Size-Independent Additive Pattern Databases for the Pancake Problem

Álvaro Torralba Arias de Reyna and **Carlos Linares López**
Planning and Learning Group
Universidad Carlos III de Madrid
Leganés (Madrid) - Spain
{*alvaro.torralba, carlos.linares*}*@uc3m.es*

Abstract

The Pancake problem has become a classical combinatorial problem. Different attempts have been made to optimally solve it and/or to derive tighter bounds on the diameter of its state space for a different number of discs. Until very recently, the most successful technique for solving different instances optimally was based on Pattern Databases. Although different approaches have been tried, solutions with Pattern Databases on Pancakes with more than 19 discs have never been reported. In this work, a new technique is introduced which allows the definition of Additive Pattern Databases for solving Pancakes of an arbitrary length. As a result, this technique solves Pancake problems with twice as many discs as the largest ones solved nowadays with other techniques based on Pattern Databases saving up to two orders of magnitude of space.

Introduction

The Pancake problem was originally introduced by Harry Dweighter as follows: (Dweighter 1975)

> The chef in our place is sloppy, and when he prepares a stack of pancakes they come out all different sizes. Therefore, when I deliver them to a customer, on the way to the table I rearrange them (so that the smallest winds up on top, and so on, down to the largest at the bottom) by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary. If there are N pancakes, what is the maximum number of flips that I will ever have to use to rearrange them?

Schematically, this problem is represented as a permutation π of the constants $\{0, 1, \ldots, N-1\}$ where each number denotes a different pancake —also called discs from now on. The goal consists of restoring the identity permutation by performing prefix reversals. Therefore, an operator f_i generates a new permutation by flipping the location of the first i discs, $1 < i \leq N$. It is trivial to show that every permutation can be solved so that the size of the state space with N discs is exactly equal to $N!$.

Here, we are exclusively concerned with the problem of optimally solving particular instances of the Pancake problem. In this setting, the most successful approach for a number of years has been Pattern Databases —or PDBs for short. Originally introduced by Culberson et al. (Culberson and Schaeffer 1998), a Pattern Database is a collection of *patterns* which are defined as abstractions of the original state space where each constant $\{0, 1, \ldots, N-1\}$ gets replaced by either a dedicated symbol or a special "*don't care*" symbol. Thus, Pattern Databases are computed as hash tables which store, for every pattern (or arrangement of symbols in the abstract state), the minimum number of moves required to place the symbols in the abstract state space in their goal location —also known as the *goal pattern*. This value can be easily computed with a backwards brute-force breadth-first search from the goal pattern. So far, Pattern Databases are admissible heuristic functions. A remarkable property of the Pattern Databases generated this way is that they consider as many symbols as there are in the original state space which is N in the Pancake problem, the number of discs. Originally, all moves were counted in so that when comparing the values retrieved from different Pattern Databases (for a collection of different patterns), taking the maximum of all values guarantees admissibility. However, when the constants appearing in the original state space can be split into disjoint sets so that each operator affects only constants in one set (as in the sliding tile puzzle or the towers of Hanoi), a usually far better informed heuristic function can be built by computing the summation of all values (Korf and Felner 2002). This idea is known as disjoint, or just additive Pattern Databases. Using techniques based on Pattern Databases, no experimental results have been reported with instances with more than 17 discs for many years and only very recently 10 instances have been solved with up to 18 and 19 discs (Helmert and Röger 2010).

However, it has been recently shown that it is possible to optimally solve instances with up to 60 discs with a very simple heuristic function known as *gap* (Helmert 2010)[1]. This heuristic function simply counts the number of gaps or adjacent locations such that $|\pi_i - \pi_{i+1}| > 1$, $0 \leq i < N$, where π_N is always equal to N and stands for the table. Since such locations contain discs which are not adjacent in the goal permutation, at least one occurrence of the op-

[1]Credit shall be acknowledged to Tomas Rokicki: http://tomas.rokicki.com/pancake/

erator that separates them shall be counted in. As a result, the number of gaps is an admissible heuristic function which has been shown to be far more informed than the best Pattern Databases known up to date.

In this work, we provide a new class of abstractions for the Pancake problem which is independent of the size of the underlying permutations. More remarkably, the new abstraction computes the number of moves required to sort specific *blocks* without affecting others so that values from different PDBs can be added. Besides, the resulting heuristic function has been found to be inconsistent and can be enhanced by comparing the values retrieved by a regular lookup and a dual lookup (Zahavi et al. 2007).

This paper is structured as follows. First, we introduce a new abstraction based on the adjacency of discs instead of the contents of each location. It is shown that these abstractions can be used for solving Pancakes with any number of discs. Next, we discuss how to extend these abstractions to consider an arbitrary number of PDBs simultaneously whose values can be added. Also, a discussion is conducted on consistency and duality of the new type of PDBs. Next, some analysis on the relations with previous heuristics for the Pancake problem are done. The experimental results show that this new class of abstractions can solve Pancakes which are twice as the largest ones solved by other PDBs before. Finally, the paper ends with some conclusions.

Block Representation

We start by defining the notion of a *block* from the idea of *adjacency* introduced by Gates et al. (Gates and Papadimitriou 1979): an adjacency is a pair of pancakes that are adjacent in the current permutation, and such that no other disc has intermediate size between the two.

Definition 1 *A* block *is defined as an arbitrary number of contiguous adjacencies such that at each extreme either there is a gap or the permutation ends*

By slightly abusing the notation, we also consider blocks of length one, i.e., single pancakes if and only if the pancakes before and after it are not contiguous in the goal state or the permutation ends there. A similar representation was suggested previously by Chitturi et al. (Chitturi et al. 2009) to derive upper bounds on the diameter of the state space.

For example, the Pancakes $\langle 01324 \rangle$ and $\langle 012376548 \rangle$ have exactly three blocks. Blocks are denoted with an *identifier* and possibly a *sign*. The *identifier* consists of a number such that block i contains discs which are all larger than pancakes of block $(i-1)$. On the other hand, if a block contains more than one pancake, it can be sorted either in ascending or descending order. The first case is said to have a positive sign whereas the latter is said to have a negative sign. Obviously, if a block consists of a single pancake it has no sign. In the previous examples, both pancakes can be represented in the block notation as $\langle 0^+1^-2 \rangle$. Since the block notation preserves all the information of the original permutation but the number of discs in each block, they do provide a convenient way to reason with pancakes of any size though one cannot recover the complete state of a problem from its block representation. It is easy to notice that the gap heuristic is exactly equal to the number of blocks minus one if the N-th disc representing the table is preserved in the block notation[2]. Thus, the gap heuristic of the preceding pancakes is equal to 2.

Blocks can be handled symbolically to either build up new blocks or to split them. In fact, the following self-explanatory relations hold (for an arbitrary pair of adjacent blocks $\mathtt{i}$ and $\mathtt{i}+1$):

$$\begin{array}{ll} \langle \mathtt{i}\ (\mathtt{i}+1) \rangle = \langle \mathtt{i}^+ \rangle & \langle (\mathtt{i}+1)\ \mathtt{i} \rangle = \langle \mathtt{i}^- \rangle \\ \langle \mathtt{i}^+(\mathtt{i}+1) \rangle = \langle \mathtt{i}^+ \rangle & \langle (\mathtt{i}+1)\ \mathtt{i}^- \rangle = \langle \mathtt{i}^- \rangle \\ \langle \mathtt{i}\ (\mathtt{i}+1)^+ \rangle = \langle \mathtt{i}^+ \rangle & \langle (\mathtt{i}+1)^-\mathtt{i} \rangle = \langle \mathtt{i}^- \rangle \\ \langle \mathtt{i}^+(\mathtt{i}+1)^+ \rangle = \langle \mathtt{i}^+ \rangle & \langle (\mathtt{i}+1)^-\mathtt{i}^- \rangle = \langle \mathtt{i}^- \rangle \end{array}$$

The block representation can be computed considering pancakes in any order since the preceding relationships can be applied *associatevily*: $(0^{s_0}1^{s_1})2^{s_2} = 0^{s_0}(1^{s_1}2^{s_2})$, where s_0, s_1 and s_2 stand for the signs of blocks 0, 1 and 2, if any, respectively. This guarantees that the representation with blocks is unique.

In this representation, operators either invert a prefix of blocks or split blocks in four different ways. Thus, any pancake represented with blocks can generate up to five different successors per block:

1. A prefix of blocks can have their sign and position inverted by applying an operator immediately after it only if it consists of more than one block or a single block with sign. For example: the permutation $\langle 1^-2^+0^+ \rangle$ generates the descendants $\langle 0^-2^-1^+ \rangle$, $\langle 2^-1^+0^+ \rangle$ and $\langle 1^+2^+0^+ \rangle$ —the latter being equivalent to $\langle 1^+0^+ \rangle$.
2. An operator can split a block in four different ways (in all the examples the block to split is 2^+):
   (a) It can separate the first pancake of the block from the rest, only if it is not the first block of the permutation. For example, $\langle 1^-2^+0^+ \rangle$ generates $\langle 21^+3^+0^+ \rangle$.
   (b) It can break up a block from the middle creating two smaller blocks. For instance, $\langle 2^-1^+3^+0^+ \rangle$ can be generated from $\langle 1^-2^+0^+ \rangle$.
   (c) If a block has length two, the previous split generates two single blocks, e.g., $\langle 21^+30^+ \rangle$ results from $\langle 1^-2^+0^+ \rangle$.
   (d) Finally, it can get all pancakes in a block from the last one as in $\langle 2^-1^+30^+ \rangle$ which is generated this way from $\langle 1^-2^+0^+ \rangle$.

Note from the preceding discussion that: two blocks can be merged into a single one when applying an operator (as in case 1); also, some blocks with a single disc naturally appear as in cases 2a, 2c and 2d.

From the relations shown above and the redefinition of the operators just discussed it is feasible to generate all the possible combinations of Pancakes as blocks from the identity permutation, which is represented as $\langle 0^+ \rangle$. However, the state space to traverse to visit all the feasible permutations

[2] Since our representation does not model the table as in the case of the gap heuristic, it will always be ommitted.

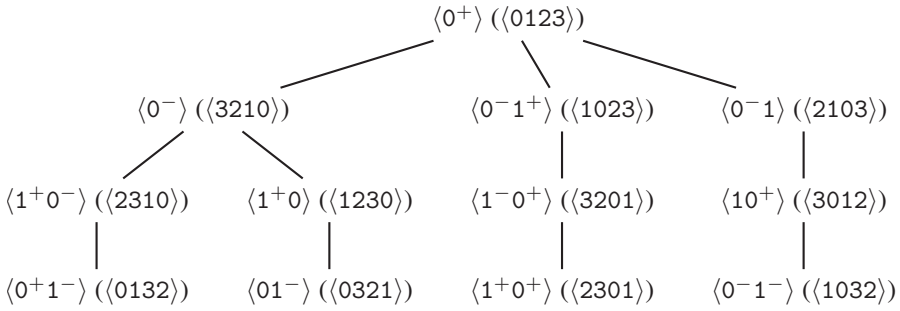

Figure 1: Abstract state space generated with 2 blocks or less (instances of the 4-Pancake are shown between parenthesis)

of blocks grows very rapidly with the number of blocks. The number of permutations that can be achieved with B blocks can be estimated by observing that B blocks can be ordered in $B!$ different ways where each block can have three different signs: positive, negative, or none. Hence, an estimation of the number of positions that can be achieved with a number of blocks less or equal than B is $\sum_{i=1}^{B} i!3^i$. Although this formula is just an upper bound (since it includes impossible cases like $\langle 0^+1^+\rangle$ which is equivalent to $\langle 0^+\rangle$) the number of positions grows with the factorial of B. Thus, a bound is set on the number of blocks when generating PDBs like this to avoid memory exhaustion. This does not mean that descendants with more than B blocks are not traversed. They should indeed (though they are not stored) since an optimal path to a Pancake with B blocks might go through a node with B' blocks, $B' > B$ [3]. Thus, the termination condition is simply that all feasible permutations with up to B blocks have been generated though the generation of these permutations can include the consideration of Pancakes with more blocks. Figure 1 shows the abstract state space generated with $B = 2$ —along with particular instances of the 4-Pancake for the sake of clarity. Pancakes with more than 2 blocks have been explicitly ommitted in the figure since none generates pancakes with 2 blocks or less. As it can be seen, this PDB improves the gap heuristic from 2 to 3 in four cases —namely, $\langle 0^+1^-\rangle$, $\langle 01^-\rangle$, $\langle 1^+0^+\rangle$ and $\langle 0^-1^-\rangle$.

All these permutations can be stored and then looked up in memory when solving a particular instance. Although this abstraction does not ignore any block the distances estimated with this PDB might not be optimal. The reason is that these abstractions get rid of the number of pancakes in each block and optimal solutions of different lengths might exist for blocks with a different number of discs.

Additive Pattern Databases

As a matter of fact, since no block is ignored in the previous representation the previous Pattern Database cannot be used, in general, when solving a particular Pancake. The reason is that there is no guarantee that for Pancakes arbitrarily large, there are a number of blocks less or equal than B, the threshold used in the generation. This difficulty is overcome making the same observation as in ordinary PDBs: ignoring some blocks. In addition to the blocks considered in the previous section, those whose contents are ignored are denoted with uppercase letters from the end of the alphabet and are known as "*don't care*" blocks in contraposition to those blocks whose contents are considered in a particular abstraction (as in the previous section), which are known as "*care*" blocks. "*Don't care*" blocks are indistinguishable, have no sign and they obey the following rule `XY = X`.

During the generation of the Pattern Database, operators are distinguished by the block they are applied to, i.e. the adjacency relationship that they modify. In order to guarantee admisibility we rely on a cost partitioning schema (Yang et al. 2008). Therefore, when ignoring some blocks, moves that change their adjacency are simply ignored. This is, their cost is assumed to be zero. Otherwise, the cost of an operator is computed as follows:

1. If the operator splits a "*care*" block or if it is applied exactly between two "*care*" blocks, the cost equals the cost of the operator, one.
2. However, if the operator is applied right between a "*care*" block and a "*don't care*" block, the cost considered in this particular abstraction is divided by the number of abstractions that will take into account the same operator. Since the "*don't care*" block shall be considered by some other abstraction, but no more than one, the cost is divided by two and 0.5 is added to each abstraction[4].

As a consequence of the consideration of "*don't care*" blocks, PDBs start now from a different specification of the goal pattern, thus allowing the simultaneous consideration of a number of PDBs. For example, a block whose contents are ignored can be considered in the identity permutation either at the beginning or the end resulting in different goal specifications, namely: $\langle 0^+\mathtt{X}\rangle$ and $\langle \mathtt{X}0^+\rangle$.

Clearly, if every abstraction maps different blocks so that the same block is considered in one and only one abstraction, then the values retrieved from different PDBs can be added instead of maximized when using the computation of costs discussed above. However, the gap between two blocks in

[3] Since the gap heuristic is not perfectly informed, there are operators that split a block somewhere along the optimal path for a number of start states. For example, the first state in the optimal path from $\langle 784523061\rangle$ is $\langle 254873061\rangle$ which splits the block containing discs 2 and 3.

[4] To avoid storing numbers in floating point precision, numbers are multiplied by two in the PDB.

different lookups is not always considered. Consider, for example, the Pancake $\langle 21034 \rangle$, which is represented in the blocks formulation as $\langle 0^-1^+ \rangle$. If two PDBs are generated with no more than one block each (i.e., $B = 1$), the first PDB would retrieve the value of $\langle 0^-X \rangle$ whereas the second one would look up the permutation $\langle X0^+ \rangle$. According to the previous rules, the first lookup would return 0.5 since the optimal path from $\langle 0^-X \rangle$ to $\langle 0^+X \rangle$ consists of a single operator that inverts the sign of the first block —indeed, if X would consist of a single positive block then this flip solves the Pancake. However, the second lookup would return 0 because $\langle X0^+ \rangle$ is the goal permutation. Adding both values yields 0.5, whereas the gap heuristic would give one. When the number of PDBs used for computing the heuristic estimate of a particular Pancake grow, these half points result in a significant loss of accuracy.

The key observation is that when mapping a permutation to different PDBs, none of these mappings take into account the relations between different lookups in the same state. The easiest solution simply consists of adding one between them —indeed, there should be a gap otherwise they would form a block. However, this solution allows the possibility of returning inadmissible estimates. This is exemplified with the same Pancake considered above, $\langle 21034 \rangle$. Since there is a gap between the last disc (0) of the block considered in the first lookup ($\langle 0^- \rangle$) and the first disc (3) of the block considered in the second lookup, one can be added. However, this results in an estimate of 1.5 when adding one to the value returned by both PDBs, which is larger than the optimal cost, one. Since we are adding the gap between the first (last) care block and the disc located at its left (right, respectively) in the goal state, we must consider that operator to have a cost equal to zero. In order to be able to identify the discs just at the left (right) of the first (last, respectively) care block in the goal state, two additional "*don't care*" blocks are defined: L and R which consist of a single disc (i.e., they are blocks with no sign) and are always placed to the left and to the right of the first and last block in the goal pattern. This is, L and R are "*don't care*" blocks (consisting of a single pancake) which can be substituted only by the discs preceding the first block of the abstraction or the last one respectively.

Thus, the new goal specifications considered before are represented instead by $\langle 0^+RX \rangle$ and $\langle XL0^+ \rangle$. Henceforth, rule 2 discussed above for computing the cost of an operator that affects a "*don't care*" block is now extended with the following remarks (similar rules follow for the consideration of the "*don't care*" block L):

2(a) Still, if an operator is applied between R and any "*don't care*" block, a cost equal to zero is computed. The reason is that R is also a "*don't care*" block.

2(b) If the operator is applied between the last "*care*" disc of an abstraction and R, zero is added as well, since the consideration of the gap between successive lookups guarantees no loss of accuracy.

2(c) Finally, if the operator is applied between a "*care*" disc other than the last one of the current abstraction and R or between a "*care*" block and any "*don't care*" block other than R, 0.5 is added.

Note there are a high number of potential arrangements of PDBs for solving Pancakes. In this work, abstractions are uniquely identified by their goal pattern. In particular, the following have been chosen: $\langle 0^+ \rangle$, $\langle 0^+RX \rangle$, $\langle XL0^+ \rangle$ and $\langle XL0^+RX \rangle$. The first one was introduced in the second section and it is the PDB of choice when the number of blocks is less or equal than B, the number of blocks used for its generation. The rest can be applied in spite of the number of blocks present in the current permutation: $\langle 0^+RX \rangle$ and $\langle XL0^+ \rangle$ have been already discussed in this section and can be applied when the mapped blocks shall be located at the beginning or the end of the goal permutation; the last one, $\langle XL0^+RX \rangle$, can be applied when the number of blocks identified exceeds B and they are neither at the beginning nor the end of the goal state.

For the sake of clarity an example follows: consider the 10-Pancake $\langle 6714508329 \rangle$ which is uniquely identified by the following permutation in the blocks notation $\langle 4^+13^+052^-6 \rangle$. Using the preceding PDBs with $B = 2$ blocks or less in $\langle XL0^+RX \rangle$ and $B = 3$ blocks or less for all the others, PDB $\langle 0^+ \rangle$ cannot be used since the current Pancake has 7 blocks. Instead, the following lookups are performed (bear in mind that each block shall appear in one PDB and only one): blocks 0, 1 and 2 are mapped in the PDB $\langle 0^+RX \rangle$ since these shall be located at the beginning in the goal state, resulting in the lookup of $\langle X1RX0X2^-X \rangle$, which returns 2; blocks 3 and 4 are used in the abstraction $\langle XL0^+RX \rangle$ because these shall be restored to intermediate positions in the goal permutation. This mapping results in the permutation $\langle 1^+X0^+XRLX \rangle$ which returns 1.5. Finally, there are only two blocks awaiting to be mapped: 5 and 6. Since these shall be restored to the end of the goal state, the PDB $\langle XL0^+ \rangle$ is used and the resulting lookup to the PDB is $\langle XLX0X1 \rangle$ which returns 1. Adding all these values yields the heuristic estimate 2 + 1.5 + 1 = 4.5 which can be enhanced further by adding two gaps, those between blocks 2 and 3 (used as the last and first ones in the first two lookups) and blocks 5 and 6 —used as the last and first ones in the last two lookups. Therefore, the resulting estimate is 4.5+2=6.5 which can be rounded up to 7. This result improves the gap heuristic by one.

Before moving on to the experimental results, a discussion is conducted now on inconsistency and weather duality can improve the regular estimates or not.

First, these PDBs easily generate inconsistencies —i.e., the absolute value of the difference between the heuristic estimates of a node and any of its children exceeds the cost of the operator. This happens when a pancake n has a descendant which creates a gap and, in addition, the PDBs realize that an extra move is still required so that the descendant has a heuristic value of $h(n) + 2$.

Second, while the gap heuristic cannot be improved with the idea of duality, these PDBs can be used to improve the heuristic estimate with a dual lookup. In fact, the dual of the gap heuristic wrt. the identity permutation is the gap heuristic itself since the definition of a gap as shown in the Introduction, $|\pi_i - \pi_{i+1}| > 1$, $0 \leq i < N$, is strictly equivalent when considering the inverse of π, π^{-1}, resulting in $|\pi_i^{-1} - \pi_{i+1}^{-1}| > 1$, $0 \leq i < N$. In words, if there is a gap

between two adjacent locations, then these locations will be placed in positions far from each other in the dual state, thus preserving the same gap. However, the dual state of a block representation can result in a different value stored in the PDB. Moreover, the dual lookup of a block representation can be readily computed: if there is a block of length k starting at location i, the same block (with the same length and sign) is preserved in the dual representation wrt. to the identity permutation starting at location π_i. The only difference is that blocks might be renamed when computing the dual representation in the blocks notation resulting in little overhead, if any.

Related Work

In (Yang et al. 2008), an additive abstraction is proposed for solving the pancake problem with two possible cost distributions over the operators: cost-splitting and location-based. The one with better results for the pancake problem is the location-based policy. This schema counts most of the gap relationships: the ones between two care blocks on the same abstraction, as to repair a gap between the two care discs, one of them has to be moved from the first position to the adjacent location. Only those gaps between discs considered in different PDBs will be ignored. However, as this kind of PDBs maps the location of each disc, their size grows with the pancake size, limiting the effectiveness of this technique in problems with a large number of discs.

The relative order abstractions (Helmert and Röger 2010) are size-independent PDBs for the pancake problem, which consider the cost of sorting a limited number of discs distinguished by their relative order (12 discs at most were used with a total memory usage of roughly 500 MB). However, when making more than one lookup the result can only be the maximum among all, limiting the maximum heuristic value that a state can have independently of the size of the complete pancake. This greatly reduces its effectiveness as the pancake size grows.

The better known heuristic for the pancake problem is the gap heuristic (Helmert 2010). Our heuristic counts the number of gaps as well but also, very importantly, it detects some patterns or relationships between blocks locations that need an extra operator that does not reduce the number of gaps. Therefore, it is more informed than the gap heuristic. An interesting question, regarding the utility of our heuristic is how many node expansions can be saved for these patterns. For example, in $\langle \mathtt{0}^{+}\mathtt{RX} \rangle$, the pattern $\langle \mathtt{0}^{+}\mathtt{XR} \rangle$ detects that none of the available operators reduces a gap. While this can be easily detected just expanding the node and realizing that all descendants preserve the number of gaps, there are more interesting patterns whose detection can lead to pruning large parts of the search tree. Take, for example, pancake $\langle \mathtt{XL0}^{+} \rangle$. The pattern $\langle \mathtt{XL0}^{-} \rangle$ needs an extra move bringing the block 0 to the first position. As a "*don't care*" symbol is at the start of the pancake and it may consist of any number of disc with any number of gaps, there could be a large number of node expansions saved. Thus, identifying such "*conflicts*" can make this heuristic significantly more informed than the gap heuristic.

	h^{b2}		h^{b3}	
	B	memory	B	memory
$\langle \mathtt{0}^{+} \rangle$	4	2,127	5	31,287
$\langle \mathtt{0}^{+}\mathtt{RX} \rangle$	3	21,648	4	643,728
$\langle \mathtt{XL0}^{+} \rangle$	3	21,648	4	643,728
$\langle \mathtt{XL0}^{+}\mathtt{RX} \rangle$	2	7,200	3	214,560
Total		52,623		1,533,303

Table 1: Threshold on the number of care blocks and overall size of each PDB (in bytes)

Concluding, this work contributes to the current state of the art in three different ways:

- Using a block representation makes our heuristic size-independent, allowing its application on bigger problems.
- The inclusion of L and R discussed in section 3 allows the consideration of other gaps between different lookups that would be otherwise overlooked.
- As shown in the Experimental Results section, our heuristic saves various orders of magnitude on the number of nodes generated when compared with any other Pattern Databases known up to date and still a significant number when being compared to the gap heuristic and various orders of magnitude in the time spent for solving particular instances when being compared to other PDBs.

Experimental Results

As mentioned in the third section, the following PDBs have been generated: $\langle \mathtt{0}^{+} \rangle$, $\langle \mathtt{0}^{+}\mathtt{RX} \rangle$, $\langle \mathtt{XL0}^{+} \rangle$ and $\langle \mathtt{XL0}^{+}\mathtt{RX} \rangle$ with two different thresholds on the maximum number of care blocks, B. The heuristic function that results from adding the values retrieved from each configuration is denoted as h^{b2} and h^{b3}. Table 1 shows the maximum number of care blocks in the stored patern selected for each PDB and their overall size in bytes. To ensure that it is possible to match all the "*don't care*" blocks when using the PDBs, B is greater for those PDBs with less care blocks, avoiding the creation of PDBs with the goal patterns $\langle \mathtt{L0}^{+} \rangle$, $\langle \mathtt{0}^{+}\mathtt{R} \rangle$, and $\langle \mathtt{L0}^{+}\mathtt{R} \rangle$. For that reason h^{b2} sets $B = 2$ for the PDB $\langle \mathtt{XL0}^{+}\mathtt{RX} \rangle$, $B = 3$ for $\langle \mathtt{XL0}^{+} \rangle$ and $\langle \mathtt{0}^{+}\mathtt{RX} \rangle$ and, finally $B = 4$ in $\langle \mathtt{0}^{+} \rangle$. On the other hand, h^{b3} expands an additional care block in each PDB, with $B = 3$ in $\langle \mathtt{XL0}^{+}\mathtt{RX} \rangle$, $B = 4$ in $\langle \mathtt{XL0}^{+} \rangle$ and $\langle \mathtt{0}^{+}\mathtt{RX} \rangle$ and $B = 5$ in $\langle \mathtt{0}^{+} \rangle$. The memory requirements listed in table 1 are larger than the upper bound computed in the second section. This results from the ranking schema chosen which wastes some space with the benefit of making the ranking procedure faster. Despite this, the largest PDB (h^{b3}) take altogether less than 2 MB of space.

Table 2 shows the optimal cost and mean number of nodes generated by the gap heuristic and h^{b2} and h^{b3} for solving 1000 instances randomly selected from $N = 10$ to 42 discs with the IDA* with the Bidirectional pathmax propagation rule. These experiments were run on an Intel Core 2 duo 1.86GHz with 8GB RAM using the Java virtual machine 1.6.1. The generated nodes shown for the gap heuristic are consistent with those reported before (Helmert 2010), but

		Generated nodes			Time (sec)		
N	h^*	h^{gap}	h^{b2}	h^{b3}	h^{gap}	h^{b2}	h^{b3}
10	8.684	294.339	121.480	87.910	0.102	0.361	0.210
11	9.717	570.040	232.331	164.630	0.131	0.745	0.435
12	10.715	927.181	425.946	306.211	0.244	1.385	0.924
13	11.685	1,476.777	676.725	488.963	0.346	2.330	1.512
14	12.723	2,073.177	1,045.742	789.024	0.537	3.841	2.626
15	13.699	2,793.450	1,495.204	1,147.949	0.747	5.826	4.095
16	14.699	4,443.078	2,516.042	1,967.638	1.177	10.304	7.429
17	15.805	7,028.581	3,829.974	3,066.605	1.899	16.629	12.239
18	16.730	9,229.122	5,341.750	4,198.285	2.569	24.546	17.695
19	17.719	12,105.556	7,538.589	6,265.455	3.511	36.244	27.652
20	18.710	20,159.721	11,398.400	9,259.529	5.980	57.848	43.441
21	19.730	24,557.891	15,496.780	12,746.851	7.615	83.016	62.878
22	20.709	33,711.946	22,027.050	18,497.313	10.585	124.214	95.929
23	21.748	48,366.230	31,689.464	26,695.949	15.488	186.298	144.250
24	22.633	50,187.097	33,755.005	28,891.763	16.283	208.918	164.142
25	23.720	79,348.358	55,607.005	47,473.662	26.236	358.042	280.647
26	24.735	94,388.283	67,548.043	58,659.569	31.392	456.075	363.845
27	25.689	113,317.403	82,122.610	72,002.949	39.268	576.897	464.681
28	26.697	153,972.289	108,525.645	94,103.649	54.362	798.533	633.978
29	27.754	183,222.217	133,295.550	116,548.775	65.879	1,019.417	818.450
30	28.675	263,108.280	177,918.376	155,474.814	95.495	1,416.736	1,135.227
31	29.702	360,065.623	209,150.386	187,876.132	132.722	1,728.874	1,426.111
32	30.750	357,276.950	262,369.433	231,016.450	134.709	2,259.041	1,825.603
33	31.639	458,478.223	345,145.443	311,125.870	173.690	3,083.068	2,544.769
34	32.680	525,150.190	397,217.293	355,743.816	202.403	3,689.623	3,025.660
35	33.664	734,044.287	553,287.486	495,887.725	286.105	5,352.354	4,391.045
36	34.665	820,331.125	632,925.129	562,590.146	328.954	6,305.066	5,114.836
37	35.682	1,290,816.553	988,602.809	872,516.678	513.398	10,139.311	8,202.614
38	36.695	1,305,943.076	1,014,136.669	918,736.036	531.696	10,792.457	8,923.332
39	37.683	1,564,876.327	1,258,145.321	1,156,110.937	645.798	13,743.394	11,496.780
40	38.703	1,708,990.215	1,368,569.706	1,259,357.089	727.621	15,482.981	12,966.377
41	39.708	2,856,850.225	2,268,045.902	2,049,308.053	1,224.695	26,505.072	21,840.799
42	40.711	2,626,988.835	2,130,661.501	1,972,564.906	1,144.392	25,833.201	21,709.817

Table 2: Mean number of nodes generated and time spent (in milliseconds) by the gap heuristic and the blocks abstraction in pancakes with a number of discs ranging from 10 to 42

the mean times reported here are slightly greater. This effect is attributed to our implementation in Java in contraposition to the most effective implementation in C used in the original experiments with the gap heuristic.

The results show that both h^{b2} and h^{b3} generate significantly fewer nodes than the gap heuristic at the cost of taking longer for the evaluation of every node. While the gap heuristic results in an elegant definition that can be evaluated very fast, the ranking mechanism imposed by Pattern Databases can become a serious bottleneck. In fact, h^{b2} and h^{b3} take from 6 (for the easiest problems) up to 20 times more than the gap heuristic for completing the search. When comparing the number of nodes, h^{b2} saves a number of nodes ranging from 40% (for the lowest values of N) to 25%, for the largest Pancakes; on the other hand, h^{b3} saves between 55% and 30% generations. Thus, as the pancake length grows, the percentage reduction on the number of nodes wrt. the gap heuristic decreases. As a consequence, the difference in runtimes between h^{gap} and our new heuristics increase with the pancake length. Besides, the differences between h^{b2} and h^{b3} does not suggest that increasing the number of care blocks could lead to a significant improvement over h^{gap} as the pancake gets larger.

Table 3 shows a comparison with the best results reported up to date[5] with Pattern Databases. This includes several configurations for the location-based PDBs (Yang et al. 2008) and the relative order abstractions (Helmert and Röger 2010). Clearly, additive PDBs generate fewer nodes and take less time than relative-order PDBs. However, the latter have constant memory requirements and the same PDB can be used for problems of any size. h^{b2} and h^{b3} take the best from each, expanding fewer nodes than any and solving problems remarkably faster. For example, pancakes of length 17, 18 and 19 are solved saving four, five and seven orders of magnitude while taking about two orders of magnitude less space in comparison with relative-order PDBs.

[5]To the best of the authors' knowledge

N	PDB technique	Nodes	PDB size (MB)	Time (s)
17	Relative order DIDA* 2x50 random pancake sets	411,830	$\approx$ 479	60.390
	Relative order IDA* 50 random pancake sets	673,340	$\approx$ 479	49.300
	Relative order DIDA* 2x4 random lookups and 2xobject-location abstraction	2,362,899	$\approx$ 479	31.620
	Additive location-based DIDA* 4-4-5	368,925	$\approx$ 1	0.195
	Additive location-based DIDA* 5-6-6	44,618	$\approx$ 18	0.028
	Additive location-based DIDA* 3-7-7	37,155	$\approx$ 196	0.026
	h^{b2}	3,830	$\approx$ 0.053	0.017
	h^{b3}	3,067	$\approx$ 1.5	0.013
18	Relative order DIDA* 2x5 random lookups	13,439,483	$\approx$ 479	209.810
	Relative order DIDA* 2x10 random lookups	7,638,885	$\approx$ 479	236.100
	h^{b2}	5,342	$\approx$ 0.053	0.025
	h^{b3}	4,198	$\approx$ 1.5	0.018
19	Relative order DIDA* 2x5 random lookups	1,236,838,871	$\approx$ 479	20,409.900
	Relative order DIDA* 2x10 random lookups	689,598,292	$\approx$ 479	22,576.390
	h^{b2}	7,539	$\approx$ 0.053	0.036
	h^{b3}	6,265	$\approx$ 1.5	0.028
$\geq$20	Relative order	Not reported		
	Additive location-based	Not reported		
	h^{b2} and h^{b3}	More than 40 pancakes		

Table 3: Comparision of various PDBs in the number of nodes generated, size and time spent in the 17, 18, 19-pancake and above

Since h^{b2} and h^{b3} can be enhanced by taking the maximum from the sum of the regular lookups and the dual lookups, we also used duality for reducing the number of nodes. Being so close to the optimal value the reduction in the number of nodes (not shown in Table 2 due to lack of space) ranged from 50% (for small values of N) to 25% —for the most difficult instances. In comparison with the gap heuristic, the dual lookups of h^{b2} save between 70% and 40% generations, whereas the dual lookups of h^{b3} result in savings between 80% and 50%. However, making two lookups affects the overall running time by a factor of two. Using DIDA* did not significantly improve our results.

Conclusions and Future Work

This work does not set a state-of-the-art heuristic for the pancake problem. Although the technique discussed here reduces significantly the number of nodes generated, the gap heuristic obtain optimal solutions faster. However, it contributes to the current state of the art in the study of Pattern Databases by showing how to create PDBs which are size independent (so that they are generated only once); which take two orders of magnitude less space than those producing the best results known so far; and which allow solving Pancakes with twice as many discs as the Pancakes solved by previous approaches with Pattern Databases.

There are several lines to research further in the context of this work in order to improve the results on the Pancake problem. First, faster heuristic evaluations could be achieved through an incremental calculation over the gap heuristic, being only neccesary to determine if some conflicts can be added to the number of gaps. Interestingly, paying attention to Figure 1, it becomes clear that all abstractions ending in $\mathtt{1}^-$ can add one to the gap heuristic —this is true, for $\langle \mathtt{0}^-\mathtt{1}^- \rangle$, $\langle \mathtt{0}^+\mathtt{1}^- \rangle$, and $\langle \mathtt{01}^- \rangle$. In fact, it is trivial to show a further generalization of this rule and all pancakes ending in $\langle \mathtt{k}^- \rangle$, with k being the larger block add one move to the gap heuristic. This and other relations can be automatically discovered with these PDBs leading to more accurate estimations. Moreover, these relations might lead to a further compression ratio of these PDBs. Finally, in order to improve the heuristic informedness, some considerations about the size of the block can be made. For example, distinguishing blocks with two discs from those with more discs can be very helpful as operators splitting a block into two single discs are only applicable in the first case.

On the other hand, while the gap heuristic only counts the number of gaps, block-based abstractions take into account their relative location. Consequently, it seems reasonable to expect that PDBs like this could be useful in those problems where adjacency is a key property of the domain. Interesting domains for applying this kind of abstractions include the burnt-pancake introduced by Gates and Papadimitriou (Gates and Papadimitriou 1979), and the genome rearrangement problem formalized as a planning problem in (Erdem and Tillier 2005). In the burnt pancake problem, the block representation can be applied just considering that all blocks are signed. In this case the gap heuristic can be extended by considering that two pancakes which are adjacent in the goal location appear together in the current permutation do generate a gap if they are not oriented in the same direction. Still, the gap heuristic is bounded by the number of pancakes. However, the current upper bound for the burnt pancake might be larger than the unburnt pancake (considered here) so that more opportunities are given to these PDBs for improving over the gap heuristic. In the genome rear-

rangement problem, operators are applied over any number of adjacent genes, so that the size of the block can be ignored. Therefore, the block representation can be expected to provide good results. The operators can be seen as follows: *transposition* swaps the location of two blocks while preserving their sign; *inversions* change the sign of a block while preserving its location; *transversions* change the location and position of a block. Of course, these operators can be applied within the blocks, splitting them in two. With operators fitting so well the block representation, it only remains to choose a good cost-splitting schema.

Acknowledgements

This research is partially supported by the Spanish Governement MICINN projects TIN2008-06701-C03-03, TIN2008-06701-C03-03 and Comunidad de Madrid - UC3M CCG10-UC3M/TIC-5597. The first author is supported by a PhD grant from Universidad Carlos III de Madrid.

References

Chitturi, B.; Fahle, W.; Meng, Z.; Morales, L.; Shields, C. O.; Sudborough, I. H.; and Doit, W. 2009. An $(18/11)n$ upper bound for sorting by prefix reversals. *Theoretical Computer Science* 410(36):3372–3390.

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

Dweighter, H. 1975. Problem E2569. *American Mathematical Monthly* 1010(82).

Erdem, E., and Tillier, E. 2005. Genome rearrangement and planning. In *Proceedings of the 20th national conference on Artificial intelligence - Volume 3*, 1139–1144. AAAI Press.

Gates, W. H., and Papadimitriou, C. H. 1979. Bounds for sorting by prefix reversal. *Discrete Mathematics* 27:47–57.

Helmert, M., and Röger, G. 2010. Relative-order abstractions for the pancake problem. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI-10)*, 745–750.

Helmert, M. 2010. Landmark heuristics for the pancake problem. In *The Third Annual Symposium on Combinatorial Search (SOCS-10)*, 109–110.

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

Yang, F.; Culberson, J. C.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A general theory of additive state space abstractions. *Journal of Artificial Intelligence Research* 32:631–662.

Zahavi, U.; Felner, A.; Holte, R. C.; and Schaeffer, J. 2007. Duality in permutation state spaces and the dual search algorithm. *Artificial Intelligence* 172:514–540.