A Novel Technique for Compressing Pattern Databases in the Pancake Sorting Problems

Morteza Keshtkaran, Roohollah Taghizadeh and Koorush Ziarati
Department of Computer Science and Engineering
Shiraz University, Shiraz, Iran
{mkeshtkaran,taghizadeh}@cse.shirazu.ac.ir
ziarati@shirazu.ac.ir

Abstract

In this paper we present a lossless technique to compress pattern databases (PDBs) in the Pancake Sorting problems. This compression technique together with the choice of zero-cost operators in the construction of additive PDBs reduces the memory requirement for PDBs in these problems to a great extent, thus making otherwise intractable problems able to be efficiently handled. Also, using this method, we can construct some problem-size independent PDBs. This precludes the necessity of constructing new PDBs for new problems with different numbers of pancakes. In addition to our compression technique, by maximizing over the heuristic value of additive PDBs and the modified version of the gap heuristic, we have obtained powerful heuristics for the burnt pancake problem.

Introduction

IDA* (Korf 1985) as a linear-space heuristic search algorithm uses a cost function f to prune nodes. f assigns to each state n, a cost $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the shortest path found so far from the starting state to state n and $h(n)$ is the heuristic estimate of the lowest cost to get from n to a goal state. If $h(n)$ is guaranteed to never overestimate the lowest cost from n to a goal state, it is admissible and the optimality of the solution is insured.

For many problems, a heuristic evaluation function can be calculated before the search and stored in a lookup table called a pattern database (PDB) (Culberson and Schaeffer 1998). For example, for the Sliding-Tile-Puzzle problem we can choose a subset of the tiles, the pattern tiles, and consider the rest of the tiles indistinguishable from each other. For each possible configuration of the pattern tiles among non-pattern ones, we store in a lookup table the minimum number of moves required to bring the pattern tiles into their goal positions. In general, a *pattern* is a projection of a state from the original problem space onto the pattern space. The projection of the goal state is called the *goal pattern*. For each pattern the minimum number of moves required to reach the goal pattern in the pattern space is stored in the PDB. PDBs are usually constructed through a backward breadth-first search from the goal pattern in the pattern space. For each pattern the entry in the PDB is the depth at which it is first generated. Under certain conditions it is possible to sum values from several PDBs without overestimating the solution cost (Korf and Felner 2002). For the Sliding-Tile-Puzzle problem we can partition all tiles into disjoint groups and construct a PDB for each of these groups. Since each operator moves only one tile, it only affects tiles in one PDB. In general, if there is a way to partition all variables into disjoint sets of pattern variables so that each operator only changes variables from one set of pattern variables, the resulting PDBs are called *additive* and such a set of PDBs are called *disjoint*. In problems such as Pancake Sorting each operator may change variables from different sets of pattern variables. Hence it is not trivial to construct additive PDBs for these problems. Yang *et al.* (2008) were first to suggest a technique to construct more general additive PDBs for problems like this.

The main drawback of PDB heuristics lies in their memory requirement. To store more accurate heuristics we need more amount of memory space while there may not be enough. To address this issue, we need to compress them in a way that they can fit into memory. As stated in (Felner et al. 2007), the best compression technique is to keep for groups of equal heuristic values in the PDB, only one value for each group. The main contribution of this paper is to introduce a lossless compression technique for the Pancake Sorting problem based on this concept. The advantage of our method is that we can easily apply other general compression techniques such as (Breyer and Korf 2010; Felner et al. 2007) in tandem to achieve even higher compression.

In additive PDBs, operators that move non-pattern tiles have zero-cost. We use these zero-cost operators, which can also be defined in general additive PDBs, to compress PDBs in the pancake sorting problem without losing any information. This enables us to solve problems that were unsolvable otherwise because of the huge memory requirement for their PDBs.

Pancake Sorting Problems

The original Pancake Sorting problem was first posed in (Dweighter 1975). The problem is to sort a given stack of pancakes of different sizes in as few operations as possible to obtain a stack of pancakes with sizes increasing from top

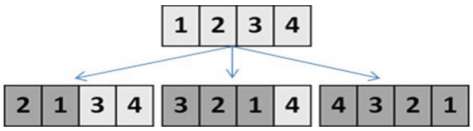

Figure 1: Three successors of the goal state in the 4-pancake problem

to bottom. The only allowed operation is taking several pancakes from top of the stack and flipping them.

In a more difficult problem called the *burnt pancake problem*, one side of each pancake in the pile is burnt, and the sorting must be completed in a way that the burnt side of every pancake is at the bottom, so that if you look from top, no burnt side is seen.

In the rest of the paper, unless stated otherwise, Pancake Sorting Problem refers to the original problem.

Formal Definition

For ease of reference, we will show the goal stack of the n-pancake problem as a sequence $< 1, 2, \ldots, n >$. In this sequence, numbers correspond to pancake sizes while indexes correspond to pancake positions in the stack, with size one denoting the smallest pancake and index one denoting the top position. Now each state of this problem can be represented as a permutation of this sequence and each operation can be done as a prefix reversal on the sequence.

If you consider $s =< s_1, s_2, \ldots, s_n >$, s will be a permutation of the goal sate and s_i is the size of a pancake located at position i from the top of the stack.

In the burnt pancake problem, the numbers corresponding to the pancake sizes can have negative sign. The negative sign shows that the burnt side of the pancake is up.

Pattern Database Heuristics

In this section we review the pattern database heuristics designed for the original pancake sorting problem. All of these methods are applicable to the burnt pancake problem, too.

The earliest pattern databases that are widely used for this problem are non-additive ones choosing usually the rightmost tiles as pattern tiles and leaving the rest of them as indistinguishable (Zahavi et al. 2006). But more accurate PDB based heuristics for this problem are two general additive pattern databases investigated by Yang *et al.* (2008), which are called "cost-splitting" and "location-based cost" methods. We briefly review these two methods on this problem.

Cost-splitting In this method the operator cost is divided proportionally among the patterns based on the share of the tiles moved by the operator that belong to each pattern. For example, consider two patterns of the 4-pancake problem, one consisting of tiles 1 and 2, the other consisting of tiles 3 and 4. Then the middle operator in Figure 1 will have a cost of $2/3$ in the first pattern space and $1/3$ in the second pattern space because it moves two tiles of the first pattern and one tile of the second pattern.

Location-based cost Another method for dividing operator costs among patterns observes one or more specific locations and assigns the full cost of the operator to the pattern which owns the tile that moves into the observed location. In the pancake sorting problem it is intuitive to use the leftmost location as the observed location since every operator changes the tile in this location. The middle operator in Figure 1 moves pancake 3, which belongs to the second pattern, to the leftmost location. Therefore, this operator has zero-cost in the first pattern space and a cost of 1 in the second one.

For the rest of the paper, we will use the "location-based cost" method, since this technique characteristically involves many zero-cost operators and this empowers us to compress PDBs for this problem.

Another PDB heuristic for this problem recently proposed is the Relative Order Abstractions (Helmert and Röger 2010). These PDBs are problem-size independent, just as our PDBs are, and we will cover them in more detail later.

The Gap Heuristic

The best recently-introduced heuristic for the original pancake sorting problem is the gap heuristic (Helmert 2010).

Consider a fixed pancake with size $n+1$ at the end of the stack of pancakes. The value of this heuristic is the number of stack positions for which the pancake at that position is not of adjacent size to the pancake below it.

$$h^{gap}(s) := |\{i | i \in \{1, 2, \ldots, n\}, |s_i - s_{i+1}| > 1\}|$$

As it can be seen in the experimental results, this heuristic is much stronger than the previously proposed PDB-based heuristics and it is the state-of-the-art for this problem.

To the best of our knowledge, this heuristic is not applied to the burnt pancake problem. So, in this part we introduce a simple modification of the gap heuristic that enables us to use it for the burnt pancake problem, too.

We define the value of the gap heuristic for this problem as the number of stack positions for which the pancake at that position is not of adjacent size to the pancake below it or the pancake at that position has its burnt side in opposite order in relation to the pancake below it.

$$\begin{aligned} h^{gap}(s) \quad = \quad & |\{i | i \in \{1, 2, ..., n\}, \\ & |s_i - s_{i+1}| > 1, or, s_i \times s_{i+1} < 0\}| \end{aligned}$$

Similar to the original gap heuristic, this modified version is a consistent and admissible heuristic, too.

Proposed Method

In this section we will put forth our lossless compression technique for the Pancake Sorting Problem. The same statements as the ones in this section can be used for the burnt pancake problem, too. So, in this section our focus is on the original pancake problem.

This compression method requires a specific partitioning of the goal state that will be introduced next.

Partitioning the Goal State into Disjoint Partitions

We will be using a simple disjoint partitioning in which each partition consists of some consecutive tiles of the goal state. Therefore, each partition accounts for a specific kind of patterns all of which have the tiles in the partition as their pattern tiles. Symbolically, consider the general form of a partitioning,

$$P_1 P_2 ... P_k$$
$$P_i's \text{ size is } s_i \text{ and } \sum s_i = n$$

If we represent a non-pattern tile (gap) with g then the goal pattern corresponding to the partition P_i can be represented as:

$$\underbrace{gg\cdots g}_{x}\overbrace{(x+1)}^{\substack{tile\ \text{number}\\ x+1}}(x+2)\cdots(x+s_i)\underbrace{gg\cdots g}_{n-x-s_i}$$

$$x = \sum_{j=1}^{i-1} s_j$$

Example: 3-4-5 partitioning of 12-pancake,

$$\underbrace{1\,2\,3}_{P_1}\ \underbrace{4\,5\,6\,7}_{P_2}\ \underbrace{8\,9\,10\,11\,12}_{P_3}$$

1^{st} goal pattern : 1 2 3 * * * * * * * * *

2^{nd} goal pattern : * * * 4 5 6 7 * * * * *

3^{rd} goal pattern : * * * * * * * 8 9 10 11 12

We will use this partitioning in other examples in the rest of the paper.

Definitions

In the following sections we will be referring to some terms that we need to specify here.

Definition 1: Gap Space We define "gap space" as a contiguous group of adjacent gaps in a pattern. In our problem, the mere existence of gap spaces is crucial and not the number of gaps in them.

Example: An instance of a pattern state of 12-pancake with pancakes 1, 2, 3 and 4 as pattern tiles is:

$$\underbrace{2}_{\substack{Pattern\\ tile}}\ \underbrace{***}_{\substack{Gap\\ space}}\ \underbrace{1}_{\substack{Pattern\\ tile}}\ \underbrace{***}_{\substack{Gap\\ space}}\ \underbrace{4}_{\substack{Pattern\\ tile}}\ \underbrace{3}_{\substack{Pattern\\ tile}}\ \underbrace{**}_{\substack{Gap\\ space}}$$

Definition 2: Compressed Pattern Consider pattern P which have m pattern tiles and more than m gaps. We define the group of Compressed Patterns (CPs) of this pattern as patterns that have the following properties:

1. They have the same pattern tiles in the same order as they appear in P.
2. They have exactly $m + 1$ gaps.
3. When reduced, with single gaps replacing gap spaces all of them should produce similar patterns.

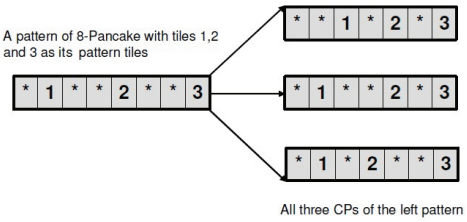

Figure 2: An example of CPs of a pattern state

Conditions 1 and 3 combined, will require that a pattern state and all its CPs to have the same order of pattern tiles and gap spaces.

As an example, all CPs of a pattern of 8-pancake are shown in Figure 2.

Definition 3: Left Sided CP of a Pattern

A left sided CP of a pattern is created by having every pattern gap space except the rightmost one, dump all their gaps but one into the rightmost gap space. As a result a left sided CP has only single-gap gap spaces except the multi-gap rightmost gap space. The last CP in Figure 2 is an example of this kind of CP.

Compression Method

Having presented the basic definitions, we now involve in establishing propositions we will need in setting up our compression framework.

Lemma 1: Each CP of a goal pattern is reachable from every other CPs of the goal pattern with zero-cost operators.

Proof: This is obvious for goal patterns corresponding to the leftmost and rightmost partitions of the goal state, because they have only a single CP. CPs of goal patterns of other partitions can be formulized as follows:

$$\underbrace{gg...g}_{k} P_i \underbrace{g...gg}_{l}$$
$$(0 < k, l) \text{ and } (k + l = n - s_i)$$

So, there is at least one gap after the pattern tiles in P_i. Reversing up to the first gap after P_i, results in:

$$gP_i^{-1}\ \underbrace{g...gg}_{n-s_i-1}$$

Now reversing up to any gap following the reversed pattern tiles of the partition, generates one of the CPs. So, each CP can reach the above CP with one zero-cost operator and then it can transform to any other CP with another zero-cost operator. ■

Corollary: PDBs constructed from each CP of a goal pattern are all the same.

Lemma 2: If an operator transforms pattern P into pattern Q in the pattern space, then there is a path with the same cost from each CP of P to one of the CPs of Q in the compressed pattern space.

Proof: Each transition in the pattern space can be shown as:

$$P \xrightarrow{R} Q$$

P and Q are two patterns and R is the reversing operator indicating the location from which the reversal should be applied to P in order to reach Q.

Consider the following definitions:

X , Y, Z	A sequence of pattern tiles and gap spaces in the pattern. It can also be empty.
X', Y', Z'	A sequence of pattern tiles and gap spaces in a compressed pattern with the same order of pattern tiles and gap spaces as in X, Y, Z. So the difference between X, Y, Z and X', Y', Z' is only in the number of gaps in each gap space.
B_i, W_i	i-th gap space in P, Q
B'_i ,W'_i	i-th gap space in the compressed pattern state that corresponds to B_i, W_i in the pattern state and may have different number of gaps in respect to B_i,W_i
p, q	A pattern tile in P, Q

Regarding the location of reversal on the pattern tile, four different situations may occur:

1. reversal is carried out from a gap space and there is a gap space at the beginning of the state,
2. reversal is carried out from a pattern tile and there is a gap space at the beginning of the state,
3. reversal is carried out from a pattern tile and there is a pattern tile at the beginning of the state,
4. reversal is carried out from a gap space and there is a pattern tile at the beginning of the state.

The general form of the pattern state corresponding to each of the above cases is formulized in Table 1.

A pattern in the pattern space*	Its CPs
(1) B_1XB_iY	$B'_1X'B'_iY'$
(2) B_1XpY	$B'_1X'pY'$
(3) $pXqY$	$pX'qY'$
(4) pXB_iY	$pX'B'_iY'$

Table 1: General form of the pattern state corresponding to each of the four different situations that may occur

* Provided that X, X', Y, Y', Z and Z' are not empty, when they come after a gap space, they should start with a pattern tile and when they appear before a gap space they should end in a pattern tile. These restrictions are due to our definition of gap spaces. As stated before, all adjacent gaps are considered as one gap space. Therefore, it is impossible to have an independent gap before or after a gap space. Consider also that in the case (1) if X is empty then we should deem B_1 and B_i as one gap space.

Now we investigate every possible transition in the pattern space in order to verify that any transition and its dual in the compressed pattern space have the same cost.

Case (1)

$$B_1X\overbrace{gg...g}^{B_i}Y \xrightarrow[1\le m\le k]{m-th\ \text{gap of}\ B_i} \underbrace{gg...g}_{W_1}X^{-1}\overbrace{B_1\underbrace{gg...g}_{k-m}}^{W_i}Y$$

$$B'_1X'\overbrace{gg...g}^{B'_i}Y' \xrightarrow[1\le n\le l]{n-th\ \text{gap of}\ B_i} \underbrace{gg...g}_{W'_1}X'^{-1}\overbrace{B'_1\underbrace{gg...g}_{l-n}}^{W'_i}Y'$$

Example

$$\underbrace{**}_{B_1}\overbrace{2*}^{B_2}\underbrace{1***}_{B_3}\underbrace{43**}_{Y} \xrightarrow{3^{rd}\ \text{gap of}\ B_3} \underbrace{***}_{W_1}\underbrace{1*2}_{X^{-1}}\underbrace{**}_{W_3}\underbrace{43**}_{Y}$$

$$\underbrace{*}_{B'_1}\underbrace{2*1}_{X'}\underbrace{**}_{B'_3}\underbrace{43*}_{Y'} \xrightarrow{1^{st}\ \text{gap of}\ B'_3} \underbrace{*}_{W'_1}\underbrace{1*2}_{X'^{-1}}\underbrace{*}_{W'_3}\underbrace{43*}_{Y'}$$

Case (2-a)

$$B_1Xp\overbrace{qZ}^{Y} \xrightarrow{p} pX^{-1}\overbrace{B_1}^{W_i}qZ$$

$$B'_1X'p\overbrace{qZ'}^{Y'} \xrightarrow{p} pX'^{-1}\overbrace{B'_1}^{W'_i}qZ'$$

Example

$$\underbrace{**}_{B_1}\underbrace{2*1***}_{X}\underbrace{4}_{p}\overbrace{\underbrace{3}_{q}\underbrace{**}_{Z}}^{Y} \xrightarrow{p} \underbrace{4}_{p}\underbrace{***1*2}_{X^{-1}}\underbrace{**}_{W_3}\underbrace{3**}_{Z}$$

$$\underbrace{*}_{B'_1}\underbrace{2*1**}_{X'}\overbrace{\underbrace{4}_{p}\underbrace{3}_{q}\underbrace{*}_{Z'}}^{Y'} \xrightarrow{p} \underbrace{4}_{p}\underbrace{**1*2}_{X'^{-1}}\underbrace{*}_{W'_3}\underbrace{3}_{q}\underbrace{*}_{Z'}$$

Case (2-b)

$$B_1Xp\overbrace{B_iZ}^{Y} \xrightarrow{p} pX^{-1}\overbrace{B_1B_i}^{W_{i-1}}Z$$

$$B'_1X'p\overbrace{B'_iZ'}^{Y'} \xrightarrow{p} pX'^{-1}\overbrace{B'_1B'_i}^{W'_{i-1}}Z'$$

Example

$$\underbrace{**}_{B_1}\underbrace{2*1***}_{X}\underbrace{4}_{p}\overbrace{\underbrace{*}_{B_4}\underbrace{3*}_{Z}}^{Y} \xrightarrow{p} \underbrace{4}_{p}\underbrace{***1*2***}_{X^{-1}}\underbrace{3*}_{Z}$$

$$\underbrace{*}_{B'_1}\underbrace{2*1*}_{X'}\underbrace{4}_{p}\overbrace{\underbrace{*}_{B'_4}\underbrace{3*}_{Z'}}^{Y'} \xrightarrow{p} \underbrace{4}_{p}\underbrace{*1*2}_{X'^{-1}}\underbrace{**}_{W'_3}\underbrace{3*}_{Z'}$$

Case (3)

$$pXqY \xrightarrow{q} qX^{-1}pY$$

$$pX'qY' \xrightarrow{q} qX'^{-1}pY'$$

Example

$$\underbrace{2}_{p}\underbrace{***1***}_{X}\underbrace{4}_{q}\underbrace{3**}_{Y} \xrightarrow{q} \underbrace{4}_{q}\underbrace{***1***}_{X^{-1}}\underbrace{2}_{p}\underbrace{3**}_{Y}$$

$$\underbrace{2}_{p}\underbrace{**1**}_{X'}\underbrace{4}_{q}\underbrace{3*}_{Y'} \xrightarrow{q} \underbrace{4}_{q}\underbrace{**1**}_{X'^{-1}}\underbrace{2}_{p}\underbrace{3*}_{Y'}$$

Case (4-a)

$$pX\overbrace{gg...g}^{B_i}Y \xrightarrow{last \text{ gap of } B_i} \overbrace{gg...g}^{W_1}X^{-1}pY$$

k, $1 \le k$ / k

$$pX'\overbrace{gg...g}^{B'_i}Y' \xrightarrow{last \text{ gap of } B_i} \overbrace{gg...g}^{W'_1}X'^{-1}pY'$$

l, $1 \le l$ / l

Example

$$\underbrace{2}_{p}\underbrace{***1***}_{X}\underbrace{43**}_{B_2}\underbrace{}_{Y} \xrightarrow{last \text{ gap of } B_2} \underbrace{***}_{W_1}\underbrace{1***}_{X^{-1}}\underbrace{2}_{p}\underbrace{43**}_{Y}$$

$$\underbrace{2}_{p}\underbrace{**1}_{X'}\underbrace{**}_{B'_2}\underbrace{43*}_{Y'} \xrightarrow{\text{last gap of } B'_2} \underbrace{**}_{W'_1}\underbrace{1**}_{X'^{-1}}\underbrace{2}_{p}\underbrace{43*}_{Y'}$$

Case (4-b)

$$pX\overbrace{gg...g}^{B_i}Y \xrightarrow[1\le m<k]{\text{m-th gap of } B_i} \overbrace{gg...g}^{W_1}X^{-1}p\overbrace{gg...g}^{W_{i+1}}Y$$

k, $2 \le k$ / m / $k-m$

$$pX'\overbrace{gg...g}^{B'_i}Y' \xrightarrow[1\le n<l]{\text{n-th gap of } B_i} \overbrace{gg...g}^{W'_1}X'^{-1}p\overbrace{gg...g}^{W'_{i+1}}Y'$$

l, $2 \le l$ / l / $l-n$

Example

$$\underbrace{2}_{p}\underbrace{***1***}_{X}\underbrace{43**}_{B_2}\underbrace{}_{Y} \xrightarrow[B_2]{2^{nd} \text{ gap of}} \underbrace{**}_{W_1}\underbrace{1***}_{X^{-1}}\underbrace{2}_{p}\underbrace{*}_{W_3}\underbrace{43**}_{Y}$$

$$\underbrace{2}_{p}\underbrace{**1}_{X'}\underbrace{**}_{B'_2}\underbrace{43*}_{Y'} \xrightarrow[B'_2]{1^{st} \text{ gap of}} \underbrace{*}_{W'_1}\underbrace{1**}_{X'^{-1}}\underbrace{2}_{p}\underbrace{*}_{W'_3}\underbrace{43*}_{Y'}$$

Case (4-c) Troublesome case

$$pX\overbrace{gg...g}^{B_i}Y \xrightarrow[1\le m<k]{\text{m-th gap of } B_i} \overbrace{gg...g}^{W_1}X^{-1}p\overbrace{gg...g}^{W_{i+1}}Y$$

k, $2 \le k$ / m / $k-m$

$$pX'\overbrace{g}^{B'_i}Y' \xrightarrow{\text{the gap of } B_i} \overbrace{g}^{W'_1}X'^{-1}pY'$$

As is evident in this last case (4-c), the result of the second transition is not a CP of the result of the first one. That is, unlike the first result, there is no gap space between p and Y' in the second result. Here, we need more than one zero-cost move in the compressed pattern space.

As stated before, the CPs of a pattern state with m pattern tiles have $m+1$ gaps. Since the gaps are one more than tiles, the only configuration that has no multiple gap space is the one with alternating gaps and pattern tiles, which inevitably has a gap on either end. Hence, the pattern state in hand which starts with a pattern tile must have at least one multiple gap space. If this gap space is located somewhere in X' then we can reformulize the pattern state as:

$$p\overbrace{X'_1ggX'_2}^{X'}\overbrace{g}^{B'_i}Y'$$

and use the following zero-cost moves to transform the pattern state into a CP of Q:

$$p\overbrace{X'_1ggX'_2}^{X'}\overbrace{g}^{B'_i}Y' \xrightarrow[X'_1 \text{ and } X'_2]{1^{st} \text{ gap between}} \overbrace{g}^{W'_1}X'^{-1}_1pgX'_2\overbrace{g}^{W'_{i+1}}Y'$$

$$\xrightarrow[p \text{ and } X'_2]{the \text{ gap between}} \overbrace{g}^{W'_1}p\overbrace{X'_1gX'_2}^{X''}\overbrace{g}^{W'_{i+1}}Y'$$

$$\xrightarrow[W'_{i+1}]{the \text{ gap of}} \overbrace{g}^{W'_1}X''^{-1}p\overbrace{g}^{W'_{i+1}}Y'$$

Since X'' has the same order of pattern tiles and gap spaces as X' (only one of its gap spaces has one gap less than the corresponding gap space in X'), it in turn has the same order of pattern tiles and gap spaces as X too.

Taking into account the complementary condition, if the gap space having multiple gaps is located somewhere in Y, similar zero-cost transitions can be applied to come by the CP.

Example

$$\underbrace{2}_{p}\underbrace{***1***}_{X}\underbrace{43**}_{B_2}\underbrace{}_{Y} \xrightarrow{2^{nd} \text{ gap of } B_2} \underbrace{**}_{W_1}\underbrace{1***}_{X^{-1}}\underbrace{2}_{p}\underbrace{*}_{W_3}\underbrace{43**}_{Y}$$

$$\underbrace{2}_{p}\overbrace{**}^{X'_1}\overbrace{*1}^{X'_2}\underbrace{*}_{B'_2}\underbrace{43*}_{Y'} \xrightarrow{the \text{ first gap after } 2} \underbrace{*}_{W'_1}\underbrace{2}_{p}\underbrace{*}_{X'_2}\underbrace{*1}_{X'^{-1}_1}\underbrace{*}_{W'_3}\underbrace{43*}_{Y'}$$

$$\xrightarrow{the \text{ first gap after } 2} \underbrace{*}_{W'_1}\underbrace{2}_{p}\overbrace{*}^{X''}\overbrace{*1}^{X'_2}\underbrace{*}_{W'_3}\underbrace{43*}_{Y'}$$

$$\xrightarrow{gap \text{ of } W'_3} \underbrace{*}_{W'_1}\underbrace{1**}_{X''^{-1}}\underbrace{2}_{p}\underbrace{*}_{W'_3}\underbrace{43*}_{Y'}$$

We demonstrated that any transition in the pattern space that takes P to Q has its parallel set of transitions in the compressed pattern space which takes a CP of P to a CP of Q with equal costs. Thus we conclude this lemma. ■

Lemma 3: If an operator transforms a CP of P into Q' in the compressed pattern space then there is a path with the same cost from P to a pattern Q for which, Q' is a CP.

Proof: The same transitions as mentioned in the previous lemma can be applied. ■

Constructing Problem-Size Independent PDBs

Now it is obvious that the cost of the shortest path between a pattern state and its goal pattern is equal to the cost of the shortest path between a CP of this pattern state and a CP of the goal pattern. So, instead of constructing PDBs from the goal patterns, we can construct them from arbitrary CPs of the goal patterns.

As we already know that all CPs of a goal pattern are reachable from each other with zero-cost operators, we are entitled to construct our PDBs from the Left Sided CPs of the goal patterns.

Lemma 4: The PDBs constructed for partitions other than the leftmost and rightmost partitions, in case of equal number of pattern tiles, are identical.

Proof: We know that the goal patterns corresponding to these partitions have the same Left Sided CP and only differ in their pattern tile numbers. So they will have the same PDBs too. ■

The CPs of goal patterns with m pattern tiles in n-pancake problems where $n \geq 2m+1$, have exactly $2m+1$ tiles. So, we only need to construct one set of PDBs for m pattern tiles in $(2m+1)$-pancake and use it for any goal with m pattern tiles in n-pancakes. That is why these kind of PDBs are problem-size independent.

Recently, another type of problem-size independent PDBs was proposed, known as relative order abstractions (Helmert and Röger 2010). In these PDBs the shortest paths from all instances of the k-pancake problem to its goal state are initially stored. Then, the heuristic value of a state of an n-pancake problem ($n \geq k$), are estimated accordingly. To that aim, a group of k pancakes of the state are considered and the heuristic value stored in the k-pancake PDB for the configuration of these k tiles as regarded independent of other $n-k$ tiles is adopted as the heuristic value for this state.

The largest storable PDB that can be constructed in this fashion is that of 12-pancake problem. Besides, the maximum heuristic value of these kinds of PDBs is 14. Therefore, the heuristic values obtained from these PDBs are weaker from that of ours for problems with large number of pancakes. That is because of the fact that we use additive PDBs which allows us to produce heuristic values greater than 14. This fact is observable in the experimental results section.

Memory Requirements for Compressed PDBs

Consider the following partitioning that was previously introduced:

$$P_1 P_2 ... P_k$$

$$P_i'\text{s size is s}_i \text{ and } \sum s_i = n$$

For this partitioning we only need to store n_1 entries instead of n_2 entries in the PDBs where n_1 and n_2 are defined as follows:

Pancake Problem

$$n_1 = \sum_{i=1}^{k} \frac{m_i!}{(m_i - s_i)!} \text{ , } m_i = \min(n, 2s_i + 1) \; 1 \leq i \leq k$$

$$n_2 = \sum_{i=1}^{k} \frac{n!}{(n - s_i)!}$$

Burnt Pancake Problem

$$n_1 = \sum_{i=1}^{k} \frac{m_i! 2^{s_i}}{(m_i - s_i)!} \text{ , } m_i = \min(n, 2s_i + 1) \; 1 \leq i \leq k$$

$$n_2 = \sum_{i=1}^{k} \frac{n! 2^{s_i}}{(n - s_i)!}$$

Table 2 and Table 3 compare the number of entries needed to be stored in PDBs of our proposed method with the numbers needed in PDBs without our compression for pancake problem and the burnt version respectively. Table 2 features instances for 17-pancake and (25–28)-pancake problems and Table 3 features instances for (17–18)-burnt pancake problems.

Partitioning	Number of Entries in Compressed PDBs	Number of Entries in PDBs without Compression
3-7-7	64,865,010	196,039,920
4-7-7-7	64,867,824	7,268,487,600
5-7-7-7	64,920,240	9,953,829,600
6-7-7-7	66,100,320	13,640,140,800
7-7-7-7	97,297,200	23,870,246,400

Table 2: Comparison of the number of entries needed to be stored in PDBs with our method and without it in the original pancake problem

Partitioning	Number of Entries in Compressed PDBs	Number of Entries in PDBs without Compression
2-5-5-5	3,548,240	71,286,848
5-6-6	159,920,640	1,164,334,080
6-6-6	237,219,840	2,566,287,360

Table 3: Comparison of the number of entries needed to be stored in PDBs with our method and without it in the burnt pancake problem

These numbers are calculated according to the assumption in Lemma 4. Note that for the 7-7-7-7 partitioning of 28-pancake problem, we need two PDBs for the leftmost and rightmost partitions and one for the two middle partitions.

If we allot one byte of memory for each entry in the PDBs, the above table suggests that for a problem like 25-pancake with 4-7-7-7 partitioning, without considering our method we would need about 6.7 GB of memory to store the PDBs while our compression shrinks this memory requirement to about 61.8 MB. Hence, it is clear that our technique decreases the memory required for this problem to a great extent.

Experimental Results

In this section, we compare different heuristic evaluation methods discussed in the previous sections.

In all cases we use standard IDA* algorithm as our search algorithm. Since, relative order abstraction produces inconsistent heuristics, we use IDA* with BPMX for this method.

Compressed PDBs in Comparison with PDBs without Compression

We used additive PDBs constructed with location-based costs to keep the heuristic information in a standard IDA* search so as to solve random instances of sizable pancake sorting problems with different numbers of pancakes.

In buying memory space by compressing patterns, we are forfeiting time. To appraise how much time we are sacrificing, we check the results of solving 100 random instances of the (15–17)-pancake problem. Table 4 holds the running times when the problem instances are solved with compressed PDBs along with running times when they are solved without compression. The compression based approach takes longer to complete, as we expect, but the excess time is not quite significant.

Problem	15-pancake	16-pancake	17-pancake
Partitioning	3-6-6	3-6-7	3-7-7
Avg. Solution Length	13.80	14.86	15.80
Avg. Initial Heuristic	11.48	12.57	13.30
Avg. Nodes Generated	340,706	526,212	1,596,538
Time (sec.) (PDBs without/with compression)	0.08/0.13	0.15/0.21	0.50/0.68

Table 4: Running times of solving 100 random instances of the (15–17)-pancake problem when the problem instances are solved with compressed PDBs along with running times when they are solved without compression

We can have a similar comparison for the burnt pancake problem.

Additive PDBs in Comparison with Relative Order Abstraction and the Gap Heuristic

In this part, we apply additive PDBs, relative order abstraction and the gap heuristic to the pancake problems.

Pancake Sorting Problem Table 5 shows the results of solving 100 random instances of (16–17)-pancake problem using additive PDBs, relative order abstraction and gap heuristic. In our implementation of the relative order abstraction, for each state, we have taken the maximum heuristic value resulted from 10 random pancake sets of size 12.

	Additive PDB	Relative-Order Abstraction	Gap Heuristic
16-Pancake (Avg. Solution Length 14.86)			
Initial Heuristic	12.57	11.96	14.19
Nodes	526,211	117,602	3,436
Time(s)	0.25	2.48	0.00
17-Pancake (Avg. Solution Length 15.8)			
Initial Heuristic	13.3	12	14.98
Nodes	1,596,538	3,539,478	9,577
Time(s)	0.79	47.12	0.00

Table 5: Results of solving instances of (16–17)-pancake problem

Being able to solve instances of this problem with about 60 pancakes and also more than that, as it was shown in (Helmert 2010), proves that the gap heuristic is state-of-the-art for this problem. The results presented here also confirm this fact.

Although using relative order abstraction may produce fewer nodes than additive PDBs for this problem, it takes time to evaluate stronger heuristics by selecting more sets of random pancakes and also to evaluate their heuristic values. The results for the relative order abstraction show that this heuristic becomes weaker whenever the number of pancakes increases. This stems from the fact that this heuristic is not scalable to larger problems and it has a maximum heuristic value that we cannot exceed that. For this problem, this maximum heuristic value is 14.

In Table 6, we present the results of solving 10 random instances of pancake sorting problem with 25–27 pancakes. We previously witnessed, in Table 2, how these instances are intractable using additive PDBs without compressing their PDBs. Here, we verify the effectiveness of our approach in tackling those problems. In this table, we also present the results of solving these problems with the gap heuristic, which shows that this heuristic is very stronger than additive PDBs for the original pancake sorting problem. As we will show in the next part, this is not true for the burnt version of this problem. Comparing the initial heuristic values in this table with the maximum heuristic value that can be obtained from the relative order abstraction, we can see that the relative order abstraction is not successful in solving these problems.

Problem	25-pancake	26-pancake	27-pancake
Compressed Additive PDBs			
Partitioning	4-7-7-7	5-7-7-7	6-7-7-7
Avg. Initial Heuristic	20.40	21.50	22.70
Avg. Solution Length	23.80	24.90	26.30
Avg. Nodes Generated	2,334,475,851	3,259,099,635	6,638,171,227
Gap Heuristic			
Avg. Initial Heuristic	23.2	24.2	25.4
Avg. Nodes Generated	28,744	34,647	150,949

Table 6: Solving 10 random instances of pancake sorting problem with 25–27 pancakes

Burnt Pancake Sorting Problem We have used additive PDBs, relative order abstraction and the modified gap heuristic for solving this problem. Table 7 shows the results of solving 100 random instances of (13–14)-burnt pancake problem using these heuristics.

In our implementation of the relative order abstraction, for each state, we have taken the maximum heuristic value resulted from 10 random pancake sets of size 9.

We observe that, in this problem, the gap heuristic is not as powerful as it was in the original pancake problem.

In relative order abstraction, the largest storable PDB that can be constructed is that of 9-burnt pancake problem. Be-

	Additive PDB	Relative-Order Abstraction	Gap Heuristic
13-Burnt Pancake (Avg. Solution Length 15.71)			
Initial Heuristic	11.8	12.38	11.98
Nodes	7,900,800	222,656	52,415,891
Time(s)	4.11	3.55	5.40
14-Burnt Pancake (Avg. Solution Length 16.89)			
Initial Heuristic	12.93	12.51	13.01
Nodes	15,389,772	5,236,687	158,458,282
Time(s)	8.98	82.19	16.81

Table 7: Results of solving 100 random instances of (13–14)-burnt pancake problem

sides, the maximum heuristic value of this PDB is 17 and the average of the heuristic values is 11.0868. So, the relative order abstraction is good for problems with small number of pancakes. Yet, evaluating the heuristic value of each state takes time, especially when the number of pancakes increases. Table 8 shows the results of solving the 17-burnt pancake problem. Although this problem can be solved with 2-5-5-5 partitioning without the necessity of compressing its PDBs, this table shows that 5-6-6 partitioning produces much fewer number of nodes. Note that PDBs of 5-6-6 partitioning cannot fit into 1 GB of memory without our compression technique. In this table, we also show that maximizing over the additive PDBs and the modified gap heuristic results in a less time-consuming performance than considering each method individually.

	Initial Heuristic	Nodes	Time(s)
2-5-5-5 Partitioning	14.68	12,274,499,928	8009.79
5-6-6 Partitioning (1)	15.68	615,546,492	375.68
Modified Gap (2)	16.00	2,213,592,360	239.42
Maximizing over (1) and (2)	16.28	52,833,226	35.53

Table 8: Results of solving the 17-burnt pancake problem

To show the effectiveness of our compression technique, we also solve 25 random instances of the 18-burnt pancake problem. The results are presented in Table 9. Since one of the instances of this problem needs a lot more time to be solved than the other ones, we do not include it in the average time and the average generated nodes. The solution of this difficult instance is also reported in Table 10.

Conclusion

In this paper we proposed a compression technique for the PDBs in the pancake sorting problems without losing information. This technique is applicable with the choice of zero-cost operators in additive PDBs and reduces the memory requirements for the PDBs in this problem to a great extent.

Maximizing over the heuristic value of additive PDBs and the modified gap heuristic, we have obtained powerful heuristics for the burnt pancake problem.

We are hopeful that our compression technique can be applied to other heuristic search domains for which general additive PDBs with zero-cost operators can be defined.

	Initial Heuristic	Nodes	Time(s)
6-6-6 Partitioning (1)	16.64	466,864,181	298.09
Modified Gap (2)	17.09	1,184,064,949	134.15
Maximizing over (1) and (2)	17.18	52,701,921	34.46

Table 9: Results of solving 25 random instances of the 18-burnt pancake problem

A problem instance with optimal solution of 24 <-6 -7 -13 9 5 -15 -18 -1 -10 -11 -8 16 -3 17 14 4 12 2>			
	Initial Heuristic	Nodes	Time(s)
6-6-6 Partitioning (1)	17	54,995,040,183	35,424.5
Modified Gap (2)	16	1,870,773,430,432	213,254
Maximizing over (1) and (2)	17	19,078,583,943	12,558.8

Table 10: Solving the most difficult case of the 25 random instances of the 18-burnt pancake problem

References

Breyer, T. M., and Korf, R. E. 2010. 1.6-bit pattern databases. In *AAAI*.

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

Dweighter, H. 1975. Problem e2569. *American Mathematical Monthly* 82:1010+.

Felner, A.; Korf, R. E.; Meshulam, R.; and Holte, R. C. 2007. Compressed pattern databases. *J. Artif. Intell. Res. (JAIR)* 30:213–247.

Helmert, M., and Röger, G. 2010. Relative-order abstractions for the pancake problem. In *ECAI*, 745–750.

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *3rd Annual Symposium on Combinatorial Search (SoCS 2010)*, 109110.

Korf, R. E., and Felner, A. 2002. Disjoint pattern database heuristics. *Artif. Intell.* 134(1-2):9–22.

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

Yang, F.; Culberson, J. C.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *J. Artif. Intell. Res. (JAIR)* 32:631–662.

Zahavi, U.; Felner, A.; Holte, R.; and Schaeffer, J. 2006. Dual search in permutation state spaces. In *AAAI*.