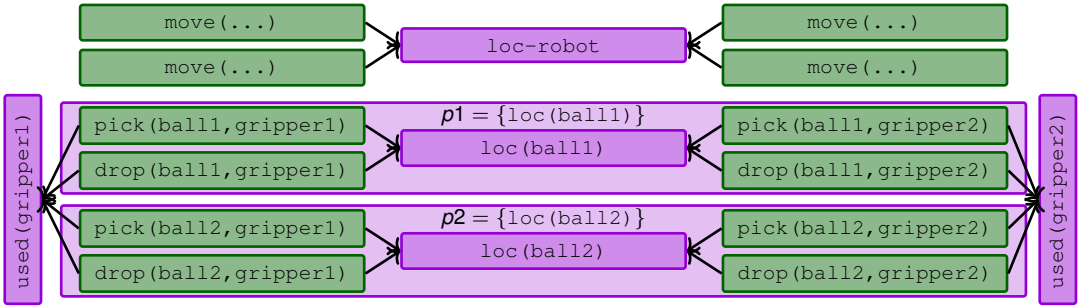


Are these problems?

$\texttt{loc}(\texttt{ball}_k) \in \{\texttt{room1}, \texttt{room2}, \texttt{gripper1}, \texttt{gripper2}\}$
$\texttt{loc-robot} \in \{\texttt{room1}, \texttt{room2}\}$
$\texttt{used}(\texttt{gripper}_k) \in \{\texttt{true}, \texttt{false}\}$

Additive PDB Heuristics (cont.)

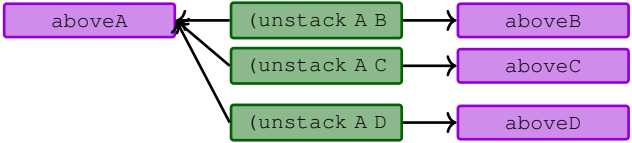
- No problem:We don't have to use all variables in patterns!



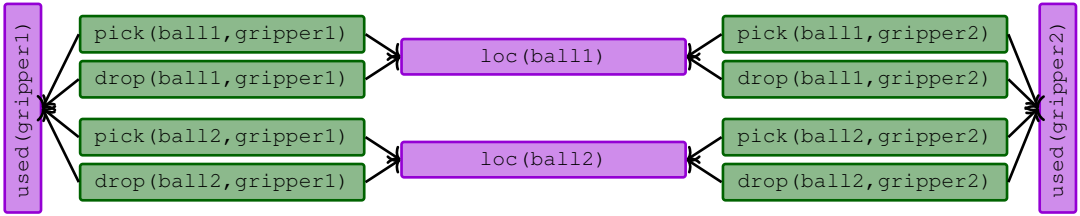
For each pattern we chose one variable
Then we have to include all actions affecting it
The other variables those actions affect [`used(...)`] don't have to be part of any pattern!

Additive PDB Heuristics (cont.)

- Notice the difference in structure!



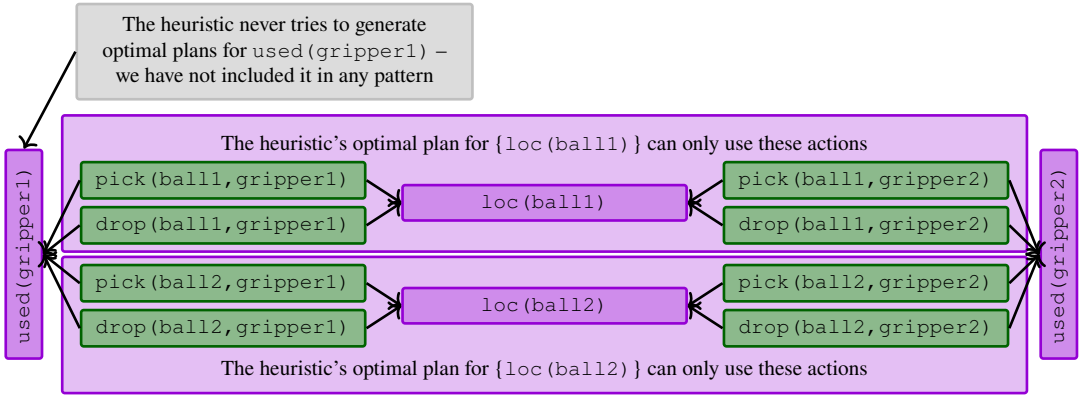
BW: Every pair of `aboveX` facts has a *direct connection* through an action



Gripper: No pair of `loc()` facts has a *direct connection* through an action

Additive PDB Heuristics (cont.)

- When every action affects facts in at most one pattern:
  - The subproblems we generated are completely disjoint
    - They achieve different aspects of the goal
    - Optimal solutions must use different actions



Additive PDB Heuristics (cont.)

- Avoids the overestimation problem

Problem:
Goal: p and q
A1: effect p
A2: effect q
A3: effect p and q // BuildTower

To achieve p Heuristic uses A1
To achieve q Heuristic uses A2

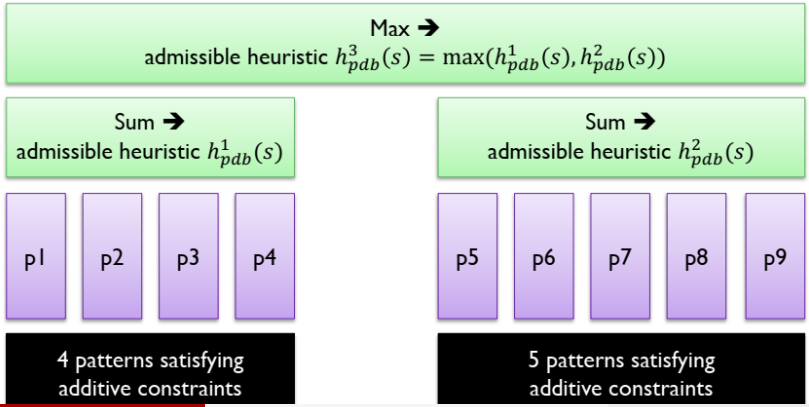
Sum of costs is 4 – optimal cost is 3, using A3

This cannot happen
when every action affects facts in at most one pattern

- $\Longrightarrow$ The costs are additive for multiple patterns
- $\Longrightarrow$ Adding costs from multiple heuristics yields an admissible heuristic!

Additive PDB Heuristics (cont.)

- Can be taken one step further...
  - Suppose we have several sets of additive patterns:
    - Can calculate an admissible heuristic from each additive set, then take the maximum of the results as a stronger admissible heuristic



Additive PDB Heuristics (cont.)

- How close to $h^*(n)$ can an additive PDB-based heuristic be?
  - For additive PDB heuristics with a single sum, asymptotic accuracy as problem size approaches infinity...

Additive PDB Heuristics (cont.)

- In Gripper:
  - In state s_n there are n balls in `room1` and no balls are carried
  - Additive PDB heuristic $h^{PDB}_{add}(s_n)$
    - Use a separate pattern for each ball location variable `loc(ball`$_k$`)`
    - For each pattern/ball, the optimal PDB cost is 2
      - `pick(ball, room1, gripper1):loc(ball)=room1` $\Longrightarrow$ `loc(ball)=gripper1`
      - `drop(ball, room2, gripper1):loc(ball)=gripper1` $\Longrightarrow$ `loc(ball)=room2`
    - $h^{PDB}_{add}(s_n)$ = sum for n balls = $2n$
  - Real cost:
    - Using both gripper
      - `pick`, `pick`, `move(room1,room2)`, `drop`, `drop`, `move(room2,room1)`
    - Repeat $n/2$ times, total cost $\approx 6n/2 = 3n$
  - $\Longrightarrow$ Asymptotic accuracy 2n/3n = 2/3

Additive PDB Heuristics (cont.)

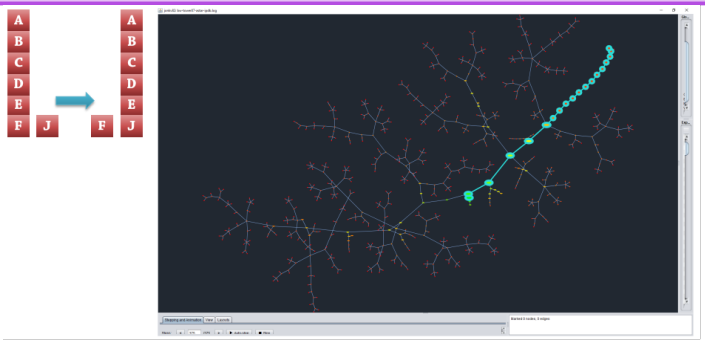
- What quality guarantees can an additive PDB heuristic give?
  - For additive PDB heuristics with a single sum, asymptotic accuracy (fraction of h^*) as problem size approaches infinity:

	h^+ (too slow)	Additive PDB
Gripper	2/3	2/3
Logistics	3/4	1/2
Blocks world	1/4	0
Miconic-STRIPS	6/7	1/2
Miconic-Simple-ADL	3/4	0
Schedule	1/4	1/2
Satellite	1/2	1/6

  - Only guaranteed if the planner finds the best combination of patterns!
    - This is a very difficult problem in itself!
  - But as usual, this is a worst-case analysis...

Example

bw-tower07-astar-ipdb: Only 7 blocks, A* search, based on PDB variation



- Blind A*: 43150 states calculated, 33436 visited
- A* + goal count: 6463 states calculated, 3222 visited
- A* + iPDB: 1321 states calculated, 375 visited

No heuristic is perfect – visiting some additional states is fine!

References I

[1] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2013. ISBN 9781608459698. doi: 10.2200/S00513ED1V01Y201306AIM022. URL https://doi.org/10.2200/S00513ED1V01Y201306AIM022.

[2] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated planning - theory and practice*. Elsevier, 2004. ISBN 978-1-55860-856-6.

[3] Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016. ISBN 978-1-107-03727-4. URL http://www.cambridge.org/de/academic/subjects/computer-science/artificial-intelligence-and-natural-language-processing/automated-planning-and-acting?format=HB.

[4] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019. doi: 10.2200/S00900ED2V01Y201902AIM042. URL https://doi.org/10.2200/S00900ED2V01Y201902AIM042.