Construction of Pattern Database Heuristics using Cost Partitioning

Simon Wang

Department of Informatics & Computational Science
University of Basel

Abstract

Pattern databases (Culberson & Schaeffer, 1998) or PDBs, have been proven very effective in creating admissible Heuristics for single-agent search, such as the A*-algorithm. Haslum et. al proposed, a hill-climbing algorithm can be used to construct the PDBs, using the canonical heuristic. A different approach would be to change action-costs in the pattern-related abstractions, in order to obtain the admissible heuristic. This the so called Cost-Partitioning.

Introduction

Planning tasks are of **fundamental importance** for single-agent search in **Artificial Intelligence**. Such tasks are given by a set of states, including an initial state and a goal description, further, a set of actions, allowing us to make transitions between the states. As we aim for an efficient, **automated handling** of **any given planning task**s, the single-agent search is often guided by an heuristic. Given an **admissible heuristic** (which never overestimates the actual cost to the goal state), the **A*-algorithm**, is guaranteed to find the optimal path through the states. In recent times there have been several **promising approaches** in creating admissible heuristics, e.g **Pattern Database Heuristics**.

The basic idea of **Pattern databases** (Culberson & Schaeffer, 1998) or PDBs, is to simplify the original search space, which is often very large, into smaller subspaces, or patterns, for which we compute the exact goal distance, and store their optimal distances in a database.

The advantage is that we can reduce the time-vasting search of the complete searchspace to a lookup-table. There has been several succesfull attempts in automatically generate these PDB Heuristics for domain-independent planning, such as Edelkamp (2001) or more recently Haslum et al. (2007). Haslum et al. propose to start with several initial patterns, each one only containing one goal variable, and then gradually extending and improving them, by exploring the

neighbourhood. These patterns typically will have the same action costs as in the original space, but to guarantee the admissible heuristic, not every pattern may be considered in the end for the heuristical estimate.

However there exists another way to obtain the admissible heuristic. Instead of dropping patterns, we may **change the action cost** in the patterns. But it's not always clear, how this **cost-partitioning** should be done exactily. This is where our focus will be. It has been shown (Katz & Domshlack, 2008) that there exists a general procedure to generate an optimal action-cost partitoning, which can be done in polynomial-time. Nevertheless the polynomial time is still very long in practice. The aim of this paper is to find a simple cost-partitioning, which can be **combined** with the original **Haslum Pattern-Collection-search**.

Before introducing the formal definitions, we will show a simple example, the 15-Puzzle, to illustrate the idea.

The 15-puzzle-example

The 15-puzzle-problem is a 4 x 4 sliding tile puzzle. In its goal state, each tile is numbered left to right, top to bottom, from 1 to 15 (one field is blank). At each state, one is allowed to move one tile (up, left, right, down) to the blank field. The target is to reach the goal state using as few moves as possible. Normally, each move has an operation cost of 1.

Now we can simplify the original problem into several ones, let's say two tiles. So with our 15-puzzle, we try to solve the problem by only regarding 1-7, the other tiles are indistinguishable. And we do the same for tiles 8-15. Formally these are two abstractions, and we solve them, targeting their goal. An admissible heuristic is an approximation of the optimal cost (e.g 12 moves), which will never overestimate. Such heuristics are easily calculated and help us to guide the search for larger problems. The two abstractions (1-7, 8-15) are said to be additive if the sum of their heuristic values is never larger than the actual cost in the original space (1-15). The question now is not how to generate these abstractions (see the Haslum algorithm section below), but how to do the cost-partitoning of the operation-costs (op-costs) if we want a good heuristic.

Preliminairies

1. Planning task

In this work we consider the $SAS+$ planning formalism (Bäckström & Nebel, 1995).

A $SAS+$ planning task is a quintuple $\Pi =< V, A, I, G >$
$V = \{v_1, ..., v_n\}$
V is a set of state variables, each having a finite domain $dom(v_i)$ of possible

values
each complete variable assignement of V is called a **state**
I is the **initial state**
G is a partial assignement, describing the set of **goal states**: $\{s \in S \mid G \subseteq s\}$

$A = \{a_1, ..., a_n\}$
is a finite set of actions where each action a is 3-tuple $< \mathit{precondition}(a), \mathit{effect}(a), \mathit{cost}(a) >$
where $\mathit{precondition}(a)$ and $\mathit{effect}(a)$ are **partial assignments**
and $\mathit{cost}(a)$ assigns to the variables **a non-negative, real value**: $\mathit{cost} : A \rightarrow \mathbb{R}_0^+$

Each planning task induces a transition-system, which allows us to label the state-to-state-transitions with costs.

For an $SAS+$task $\Pi =< V, A, I, G >$ the **induced transition-system** (or TS) is a 4-tuple
$T = (S, A, Tr, I, G, \mathit{cost})$
S is the finite set of all possible states **induced by the** $SAS+$ **planning task**
$\mathit{cost} : A \rightarrow \mathbb{R}_0^+$ is a transition cost-function.
$Tr = \{< s, a, s' > | s, s' \in S, a \in A\}$ is a set of **action-linked transitions**
where a is **applicable** in s (means $\mathit{precondition}(a) \subseteq s$) and
s' is the result of applying a in s, where applying a in s is defined for each assigned variable v in s as follows:

$$\mathit{app}_a(s)(v) = \begin{cases} s(v) & \text{if } v\text{is unspecified in } \mathit{effect}(a)\text{is unspecified} \\ \mathit{effect}(a)(v) & \text{otherwise} \end{cases}$$

Furthermore:

$\mathit{distance}(s, S')$ in T, is given by the cost of a cheapest (with respect to cost) path from s, (with $s \in S$) to a state in S' (with $S' \subseteq S$) along the transitions of T.

A **plan** for T is any path from the initial state to a goal state.
and cheapest paths are called **optimal plans**.

The aim of a planning task is to find an optimal plan of a given problem.

2. Heuristics

For computational reasons, we will calculate an estimate of the optimal search cost, called **Heuristic**, to guide the optimal search.

The heuristic function $h : S \rightarrow \mathbb{R}_0^+$ for an transition-system $T = (S, A, Tr, I, G, \mathit{cost})$, assignes each state $s \in S$ a **real, non-negative value**.

For our purposes, we need additionally the properties **admissible** and **consis-**

tent, in order to guarantee **optimality of the reslut** and **avoid reopening of states with the A* search**.

The heuristic function $h : S \rightarrow \mathbb{R}_0^+$ defines an **admissible heuristic** $h(s, G)$ for an transition-system $T = (S, A, Tr, I, G, cost)$
if for all $s \in S$ holds:

$h(s) \leq distance(s, G)$

The heuristic function $h : S \rightarrow \mathbb{R}_0^+$ defines an **consistent heuristic** h for an transition-system $T = (S, L, Tr, I, G, cost)$
if for all $s, s' \in S$ with $distance(s', G) < distance(s, G)$ holds:

$h(s) \leq h(s') + distance(s, s')$

One possibility to define a heuristic is by **simplifying the transition system**. That means, we map our set of possible states to a **smaller**
set, in such a way that the costs of each state-transition in our **simplified** transition system, **don't exceed** the corresponded **original costs**. The gain of this constraint, is that each admissible, consistent heuristic for the abstraction, will still be admissible, consistent in the original transititon-system. Formally:

An **abstraction-projection** α is a surjective mapping function $\alpha : S \rightarrow S'$ which defines for a transition-system $T = (S, A, Tr, I, G, cost)$
an **abstract transition-system, or abstraction**
$T_\alpha = (S', A, Tr', I', G', cost)$
$S' = \{\alpha(s) | s \in S\}$
$T' = \{< \alpha(s), a, \alpha(s') > | < s, a, s' > \in Tr\}$
$I' = \alpha(I)$
$G' = \{\alpha(s) | s \in G\}$

This allows us to define:

An **abstraction-heuristic** $h_\alpha(s)$ with $s \in S$ is given by the cost of a cheapest path,
from $\alpha(s)$ towards a goal state in T_α.

3. Pattern Database Heuristics

One way of defining abstraction-heuristics are **Pattern Database Heuristics (or PDB Heuristics)**.
A **pattern P** is a partial specification of a complete set of variables (Culberson & Schaeffer, 1998).

For $P \subseteq V$, where V is the set of state variables of a given $SAS+$ planning task Π, with **state set** S,

α^P is the **abstraction-projection induced by** P, which is defined as
$\alpha^P = s \mid_P$ for all $s \in S$
We denote h_{a^P} by h_P

The idea is to construct a Pattern Collection $C = \{P_1, \ldots, P_k\}$ with corresponding abstraction-projections α^{P_i} and abstractions T_{α_i}
with $i = 1, ...k$, in such way, that the following **property** holds:

$\sum_{i=1}^{k} h_{P_i}$ is an admissible and consistent heuristic for Π

A **sufficient criterium** for this property is:

There exists no operator in Π which has an effect on a variable v_i in P_i, **and** on a variable v_j in P_j, for $i \neq j$
The patterns in C are then, said to be **additive.**

This leads us to **2 possible approaches**, dealing with the case,
where the **property is not given, through our abstractions** :

Approach I

The PDB-heuristics are non-additive, in order to keep the patterns, we use **Cost-Partitioning**. This means we change the $cost_i$ of our abstractions $T_{\alpha_i} = (S_i, A_i, Tr_i, I_i, G_i, cost_i)$**,** in such a way that:

for all $< s, L, s' > \in Tr$ of a transition-system $T = (S, A, Tr, I, G, cost)$

$$\sum_{i=1}^{n} cost_i(\alpha_i(s), L, \alpha_i(s')) \leq cost(< s, L, s' >) \tag{1}$$

with $\alpha_i(s) \neq \alpha_i(s')$

Now, this allows us some freedom of the choise of the costs. One possible approach could be the **Zero-One-Cost-Partitioning**,
where we can simplify (1) to some extend:

for all $< s, a, s' > \in Tr$ of a transition-system $T = (S, A, Tr, I, G, cost)$:
$cost_i(\alpha_i(s), a, \alpha_i(s')) = cost(< s, a, s' >)$ for at most one $i \in \{1, ..., n\}$ with $\alpha_i(s) \neq \alpha_i(s')$
and $cost_j(\alpha_j(s), L, \alpha_j(s')) = 0$
for all $j \neq i$ if $\alpha_j(s) \neq \alpha_j(s')$

Approach II

The PDB-heuristics are not all (or partially) additive, in order to combine them optimally, we use the canonical heuristic

where the **canonical heuristic** of a pattern collection $C = \{P_1, ..., P_k\}$is defined as

$h^C(s) = \max_{S \in A} \sum_{P \in S} h^P(s)$

A being the set of all maximal (w.r.t. set inclusion) additive subsets of C

4. The Haslum Algorithm

One way to generate a good Pattern Collection $C = \{P_1, ..., P_k\}$ with the **canonical heuristic**, given a $SAS+$ planning task $\Pi =< V, A, I, G >$,
is the Haslum method (2007).

The search effort, which we aim to minimize, given cost bound L, and admissible and consistent heuristic h, can be estimated by the number of expanded states in a tree search (using IDA*) with the following formula (Korf,Reid & Edelkamp, 2001):

$$\sum_{i=0}^{L} n(L-i)P(i) \tag{2}$$

L is the cost bound
$n(i)$ gives the number of states s, where $distance(I, s) = i$,
$P(i)$ gives the probability that $h(t) < i$, where t is a random state, drawn uniformly from the set of all states
occuring in the search-tree up to depth L

Inspired by this aim the algorithm performs a hill-climbing search in the space of pattern collections as follows:

(a) We start the search with a Pattern Collection $C = \{P_1, ..., P_n\}$ containing one pattern for each goal variable $V = \{v_1, \ldots, v_n\}$, each containing only that variable. (So with the 15-puzzle we would start with 15 1x1 puzzles each having one tile to move.)

(b) From the given Collection C, a new collection C' can be constructed, by selecting a pattern $P_i \in C$, a variable $v \notin P_i$, and adding the new pattern

$P_{k+1} = P_i \cup \{v\}$ to the collection. The range of possible patterns, constructed this way, defines our **neighbourhood**.

Also we can simplify the neighbourhood-evaluation, by considering only variables v, that will influence some variable v' in our Pattern P_i, which means there exists a action, which changes v', **and** has a precondition or effect on v.

(c) Evaluating the neighbourhood: this step is **crucial for the algorithm**, for the decision which new collection C' is the best.
In order to evaluate the neighbours of our pattern collection, we need a **random sample** of states, **drawn uniformly** from the search space up to the cost bound. For a complete, uniform tree, a uniformly sampled state can be found by a **random walk** trough the tree. However the search space of a planning problem is **rarely** uniform or a tree, and additionally the depth is unknown. Still, random walks perform good results, if we **estimate the solution depth** of the uniform drawn sample by the current heuristic, **multiplied by a constant** (e.g 2), since the heuristic is underestimating. (For problems with non-uniform action costs, the depth-estimate is done with the **average cost**.)

Using these sampled states we can **measure the quality** of a pattern collection using formula (2). Since the new collection C' differs from the old C, only by one new pattern, the **canonical heuristic** $h^{C'}$ can never be less then h^C, and the gained improvement is given by

$$\frac{1}{m} \sum_{s \in \{s_i,...,s_m\}} \sum_{h^C \leq k < h^{C'}} n(L-k) \tag{3}$$

$\{s_1, ..., s_m\}$ are the **sampled states**

Assuming that $n(k)$ dominates $\sum_{i<k} n(i)$, we can simplify (3), to get the so called **counting approximation**:

$$\frac{1}{m} \sum_{\{s | h^C(s) < h^{C'}(s)\}} n(L - h^C(s))$$

s is a **random** state within the **uniform drawn sample**

(d) The search is ended, when **no pattern within our size limit improves the current pattern collection**, or the **search-effort exceeds the value of the gain in heuristic** accuracy. The second case occurs more in practice.

This is the common algorithm with the **original action-costs unchanged**. What we do is to change the action-costs in **(c)** after new abstractions have

been created. For testing and implementing, the **Fast Downward planning system** was used.

The crucial step **(c)**, refered as hill-climbing, is shown below in a simplified code.

Pseudo-Code of the Haslum Hill-climbing

```
hill_climbing()

vector new_candidates //contains the new candidate pdbs as described above
vector candidate_pdbs //contains all the candidate pdbs

while (true)

        samples=sample_states()
        update_candidates( new_candidates )

        foreach pdb in candidate_pdbs

                improvement=0
                count=0

                foreach sample in samples
                        if is_heuristic_improved( pdb, sample )
                                ++count

                if (count > improvement)
                        improvement = count
                        best_pdb = pdb

                if (improvement < min_improvement)
                break

add_pattern_to_canonical_heuristic( best_pdb )
```

where update_candidates is differently implemented. For the original canonical Heuristic it is defined as:

```
Canonical_Heuristic::update_candidates( new_candidates )
```

```
foreach pdb in new_candidates
        if is_new_candidate( pdb )
                insert_into_candidate_pdbs( pdb )
```

Experiment

1. Zero-One Cost-Partitioning

Now a simple approach of partitioning the costs, in order to **guarantee additiviy of the abstractions**, could be the zero-one cost partitioning. The **idea** is that the operation-cost are updated each time a new pattern is added to our current pattern collection (inside one hill-climbing-step), in such a way, that **operators** which have been **used in a previous PDB**, will be assigned a **cost of zero**.

Or in pseudo-code:

```
ZeroOneHeuristic::add_pattern_to_zero_one_heuristic( pattern )
        pdb = new PDBHeuristic( pattern, operator_costs )
        insert_into_pattern_databases( pdb )

        foreach operator in operators
                if used_in_a_previous_PDB( operator )
                        operator.cost=0
```

We remember from step **(b)** of the Haslum Algorithm, that only variables are considered to be included in the new Pattern, which will affect our current set of variables over some action. In setting the cost of these actions, which are our **used operators,** to 0, we can guarantee the additivity-constraint:

We change the $cost_i$ of our abstractions $T_{\alpha_i} = (S_i, A_i, Tr_i, I_i, G_i, cost_i)$, in such a way that:
for all $< s, a, s' > \in Tr$ of a transition-system $T = (S, A, Tr, I, G, cost)$:
$cost_i(\alpha_i(s), a, \alpha_i(s')) = cost(< s, a, s' >)$ for at most one $i \in \{1, ..., n\}$ with $\alpha_i(s) \neq \alpha_i(s')$
and $cost_j(\alpha_j(s), L, \alpha_j(s')) = 0$
for all $j \neq i$ if $\alpha_j(s) \neq \alpha_j(s')$

2. Implementation

Several things will change slightly in the implementation of the original Haslum-algorithm:

(a) is_heuristic_improved( ) becomes much simpler for the zero-one-cost-partitioning, since we don't have to go over the maximal additive subsets to obtain the canonical heuristic anymore. The heuristics of the subsets are always additive, thus we simply need to check, wheather the heuristic of the added pattern is greater zero: $h_{P_{new}} > 0$.

(b) Initial Pattern collection: We remember from the Haslum algorithm, we start the hill-climbing with one pattern, for each goal variable, with Zero-One now will be initially empty (constant zero-heuristic).

(c) Initial candidate patterns: With the Haslum method, we will start right away, with each possible extension of a pattern $\{v\}$
in the initial collection with one variable v', if v' influences v in the causal graph. However with Zero-One, we have now a candidate pattern for each goal variable, since our inital pattern collection is empty.

(d) update_canditates: Since we modify the costs of our newly added pattern, we have to do so, for the PDBs of the other candidate patterns in our current collections as well, since they may still have the original costs. In the implementation we first recompute the old candidate pdb's with the updated costs, before adding the pdb's for the new patterns.

```
ZeroOneHeuristic::update_candidates( pattern )

foreach pdb in candidate_pdbs
recomputePDB( pdb, updated_operator_costs ) //update the old pdb operator cos

foreach pdb in new_candidates
insert_into_candidate_pdbs( pdb )
```

3. Experiment Preparation

For more generalizable results, a range of search problems (Grid) have been tested, using the **Zero-One-cost-partioning**.
The time limit was set to 30 minutes for the search component of Fast Downward (including the computation of the heuristic).
The memory limit was set to 2GB.

(a) We compared the original **haslum algorithm** using the **canonical heuristic (I)**, with the modified haslum algorithm
using **Zero-One-heuristic (II)**.

(b) We run both algorithms with following the **different initial parameters**:

pdb_max_size = 500000, 1000000, 2000000, 5000000, 10000000
and **collection_max-size** always **10 times greater**
refered as plan11, plan12, plan13, plan14, plan15 in the result tables **for the canonical**
and refered as plan31, plan32, plan33, plan34, plan35 **for the zero-one**

Note: Since the different parameter only showed minor variations, only plan11 and plan 31 were included in this work.

(c) Different result attributes were compared: most importantly:
coverage (how many problems solved)
expansions (number of searchnodes in the closed list, giving a measure for the quality of the heuristic)
generated (number of searchnodes that have been in the open list, typically a exponential relation to expansions)
iPDB:time (hill-climbing time)
iPDB:iterations (hill-climbing iterations)
inital h-value (inital heuristic value)
search_time

(d) We run the experiment with **14 different set of benchmarks** each containing 20 planning problems, overall **280 planning problems** (which have been used as the benchmark instances of the optimal tracks in the International Planning Competition 2011).

4. Experiment Results

coverage	WORK-plan11	WORK-plan31
barman-opt11-strips (20)	4	4
elevators-opt11-strips (20)	**16**	9
floortile-opt11-strips (20)	2	2
nomystery-opt11-strips (20)	16	14
openstacks-opt11-strips (20)	14	14
parcprinter-opt11-strips (20)	7	**11**
parking-opt11-strips (20)	5	5
pegsol-opt11-strips (20)	0	**17**
scanalyzer-opt11-strips (20)	**10**	9
sokoban-opt11-strips (20)	**20**	18
tidybot-opt11-strips (20)	14	14
transport-opt11-strips (20)	6	6
visitall-opt11-strips (20)	**12**	9
woodworking-opt11-strips (20)	2	**5**
SUM (280)	128	**137**

dead_ends	WORK-plan11	WORK-plan31
barman-opt11-strips (4)	0	0
elevators-opt11-strips (9)	0	0
floortile-opt11-strips (2)	0	0
nomystery-opt11-strips (14)	**28134**	348258
openstacks-opt11-strips (14)	0	0
parcprinter-opt11-strips (7)	**89606**	1234895
parking-opt11-strips (5)	2721620	2721620
scanalyzer-opt11-strips (9)	0	0
sokoban-opt11-strips (18)	**197230**	584356
tidybot-opt11-strips (14)	0	0
transport-opt11-strips (6)	0	0
visitall-opt11-strips (9)	0	0
woodworking-opt11-strips (2)	**1704**	48753
SUM (113)	**3038294**	4937882

expansions	WORK-plan11	WORK-plan31
barman-opt11-strips (4)	**16760296**	20153723
elevators-opt11-strips (9)	**1666477**	21385762
floortile-opt11-strips (2)	319348	1203126
nomystery-opt11-strips (14)	**260595**	3438968
openstacks-opt11-strips (14)	33488248	33488248
parcprinter-opt11-strips (7)	**192091**	1833722
parking-opt11-strips (5)	1771523	1771523
scanalyzer-opt11-strips (9)	17263329	58624853
sokoban-opt11-strips (18)	**4332897**	56062414
tidybot-opt11-strips (14)	2294879	**2173777**
transport-opt11-strips (6)	**1329539**	5984270
visitall-opt11-strips (9)	**24675**	7810517
woodworking-opt11-strips (2)	**1257**	146951
SUM (113)	79705154	214077854

generated	WORK-plan11	WORK-plan31
barman-opt11-strips (4)	**73367303**	88189263
elevators-opt11-strips (9)	**41157505**	398213753
floortile-opt11-strips (2)	2128490	7698475
nomystery-opt11-strips (14)	**1978035**	23670553
openstacks-opt11-strips (14)	169608236	169608236
parcprinter-opt11-strips (7)	**1400188**	13529516
parking-opt11-strips (5)	23848596	23848596
scanalyzer-opt11-strips (9)	310547468	846977345
sokoban-opt11-strips (18)	**11243074**	148077198
tidybot-opt11-strips (14)	7521817	**7186790**
transport-opt11-strips (6)	**10216603**	43913733
visitall-opt11-strips (9)	**84459**	25028583
woodworking-opt11-strips (2)	**15545**	1954140
SUM (113)	653117319	1797896181

iPDB_generated	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	3509	**1382**
elevators-opt11-strips (20)	1567	**93**
floortile-opt11-strips (10)	1497	**966**
nomystery-opt11-strips (20)	**183**	320
openstacks-opt11-strips (20)	1041	**390**
parcprinter-opt11-strips (7)	1607	**1011**
parking-opt11-strips (20)	**15240**	15580
scanalyzer-opt11-strips (20)	3240	**1314**
sokoban-opt11-strips (20)	4569	**3194**
tidybot-opt11-strips (20)	**7662**	8694
transport-opt11-strips (20)	921	**715**
visitall-opt11-strips (20)	**753**	880
woodworking-opt11-strips (2)	477	**162**
SUM (207)	42266	**34701**

iPDB_improvement	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	61	**0**
elevators-opt11-strips (20)	95	**0**
floortile-opt11-strips (10)	41	**0**
nomystery-opt11-strips (20)	84	**0**
openstacks-opt11-strips (20)	0	0
parcprinter-opt11-strips (7)	**22**	27
parking-opt11-strips (20)	0	0
scanalyzer-opt11-strips (20)	56	**0**
sokoban-opt11-strips (20)	**22**	41
tidybot-opt11-strips (20)	**10**	13
transport-opt11-strips (20)	110	**0**
visitall-opt11-strips (20)	60	**0**
woodworking-opt11-strips (2)	0	0
SUM (207)	561	**81**

iPDB_iterations	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	186	**132**
elevators-opt11-strips (20)	181	**20**
floortile-opt11-strips (10)	179	**154**
nomystery-opt11-strips (20)	**114**	210
openstacks-opt11-strips (20)	20	20
parcprinter-opt11-strips (7)	**80**	112
parking-opt11-strips (20)	**20**	360
scanalyzer-opt11-strips (20)	**110**	119
sokoban-opt11-strips (20)	**82**	119
tidybot-opt11-strips (20)	**132**	213
transport-opt11-strips (20)	**106**	145
visitall-opt11-strips (20)	**61**	166
woodworking-opt11-strips (2)	23	**18**
SUM (207)	**1294**	1788

iPDB_max_pdb_size	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	1336	**208**
elevators-opt11-strips (20)	132783	**308**
floortile-opt11-strips (10)	**0**	712
nomystery-opt11-strips (20)	**0**	1740
openstacks-opt11-strips (20)	1230	**60**
parcprinter-opt11-strips (7)	5886	**4368**
parking-opt11-strips (20)	1060	1060
scanalyzer-opt11-strips (20)	4884	**2956**
sokoban-opt11-strips (20)	44660	**33360**
tidybot-opt11-strips (20)	2044	**992**
transport-opt11-strips (20)	172271	**4335**
visitall-opt11-strips (20)	1660	**8**
woodworking-opt11-strips (2)	140	**84**
SUM (207)	367954	**50191**

iPDB_num_patterns	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	206	**124**
elevators-opt11-strips (20)	254	**0**
floortile-opt11-strips (10)	313	**144**
nomystery-opt11-strips (20)	244	**190**
openstacks-opt11-strips (20)	390	**0**
parcprinter-opt11-strips (7)	193	**105**
parking-opt11-strips (20)	340	340
scanalyzer-opt11-strips (20)	522	**99**
sokoban-opt11-strips (20)	145	**99**
tidybot-opt11-strips (20)	**192**	193
transport-opt11-strips (20)	211	**125**
visitall-opt11-strips (20)	794	**146**
woodworking-opt11-strips (2)	51	**16**
SUM (207)	3855	**1581**

iPDB_rejected	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	376	**0**
elevators-opt11-strips (20)	0	0
floortile-opt11-strips (10)	2814	**0**
nomystery-opt11-strips (20)	29	**0**
openstacks-opt11-strips (20)	0	0
parcprinter-opt11-strips (7)	0	0
parking-opt11-strips (20)	0	0
scanalyzer-opt11-strips (20)	322	**0**
sokoban-opt11-strips (20)	0	0
tidybot-opt11-strips (20)	0	0
transport-opt11-strips (20)	0	0
visitall-opt11-strips (20)	0	0
woodworking-opt11-strips (2)	0	0
SUM (207)	3541	**0**

iPDB_size	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	1422232	**888**
elevators-opt11-strips (20)	144756	**0**
floortile-opt11-strips (10)	10370944	**576**
nomystery-opt11-strips (20)	320400	**294764**
openstacks-opt11-strips (20)	1170	**0**
parcprinter-opt11-strips (7)	4492	**1878**
parking-opt11-strips (20)	9280	9280
scanalyzer-opt11-strips (20)	2314872	**994**
sokoban-opt11-strips (20)	3852	**1778**
tidybot-opt11-strips (20)	1628	**1286**
transport-opt11-strips (20)	60427	**2031**
visitall-opt11-strips (20)	2992	**2266**
woodworking-opt11-strips (2)	503	**93**
SUM (207)	14657548	**315834**

iPDB_time	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	3068.82	**3.85**
elevators-opt11-strips (20)	75.54	**0.20**
floortile-opt11-strips (10)	3021.93	**6.13**
nomystery-opt11-strips (20)	**9.33**	9.52
openstacks-opt11-strips (20)	1.64	**0.27**
parcprinter-opt11-strips (7)	894.60	**2.59**
parking-opt11-strips (20)	**53.98**	449.72
scanalyzer-opt11-strips (20)	84.60	**49.91**
sokoban-opt11-strips (20)	63.54	**3.75**
tidybot-opt11-strips (20)	287.68	**201.38**
transport-opt11-strips (20)	74.58	**3.00**
visitall-opt11-strips (20)	6.82	**2.34**
woodworking-opt11-strips (2)	1298.80	**0.37**
SUM (207)	8941.86	**733.03**

initial_h_value	WORK-plan11	WORK-plan31
barman-opt11-strips (4)	**79**	36
elevators-opt11-strips (9)	**244**	0
floortile-opt11-strips (2)	**43**	38
nomystery-opt11-strips (14)	**232**	231
openstacks-opt11-strips (14)	0	0
parcprinter-opt11-strips (7)	**4004359**	2975592
parking-opt11-strips (5)	61	61
scanalyzer-opt11-strips (9)	**194**	52
sokoban-opt11-strips (18)	**246**	63
tidybot-opt11-strips (14)	**114**	113
transport-opt11-strips (6)	**712**	50
visitall-opt11-strips (9)	**91**	50
woodworking-opt11-strips (2)	**360**	205
SUM (113)	**4006735**	2976491

memory	WORK-plan11	WORK-plan31
barman-opt11-strips (8)	**10151876**	10469664
elevators-opt11-strips (20)	**11124968**	26623256
floortile-opt11-strips (10)	**12300728**	16428284
nomystery-opt11-strips (20)	**8719552**	13088256
openstacks-opt11-strips (20)	17888272	**17883680**
parcprinter-opt11-strips (9)	**116152**	4645504
parking-opt11-strips (20)	31672172	**31666792**
scanalyzer-opt11-strips (20)	**22224944**	25983052
sokoban-opt11-strips (20)	**590360**	9115200
tidybot-opt11-strips (20)	**13647648**	13658024
transport-opt11-strips (20)	**28681632**	29226168
visitall-opt11-strips (20)	**16420748**	22899824
woodworking-opt11-strips (2)	**7228**	62304
SUM (209)	**173546280**	221750008

□

search_time	WORK-plan11	WORK-plan31
barman-opt11-strips (4)	91.09	**37.41**
elevators-opt11-strips (9)	**0.97**	14.07
floortile-opt11-strips (2)	**2.05**	4.62
nomystery-opt11-strips (14)	**0.16**	0.34
openstacks-opt11-strips (14)	4.69	**4.62**
parcprinter-opt11-strips (7)	**0.31**	0.34
parking-opt11-strips (5)	**5.00**	5.42
scanalyzer-opt11-strips (9)	**1.31**	15.13
sokoban-opt11-strips (18)	**0.48**	1.80
tidybot-opt11-strips (14)	1.75	**1.62**
transport-opt11-strips (6)	**0.75**	2.13
visitall-opt11-strips (9)	**0.10**	0.22
woodworking-opt11-strips (2)	**0.43**	1.26
GEOMETRIC MEAN (113)	**1.17**	2.54

search_error	WORK-plan11	WORK-plan31
barman-opt11-strips (20)	16	16
elevators-opt11-strips (20)	**4**	11
floortile-opt11-strips (20)	18	18
nomystery-opt11-strips (20)	**4**	6
openstacks-opt11-strips (20)	6	6
parcprinter-opt11-strips (20)	13	**9**
parking-opt11-strips (20)	15	15
pegsol-opt11-strips (20)	20	**3**
scanalyzer-opt11-strips (20)	**10**	11
sokoban-opt11-strips (20)	**0**	2
tidybot-opt11-strips (20)	6	6
transport-opt11-strips (20)	14	14
visitall-opt11-strips (20)	**8**	11
woodworking-opt11-strips (20)	18	**15**
SUM (280)	152	**143**

total_time	WORK-plan11	WORK-plan31
barman-opt11-strips (4)	267.79	**37.78**
elevators-opt11-strips (9)	**2.79**	14.11
floortile-opt11-strips (2)	11.75	**4.85**
nomystery-opt11-strips (14)	**0.40**	0.51
openstacks-opt11-strips (14)	4.81	**4.67**
parcprinter-opt11-strips (7)	41.51	**0.68**
parking-opt11-strips (5)	**8.06**	14.95
scanalyzer-opt11-strips (9)	**8.90**	15.78
sokoban-opt11-strips (18)	**1.72**	2.46
tidybot-opt11-strips (14)	16.30	**10.63**
transport-opt11-strips (6)	**1.08**	2.18
visitall-opt11-strips (9)	**0.16**	0.23
woodworking-opt11-strips (2)	614.44	**1.45**
GEOMETRIC MEAN (113)	7.24	**3.63**

5. Results Discussion

	without cost-partitioning I	zero-one-cost-partitioning II
initial parameter	almost no differences	minor variations
iPDB: iterations	-	more (alternating for benchmarks)
iPDB: size	-	always, much smaller
iPDB: improvement	-	mostly 0 but if not, then greater
iPDB: generated	-	less (alternating for benchmarks)
iPDB: rejected	some	0
iPDB: max_pdb_size	-	less
iPDB: num_patterns	-	mostly less
iPDB: rejected	sometimes greater 0	always 0
iPDB: time	-	much less
initial h-value	always greater	-
coverage	-	slightly more (caused by one benchmark)
dead ends	fewer dead-ends	-
expansions	less	-
generated	less	-
search time	less	-
total time	-	less overall (but alternating benchmarks)

(a) One can observe that the **iPDB construction time** (including hill-climbing) **was much shorter with the 0-1**, but the **actual search time was longer** (except in 3 benchmarks: barman, openstacks, tidybot), indicating that the **canonical heuristics had a better quality overall. But the total time was still shorter for the 0-1.**

(b) more search-nodes generated and expanded in 0-1, and smaller

iPDB size.

(c) The different initial parameters showed almost no differences for 0-1, **and minor variations in the original.** This indicates that the size limits were rarely reached**, which can be confirmed**, by the fact**, that in 0-1, Patterns were never rejected, and 0-1 iPDB size was mostly smaller.** There haven been more expansions in 0-1, and thus also more dead ends.

(d) In elevator and openstacks: the initial h-value for 0-1 was zero, that's why the iPDB time is almost 0, thus the guided search, was very long, and overall time was also much longer. **The problem is that with 0-1, the initial candidate patterns are just the goal variables, and may all be reachable with zero-action-costs**, which will lead to no improvement, and a entirely uninformed heuristic.

(e) It's interesting that in the parking-opt, almost all the attributes (coverage, size, search time, expansions...) were the same, but the iPDB time was much longer for 0-1, (as one of the few benchmarks where iPDB time was longer in 0-1)**. So the hill-climbing took more time with the 0-1-cost-partitioning in that benchmark (since the iPDB iterations were more for 0-1).**

(f) Though 0-1 had more expansions, it needed more in search time. Since iPDB size was always smaller in 0-1, the **PDB size** seems a better **measure for the quality of the heuristic**.

(g) Example, where total time of canonical was much shorter: If one looks at the **elevator-benchmarks** more closely, one can see that in 0-1 less problems were covered, iPDB size was mostly zero, iPDB time was thus smaller and search time was much longer, caused by the bad heuristic.

(h) Example, where total time of zero was much shorter: In the **woodworking benchmark**, the coverage was slightly bigger, iPDB size slightly less, but iPDB time much less, and search-time slightly more. iPDB iterations about the same.

Conclusion

Constructing PDBs using cost-partitoning is a rather unexplored field.

We have compared the standard algorithm by Haslum et. Al (2007), which uses the unchanged original-operatorcosts, with a simple zero-one-costpartitioning. We have seen, that the 0-1 is a rather simple method to guarantee additivity, thus the PDB construction time is much shorter compared with the canonical heuristic. For the same reason, the quality of the 0-1 heuristic is not so well as the canonical, leading to a longer search-time. In our experiments, the total

time was still shorter for the 0-1, but one can expect that this changes, as the search space grows. Mostly the 0-1 PDB had a much smaller size. In one of the few benchmarks, where 0-1 was actually equal in PDB size, the hill-climbing also took longer, but this may be benchmark-related.

But we have seen that not all the benchmarks used, delievered the same comparisson for the search time/ hill-climbing time though. One may assume, that the 0-1 heuristic is better fitted for some special sort of domains. However there have also been, obvious problems, like the initial h values of 0, caused by the fact, that all goal variables, may be reachable within the first step. In future implentations, this should be dealt as a special case, avoiding the hill-climbing to break up immediately.

The problem seems to be, that the zero-one-cost-partitioning is too simplistic, for the sometimes rather complex search spaces, and the results are thus not predictable. But it may be, that even a cost-partitioning that makes more assumptions about the search problem (which is not easy, since the procedure should be domain-independent), will have rather random results.

Other challenges for the future may be:

Since the cost-partitioning may save time, in the PDB construction, one could also try more sophisticated approaches, then the 0-1 approach (e.g. weighted heuristics), also one could run the hill-climbing with normal Haslum method, and then try to find a Cost-Partitioning that will improve our obtained heuristic.

References

Helmert, M. (2006). The Fast Downward Planning System. Journal of Artificial Intelligence Research 26, pp. 191-246.

Culberson, J. C., & Schaeffer, J. (1998). Pattern databases. Computational Intelligence, 14(3), pp. 318–334.

Edelkamp, S. (2001). Planning with pattern databases. In Proceedings of the 6th European Conference on Planning, pp. 13–24.

Haslum, P., Botea, A., Helmert, M., Bonet, B., & Koenig, S. (2007). Domain-independent construction of pattern database heuristics for cost-optimal planning. In Proceedings of The Twenty-Second National Conference on Artificial Intelligence (AAAI-07), pp. 1007–1012.

Katz, M, & Domshlak, C. (2008). Optimal Additive composition of abstraction-based admissible heuristics. ICAPS-08. 18th International Conference on Automated Planning and Scheduling. pp. 174-181.