Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)

Value Compression of Pattern Databases

Nathan R. Sturtevant
Computer Science Department
University of Denver
sturtevant@cs.du.edu

Ariel Felner
ISE Department
Ben-Gurion University
Be'er-Sheva, Israel
felner@bgu.ac.il

Malte Helmert
Dept. of Math. and Computer Science
University of Basel
Switzerland
malte.helmert@unibas.ch

Abstract

One common pattern database compression technique is to merge adjacent database entries and store the minimum of merged entries to maintain heuristic admissibility. In this paper we propose a compression technique that preserves every entry, but reduces the number of bits used to store each entry, therefore limiting the values that can be represented. Even when this technique throws away low values in the heuristic, it can still have better performance than the traditional approach. We develop a theoretical basis for selecting which values to keep and show improved performance in both unidirectional and bidirectional search.

1 Introduction

On approach to improving heuristic search algorithms, used to find a paths between states in a state space, is to improve the heuristic used to guide the search. Algorithms in the A* family use a cost function of $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the least-cost known path to n and $h(n)$ is an admissible (lower bound) estimation on the remaining cost to the goal.

Pattern Databases (PDBs) (Culberson and Schaeffer 1996) are *memory-based heuristics* which store lookup tables in memory containing the distance to the goal in an abstracted state space. If the size of the lookup table exceeds the size of main memory, it is usually compressed to fit in memory. The most common class of lossy compression techniques for memory-based heuristics (Felner et al. 2007), denoted here as *entry compression* (EC), compresses a group of entries into a single entry by storing the minimum of the group, guaranteeing admissibility. Entry compression is very effective in many domains, but it is also very sensitive to how entries are grouped together, which controls the loss of information from compression.

In this paper we introduce an orthogonal compression technique called *Value Compression* (VC). PDB entries are traditionally stored using sufficient bits to represent every possible value in the PDB. Unlike EC, VC retains all entries in the PDB but reduces the number of bits used to store each entry. VC partitions the range of values into subranges and stores one value (the minimum) for each subrange. VC can be used alone or together with EC.

This paper makes the following contributions. First, we introduce and describe the idea of VC. Second, we present an efficient algorithm that, given a distribution of heuristic values, partitions the VC ranges in a way that maximizes the average heuristic value. Third, we present extensive experimental results that illustrate the strengths and weaknesses of VC. We show that the static and dynamic distribution of heuristic values of the PDB are key for predicting the usefulness of VC. When they correlate VC is more effective. VC is particularly effective when the majority of values in the PDB fall in a small range, as it can eliminate many of the other values and still perform well. When heuristics do not fall in a small range, we show that the distribution of values in a heuristic can be shifted with *delta heuristics*. Finally, we show that VC is particularly effective in the new meet-in-the-middle (MM) family of bidirectional search algorithms (Holte et al. 2016) where only high values matter. We provide experimental results that support these trends.

2 Background

A heuristic h is *consistent* if, for all states a and b, $h(a) \leq c(a, b)+h(b)$, where $c(a, b)$ is the cost of a shortest path from a to b (Felner et al. 2011). Inconsistent heuristics can cause complications during search, but these can be remedied by using *pathmax* or *bidirectional pathmax* (BPMX) (Felner et al. 2005) which propagate heuristic values along edges to smooth local inconsistencies.

A pattern database (PDB) (Culberson and Schaeffer 1996) is an admissible heuristic built by abstracting the underlying state space S into an abstract state space S', usually by ignoring some of the details of states in S. The abstract state-space is usually exponentially smaller than the underlying search space. The shortest (abstract) distance from any abstract state $s' \in S'$ to the abstract goal is computed and then stored in a lookup table – the *pattern database*. When the search algorithm arrives at a state s, s is then abstracted into s', and s' is used as an index into the PDB. The value stored in *PDB*(s') is used as an admissible heuristic for s.

PDBs are usually generated in a preprocessing phase and stored in a lookup table of h-values, with one entry per abstract state, using a *ranking function* (e.g., Myrvold and Ruskey 2001) which maps abstract states to consecutive integers. Once the PDB is created, it can be used to provide heuristics for an infinite number of problems, either to the same specific goal or to other goal states that can also benefit from the same PDB (e.g., by symmetries etc).

2.1 Compressed PDBs

When a PDB is too large, it can be compressed to fit into memory, using either lossy or lossless approaches.

A common *lossy compression* method (Felner et al. 2007) is denoted here by *entry compression*. To compress a PDB with E entries by a factor of f, the entire PDB is divided into E/f buckets, each containing f entries. The compressed PDB is of size E/f and only stores one value for each bucket. To guarantee admissibility, the *minimum* value of all entries in the bucket in the original PDB is stored (hence the term *entry compression* (EC)). The main challenge is to minimize the *loss of information* – we want the compressed heuristic to be as close as possible to the original uncompressed heuristic. Buckets can be determined deterministically, e.g., using *mod* or *div* operators to map states into buckets (Felner et al. 2007) or randomly, as done in Compressed Partial Pattern Databases (Anderson, Holte, and Schaeffer 2007). *Div* compresses adjacent entries while *mod* compresses entries that are E/f entries apart. It is important to note that EC might result in an inconsistent heuristic even if the original heuristic was consistent because two neighboring nodes might have heuristic values there were compressed to different buckets.

In *lossless compression*, the challenge is to minimize the memory needs per entry while preserving the original values and keeping a reasonable time overhead for the unpacking. Felner et al. (2007) describe a lossless compression approach which stores the delta over an entry-compressed PDB. Another lossless compression technique is *1.6 bit PDBs* (Breyer and Korf 2010) that store the PDB values modulo 3. This value essentially stores whether a state's heuristic is larger, smaller or equal to the parent. 1.6 bit PDBs can only be used on top of a consistent heuristic and only for undirected and unweighted state spaces.

2.2 Delta Heuristics

Delta heuristics ($h_\triangle$) have been used as implementation tricks, but have not been deeply studied. Let h_1 and h_2 be two heuristic functions where h_1 *weakly dominates* h_2, i.e., $h_1(s) \geq h_2(s)$ for all states s. Then we can define a (non-negative) *delta heuristic* ($h_\triangle$) by $h_\triangle(s) = h_1(s) - h_2(s)$. Clearly, we can exactly recover h_1 using $h_1(s) = h_2(s) + h_\triangle(s)$. A common $h_\triangle$ is a PDB where the pattern used for h_2 is a strict subset of the pattern used for h_1. A PDB for the 15-puzzle which stores the delta over Manhattan distance (MD) is a known form of lossless compression to reduce the number of bits per PDB entry (Felner, Korf, and Hanan 2004; Felner and Adler 2005; Samadi et al. 2008).

2.3 Meet-in-the-Middle Bidirectional Search

Bidirectional search algorithms interleave a search forward from the start state ($start$) and a search backward (i.e. using reverse operators) from the goal state ($goal$). There is a long history of research into bidirectional search algorithms (Nicholson 1966; Pohl 1969), justified by the potential for an exponential reduction in the size of a bidirectional search over a unidirectional search.

The `MM` algorithm (Holte et al. 2016) is a recently introduced bidirectional search algorithm. `MM` runs an A*-like search in both directions but prioritizes nodes in its open-lists with the priority function $pr(n) = \max(g(n) + h(n), 2g(n))$ This priority function guarantees that the search frontiers will *meet in the middle*. That is, the search will not expand any nodes with g-cost that exceeds $C^*/2$ in either direction, where C^* is the optimal solution cost. Because it meets in the middle, `MM` treats the g-cost of a node as another heuristic; the priority rule can be re-written as $pr(n) = g(n) + \max(h(n), g(n))$. When $g(n) \geq h(n)$ then $h(n)$ does not provide any guidance to the search as it is dominated by $g(n)$. In order to prune a node because of the heuristic (i.e., to have $pr(n) > C^*$) it must be that $h(n) > C^*/2$. Heuristic values smaller than $C^*/2$ can be treated as 0. VC can take advantage of this fact and compress small values to 0 with no loss in performance.

2.4 Top Spin Domain

The (N,K)-TopSpin puzzle has N tokens arranged in a ring. Any set of K consecutive tokens can be reversed (rotated 180 degrees in the physical puzzle). The goal state has the tokens sorted. A q-token PDB abstracts the puzzle by keeping the first q tokens and abstracting away the rest. This puzzle will be used as our main testbed because tuning N, K, and q can create state spaces with different properties.

3 Value Compression

Consider the distribution of values for an 8-token PDB of the (18,4)-TopSpin puzzle shown in Table 1 (column *Total*). This PDB has 1.76 billion entries ranging from 0 to 17. Storing this PDB losslessly requires at least 5 bits per entry. In practice, for the purpose of speed and simplicity of memory access, implementers usually round up the number of bits to the nearest power of 2, requiring 8 bits per entry in this case. In this particular PDB, 4 bits would be sufficient for all values from 0 to 15, and the other 4 bits are used just to be able to store heuristic values of 16 and 17, which are very rare (70,654 entries, only 0.004% of the total PDB). Although, we cannot efficiently remove just these entries from the PDB, we can compress the entire PDB by replacing these values with 15 as shown in the $\underline{VC2}$ column of Table 1. This reduces the number of bits used for the PDB from 8 to 4, halving memory requirements while hardly affecting the average heuristic value (it decreases by less than 0.0001).

Consider instead what happens when we apply EC using *div* to compress the same PDB by a factor of 2 as shown in the *EC2* column. To compare with the other columns, the numbers in that column are the total number of entries in the original PDB that are mapped to the given value (because EC2 groups pairs of entries together, only even numbers are seen). The average value here is 11.59, much worse than $\underline{VC2}$ while using the same amount of memory. This shows the potential benefit of VC.

	1,760MB	880MB			440MB	
D	Total	$\underline{VC2}$	VC2$_{\bar{h}}$	EC2	VC4$_{\bar{h}}$	EC4
0	1	1	12	2	10,188,753	4
1	11	11		22		40
2	94	94	94	186		340
3	731	731	731	1,430		2,596
4	5,353	5,353	5,353	10,340		18,736
5	37,275	37,275	37,275	70,894		127,756
6	245,468	245,468	245,468	457,304		813,700
7	1,508,099	1,508,099	1,508,099	2,722,458		4,724,408
8	8,391,721	8,391,721	8,391,721	14,408,820		23,870,392
9	40,012,497	40,012,497	40,012,497	63,502,746	190,013,262	97,318,252
10	150,000,765	150,000,765	150,000,765	212,692,340		290,434,356
11	393,482,172	393,482,172	393,482,172	478,114,034	393,482,172	553,276,900
12	612,084,904	612,084,904	612,084,904	601,419,722	1,170,638,373	549,750,508
13	440,655,534	440,655,534	440,655,534	328,304,534		217,340,348
14	110,437,757	110,437,757	110,437,757	59,883,892		26,009,144
15	7,389,524	7,460,178	7,389,524	2,721,910		634,464
16	**70,633**		70,654	11,924		616
17	**21**			2		
Avg.	11.90	11.90	**11.90**	11.59	**11.38**	11.27

Table 1: (18-4)-TopSpin. Distribution of values for different types of compression

3.1 General Description of Value Compression

Using b bits per PDB entry, we can store 2^b different heuristic values. The main idea in VC is to compress a range of values together, storing the minimal value of each range to preserve admissibility. In the $\underline{VC2}$ example above, we compress the range $\{15, \dots, 17\}$ to the value 15.

In general, let R be the range of distinct h values of a memory-based heuristic such as a PDB. To store this, we need $\lceil \log_2 |R| \rceil$ bits per entry. VC partitions range R into M disjoint contiguous subranges $R = R_1 \cup \cdots \cup R_M$. By "contiguous" we mean that no value of a given range falls between the minimum and maximum value in another range. In the compressed PDB we only store the identity of each subrange for each entry, requiring $\lceil \log_2 M \rceil$ bits. Of course this loses some information: if the heuristic lookup determines that the heuristic value falls into range R_i, we have to use the minimum value in R_i as an admissible heuristic. For example, assume $R = \{0, \dots, 99\}$. Without compression, we need at least 7 bits per entry. One possible value compression groups any 10 consecutive values together: $R_1 = \{0, \dots, 9\}$, $R_2 = \{10, \dots, 19\}$, ..., $R_{10} = \{90, \dots, 99\}$. This reduces space usage to 4 bits per entry at the loss of some heuristic accuracy (e.g., all heuristic values between 30 and 39 are compressed to 30). We expect VC to be particularly effective when few bits can be used to capture the majority of the values in the PDB. Similar to EC, VC may cause the heuristic to be inconsistent if two neighboring states are mapped to different ranges.

VC can be seen as generalizing the idea of partial pattern databases (PPDB) (Anderson, Holte, and Schaeffer 2007; Edelkamp and Kissmann 2008), which only store heuristic values up to a threshold V, assigning a heuristic value of $V + 1$ to all other entries.

3.2 General Optimized Range Partitioning

VC is flexible regarding which values to group together. There are $\frac{(|R|-1)!}{(|R|-M)!(M-1)!}$ ways to partition a range R into M nonempty contiguous subranges. (One range must start at 0 to preserve admissibility.) Which one should be used? We define an *optimal partition* as one that maximizes the average heuristic value of the compressed heuristic among all possible partitions into M subranges. This is equivalent to the minimal average *loss of information* over all values in the compressed PDB.

We now descibe how to compute an optimal partition, which we denote by $VC_{\bar{h}}$, in time polynomial in $|R|$ and M. Consider an arbitrary contiguous partition $\mathcal{P} = \{R_1, \dots, R_M\}$ of the range R. For $k \in R$, Let $N(k)$ denote the number of PDB entries with heuristic value k. The *quality* (= cumulative heuristic value) under partition $\mathcal{P}$ is:

$$\textit{Quality}(\mathcal{P}) = \sum_{R_i \in \mathcal{P}} \left(\sum_{k \in R_i} N(k) \cdot \min R_i \right). \qquad (1)$$

In words, we sum over all subranges, and for each subrange R_i we count how many PDB entries fall into R_i and multiply the total by the value stored for this range ($\min R_i$). Optimizing quality is equivalent to optimizing the average heuristic value, as the average is the quality divided by the number of PDB entries. The quality metric is easier to work with than the average because it is additive: if we divide a partition into two parts $\mathcal{P} = \mathcal{P}' \cup \mathcal{P}''$, then we have

$$\textit{Quality}(\mathcal{P}) = \textit{Quality}(\mathcal{P}') + \textit{Quality}(\mathcal{P}''). \qquad (2)$$

This additivity property suggests a dynamic programming approach for finding a partition into M subranges that maximizes quality. For any set X and value $k \in X$, we define $X_{\leq k} := \{x \in X \mid x \leq k\}$ and $X_{>k} := \{x \in X \mid x > k\}$. Every contiguous partition $\mathcal{P}$ of R into $M \geq 2$ subranges

Algorithm 1: Optimal Partitioning

```
OptPart(max_h, M)
    for $s$ = 0 to max_h do
        Pivot[s, 1] = max_h;
        Qual[s, 1] = CalcQ(s, max_h);
    end
    for $m$ = 2 to $M$ do
        for $s$ = 0 to max_h $+ 1 - m$ do
            bestQ = $-\infty$;
            for $p$ = $s$ to max_h $+ 1 - m$ do
                currQ = CalcQ(s, p) + Qual[$p + 1, m - 1$];
                if currQ > bestQ then
                    bestQ = currQ;
                    pivot = p;
                end
            end
            Qual[s, m] = bestQ;
            Pivot[s, m] = pivot;
        end
    end
end
```

can be written as $\mathcal{P} = \{R_{\leq p}\} \cup \mathcal{P}'$ where p is the largest value of the smallest subrange of $\mathcal{P}$ (we call this the *pivot* of $\mathcal{P}$), and $\mathcal{P}'$ is a contiguous partition of $R_{>p}$.

If $\mathcal{P}$ is an *optimal* partition (one that maximizes $\mathit{Quality}(\mathcal{P})$ over all partitions of R into M subranges), then from $\mathit{Quality}(\mathcal{P}) = \mathit{Quality}(\{R_{\leq p}\}) + \mathit{Quality}(\mathcal{P}')$, we get that $\mathcal{P}'$ must be an *optimal* partition of $R_{>p}$ into $M-1$ subranges. Otherwise, the quality of $\mathcal{P}$ could be improved by replacing this subpartition by another one of higher quality.

For a general range R and $M \geq 1$, let $\mathit{OptPart}(R, M)$ denote an optimal partition of R into at most M subranges. Clearly, $\mathit{OptPart}(R, 1) = \{R\}$ for all R. Another base case is $\mathit{OptPart}(\emptyset, M) = \emptyset$ for all M. For $R \neq \emptyset$ and $M > 1$, we obtain $\mathit{OptPart}(R, M)$ by computing $\mathit{Quality}(\{R_{\leq p}\}) + \mathit{Quality}(\mathit{OptPart}(R_{>p}, M-1))$ for all possible $p \in R$. If $p \in R$ maximizes this quantity, we can set $\mathit{OptPart}(R, M) = \{R_{\leq p}\} \cup \mathit{OptPart}(R_{>p}, M-1)$. That is, an optimal solution can be obtained by trying out all possible pivots p, recursively computing an optimal partition for each subproblem, and selecting the best partition among these candidates.

It is easy to see that all subproblems generated when computing $\mathit{OptPart}(R, M)$ in this fashion are of the form $\mathit{OptPart}(R_{>p}, M')$ for some $p \in R$ and some $M' \in \{1, \ldots, M\}$, and hence the total number of subproblems is bounded by $|R| \cdot M$, giving rise to a dynamic programming algorithm with runtime polynomial in $|R|$ and M.

Algorithm 1 gives pseudo-code for this algorithm. We assume $R = \{0, \ldots, \mathit{max\_h}\}$ for some number *max_h*, but the algorithm can be easily adapted to arbitrary ranges. $\mathit{Pivot}[s, m]$ and $\mathit{Qual}[s, m]$ store the pivot and quality of partitioning the range $\{s, \ldots, \mathit{max\_h}\}$ into m subranges. Subprocedure *CalcQ(s, u)* calculates the quality for the subrange $\{s, \ldots, u\}$, which is $\sum_{k=s}^{u} N(k) \cdot s$. The optimal partition is then obtained by collecting the pivots from the array.

Table 1 also shows the optimal value compression for

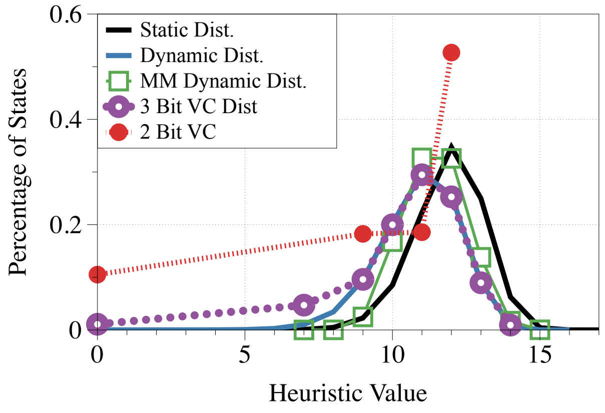

Figure 1: Distribution curves for (18-4)-TopSpin

Memory		EC	VC	VC-bits	Nodes	Time
1		1	1	8	3.88M	15.29
0.5	(A)	1	2	4	**3.88M**	**15.32**
0.375		1	2.66	3	4.03M	15.44
0.25	(B)	1	4	2	10.39M	33.63
0.5	(A)	2	1	8	7.11M	27.70
0.25	(B)	2	2	4	**7.11M**	**27.88**
0.1875		2	2.66	3	7.37M	28.44
0.125	(C)	2	4	2	30.43M	80.04
0.25	(B)	4	1	8	13.75M	51.06
0.125	(C)	4	2	4	**13.74M**	**50.97**
0.094		4	2.66	3	14.31M	51.52
0.0625		4	4	2	30.48M	77.68

Table 2: Results for (18-4)-TopSpin

our 8-tile PDB. Column VC$2_{\bar{h}}$ represents compression by a factor of 2, i.e., to 4 bits ($M = 2^4 = 16$), and column VC$4_{\bar{h}}$ is compression by a factor of 4, i.e., to 2 bits ($M = 2^2 = 4$). We see that for VC$2_{\bar{h}}$, it is optimal to use the subranges $\{0, 1\}, \{16, 17\}$ and singleton ranges $\{2\}, \ldots, \{15\}$, while for VC$4_{\bar{h}}$ the optimal subranges are $\{0, \ldots, 8\}$, $\{9, 10\}$, $\{11\}$ and $\{12, \ldots, 17\}$. The average heuristic values for VC$2_{\bar{h}}$ and VC$4_{\bar{h}}$ are 11.90 and 11.38, both significantly better than entry compression with the same amount of memory (columns EC2 and EC4).

4 Heuristic Distributions

Two different types of heuristic distributions are described in the literature (Holte et al. 2006; Felner et al. 2005). In the context of PDBs the *static distribution* is the distribution of values in the PDB, while the *dynamic distribution* is the distribution of the heuristic values that are seen during the process of solving a given problem instance.

For unidirectional searches such as A* or IDA*, the dynamic distribution may contain lower values than the static distribution (see for example (Felner et al. 2011), figure 24, pp 1592). Holte et al. (2006) explained that for a given f-value, the search tree expanded by IDA* contains many more nodes with large g-values and small h-values compared to nodes with small g-values and large h-values (of the same f-cost) as there are exponentially more nodes with high g-values. VC$_{\bar{h}}$ is calculated (trained) on the static distribution and this should usually be done in the preprocessing phase when the PDB is built.

4.1 Calculating the Distributions

The user certainly knows the static distribution from the PDB and can implement VC$_{\bar{h}}$ in the preprocessing phase. A key point is that the effectiveness of VC$_{\bar{h}}$ will increase when the dynamic distribution correlates with the static distribution. In this case the optimization of values has selected the values that are most important for effective search. Indeed, in many cases the static and dynamic distributions are known to be close together. For example, using multiple PDBs (Holte et al. 2006) or using inconsistent heuristics and BPMX (Felner et al. 2011) causes the dynamic distribution to be very close to the static distribution. Therefore, VC$_{\bar{h}}$ may be safely used if the two distributions are known to correlate. But, when the distributions are far apart, the effectiveness of VC$_{\bar{h}}$ will be weakened. In the remaining sections we demonstrate this on a number of search scenarios.

If nothing is known about the dynamic distribution one will need to solve or to sample a set of representative problem instances in order to learn the dynamic distribution in a preprocessing phase. Our optimization algorithm may be applied on top of the dynamic distribution if it is known.

5 Experiments: VC in Unidirectional Search

To begin, we experiment with IDA* (+BPMX) on (18,4)-TopSpin with a heuristic that takes the maximum of three 8-token PDBs. All the results reported below are averages over 50 random instances which were created by 200 random steps. Running time is reported in seconds.

Five curves are shown for this domain in Figure 1. Notice that the static distribution of values in the PDB (black curve) and the dynamic distribution of an IDA* search with that PDB (blue curve) follow closely. The dotted purple curve corresponds to the values that are seen during the search when using $VC_{\bar{h}}$ with 8 values (3 bits). This curve tends to correlate with the dynamic distribution and this is a clear sign that $VC_{\bar{h}}$ will work well here. Not shown is the curve with 16 values (4 bits) which is almost identical to the dynamic distribution. The red curve corresponds to the values that are seen during the search when using $VC_{\bar{h}}$ with 4 values (2 bits). Clearly, now, there is loss of information which predicts that compressing to 2 bits will be significantly weaker compared to 3 or more bits. Nevertheless, we will compare this to EC shortly.

The rows of Table 2 combine EC and $VC_{\bar{h}}$ in different ways. The first column gives the memory needs relative to the basic PDB with no compression (using 8 bits per entry).[1] Each row is uniquely defined by the EC and VC compression factors given in the second and third columns (in **bold**). The

[1]Breyer and Korf (2010) note that researchers have traditionally used powers of two when storing PDBs. Our results in Table 2 are under this assumption.

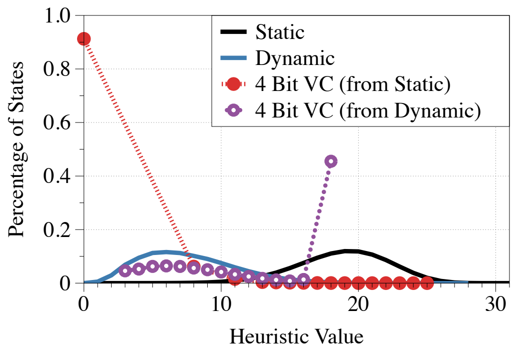

Figure 2: Distributions curves for (18-2)-TopSpin

fourth column gives the number of bits used by the VC compression. The last two columns give the number of nodes expanded and the running time in seconds. Together, these confirm the trends predicted in Figure 1. For a fixed value of EC it can be seen that using $VC_{\bar{h}}$ and going down from 8 bits to to 4 bits (16 values) and 3 bits (8 values) is beneficial and the loss of information is small. While there is a jump when going down to 2 bits (4 values), comparing 4x VC (10.39M nodes) to 4x EC (13.75M nodes) still shows an advantage for VC. This may be a surprising result, as this heuristic has no values between 1 and 8. But, BPMX is sufficient to limit the overhead of this compression. (Without BPMX the number of node expansions increases by two orders of magnitude.)

Rows with memory 0.5, 0.25 and 0.125 are labeled with A, B and C, respectively. For each of these groups, the best variant is in bold. For example, for 0.5 memory (group A, compression factor of 2) $VC_{\bar{h}}$ (3.88M) outperforms EC (7.11M). In fact, even compressing to 0.375 memory (3 bits) with $VC_{\bar{h}}$ outperforms EC2 both in memory and in runtime. Similarly, for 0.25 when only one method is used, $VC_{\bar{h}}$ (10.39M) outperforms EC (13.75M). For both 0.25 and 0.125, it is always better to compress the last factor of 2 by $VC_{\bar{h}}$ and not by EC. Further compressing this to 3 bits is still beneficial.

Next, we move to (18,2)-TopSpin. Again we use three 8-token PDBs (8 bits per entry, 32 values). Figure 2 shows that the static (black curve) and dynamic (blue curve) distributions using the original PDB differ greatly. The reason is that this domain is more difficult than the (18,4)-TopSpin and its PDB is much weaker. The red curve gives the values seen during the search by $VC_{\bar{h}}$ with four bits trained on the static distribution. This compressed heuristic is more concentrated on high values (around 20) but these are useless in the search which encounters more low values (in the range $1 \ldots 8$). Hence, 90% of the states encountered during the search have had their heuristic value compressed to 0. The blue dotted curve corresponds to the values seen by $VC_{\bar{h}}$ with 4 bits when trained on the dynamic distribution. This has a better fit with the dynamic distribution, but too many large values are lost and performance is still poor.

Table 3 shows different ways of compressing this PDB

MEM	**EC**	**VC-bits**	Nodes	Time
1	1	8	367,225	0.44
0.5	1	4s	3,645,502	4.18
0.5	1	4d	684,846	0.94
0.5	1	4c	**394,603**	**0.43**
0.5	2	8	416,014	0.49

Table 3: Results for (18-2)-TopSpin

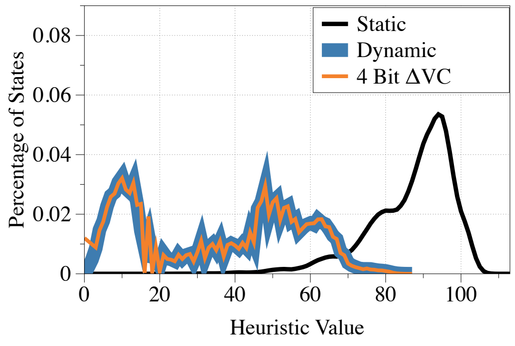

Figure 3: Distributions for the 16-peg Towers of Hanoi

with VC to 4 bits (a factor of 2). 4s/4d is $VC_{\bar{h}}$ trained on the static/dynamic distributions. As predicted 4d outperforms 4s but they are both have worse performance than EC with the same amount of memory (last line). Finally, 4c is a manually tuned range selection which achieves slightly better performance than EC.

6 VC on Top of Delta Heuristics

Recall that $h_\triangle = h_1 - h_2$ and that h_1 can be recovered by $h_1 = h_2 + h_\triangle$. Here, we briefly study the question of using $h_\triangle$ when the distribution of values in h_1 is large. The main advantage of using $h_\triangle$ is that it contains a smaller range of values than h_1. This may require fewer bits per entry and thus reduce memory. In addition, smaller ranges reduce the *loss of information* that occurs when performing EC or VC on $h_\triangle$ compared to a straight compression of the original PDB. We have experimental evidence that shows this general trend but focus here on applying VC to $h_\triangle$.

An ideal domain for using $h_\triangle$ is the 4-peg Towers of Hanoi (TOH4) (Korf and Felner 2007). The aim is to move all discs, one at a time, to the goal peg while never placing a large disc on top of a small disc. TOH4 has very long solution lengths and the range of heuristic values is very large.[2] We study the the 16-disk TOH4 problem.

h_1 is set to be a 14-disk PDB. h_2 is generated by an entry compression of a 14-disk PDB by a factor of 8192. This is equivalent to compressing the smallest 6.5 discs (Felner et al. 2007). h_2 is still very accurate because in TOH4 the loss-of-information is still very small. Nevertheless, h_1 has 113 values and its static distribution is shown in the black curve of Figure 3. By contrast, $h_\triangle$ has only 26 values ranging from 0 to 25 (not shown in the figure). There is not necessarily a correlation between large $h_\triangle$ values and large h_2 values. Therefore, the static and dynamic distributions of $h_\triangle$ are not necessarily correlated to the original PDB.

[2]TOH4 has many cycles so IDA* will not be effective here; A* with BPMX is required.

The dynamic distribution of values when using h_1 is shown in the figure (thick blue curve) and is not close to the static distribution. When we tried VC directly on h_1, it could not solve many instances within our time/memory limits because of the gap between the dynamic and static distribution. However, performing VC on $h_\triangle$ is very effective. The thin orange curve shows that the distribution of values of VC of $h_\triangle$ with 4-bits per entry (then added to h_2) matches with the dynamic distribution of the original PDB. We observed this trend in other domains as well including the 15 puzzle and TopSpin but we omit the results here.

To summarize the unidirectional research section on whether to use VC or RC we can provide the following general rule: when the static and dynamic distribution correlate $VC_{\bar{h}}$ will be very effective and will tend to outperform EC, especially when we compress small ranges and not too many values are lost.

Memory	**EC**	**VC**	VC-bits	Nodes	Time
1	1	1	8	2.07M	30.60
0.5	2	1	8	3.55M	55.61
0.5	1	2	4	**2.07M**	**30.93**
0.25	4	1	8	5.42M	83.87
0.25	2	2	4	**3.55M**	**55.55**
0.25	1	4	2	4.63M	66.46
0.125	4	2	4	5.42M	84.06
0.125	2	4	2	**5.19M**	**79.17**

Table 4: MM: (18,4)-TopSpin. EC vs. VC

Mem	**EC**	**VC**	bits	Nodes	Time	Nodes	Time
				(18-6)		(18-10)	
1	1	1	4	9.00M	163	9.11M	170
0.5	2	1	4	16.27M	316	17.30M	329
0.5	1	2	2	**12.48M**	**224**	**9.31M**	**184**
0.25	4	1	4	29.76M	636	31.93M	604
0.25	2	2	2	**17.68M**	**330**	**28.14M**	**528**
0.125	4	2	2	**30.09M**	**626**	**37.62M**	**707**

Table 5: MM: (18,6) and (18-10)-TopSpin. EC vs. VC

7 Experimental Results: VC for MM

Since MM meets in the middle, only heuristic values that are larger than $C^*/2$ may prune nodes that would otherwise be expanded. Nodes with $h(n) \leq C^*/2$ fall into two cases. If $h(n) < g(n)$, then $h(n)$ is dominated by $g(n)$ in the priority function and $h(n)$ can be treated as 0. If $g(n) \leq h(n) \leq C^*/2$, then $pr(n) = g(n) + h(n) \leq C^*$. Such nodes will be expanded at some point and their heuristic values can also be treated as 0. Therefore, for MM low heuristic values may be fully compressed away by VC without losing anything.

The green curve in Figure 1 shows the dynamic distribution of h-values that actually influenced the priority function, i.e., for nodes where $h(n) > g(n)$. Clearly, while the static heuristic (black curve) had values ranging from 0 to 17, only h-values of 7 and higher belongs to nodes with smaller g-values. Furthermore, even some of these h-values are not important if they are $\leq C^*$. The red curve ($VC_{\bar{h}}$ trained on the static distribution and compressed to 4 values (2 bits)) is biased towards the high values and all values ≤ 8 are compressed to 0. This was problematic for unidirectional search (see the fourth line in Table 2) as it missed many of the small value. For `MM` this does not hurt as small values do not matter, but the large values that really matter are better preserved.

Table 4 combines EC with the $VC_{\bar{h}}$ in different ways on the (18-4)-TopSpin domain for `MM`. Rows are grouped according to their memory usage and the best variant is shown in bold. For example, for 0.5 memory, VC was much better than EC. Similarly, for 0.125 memory, 4x compression by VC is faster than 4x compression by EC. For 0.25 memory, still VC alone (4x compression) was better than EC alone (4x compression) but it is best to combine them, each with a compression factor of 2.

Table 5 presents similar results on the (18-6)- and (18-10)-TopSpin domains. Here, the range of heuristic values was from 0 to 15 and the uncompressed heuristic needed 4 bits. Again, the best variant for a given amount of memory is in bold. VC outperformed the corresponding EC by up to a factor of 2.

It is important to note that when C^* is small many of the high values of the PDB will never be used and since we compressed away the low values then these will not be seen too. But, when C^* is small, the problems are very easy to solve and the need of a strong heuristic is less important. Strong heuristics are needed for large values of C^*.

8 Conclusions

We showed that $VC_{\bar{h}}$'s effectiveness depends on the correlation between the static and dynamic heuristic distributions. In many cases, even a sparse set of PDB values used by VC can outperform standard EC. When the static and dynamic distribution do not correlate we can either train on the dynamic distribution or use a delta PDB and compress the delta with VC. VC is also valuable in `MM` where it can compress PDB values that will never be used in practice. Ultimately, the magnitude of the gains by VC depend on many properties of each domain, but VC is an effective way of compressing away extra bits that would otherwise be wasted.

9 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1551406. Financial support for this research was in part provided by Israel Science Foundation (ISF) grant #417/13. This work was supported by the Swiss National Science Foundation (SNSF) as part of the project "Reasoning about Plans and Heuristics for Planning and Combinatorial Search" (RAPAHPACS).

References

Anderson, K.; Holte, R.; and Schaeffer, J. 2007. Partial pattern databases. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 20–34.

Breyer, T. M., and Korf, R. E. 2010. 1.6-bit pattern databases. In *AAAI Conference on Artificial Intelligence*, 39–44.

Culberson, J. C., and Schaeffer, J. 1996. Searching with pattern databases. *Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081)* 402–416.

Edelkamp, S., and Kissmann, P. 2008. Partial symbolic pattern databases for optimal sequential planning. In Dengel, A.; Berns, K.; Breuel, T. M.; Bomarius, F.; and Roth-Berghofer, T., eds., *KI*, volume 5243 of *Lecture Notes in Computer Science*, 193–200. Springer.

Felner, A., and Adler, A. 2005. Solving the 24-puzzle with instance dependent pattern databases. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 248–260.

Felner, A.; Zahavi, U.; Schaeffer, J.; and Holte, R. C. 2005. Dual lookups in pattern databases. In *International Joint Conference on Artificial Intelligence (IJCAI-05)*, 103–108.

Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. C. 2007. Compressed pattern databases. *Journal of Artificial Intelligence Research* 30:213–247.

Felner, A.; Zahavi, U.; Holte, R.; Schaeffer, J.; Sturtevant, N. R.; and Zhang, Z. 2011. Inconsistent heuristics in theory and practice. *Artif. Intell.* 175(9-10):1570–1603.

Felner, A.; Korf, R. E.; and Hanan, S. 2004. Additive pattern database heuristics. *Journal of Artificial Intelligence Research* 22:279–318.

Holte, R. C.; Felner, A.; Newton, J.; Meshulam, R.; and Furcy, D. 2006. Maximizing over multiple pattern databases speeds up heuristic search. *Artificial Intelligence* 170:1123–1136.

Holte, R.; Felner, A.; Sharon, G.; and Sturtevant, N. 2016. Bidirectional search that is guaranteed to meet in the middle. In *AAAI*.

Korf, R. E., and Felner, A. 2007. Recent progress in heuristic search: A case study of the four-peg towers of hanoi problem. In *IJCAI*, 2324–2329.

Myrvold, W., and Ruskey, F. 2001. Ranking and unranking permutations in linear time. *Information Processing Letters* 79:281–284.

Nicholson, T. A. J. 1966. Finding the shortest route between two points in a network. *The Computer Journal* 9(3):275–280.

Pohl, I. 1969. Bi-directional and heuristic search in path problems. Technical Report 104, Stanford Linear Accelerator Center.

Samadi, M.; Siabani, M.; Felner, A.; and Holte, R. 2008. Compressing pattern databases using learning. In *European Conference on Artificial Intelligence (ECAI-08)*, 495–499.