Proceedings of the Twenty-Sixth International Conference on
Automated Planning and Scheduling (ICAPS 2016)

Efficient Representation of Pattern Databases Using Acyclic Random Hypergraphs

Mehdi Sadeqi
Department of Computer Science
University of Regina
Regina, SK, Canada S4S 0A2
sadeqi2m@cs.uregina.ca

Howard J. Hamilton
Department of Computer Science
University of Regina
Regina, SK, Canada S4S 0A2
hamilton@cs.uregina.ca

Abstract

A popular way to create domain-independent heuristic functions is by using abstraction, where an abstract (coarse) version of the state space is derived from the original state space. An abstraction-based heuristic is usually implemented using a pattern database, a lookup table of (abstract state, heuristic value) pairs. Efficient representation and compression of pattern databases has been the topic of substantial ongoing research. In this paper, we present a novel domain-independent algorithm for this purpose using acyclic r-partite random hypergraphs. The theoretical and experimental results show that our proposed algorithm provides a consistent representation that works well across planning problem domains and provides a good trade-off between space usage and lookup time. Thus, it is suitable to be a standard efficient representation for pattern databases and a benchmark method for other pattern database representation/compression methods.

Introduction

The A* (Hart, Nilsson, and Raphael 1968) and IDA* (Korf 1985) search algorithms use a function $f(n) = g(n) + h(n)$ as their node prioritization mechanism, where $g(n)$ is the cost of reaching node n from the start node and $h(n)$ is a heuristic estimate of the cost of the optimum path from node n to the goal node. By definition, a heuristic function h is *admissible* if it never overestimates the cost of reaching the goal state from any given state (a lower bound on the actual cost). If $h(n)$ is an admissible heuristic function, A* and IDA* will find an optimal solution (Hart, Nilsson, and Raphael 1968; Korf 1985).

With abstraction, an abstract (coarse) version of a state space is used to create a domain-independent heuristic function. The actual costs in the abstract space are used as a heuristic function in the original space. A heuristic function derived in this manner is usually stored in memory in an efficient lookup table of (abstract state, heuristic value) pairs called a *pattern database* (PDB) (Culberson and Schaeffer 1998) or a memory-based heuristic. The pattern database is then consulted by a heuristic search algorithm for efficient extraction of heuristic values. The effectiveness of heuristic search algorithms using pattern databases is highly dependent on their compact and efficient representation. Typically, larger PDBs contain more accurate heuristic values and therefore their efficient representation with respect to memory requirements and lookup speed is a crucial component of heuristic search that uses a PDB to retrieve heuristic values.

Hash tables are among the most popular data structures used for PDB implementation. Regular hashing, however, can have the problem of *address collision* where two (or more) abstract states are mapped to the same address in the lookup table. This problem is usually resolved using either *open hashing (separate chaining)* or *closed hashing (open addressing)*. Alternatively, a perfect hash function (PHF) can be designed to avoid collisions completely. By using a perfect hash function, no two abstract states can ever be mapped to the same address in the lookup table. Furthermore, a minimum perfect hashing function (MPHF) can be designed to avoid collisions with no unused slots. In the context of PDB implementation, a perfect hash function has typically been created based on a unique id for each abstract state, generated from its lexicographic rank. For domain-independent planning and for an arbitrary abstraction, however, it is generally not possible to find such a function that is surjective and therefore this approach suffers from the problem of an excessive number of unused slots (Schmidt and Zhou 2011). To achieve an efficient domain-independent implementation, we need to turn our attention to generic PHF and MPHF hash functions such as those suggested in the FKS (Fredman-Komlós-Szemerédi) scheme and the BDZ (BPZ) (Botelho, Pagh, and Ziviani 2007) and Compress, Hash and Displace (CHD) (Belazzougui, Botelho, and Dietzfelbinger 2009) algorithms.

The state-of-the-art minimal perfect hash algorithms that can be used for PDB implementation are the BDZ and CHD algorithms. Although they require knowledge of all keys in advance they are suitable for PDB implementation because the set of abstract keys in an abstraction is constant (i.e., a PDB contains an invariant set of abstract states). Although these two algorithms are the fastest existing minimal perfect hash algorithms, they are still much slower than an efficient regular hash table implementation of a PDB (e.g. statemap implementation in PSVN (Holte, Arneson, and Burch 2014)).

Efficient retrieval of values associated with a given key set is a common need in many areas of computer science.

An example would be a static set S of $100,000,000$ URL addresses with their corresponding popularity value. This can be implemented efficiently using a minimal perfect hash function obtained from the BDZ algorithm, i.e., a unique index to a table containing popularity values associated with the URL addresses. To store this unique index, BDZ requires around 2 bits per URL address. We can then omit the URL addresses and only keep the minimal perfect hash function along with the table containing popularity values. The same process can be applied for implementing pattern databases. For PDB representation, however, heuristic value lookup time is an important factor and the original BDZ algorithm is not fast enough for this purpose.

Successfully applied to planning and model checking, Binary Decision Diagrams (BDDs) are an effective approach for implementing pattern databases (Edelkamp 2002; Jensen, Bryant, and Veloso 2002). The heuristic lookup time in a BDD-implemented PDB, however, is usually higher than an efficient regular hash table and their space efficiency depends on the structural properties of the problem domain, such as encoding disruption and the branching factor (Ball and Holte 2008). The Level-Ordered Edge Sequence (LOES) was another approach introduced for the purpose of representing pre-computed heuristics and provides an effective representation of pattern databases in domain-independent planning (Schmidt and Zhou 2011).

The goal of this paper is to introduce a new direction for representing (compressing) PDBs that can be used in conjunction with other state-of-the-art approaches for this purpose and be an effective part of the heuristic search toolbox. It can be particularly useful when other approaches like BDD or LOES do not perform well for a given problem domain or when we need a fast lookup while using a reasonable amount of space for PDB storage. The main advantages of this approach is its theoretical predictability of space usage along with its fast lookup speed. It can also be used as a benchmark method for other PDB representation (compression) methods because it provides a good trade-off between space usage and lookup time for PDB implementation.

An Illustrative Example

We explain the main idea of this paper using a simple example. Assume we want to represent a PDB corresponding to an abstraction Φ containing three abstract states key_1, key_2 and key_3 and their associated heuristic values 1, 0 and 2, respectively. If we use a regular hash table with an arbitrary hash function to map the keys to slots in the PDB, there is a possibility of address collision(s), which would need to be addressed accordingly. Instead, we propose to resolve the address collision problem using a combination of three hash functions (one can use any number of hash functions greater than or equal to two but three is the most appropriate number due to a theoretical property of hypergraphs, as explained later).

To facilitate the discussion, suppose we have three hash functions that map each abstract state in the abstraction to three distinct addresses in a lookup table (Figure 1). The keys associated with the abstract states, shown at left, are distinguished by different style of arrows. For example, the values for key_2 are 0, 1 and 2, which are located at indices 0, 3 and 5 of the lookup table (illustrated using solid arrows in Figure 1). **The idea here is to populate this table such that a heuristic value associated with an abstract state can be obtained from the combination of the three values in this table determined by the three hash values of the abstract state.** The combining of these values should be done using a simple efficient function such as the summation function. Figure 1 shows this process for three abstract states of abstraction Φ introduced earlier. The heuristic value for this abstract state is calculated as $(0+1+2)\,mod\,3 = 0$. We will later discuss the conditions under which a suitable assignment of values for this table can be found.

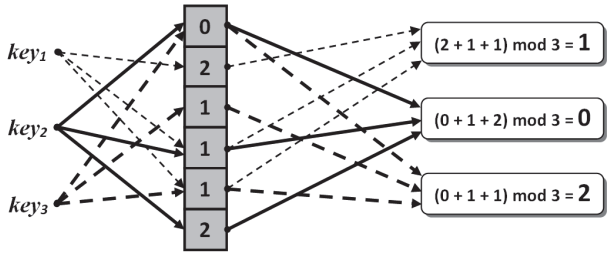

Figure 1: Every heuristic value associated with an abstract state is obtained from a combination of three values in the lookup table.

We now turn our attention to the population of the lookup table in Figure 1. The assignment of values in this table should be done in such a way that we can derive the heuristic value associated with any abstract state from a combination of three values in this table. Figure 2 shows one approach. Here we see an acyclic 3-partite hypergraph where each vertex corresponds to a table entry and each edge of this hypergraph connects three vertices (acyclic r-partite hypergraphs will be discussed in more detail later). The three vertices of each hyperedge correspond to the three different entries—calculated using three hash functions—associated with a given abstract state. Now consider an arbitrary ordering of these edges such as (E_2, E_3, E_1). If we assign values to the vertices of each edge starting from E_2 and ending with E_1, we can obtain the lookup table in Figure 1. In the next section, we will explain in detail how to create an acyclic r-partite hypergraph for a set of abstract states, how to choose an appropriate ordering of hyperedges, and how to assign values to the vertices[1].

Approach

We propose to implement a PDB based on the idea of a near optimal implementation of an associative array where $|S| \gg |T|$ (S and T are the key and value sets, respectively). When the key set is fixed or rarely updated and $|S| \gg |T|$,

[1]The order of selecting edges does not matter in this example. In general, this is not the case and we have to follow a particular ordering. This will be discussed later.

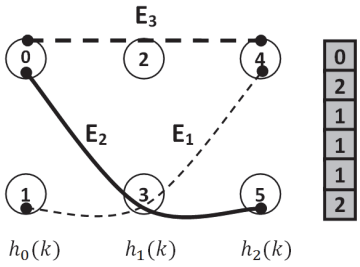

Figure 2: The 3-partite hypergraph used as the basis for creating the lookup table. Each heuristic value is calculated from the three vertices of an edge connecting these vertices.

one can create a compact representation of an associative array. For memory-based abstraction heuristics implemented by a PDB, in particular, we are interested in finding the optimum (or at least a good) trade-off between memory usage and lookup time. We describe our approach by discussing an algorithm for the general case of associative array implementation and then explaining how to adjust some elements of the algorithm to be effective for PDB implementation.

The General Case Algorithm

Assume U is a key universe and $A : S \rightarrow T$ is an associative array where $S \subseteq U$ is a key set and T is value set.

Theorem 1 *For every representation scheme and every set S and T where $|S| = s$ and $|T| = t$, there exists an associative array $A : S \rightarrow T$ that requires at least $\lceil s \log_2 t \rceil$ bits under this representation scheme.*

Proof Given a fixed representation scheme and fixed sets S and T with $|S| = s$ and $|T| = t$, there exist t^s different associative arrays, all of which need to be distinguishable and hence require a different encoding. Let $M = \lceil s \log_2 t \rceil$. Since the total number of distinct bit sequences of length less than M is smaller than t^s, there is at least one of the t^s possible associative arrays that requires M bits or more.

■

We now propose an algorithm that tries to achieve the lower bound proved in Theorem 1. In the following paragraphs, we explain the detailed steps of our algorithm (*mapping*, *assigning* and *compressing*) for efficient representation of an associative array where $|S| \gg |T|$. Figure 3 shows the output of the three steps of the algorithm for the example associative array of $\{key_1 \rightarrow 1, key_2 \rightarrow 0, key_3 \rightarrow 2\}$ (see also Figures 1 and 2). Notice that although the generated 3-partite random hypergraphs in Figures 2 and 3(b) are equal, the output of the assigning step (Figure 3(c)) is different from the lookup table of Figure 2. In other words, the assignment of values to vertices can be done in different ways. As we will see later, the assigning affects the compression rate we achieve in the compressing step. Furthermore, for associative array implementation in general, we also need a table that translates every value to the actual data associated with a key. Since we assume $|key\,set| \gg |value\,set|$, the space required by this table is negligible. For PDB implementation, however, the heuristic values can be used directly and a translation table is not needed.

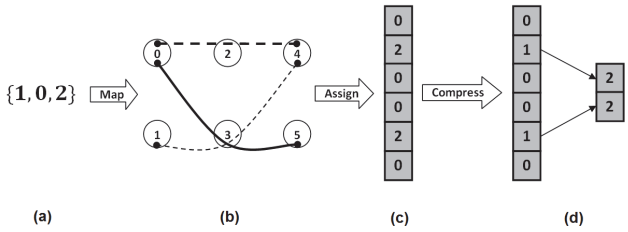

Figure 3: The output of the three steps of the proposed algorithm applied to an associative array with three keys and their associated values: (a) the associative array of $\{key_1 \rightarrow 1, key_2 \rightarrow 0, key_3 \rightarrow 2\}$, (b) the mapping step (generation of an acyclic 3-partite random hypergraph), (c) the assigning step (assignment of values to the hypergraph vertices) and (d) the compressing step (conversion to a more compact data structure using a bit array).

Step 1: Mapping

In the mapping step, an acyclic r-partite random hypergraph is generated. A conventional undirected graph can be generalized to a hypergraph where each edge connects $r \geq 2$ vertices. A random r-partite r-uniform hypergraph has a vertex set $V = \cup_{i=0}^{r-1} V_i$ where $\forall i, j, i \neq j : V_i \cap V_j = \emptyset$ and each edge is generated by randomly choosing one from all possible edges with repetitions allowed. We are in particular interested in a random r-partite r-uniform hypergraph that contain no cycles. A cycle in a hypergraph can be defined in many ways but—in our context—its strongest definition leads to the proper definition of a acyclicity: a hypergraph is acyclic if and only if some sequence of repeated deletions of edges containing at least one vertex of degree 1 yields a hypergraph with no edges (Czech, Havas, and Majewski 1997).

To find an acyclic hypergraph, an r-partite random hypergraph is generated and tested for acyclicity. If the hypergraph contains any cycles, another hypergraph is generated. This process is continued until an acyclic graph is found. For a graph $G_r = (V, E)$, it has been shown that there exists a constant $c_r = c(r)$ such that if $|V| \geq c_r|E|$, the space of random r-graphs is dominated by acyclic r-graphs and c_r has a minimum at $r = 3$ ($c_3 \approx 1.23$), i.e., the smallest acyclic hypergraphs are achieved with $r = 3$ (Czech, Havas, and Majewski 1997). Figure 4, adopted from (Botelho 2008), illustrates a plot of c_r for $2 \leq r \leq 10$.

To detect whether a hypergraph G_r has any cycles, we use the following algorithm (Majewski et al. 1996):

1. Every edge of hypergraph G_r that has at least one of its vertices with degree one is stored in a queue Q.
2. The edges in Q are dequeued one by one, removed from the hypergraph G_r and stored in a list L. If any of the vertices of a removed edge has degree one (after its removal

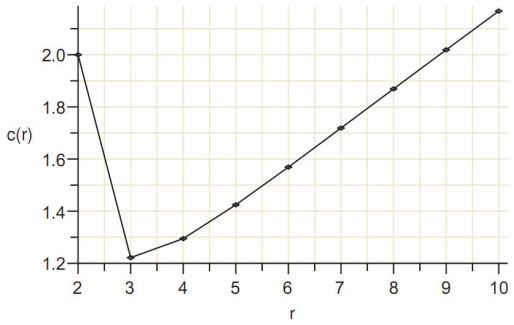

Figure 4: Plot of c_r for $2 \leq r \leq 10$, adopted from (Botelho 2008). c_r has a minimum of ≈ 1.23 at $r = 3$.

from the hypergraph), the edge that contains that vertex is enqueued in Q. This is repeated until Q is empty.

If all the edges of G_r are removed in this process, it is an acyclic hypergraph. Figure 3(b) shows an acyclic 3-partite random hypergraph generated by the mapping step of our algorithm.

Step 2: Assigning

In the assigning step, the appropriate values are assigned to the vertices of the acyclic random r-partite hypergraph G_r generated in the mapping step. The output of the assigning step is represented in a lookup table g. All the entries in this lookup table are initialized to value 0. The assignment of values in g can be done similarly to what is proposed in (Botelho, Pagh, and Ziviani 2007). Given the list of edges L created in the mapping step, it is possible to make a perfect assignment of the lookup table g by traversing L from tail to head. To put it another way, by following a particular ordering of edges in the assigning step, we never encounter an edge such that all its vertices have already been assigned values. For every edge e dequeued from L moving backwards in this list, we find and set the value for unassigned $g[v]$ entries such that $(\Sigma_{v \in e}\, g[v])\, mod\, t = value[key_e]$ where $t = |T|$ is the size of value set, v is a vertex of e, and key_e is the key corresponding to edge e in the hypergraph[2]. Botelho et al. showed such an unassigned vertex always exists if we move backwards in L (Botelho, Pagh, and Ziviani 2007). For every given edge e_i in L, there is at least one vertex v_i with degree 1 when considered for deletion in the cycle detection of the mapping step. After removing this edge, for all the edges that are deleted after e_i, the degree of v_i will be 0. This means that v_i will be unassigned if we perform the assignment from tail to head in L. This enables us to have a perfect assignment in the lookup table g. Figure 3(c) illustrates this assignment for the acyclic 3-partite random hypergraph of Figure 3(b). One possible order of assignment for this graph is shown in Figure 5. This figure shows how the vertices of the hypergraph—and thus the entries of array g—are assigned with appropriate values in the range $[0, 3)$.

[2]If there are more than one unassigned $g[v]$, all but one is set to 0 and the last one is set such that $(\Sigma_{v \in e}\, g[v])\, mod\, t = value[key_e]$. This is in order to have as many 0 in the table as possible to make the compressing step more effective.

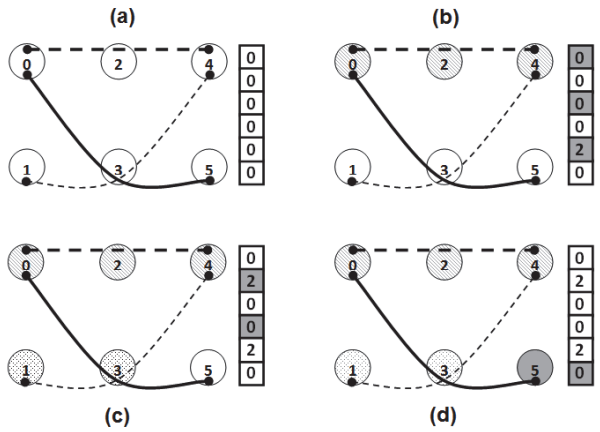

Figure 5: One possible order of assignment for the acyclic 3-partite random hypergraph of Figure 3. Steps (a)–(d) show how the vertices of this hypergraph—and the corresponding entries of array g—are assigned with appropriate values in the range $[0, 3)$.

Step 3: Compressing

To make the lookup table g generated in the assigning step more compact, a simple but efficient compression scheme from (Pagh 2001) is used. We use a bit array corresponding to the entries in g representing whether there is a non-zero item in that index or not. The rank of every k^{th} index in g is defined as the number of non-zero entries in this table before that entry. A rank table is then used to store the rank of every k^{th} index in g. k should be set carefully to achieve a good trade-off between space and evaluation time. For example, with $k = 256$, the rank table stores the number of assigned entries in g before every 256^{th} entry in this lookup table. Figure 3(d) shows the final compressed table achieved after the application of the compressing step. Further, one should notice that zeroes in g have two separate sources: (a) initialization of all g entries to 0 at the start of the assigning step, and (b) zeroes assigned to g entries corresponding to the edges of the acyclic hypergraph generated in the mapping step. All of these zeroes are then eliminated to make the final compressed table.

PDB Implementation Considerations

In this section we explain why the general case algorithm is a suitable candidate for implementing pattern databases. We also discuss the adjustments required to make this algorithm efficient for PDB representation with respect to space and lookup time. Our main concern about using our proposed approach for PDB implementation is the lookup time. The two main factors affecting the lookup time in this approach are: 1) the time required by uniform hash values calculations (used both in generating a random acyclic r-partite r-uniform hypergraph and for heuristic value lookup), and 2)

the overhead time added by the compressing step. These two components are discussed in the following paragraphs.

The time required for generating a random acyclic r-partite hypergraph as well as for heuristic value lookup can be reduced using Zobrist hashing (Zobrist 1970). Although full randomness is theoretically desired for the mapping step to generate an acyclic r-partite r-uniform hypergraph, in practice limited randomness using universal hashing is sufficient (Botelho, Pagh, and Ziviani 2007; Botelho 2008). Zobrist hashing is an ideal candidate for the purpose of PDB representation because 1) it is from the universal hashing family (it is strongly universal and also 3-wise independent) and, 2) the calculation of the hash values of a child state is very fast: the hash values of a child state can be calculated from the hash values of its parent state using only the parts of the child state that are different from its parent state.

The overhead time added by the compressing step is discussed later by comparing the memory needed by our approach, the BDZ algorithm and the CHD algorithm.

PDB Construction Process

The PDB creation process can be summarized as follows (since the smallest acyclic r-partite hypergraphs are achieved at $r = 3$, we use three Zobrist hash functions to achieve the best compromise in terms of memory requirement and lookup speed):

1. Enumerate all the states in the abstraction to find the number of abstract states m and the number of distinct heuristic values v: if all the operators are invertible, this can be done by applying the inverse abstract operators in a breadth-first manner starting from the abstract goal state; otherwise, a forward pattern database construction process, as explained in (Edelkamp and Schrödl 2011, 170–171) can be used. An integer number n is chosen such that n is the smallest integer number greater than or equal to $1.23m$ where $n \, mod \, 3 = 0$. A table g of size n is then constructed corresponding to a hypergraph with n entries.
2. Generate 3 new Zobrist hash functions, h_1, h_2 and h_3.
3. Similar to step 1, enumerate all the states in the abstraction in order to generate the 3-partite random hypergraph[3]:
   (a) For every abstract state s, a hyperedge that connects three Zobrist hash values, $h_1(s)$, $h_2(s)$ and $h_3(s)$, is added to the hypergraph ($h_1(s)$, $h_2(s)$ and $h_3(s)$ will have integer values in $[0, \frac{n}{3}-1]$, $[\frac{n}{3}, \frac{2n}{3}-1]$ and $[\frac{2n}{3}, n-1]$, respectively).
   (b) The generated hypergraph is tested for acyclicity as is explained in the mapping step. In doing so, a queue of edges is constructed. If the graph contains any cycles, return to step 2.
4. All entries of g are initialized to 0. The nodes in the graph are then assigned values as explained in the assigning step:
   (a) Hyperedges are removed one by one from the queue created in the mapping step. Each hyperedge corresponds to an abstract state s (since the generated hypergraph has no cycles, it is guaranteed that at least one of the vertices of the removed hyperedges has a corresponding unassigned entry).
   (b) Values are entered in unassigned table entries $g[h_1(s)]$, $g[h_2(s)]$ and $g[h_3(s)]$ such that the sum of these values—modulo the number of distinct entry values v—is equal to the heuristic value of the corresponding abstract state s.
5. The table can then be compressed as explained in the compressing step.

From a PDB constructed using the above procedure, the heuristic value of a state mapping to an abstract state s is obtained by calculating $g[h_1(s)] + g[h_2(s)] + g[h_3(s)]$ modulo the number of distinct heuristic values v in the abstraction. Although the process of finding acyclic hypergraphs is a random process, in our experiments, we always obtained an acyclic hypergraph the first time we generated a random 3-partite hypergraph of the size recommended in the step 1 of the procedure.

Memory Usage Comparison

We compare the memory required by our approach with and without the compressing step to the memory requirements of the most efficient (minimum) perfect hash implementations for a PDB, i.e., the BDZ and CHD algorithms. BDZ needs around 1.95 and 2.6 bits per entry to store PHF and MPHF, respectively (Botelho, Pagh, and Ziviani 2007; Botelho 2008). CHD needs around 1.40 bits per entry for PHF with a load factor of 0.81 (equal to the load factor of BDZ in the PHF setting and our approach without the compressing step, corresponding to the optimum hypergraph size achieved at $c_r = 1.23$) and 2.07 bits for MPHF (Belazzougui, Botelho, and Dietzfelbinger 2009). Our approach without the compressing step does not need any information per entry (for a load factor of 0.81) and with the compressing step requires—at least—one bit less than BDZ in the MPHF setting. This means that, our proposed data structure without the compressing step can store up to $2^{(2.6/0.23 \approx 11)} = 2,048$ different values before its memory requirement exceeds the memory usage by BDZ algorithm in the MPHF setting (for CHD, this number is $2^{(2.07/0.23=9)} = 512$). In other words, as long as an abstraction has less than or equal $2^{11} = 2,048$ different heuristic values, the memory usage of our approach without the compressing step is less than or equal the BDZ in the MPHF setting (the corresponding number for CHD is $2^9 = 512$). This is a desirable property since the compressing step has a substantial effect on the lookup time and for all the practical abstractions in benchmark planning problem domains (to the best of our knowledge), the above property holds true. In other words, for all experimented abstractions, our approach without the compressing step needs less memory than BDZ and CHD in their MPHF setting, making its lookup speed considerably faster than a BDZ and CHD implementation of a PDB.

[3]For simplicity, we have explained the PDB construction process as performing two separate enumerations of the abstract states in steps 1 and 3. In our implementation, however, we only enumerate the states in the abstraction once.

Experimental Results

Experimental results in three problem domains, Sliding-Tile Puzzle, Blocks World with Table Positions and Scanalyzer, are presented in this section. We used production system vector notation (PSVN) (Holte, Arneson, and Burch 2014) for the state space representation of these problem domains. We compare the efficiency of our proposed approach with a fast regular hash table implementation of PDBs in these three benchmark planning problem domains. For this purpose, we used IDA* to solve $1,000$ uniformly generated random problem instances in each of these problem domains and reported the average time for each. IDA* is implemented in such a way that the Zobrist hash values for a child state are efficiently calculated from the hash values of its parent. Before introducing the experimented representations and abstractions, we explain how abstractions are defined here.

An abstraction is typically defined implicitly by defining a rule/rules describing the abstraction. Two types of abstraction are discussed here: *domain abstraction* and *projection abstraction*. A domain abstraction is simply a mapping from the original state space alphabet to a new smaller one. A domain abstraction is formally defined by a mapping $\Phi : \Sigma \rightarrow \Gamma$ where Σ is the original state space alphabet and Γ is the new alphabet of the abstract state space and $|\Gamma| \leq |\Sigma|$. Applying this mapping to the seed state s_0 and the operator set of a state space results in the new abstract state space. In other words:

$$\Phi(S) = (\Phi(s_0), \Phi(O), \Gamma)$$

The other type of abstraction we are considering here is projection abstraction (Edelkamp 2001). In projection, the original state space alphabet remains unchanged ($\Gamma = \Sigma$) but some variables from the state representation are ignored. The remaining variables are specified via a subset $M = \{i_1, ..., i_m\} \subset \{1, ..., n\}$ where n is the number of variables in a state representation in the original state space and $m < n$. For example, assuming the original state space alphabet is $\{a, b, c, d, e\}$ and having a projection abstraction defined as $M = \{1, 4, 5\} \subset \{1, 2, 3, 4, 5\}$, we simply keep variables at indexes 1, 4 and 5 and ignore the rest while the alphabet is kept intact. This means that the state $\langle a, c, d, b, e\rangle$ in the original state space will be mapped to the abstract state $\langle a, -, -, b, e\rangle$ in the abstract state space where $-$ refers to the ignored variables.

We use rules to define both domain and projection abstractions. A rule

$$a_1 \leftarrow a_1, a_2, \ldots, a_k$$

means that the symbols $a_1, a_2, \ldots, a_k$ are no longer distinguishable and are all mapped to the symbol a_1 (domain abstraction). A rule

ignore [*facts*]

means that the variables encoding the listed facts are ignored (projection abstraction).

Sliding-Tile Puzzle In the $n \times m$-Sliding-Tile Puzzle there is an $n \times m$ grid, in which tiles numbered 1 through $n \cdot m - 1$ each fill one grid position and the remaining grid position is blank. A move consists of swapping the blank with an adjacent tile. The goal is to have the numbered tiles in increasing order from top left corner to bottom right corner with the blank tile in the bottom right position.

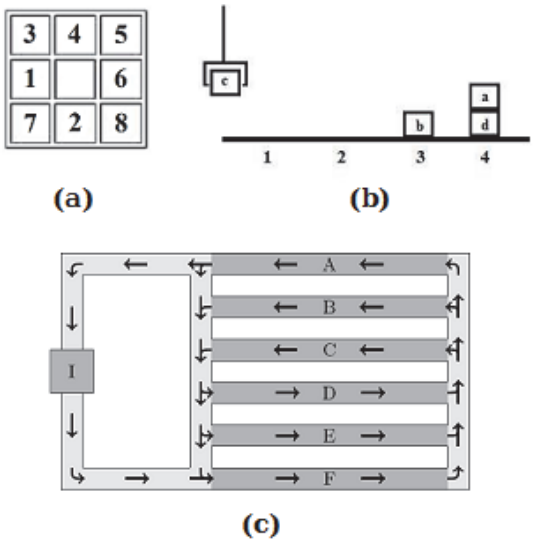

Figure 6: The experimented problem domains: (a) Sliding-Tile Puzzle, (b) Blocks World with Table Positions and (c) Scanalyzer.

In one representation of this puzzle, states are vectors of length $n \cdot m$ where each component corresponds to either one of the numbered tiles or the blank. The value of a vector component represents the grid position at which the corresponding tile is located. For example, the state in Figure 6(a) would be encoded as $\langle 4, 8, 1, 2, 3, 6, 7, 9, 5\rangle$, where the i^{th} component, for $i \leq 8$, holds the position of tile i, and the 9^{th} component holds the position of the blank.

Blocks World with Table Positions In the n-Blocks World with p Table Positions, a state describes the constellation of n blocks stacked on a table with p named positions, where at most one block can be located in a "hand." In every move, either the empty hand picks up the top block off one of the stacks on the table, or the hand holding a block places that block onto an empty table position or on top of a stack of blocks. The goal is to stack up all numbered blocks in increasing order, from bottom to top, on the goal position from a given start state using the legal moves.

We consider a PSVN representation of the n-Blocks World with p distinct table positions where a state vector has $1 + p + n$ components, each containing either the value 0 or one of n block names: (i) the first component is the name of the block in the hand, (ii) the next p components are the names of the blocks immediately on table positions 1 through p, (iii) the last n components are the names of the blocks immediately on top of blocks $a, b, c, \ldots$. In each case, the value 0 means "no block." For example, Figure 6(b) illustrates a state of the 4-Blocks World with 4 table positions, encoded as $\langle c, 0, 0, b, d, 0, 0, 0, a\rangle$.

Scanalyzer In the n-Belt Scanalyzer domain, a state de-

Problem Domain	Abstraction	Regular Hash Table		Our Approach		Trivial Perfect Hash
		PDB Size	Average Time (S)	PDB Size	Average Time (S)	PDB Size
Sliding-Tile Puzzle	ignore [tiles 1,3,6,9,11]	1.1 GB	0.74	32 MB	0.76	≈ 26 MB
	ignore [tiles 1,6,7,8,9]	1.1 GB	0.38	32 MB	0.39	≈ 26 MB
Blocks World with Table Positions	$1 \leftarrow 1,2,3,4,5,6$ $2 \leftarrow 7,8$	11 GB	48.05	275 MB	51.78	$\approx 24,000,000$ MB
	$1 \leftarrow 1,2,3,4,5,6$ $2 \leftarrow 10,11$	11 GB	74.70	275 MB	78.14	$\approx 24,000,000$ MB
Scanalyzer	ignore [belts 0,1,2,6,8]	6.1 GB	10.71	76 MB	11.38	≈ 61 MB
	ignore [belts 1,3,4,7,9]	6.1 GB	8.01	76 MB	8.08	≈ 61 MB

Table 1: Different abstractions of 3×4-Sliding-Tile Puzzle, Blocks World with 12 blocks and 3 table positions and 10-Belt Scanalyzer. The corresponding PDB sizes in bytes for both regular hash implementation and our proposed approach along with their average time needed for solving $1,000$ uniformly generated random problem instances using IDA* are illustrated here.

scribes the placement of n plant batches on n conveyor belts along with information indicating which batches have been "analyzed" (for a detailed description of this domain, see (Helmert and Lasinger 2010)). In a *rotate* move, a batch can be switched from one conveyor belt in the upper half (A, B and C in Figure 6(c)) to one in the lower half (D, E and F in Figure 6(c)) and vice versa. In a *rotate-and-analyze* move, a batch can simultaneously be transferred and analyzed from the topmost conveyor belt to the bottommost one while the batch at the bottommost conveyor belt is moved to the topmost one without any change to its "analyzed" state. Once a batch is analyzed, it will remain analyzed henceforward.

In a representation of the n-Belt Scanalyzer, a state is encoded as a vector of length $2n$ in which each conveyor belt corresponds to two components: the name of the batch on that belt and a flag indicating whether that batch is analyzed or not. The goal state corresponds to having all plant batches analyzed and replaced back on their original conveyor belts.

In what follows, we evaluated the efficiency of our proposed algorithm. In order to have higher lookup speed, we have not used the compressing step in our experiments; one can achieve a smaller representation if some lookup speed is sacrificed. In the regular hash table implementation we have chosen linear probing for its simplicity of implementation and high cache performance[4]. However, for linear probing to be efficient, one should choose a good hash function that is simple enough to be evaluated fast and uniformly distributes keys in the hash table (minimum number of collisions). The Jenkins hash functions (Jenkins 1997) are known to be very efficient for this purpose. As well, the hash table sizes are chosen to be powers of two for efficiency reasons. We double the hash table size whenever we reach a load factor of 0.75 to avoid the performance degradation that generally happens with open addressing if the hash table becomes nearly full (considering the time-space trade-off, a load factor of 0.75 seems to be a good threshold for rehashing).

We considered two projection abstractions in the 3×4-Sliding-Tile Puzzle (both containing $35,831,808$ abstract states with maximum heuristic values of 43 and 44 in the order they appear in Table 1), two domain abstractions of the Blocks World with 12 blocks and 3 table positions (both with $311,594,640$ abstract states and maximum heuristic values of 41), and two projection abstractions in the 10-Belt Scanalyzer (both having $102,400,000$ abstract states and maximum heuristic values of 18) in the representations described earlier. All these abstractions contain spurious states, which increases their number of abstract states (for more on spurious states, how they may effect the size and quality of an abstraction and the computational complexity of avoiding them, see (Hernádvölgyi and Holte 2004; Zilles and Holte 2010)). The definition of these abstractions, along with the average timing results of our proposed algorithm for solving $1,000$ uniformly generated random problem instances using IDA*, are shown in Table 1. These results are compared to a fast regular hash table implementation of these PDBs. Our implementation is slightly slower (about 3% for the two abstractions of the Sliding-Tile Puzzle, about 7% and 4% for the Blocks World with Table Positions, and about 7% and 4% for the Scanalyzer), but its storage requirement is substantially less than that of the regular hash table implementation (around 3%, 2% and 1%, respectively).

In the above experiments, the default regular hash table implementation integrated into PSVN uses 1 byte for each state variable and 4 bytes for each heuristic value. However, one can achieve a more compact regular hash table representation of an abstraction by considering the total possible values each state variable can adopt and the number of distinct heuristic values in that abstraction. For example, for the projection abstractions of the Sliding-Tile Puzzle discussed earlier, each tile only needs 4 bits to represent its

[4]It has been shown that linear probing can outperform other address resolution mechanisms when dealing with load factors of 30%-70% because of its cache friendly property (Heileman and Luo 2005; Black, Martel, and Qi 1998).

grid location and since there are 12 tiles (including blank), a state only requires 48 bits. If we add another byte for the heuristic value, we need 7 bytes to store each state and $2^{26} \times 7 = 469,762,048 \approx 469$ MB to represent the corresponding PDBs[5] (in order to avoid the expensive modulo operator, the hash table size is chosen the smallest power of two such that the load factor of the hash table is less than 0.75). For the domain abstractions of the Blocks World experiments, an abstract state has 16 variables. Since there are 7 distinct values each abstract state variable can adopt (requiring 3 bits to distinguish), each abstract state can be represented using $16 \times 3 = 48$ bits. Adding another byte for the heuristic value, we need 7 bytes to store each abstract state. Since the hash table size is chosen the smallest power of two such that the load factor of the hash table is less than 0.75, these PDBs can be represented using $2^{29} \times 7 = 3,758,096,384$ bytes (almost 3.7 GB).

Instead of a regular hash table, we could have used a trivial perfect hash function based on the number of variables and their corresponding domain in an abstract state. Using this approach, the projection abstractions of the Sliding-Tile Puzzle will have $12^7 = 35,831,808$ entries and with 6 bits for each heuristic value, 26,873,856 bytes are needed for the corresponding PDBs. As for the projection abstractions of the Scanalyzer, $2^{10} \times 10^5 = 102,400,000$ entries are required and with 5 bits for each heuristic value, the corresponding PDBs need 64,000,000 bytes. The domain abstractions of the Blocks World experiments will have $7^{16} = 33,232,930,569,601$ entries and with 6 bits for each heuristic value, almost 2×10^{13} bytes are required for implementing the corresponding PDBs. Table 1 shows these PDB sizes using this trivial perfect hash function. From these results, we see that the trivial perfect hash function yields very good results in the case of the projection abstractions of the Sliding-Tile Puzzle and the Scanalyzer abstractions but is completely ineffective for the domain abstractions of the Blocks World experiments. This illustrates the fact that this approach is inapplicable for domain-independent planning.

To better illustrate the ineffectiveness of trivial perfect hashing in a domain-independent setting, we modified the Sliding-Tile Puzzle by disallowing some of the tile movements, based on the blank location. Two versions of this puzzle are used here (see Figure 7). We have used a natural encoding for the states where each component in a state corresponds to a grid position and represents the number of the tile (or blank) in this position.

In the 3×4 version of this puzzle, the domain abstraction that maps tile 11 to blank contains 239,500,800 abstract states. Also, in the 4×5 version of this puzzle, the domain abstraction that maps tiles 3 and 13 to blank contains 54,432,000 abstract states. To implement the corresponding pattern databases using trivial perfect hashing, we need tables containing $11^{12} \approx 3 \times 10^{12}$ and $18^{20} \approx 10^{25}$ heuristic value entries, respectively. These are obviously unacceptable

[5]At the expense of some lookup speed, an even more compact representation of these abstractions can be achieved knowing there are only 7 variables in each abstract state and only 6 bits are required to store each heuristic value.

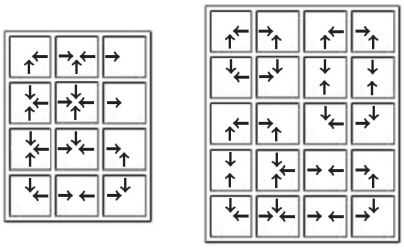

Figure 7: 3×4 and 4×5 Constrained-Movement Sliding-Tile Puzzle. The arrows indicate the possible movements of the tiles, based on the blank location.

implementations for a PDB, which demonstrates the ineffectiveness of trivial perfect hashing in domain-independent planning.

We need to emphasize that we are only interested in showing how a PDB implemented using this approach compares to a fast regular hash implementation; we are not concerned that the abstractions used in our experiments are the best ones for solving problem instances compared to fine-tuned abstractions created using specific problem domain properties. The 3×4-Sliding-Tile Puzzle is appropriate for the experiments because the memory required by a regular hash implementation of an arbitrary domain/projection abstraction capable of solving 1,000 uniformly generated random problem instances in 4×4-Sliding-Tile Puzzle is huge.

From the above results, we observe that the heuristic lookup speed of our algorithm using the proposed PDB representation is comparable to that of a fast regular hash table implementation (with linear probing using fast Jenkins hash functions) while it requires a lot less memory. It can therefore be considered as a standard PDB representation approach with efficient space usage and fast heuristic lookup: In some cases, such as in the Sliding-Tile Puzzle, it provides better compression than the BDD approach. This means that this algorithm can be effective in situations where other approaches fail to provide a good trade-off for space usage and lookup time in their PDB representation.

Conclusions

Using acyclic r-partite random hypergraphs, we have introduced a simple yet efficient new approach for efficient representation of associative arrays where $|key\,set| \gg |value\,set|$. We have shown that this approach can be adjusted for efficient representation of pattern databases in memory-based heuristic search. Although theoretical results require full randomness to generate an acyclic random hypergraph with high probability in the mapping step, we have observed in practice that the limited randomness of some universal hash functions is sufficient for this purpose. In particular, a class of strongly universal hash functions suitable for heuristic search algorithms, Zobrist hash functions, is in practice able to generate a random hypergraph that is acyclic with high probability for our proposed algorithm. The Zobrist hashing is especially useful for fast lookup of heuristic values in heuristic search algorithms like A* and IDA*.

The algorithm proposed here has three advantages over the most efficient PHF/MPHF algorithms applied to PDB representation (BDZ and CHD): (a) we require less memory to represent a PDB in both PHF and MPHF settings, (b) we can use Zobrist hashing, which is quite fast when calculating the hash function of a child state from its parent state (we only need to consider changes of the child state from its parent to calculate the child state hash value), and (c) our algorithm without the compressing step requires less than or equal memory than an MPHF generated by BDZ (CHD) as long as $|value\,set| \leq 2,048$ ($|value\,set| \leq 512$), which is always the case for pattern databases in our experiments. Since the compressing function and table is not needed in this setting, the lookup time is substantially improved. The experimental results illustrate that heuristics lookup speed is comparable to a fast regular hash table implementation using Jenkins hash functions and linear probing for collision resolution strategy. By using IDA* to solve $1,000$ uniformly generated random problem instances, in three benchmark problem domains, we demonstrated the efficiency of our proposed approach for PDBs.

Further, our approach provides a compact and efficient domain-independent perfect hash function that can be used in combination with other PDB compression algorithms that require a perfect hash function as part of their schemes. A promising example would be the combination of our approach with 1.6-bit pattern databases (Breyer and Korf 2010) to achieve even more compact PDBs in a domain-independent setting. The exact amount of compression and lookup efficiency could be investigated with comprehensive experimental analysis.

References

Ball, M., and Holte, R. C. 2008. The compression power of symbolic pattern databases. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008*, 2–11.

Belazzougui, D.; Botelho, F. C.; and Dietzfelbinger, M. 2009. Hash, displace, and compress. In Fiat, A., and Sanders, P., eds., *ESA*, volume 5757 of *Lecture Notes in Computer Science*, 682–693. Springer.

Black, J. R.; Martel, C. U.; and Qi, H. 1998. Graph and hashing algorithms for modern architectures: Design and performance. In *Algorithm Engineering*, 37–48.

Botelho, F. C.; Pagh, R.; and Ziviani, N. 2007. Simple and space-efficient minimal perfect hash functions. In *Proc. of the 10th Intl. Workshop on Data Structures and Algorithms*, volume 4619 of *Lecture Notes in Computer Science*, 139–150. Springer.

Botelho, F. C. 2008. Near-optimal space perfect hashing algorithms. *The thesis of PhD. in Computer Science of the Federal University of Minas Gerais*.

Breyer, T. M., and Korf, R. E. 2010. 1.6-bit pattern databases. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*.

Culberson, J., and Schaeffer, J. 1998. Pattern databases. *Computational Intelligence* 14(3):318–334.

Czech, Z. J.; Havas, G.; and Majewski, B. S. 1997. Perfect hashing. *Theoretical Computer Science* 182(1-2):1 – 143.

Edelkamp, S., and Schrödl, S. 2011. *Heuristic Search: Theory and Applications*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Edelkamp, S. 2001. Planning with pattern databases. In *Proceedings of the European Conference on Planning*, 13–24.

Edelkamp, S. 2002. Symbolic pattern databases in heuristic search planning. In *AIPS*, 274–283.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* SSC-4(2):100–107.

Heileman, G. L., and Luo, W. 2005. How caching affects hashing. In *ALENEX/ANALCO*, 141–154.

Helmert, M., and Lasinger, H. 2010. The Scanalyzer domain: Greenhouse logistics as a planning problem. In *ICAPS*, 234–237.

Hernádvölgyi, I., and Holte, R. 2004. Steps towards the automatic creation of search heuristics. Technical Report TR04-02, Dept. of Computing Science, Univ. of Alberta.

Holte, R.; Arneson, B.; and Burch, N. 2014. PSVN manual (june 20, 2014). Technical Report 14-03, Department of Computing Science, University of Alberta.

Jenkins, B. 1997. Algorithm alley: Hash functions. *Dr. Dobb's Journal of Software Tools* 22(9):107–109, 115–116.

Jensen, R. M.; Bryant, R. E.; and Veloso, M. M. 2002. Seta*: An efficient BDD-based heuristic search algorithm. In *AAAI/IAAI*, 668–673.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Majewski, B. S.; Wormald, N. C.; Havas, G.; and Czech, Z. J. 1996. A family of perfect hashing methods. *Comput. J.* 39(6):547–554.

Pagh, R. 2001. Low redundancy in static dictionaries with constant query time. *SIAM J. Comput.* 31(2):353–363.

Schmidt, T., and Zhou, R. 2011. Representing pattern databases with succinct data structures. In *SOCS*, 142–149.

Zilles, S., and Holte, R. C. 2010. The computational complexity of avoiding spurious states in state space abstraction. *Artificial Intelligence* 174:1072–1092.

Zobrist, A. L. 1970. A new hashing method with application for game playing. *Technical Report 88*.