Efficient Implementation of Pattern Database Heuristics for Classical Planning

Silvan Sievers[1], Manuela Ortlieb[1] and Malte Helmert[2]

[1]Albert-Ludwigs-Universität Freiburg [2]Universität Basel

Classical Planning

A deterministic planning task is a 4-tuple $\Pi = \langle \mathcal{V}, \mathcal{I}, \mathcal{O}, s_\star \rangle$ where

- $\mathcal{V}$ is a finite set of state variables with an associated finite domain $\mathcal{D}_v$ for each variable $v \in \mathcal{V}$
- $\mathcal{I}$ is the initial state (a state is an valuation over $\mathcal{V}$)
- $\mathcal{O}$ is a finite set of operators where each operator $o \in \mathcal{O}$ (with associated cost $cost(o) \in \mathbb{N}_0$) possibly changes the value of one or several variables
- $s_\star$ is a goal description which is a partial variable assignment

Objective for optimal planning: Find an optimal (i.e. a cheapest) plan which leads from the initial state to a goal state.

PDBs for Classical Planning

- Pattern databases heuristics for a planning task are abstraction heuristics defined by a subset of variables $P \subseteq \mathcal{V}$ called the pattern:
  - Only variables in P are perfectly represented in the abstract planning task.
  - All other variables are not represented at all.
- A PDB is a lookup table which stores $h^*(s)$ for all (abstract) states s, implemented as a one-dimensional array of size $N := \prod_{i=1}^{k} |\mathcal{D}_i|$.
- A perfect hash function maps states to table indices in $\{0, \dots, N-1\}$, called ranks.
- Computing ranks from states is called ranking, the inverse process is called unranking.

Basic PDB Construction Algorithm

$N := \prod_{i=1}^{k} |\mathcal{D}_i|$
PDB := array of size N filled with ∞
heap := *make-heap*()
graph := *make-array-of-vectors*()

/* phase 1: create graph of backward transitions and identify goal states */

```
for r ∈ {0, ..., N − 1} do
    s := unrank(r)
    if s⋆ ⊆ s then
        PDB[r] := 0
        heap.push(0, r)
    for o ∈ O do
        if o applicable in s then
            s′ := successor of s
            r′ := rank(s′)
            graph[r′].append(⟨r, cost(o)⟩)
```

/* phase 2: perform Dijkstra search with *graph* and *heap* to complete the entries in *PDB* */
... (Dijkstra pseudo-code omitted)

Inefficiencies of the Basic Algorithm

1. Creating the complete transition graph has a significant space cost.
2. Testing each operator for applicability in each state is expensive.
3. A complexity analysis shows that the computation of many states and ranks of these states (lines 13 and 14) can form the bottleneck of the overall algorithm.

Efficient PDB Construction Algorithm

1. Required: efficient way to regress over states to avoid constructing the transition graph
   Solution: multiply out all non-injective operators so that all operators can be also applied "backwards".
2. Required: efficient way to determine all applicable operators for a given state to avoid checking all operators individually
   Solution: use a successor generator for an efficient computation of the set of applicable operators for a given state.
3. Required: avoid ranking and unranking of states while running Dijkstra's algorithm.
   Solution:
   - Successor generator works directly on ranked states.
   - The effects of (backward) operators are expressed in a change of rank, i.e., a simple addition is sufficient.

Experimental results

Domain	basic algorithm				efficient algorithm			
	100k	1m	10m	100m	100k	1m	10m	100m
Barman (20)	20	20	0	0	20	20	20	20
Elevators (20)	20	20	18	0	20	20	20	20
Floortile (20)	20	20	2	0	20	20	20	20
Nomystery (20)	20	20	18	10	20	20	20	20
Openstacks (20)	20	20	3	0	20	20	20	20
Parcprinter (20)	20	20	4	0	20	20	20	20
Parking (20)	20	20	10	0	20	20	20	20
Pegsol (20)	20	20	0	0	20	20	20	20
Scanalyzer (20)	17	12	3	3	20	20	19	18
Sokoban (20)	20	20	20	7	20	20	20	20
Tidybot (20)	13	0	0	0	20	20	20	4
Transport (20)	20	20	18	2	20	20	20	20
Visitall (20)	20	20	8	8	20	20	20	20
Woodworking (20)	20	20	2	0	20	20	20	20
Total (280)	**270**	**252**	**106**	**30**	**280**	**280**	**279**	**262**

Number of instances where a PDB could be constructed within 30 min and 2GB memory by the basic and efficient construction algorithm for different PDB size limits.

Domain	HSP_f-iPDB	FD-iPDB	M&S-2011
Barman (20)	4	4	4
Elevators (20)	19	15	10
Floortile (20)	6	2	7
Nomystery (20)	18	16	18
Openstacks (20)	6	14	13
Parcprinter (20)	13	11	13
Parking (20)	5	5	5
Pegsol (20)	5	18	19
Scanalyzer (20)	7	10	9
Sokoban (20)	15	20	19
Tidybot (20)	14	14	7
Transport (20)	7	6	7
Visitall (20)	16	16	16
Woodworking (20)	6	5	9
Total (280)	**141**	**156**	**156**

Number of tasks solved by the original iPDB implementation in HSP_f, our new implementation of iPDB in Fast Downward and the IPC 2011 merge-and-shrink planner.