Optimally solving permutation sorting problems with efficient partial expansion bidirectional heuristic search

Marco Lippi
Dipartimento di Informatica – Scienza e Ingegneria, Università degli Studi di Bologna, viale Risorgimento 2, 40136 Bologna, Italy
E-mail: marco.lippi3@unibo.it

Marco Ernandes
Quest-IT s.r.l., strada Massetana Romana, 44, 53100 Siena, Italy
E-mail: ernandes@quest-it.com

Ariel Felner
Department of Information System Engineering, Ben Gurion University, Israel
E-mail: felner@bgu.ac.il

In this paper we consider several variants of the problem of sorting integer permutations with a minimum number of moves, a task with many potential applications ranging from computational biology to logistics. Each problem is formulated as a heuristic search problem, where different variants induce different sets of allowed moves within the search tree. Due to the intrinsic nature of this category of problems, which in many cases present a very large branching factor, classic unidirectional heuristic search algorithms such as A* and IDA* quickly become inefficient or even infeasible as the problem dimension grows. Therefore, more sophisticated algorithms are needed. To this aim, we propose to combine two recent paradigms which have been employed in difficult heuristic search problems showing good performance: *enhanced partial expansion* (EPE) and *efficient single-frontier bidirectional search* (eSBS). We propose a new class of algorithms combining the benefits of EPE and eSBS, named efficient Single-frontier Bidirectional Search with Enhanced Partial Expansion (eSBS-EPE). We then present an experimental evaluation that shows that eSBS-EPE is a very effective approach for this family of problems, often outperforming previous methods on large-size instances. With the new eSBS-EPE class of methods we were able to push the limit and solve the largest size instances of some of the problem domains (the pancake and the burnt pancake puzzles). This novel search paradigm hence provides a very promising framework also for other domains.

Keywords: heuristic search, bidirectional search, permutation sorting

1. Introduction and background

Sorting permutations of integers is a classical problem in optimization and discrete mathematics. Given a permutation of the first M integers, the aim is to transform it into another *goal* permutation, which is typically the sorted configuration $\{1, \dots, M\}$, by minimizing some cost function, such as the number of required moves. Depending on which moves are allowed, several variants of this sorting problem can be conceived. If the only allowed operators consist in flipping a subsequence of the first k integers within the permutation, then the resulting problem is formally defined as *sorting by prefix reversal* (Figure 1a on page 5). This problem is known as the pancake puzzle (P1) [16], since it resembles the way in which a stack of pancakes can be sorted on a plate using a spatula. In the *burnt pancake puzzle* (P2) variant (also introduced in [16]), a move also changes the sign of the integers involved in the flip (Figure 1b). In that case, the sign indicates whether the burnt side of the pancake is faced up or down, and the goal is always to have all pancakes with burnt face down (thus no integer with negative sign in the final configuration). Another possibility allows moves which swap two non-overlapping subsequences of any length: this problem is known as *sorting by block interchanges* (P3) [8] (Figure 1c). If the two substrings are required to be adjacent, then we speak of *sorting by transpositions* (P4) [2]

(Figure 1d). The *sorting by reversals* (P5) [1] problem can be seen as a generalization of the pancake puzzle, where any subsequence, and not necessarily a prefix, can be reversed (Figure 1e). The *signed* variant of this problem is a natural generalization of the burnt pancake puzzle. By merging reversals and block interchanges, the problem of *sorting by translocations* (P6) is defined [4] (Figure 1f), again with the possibility of signed configurations. We provide detailed definitions of these six problem variants below.

The aforementioned problems can be applied also to circular permutations, where elements are arranged in a circular array rather than in a sequence [36]. A special case here is given by the Top-Spin puzzle [25], where only flips of a fixed length k are allowed. Another variant of all permutation problems consists in limiting the size of the allowed move, in which case the problem is commonly referred to as *bounded* (e.g., sorting permutations by bounded transpositions) [22]. A full consideration of circular and bounded problems is beyond the scope of this paper, where we will focus on the plain version of each problem.

Practical applications of permutation sorting problems are found in routing algorithms and parallel computing [34], logistics [38], and particularly in computational biology for genome rearrangement and evolution studies [1,8,2,27,18,37,39].

For most of these problems it is easy to construct a solution by a set of macros or procedures. However, such solutions are only sub-optimal, even though often with good approximation bounds (e.g., see [11] and references therein). By contrast, finding optimal solutions (i.e., transforming the initial permutation in the goal using a minimal number of moves) is far more complicated, but it strongly depends on the problem variant. Sorting by blocks interchanges (P3) has been proven to be solvable in polynomial time [8], while NP-completeness has been proven for the pancake problem (P1) [5], and NP-hardness for sorting by transpositions [6]. Sorting by reversals (P5) and by translocations (P6) can be solved in polynomial time only in the signed case [19,20], while being NP-hard in the unsigned case [7,41]. No result has been found yet for the burnt pancake problem (P2), yet it has been shown that some sets of configurations can be optimally solved in polynomial time [31]. The literature on approximation algorithms for all these variants of permutation sorting problems is extremely wide: for example, see the survey in [15,32] and references therein. The focus of this paper is instead on optimal solutions for such problems, and in particular on the application of heuristic search algorithms for these tasks.

1.1. Background: heuristic search algorithms

Consider the task of finding the shortest path between two states s and g on an undirected graph. Traditionally, unidirectional search algorithms build a search tree where each node of the tree includes one state of the graph. The root node S includes the start state s. Assume that node X in the tree corresponds to state x in the graph: the task at X is to find a (shortest) path between x and g. Heuristic (sometimes also called informed) search algorithms exploit the use of a *heuristic function* $h(x)$ to estimate the cost of the path from x to g, so that the search tree can be efficiently traversed in order to find the solution. A* [21] and IDA* [30] are among the most used algorithms in this context. A* uses a *best-first* approach to choose the states to be visited during the search. An ordered queue of nodes (called OPEN) is maintained, based on function $f(x) = g(x) + h(x)$, where g is the cost of the path from the start node up to x, and h is the heuristic function. The first node in this queue is removed and expanded at each step of the algorithm. IDA* is the Iterative-Deepening version of A*, which performs iterative-deepening depth-first search guided by the function f. The optimality of A* and IDA* is guaranteed if the heuristic function h driving the search is *admissible*. This means that it never overestimates the distance between the current state and the goal:

$$h(x) \leq h^*(x)$$

where $h^*(x)$ is the optimal distance between node x and the goal. In order to speed up the search, A* can also employ a list named CLOSED. CLOSED is used to construct the solution path and to prune nodes which have already been visited. CLOSED can be used to prune nodes when seen for the second time in A* only in the case that the heuristic function is *consistent*[1]. Otherwise nodes already

[1]Given two nodes x and y in the search tree, such that y is a descendant of x, and $g(x, y)$ is the cost of the path from x to y, a heuristic h is said to be consistent if $h(x) \leq g(x, y) + h(y)$ [14]

visited could be rediscovered with an improved cost and should be re-opened (i.e., moved from CLOSED to OPEN).

The pancake puzzle has become a classic benchmark within this field [23,33], and a few works have also considered the burnt pancake problem [29]. Nevertheless, other variants of sorting problems have so far received no attention. Since this family of problems presents many cases with a very large branching factor, sophisticated algorithms have to be employed in order to build efficient solvers.

1.2. New search algorithms

In this paper we consider two recently introduced search paradigms which can greatly speed up the search process and which perfectly fit the considered problem domains mention above: *enhanced partial expansion* (EPE) and *efficient single-frontier bidirectional heuristic search* (eSBS). EPE [12,17] was introduced as an enhanced version of partial expansion (PE) [40]. The key idea of this algorithm is to avoid the generation of those nodes having an f-value larger than the optimal solution. Coupled with A^* and IDA^*, the EPE search paradigm has shown very strong results on a number of puzzles (including the pancake puzzle), as well as on single-agent and multi-agent path-finding problems [17,35].

Additionally, since the goal state is known, bidirectional search can be naturally applied within the context of permutation sorting. Yet, applying bidirectional heuristic search to these problems so far has been limited to the pancake puzzle only [33] in the form of *Single-frontier bidirectional search* (SBS) [13]. SBS is a bidirectional search algorithm which works in the state-space of *tasks* – nodes that include a pair of states. The task inside a node is to find a shortest path between the states of the corresponding pair. At each node, SBS chooses which of the two states to expand, potentially changing the side of the search. *Efficient single frontier bidirectional search* (eSBS) [33] is an enhanced variant of of SBS, which exploits several pruning and caching techniques in order to speed up the search process while maintaining low memory consumption in comparison to classic unidirectional algorithms.

1.3. Contributions

The main contributions of the paper can be summarized as follows:

- We propose permutation sorting problems as new challenging benchmarks for testing heuristic search algorithms, due to the diversity of the problems that can be conceived and the number of practical applications that can be drawn from them.
- We present heuristic functions for each of the problem variants, with a special mention for the burnt pancake puzzle, for which we introduce a novel, powerful heuristic, named the *oriented gap* heuristic. As we show below, this heuristic pushes the limit on the size of problems that can be solved.
- We introduce a new class of algorithms, which combine enhanced partial expansion and efficient single-frontier bidirectional search. The resulting combination is a framework for heuristic search, that perfectly fits the domain of permutation sorting, but which could also be successfully applied to other tasks.
- We provide experimental results over a variety of different settings that show the benefits of the new framework, which often outperforms any of the previous approaches. In particular we obtained state-of-the-art results for both the pancake and the burnt pancake puzzle by solving problems with the largest dimension ever addressed until now.

The paper is structured as follows: Section 2 provides a formal definition of the different types of permutation sorting problems addressed in this work, together with heuristic functions for each proposed problem. Sections 3 and 4 will then review the Enhanced Partial Expansion (EPE) heuristic search method, and the efficient Single-frontier Bidirectional Search (eSBS) paradigm. These two algorithms will be combined into the efficient Single-frontier Bidirectional Search with Enhanced Partial Expansion (eSBS-EPE) framework, described in Section 5. In Section 6 we present and discuss experimental results. Finally, Section 7 concludes this paper.

2. Sorting problems and their heuristics

In this section we formally define the different variants of the integer sorting problem, and we introduce heuristic functions for each of them. To ease the notation in the description and following the approach in [23], a sorting problem of size M is defined with $M+1$ integers, where the last element is always $M+1$ and is never moved.[2] Such element will be used in some definitions below.

Consider a permutation $\pi = \{\pi_1, \ldots, \pi_M, \pi_{M+1}\}$ of the first M integers (assuming $\pi_{M+1} = M+1$). The goal of *permutation sorting* is to find a sequence of allowed operators $m = \{m_1, \ldots, m_k\}$ which transform π into a given *goal* configuration γ, which is usually the sorted configuration: $\gamma = \{1, \ldots, M, M+1\}$. Optimal solutions aim at identifying m so that the sequence of moves minimizes some cost function. The most typical case is to consider all allowed moves as equally costly, and therefore to minimize the number k of total needed moves. For simplicity in the reminder of this paper we assume this unit operator cost function. The problem of permutation sorting with an optimal number of moves can be easily formulated as a search problem, where the starting node contains the initial configuration, the goal node is represented by the goal configuration, and a shortest path has to be found between these two nodes. Starting from the initial state, a search tree is built, where the children of a certain node are all the possible successors of that node, as a function of the allowed moves.

As already stated in Section 1, different variants of this problem can be modeled by different sets of allowed moves. The average number of successors within the search tree determines the average *branching factor* of the problem set. Note that the branching factor greatly changes within the variety of permutation sorting problems, leading to tasks with different complexity for heuristic search.

We now turn to define the different variants of permutation sorting problems that we work with in this paper.

[2]For example, in the pancake problem, this last element can be seen as the *plate* or *table* on which the M pancakes are arranged.

2.1. Pancake problem (P1)

In the pancake puzzle, the allowed moves just flip a permutation prefix of length k, where $1 \leq k \leq M$. Therefore, a k-flips transforms $\pi = \{\pi_1, \ldots, \pi_{M+1}\}$ into

$$\hat{\pi} = \{\pi_k, \pi_{k-1}, \ldots, \pi_1, \pi_{k+1}, \ldots, \pi_{M+1}\}$$

The problem is depicted in Figure 1 (a), where a 4-flip move is illustrated. The branching factor is $M-1$ as flipping only the first pancake is useless.

For P1, a known very effective heuristic function is the *gap* heuristic [23]. This function simply counts the *gaps* within a given sequence $\{\pi_1, \ldots \pi_M, \pi_{M+1}\}$, where a position in the sequence contains a gap if the pancake in that position is not adjacent to the pancake below:

$$\begin{aligned} h_1(s) = h^{gap}(s) := |\{i \ \ s.t. \ \ i \in \{1, \ldots, M\}, \\ |\pi_i - \pi_{i+1}| \neq 1\}\,| \end{aligned} \tag{1}$$

For example, the following sequence s_1:

$$3 \quad 6 \quad 5 \quad 4 \quad 1 \quad 2 \quad 7$$

with $M = 6$, contains three gaps, one between 3 and 6, another one between 4 and 1, and finally the last one between 2 and 7 (where 7 is "the table"). Every move in the pancake puzzle flipping k positions can potentially eliminate only *one* gap, namely the one between position k and $k+1$, which ensures admissibility. Thus, $h_1(s_1) = 3$.

2.2. Burnt pancake problem (P2)

In the burnt pancake variant, integers are signed and therefore the same move mentioned for P1 would end up with configuration

$$\hat{\pi} = \{-\pi_k, -\pi_{k-1}, \ldots, -\pi_1, \pi_{k+1} \ldots, \pi_{M+1}\}$$

Figure 1 (b) shows an example of a 3-flip move, where each integer involved in the flip also changes its sign. Note that, since the integers are signed, this problem cannot be strictly considered as a permutation sorting task. In P2 the branching factor is always M (one can flip a number of pancakes ranging from 1 up to M): unlike P1, here flipping the first pancake flips its sign.

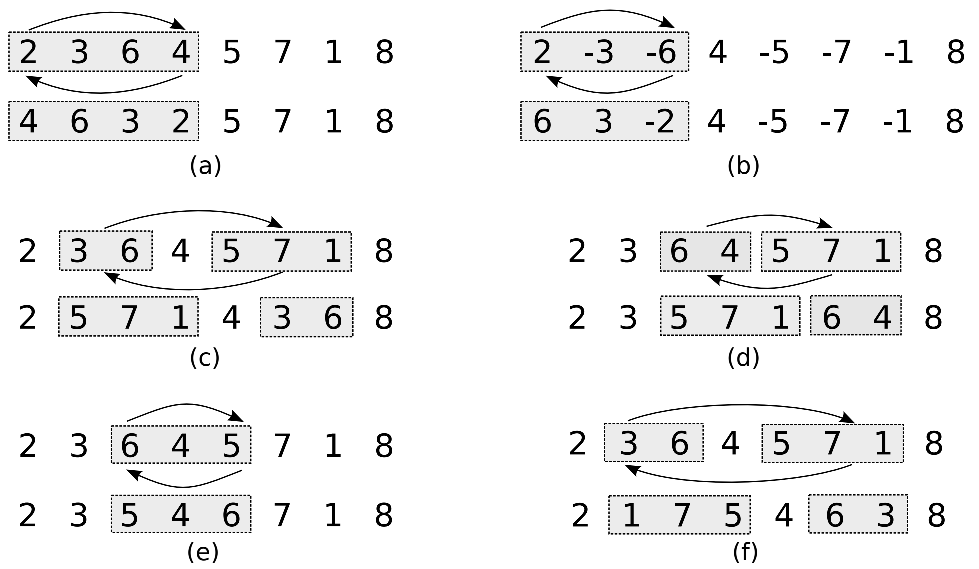

Fig. 1. Examples of different permutation sorting problems: (a) pancake puzzle, or sorting by prefix reversal; (b) burnt pancake puzzle; (c) sorting by block interchanges; (d) sorting by transpositions; (e) sorting by reversals; (f) sorting by translocations.

In the burnt pancake puzzle an admissible heuristic can be conceived by extending the same principle driving the gap heuristic, thus counting the gaps between the *absolute values* of the elements in each position:

$$h^{gap}(s) := |\{i \;\; s.t. \;\; i \in \{1,\ldots,M\}, \\ ||\pi_i| - |\pi_{i+1}|| \neq 1\}|$$

Yet, such a heuristic is much less informative than in the pancake puzzle case, because it does not consider the side (sign) of the pancakes. In fact, a more informative heuristic has been defined in [29] as follows:

$$h^{gap}_{burnt}(s) := |\{i \;\; s.t. \;\; i \in \{1,\ldots,M\}, \\ ||\pi_i| - |\pi_{i+1}|| > 1 \vee \pi_i \cdot \pi_{i+1} < 0\}| \tag{2}$$

where a gap is counted for a certain position either if the pancake in that position is not of adjacent size to the pancake below, or if they have opposite signs. As a matter of fact, even h^{gap}_{burnt} is not a highly informative heuristic. For example, given the following configuration for a burnt pancake puzzle problem with $M = 7$:

$$-1 \quad -2 \quad -3 \quad -4 \quad -5 \quad -6 \quad 7 \quad 8$$

only one gap will be detected by h^{gap}_{burnt} (between pancakes 6 and 7 which have opposite signs), while the depth of the solution is 12. What is missing in this heuristic, in fact, is that in the burnt pancake also the *orientation* of the gap has to be considered: there is no gap between -1 and -2, but actually *they are in correct order but with wrong signs.* Any flip involving both pancakes will not change the situation, since, when flipped into the subsequence "2 1" they will be *with correct signs but in wrong order.* The only way to solve this situation is a move which separates the two pancakes.

In general, we introduce the *oriented gap* (*ogap*) heuristic for the burnt pancake problem (P2) defined as follows. Although others have previously mentioned this idea [10], we provide the first formal definition of the heuristic function, and experiments with it. We say that there is an *oriented* gap between two pancakes if at least one of the following conditions holds:

1. There is a gap (as in the pancake puzzle);
2. They have opposite signs;
3. They are wrongly ordered, with correct signs;
4. They are correctly ordered, with wrong signs.

For example, in the following sequence s_2:

$$1 \quad 2 \quad 3 \quad -4 \quad -5 \quad 7 \quad 6 \quad 8 \quad 9$$

there is an oriented gap of the first type between 6 and 8 (standard gap as in P1), of the second type between 3 and -4 (opposite signs), of the third type between 7 and 6 (correct signs, wrong order), of the fourth type between -4 and -5 (wrong signs, correct order). There is also an additional gap between -5 and 7 which falls in both the first and the second category. This gap is of course only counted once to maintain admissibility.

Formally the *ogap* heuristic is defined as follows (each line corresponds to a different condition just mentioned):

$$\begin{aligned} h_2(s) = h^{ogap}_{burnt}(s) := |\{i \;\; s.t. \;\; i \in \{1, \dots, M\}, \\ ||\pi_i| - |\pi_{i+1}|| > 1 \vee \\ \pi_i \cdot \pi_{i+1} < 0 \vee \quad (3) \\ (\pi_{i+1} - \pi_i = 1 \wedge \pi_{i+1} < 0 \wedge \pi < 0) \vee \\ (\pi_i - \pi_{i+1} = 1 \wedge \pi_{i+1} > 0 \wedge \pi > 0)\}| \end{aligned}$$

for the example above we have $h_2(s_2) = 5$. Admissibility is straightforward to prove, since all the oriented gaps have to be removed, and each move can remove at most one of them (for a k-flip, the oriented gap between $k-1$ and k). Since the cost of each move is equal to 1, and since the decrease of the heuristic function Δh produced by any move is at most 1, consistency is also ensured. As the experiments will show in Section 6, our new heuristic function can solve much more complex burnt pancake puzzles with respect to the ones solved until now with weaker heuristics.

2.3. Block interchanges problem (P3)

Block interchanges swap two non-overlapping blocks of arbitrary lengths. Given i and j as the starting positions of the two blocks and k and l as the two blocks lengths, such that:

$$1 \leq i \leq M - 1$$

$$1 \leq k \leq M - i - 1$$
$$i + k + 1 \leq j \leq M - 1$$
$$1 \leq l \leq M - j - 1$$

an allowed move would transform

$$\pi = \{\pi_1, \dots, \pi_{i-1}, \pi_i, \dots, \pi_{i+k}, \pi_{i+k+1}, \dots, \\ \pi_{j-1}, \pi_j, \dots, \pi_{j+l}, \pi_{j+l+1}, \dots, \pi_{M+1}\}$$

into

$$\hat{\pi} = \{\pi_1, \dots, \pi_{i-1}, \pi_j, \dots, \pi_{j+l}, \pi_{i+k+1}, \dots, \\ \pi_{j-1}, \pi_i, \dots, \pi_{i+k}, \pi_{j+l+1} \dots, \pi_{M+1}\}$$

This problem is exemplified in Figure 1 (c), where $i = 2$ and $j = 5$ are the two starting positions of the moved blocks, which have lengths $k = 2$ and $l = 3$, respectively. By contrast to P1 and P2, in P3 the moved blocks are not flipped. In sorting by blocks interchanges (P3), there are four degrees of freedom for selecting moves: the starting position i and j of the two blocks, and their lengths k and l: the branching factor is therefore $O(M^4)$.

For P3, an admissible heuristic can again be derived from the number of gaps in a certain sequence, but in this case some additional considerations have to be taken into account. First, k-flips are not among the allowed moves, differently from the pancake puzzles, and therefore one should count the number of *breakpoints*, that is, gaps where the order of the consecutive elements is also not correct (i.e., there is a breakpoint, but not a gap, between 4 and 3).

$$h^{bp}(s) := |\{i \;\; s.t. \;\; i \in \{1, \dots, M\}, \\ \pi_{i+1} - \pi_i \neq 1\}|. \quad (4)$$

This heuristic function would clearly be non-admissible, since, in the case of blocks interchanges, a move can eliminate more than a single breakpoint within the sequence. More precisely, the maximum number of breakpoints eliminated with a single move is four (see Figure 2a) due to the non-adjacency of the swapped blocks. Therefore, an admissible and consistent heuristic function for P3 can be computed as follows:

$$h_3(s) = h^{bp}_{blocks}(s) := \left\lceil \frac{h^{bp}}{4} \right\rceil \quad (5)$$

For example, the sequence in Figure 2a s_3:

$$2 \quad 6 \quad 7 \quad 5 \quad 3 \quad 1 \quad 4 \quad 8$$

has six breakpoints (2–6; 7–5; 5–3; 3–1; 1–4; 4–8), thus $h_3(s_3) = 2$.

2.4. Transpositions problem (P4)

Sorting by transpositions is similar to P3, but parameter j needs not to be chosen, since the two moved blocks need to be adjacent, and therefore

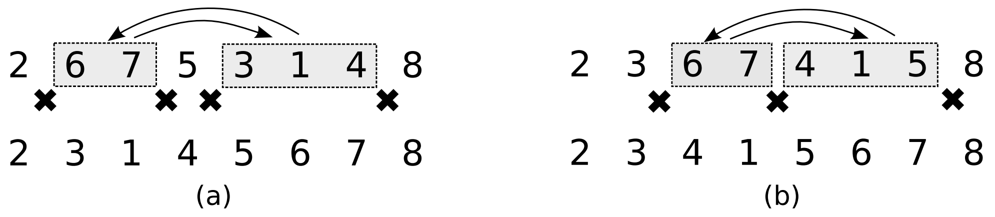

Fig. 2. In different permutation sorting problems, a single move can remove more than one breakpoint. In particular, in sorting by blocks interchanges (a) at most four breakpoints (indicated by black crosses) can be eliminated by a move, while the number is only three in sorting by transpositions (b).

the second block starts at position $i+k+1$. An example is given in Figure 1 (d). Since the two subsequences are required to be adjacent, the branching factor is $O(M^3)$, with one degree of freedom less than the previous case. It is worth highlighting that, for this problem, the set of allowed moves at each node is a subset of the corresponding set in the sorting by blocks interchanges problem (P3). For this reason, given a starting configuration of integers, the optimal number of moves required for P3 will be *smaller than or equal to* the optimal solution for the transposition case (P4). The depth of the search tree for P3 will therefore be *smaller than or equal to* that of P4, albeit with a much larger branching factor.

An admissible and consistent heuristic function for P4 can be computed similarly to P3, thus considering breakpoints. In particular, in P4 a move can eliminate up to three breakpoints, and not four as in P3 (see Figure 2b), namely the one before the first block, the one at the end of the second block, and the one between the two swapped blocks. This heuristic function can therefore be computed as:

$$h_4(s) = h^{bp}_{transp}(s) := \left\lceil \frac{h^{bp}}{3} \right\rceil \ , \qquad (6)$$

For example, the sequence in Figure 2b s_4:

2 6 7 5 3 1 4 8

has six breakpoints (2–6; 7–5; 5–3; 3–1; 1–4; 4–8), thus $h_4(s_4) = 2$.

2.5. Reversals problem (P5)

Sorting by reversals is simply a generalization of the (burnt) pancake problem (P2). Here too, allowed moves correspond to flips, but besides being of any length (as in P1 and P2), they can also start at any position in the sequence. For example, in Figure 1 (e), a 3-flip is performed, starting at position $i = 3$ in the sequence. A special case of this problem is known as the Top-Spin puzzle, where the permutation to be sorted is circular, and the allowed flips can be of a fixed, given length K only (typically $K = 4$).

In sorting by reversals, the degrees of freedom of the allowed moves are the starting position and the length of the flip: the branching factor is therefore $O(M^2)$.

The case of reversals is similar to the pancake puzzle, as it allows k-flips, but not necessarily in the configuration prefix: a move can choose both the starting element of the block to flip, say position i, and the length of the flipped block, k. As in the pancake puzzle, it is possible to count the number of *gaps*, but this time a single move can remove two gaps, the one between $i-1$ and i, and the other between $i+k-1$ and $i+k$. Therefore, an admissible and consistent heuristic for P5 is:

$$h_5(s) = h^{gap}_{rev}(s) := \left\lceil \frac{h^{gap}}{2} \right\rceil \qquad (7)$$

For example, in the following sequence s_5:

3 2 4 6 1 5 7 8 9

we have five gaps (2–4; 4–6; 6–1; 1–5; 5–7) thus $h_5(s_5) = 3$.

2.6. Translocations problem (P6)

Translocations problems merge the rules of transpositions and reversals: allowed moves both swap two blocks, and also flip their content. In Figure 1 (f), the same blocks of Figure 1 (c) are moved, but this time the configuration that is obtained has flipped elements in the moved blocks.

The space of allowed moves for the problem of translocations is identical to P4, with a branching factor of $O(M^4)$, the only difference being in reversing the moved blocks. For P5 and P6 the same consideration made for P3 and P4 holds: the set of allowed moves in reversals is a subset of the allowed moves for translocations. Therefore, if g_5 is the optimal solution for a given problem of P5, then the same configuration for P6 will have an optimal $g_6 \leq g_5$, since a solution will certainly be found at most at g_5 (by using only the set of allowed moves of P5).

As for the heuristic function, since the moved blocks are reversed, gaps have to be counted instead of breakpoints (as in P5), whereas four different gaps can be removed by a single move, analogously to P3. Therefore, an admissible and consistent heuristic function for P6 is the following:

$$h_6(s) = h^{gap}_{transl}(s) := \left\lceil \frac{h^{gap}}{4} \right\rceil \tag{8}$$

For example, in the following sequence s_6:

$$3 \quad 2 \quad 4 \quad 1 \quad 5 \quad 6 \quad 7 \quad 8 \quad 9$$

we have three gaps (2–4; 4–1; 1–5 thus $h_6(s_6) = \lceil \frac{3}{4} \rceil = 1$.

Several theoretical works exist that have studied lower bounds for these permutation sorting problems: for example, see [9] for the burnt pancakes and [28] for reversals. Yet, such bounds are often computationally expensive, and moreover not straightforward to adapt to the search paradigms discussed in this paper. Conversely, all the presented heuristics can be easily formulated and computed between any two states (similar to what happens with the classic gap heuristic for the pancake puzzle) and thus they can be naturally applied also to the bidirectional search paradigm we use in this paper.

3. Enhanced partial expansion unidirectional search

Partial expansion (PE) heuristic search was introduced by Yoshizum et al. [40] as an efficient solution to multiple sequence alignment problems. An enhanced version was presented by Felner et al. [12,17] and introduced a number of smart improvements which significantly contributed to the applicability of the algorithm to several different domains.

3.1. Partial expansion A*

It is well known that A* only expands nodes n with $f(n) \leq C^*$ where C^* is the cost of the optimal solution. Define the set of *surplus* nodes [12] as the nodes which have an f-value greater than C^*. A* might generate surplus nodes but it will never expand them. The key idea of PEA* and EPEA* is to avoid the generation of surplus nodes.

*Partial expansion A**(PEA*) [40] works as follows. When expanding node n, PEA* first generates a list of all the children of n, $CH(n)$. Only nodes c from $CH(n)$ with $f(c) = f(n)$ are added to OPEN. The remaining children are discarded but n is added back to OPEN with the smallest f-value greater than $f(n)$ among the remaining children (denoted f'). n may be chosen for expansion again with its new f-value f'. In this case, we again generate the list of all children but only add to OPEN those children with $f = f'$ and the rest are discarded. Finally, when n is chosen for expansion with $f(n) = \hat{f}$ and there are no children with f-value larger than $\hat{f}$ then n is moved into CLOSED.

It is easy to see that surplus nodes will never be added to OPEN because a node is added to OPEN only when its parent stores a similar f-value. Therefore, the goal will be chosen for expansion and the search will halt before any surplus node is ever added to OPEN. Consequently, the main advantage of PEA* over A* is the memory requirements. While A* adds many surplus nodes to OPEN (but never expands them) PEA* never adds these nodes to OPEN (although it may generate them in the children lists). In domains with large branching factor when many nodes may have f-values larger than their parents, significant memory savings can be offered with PEA*.

3.2. Enhanced partial expansion A*

A disadvantage of PEA* is its time requirements. When expanding a node PEA* scans through the entire set of its children (including surplus nodes) every time it is chosen for expansion. *Enhanced Partial Expansion A** (EPEA*) [12,17] aims to remedy this. Instead of scanning through the entire list of children as in PEA*, EPEA* directly generates only the children that will be surely added to OPEN, i.e., only those with the same f-value of their parent. Thus, EPEA* will never generate any of the surplus nodes. EPEA* uses *a priori* domain knowledge in order to make the current choice among the children and only generate the relevant set of children. We explain this next.

When expanding a node, n, EPEA* generates only those children n_c with $f(n_c) = f(n)$. The other children of n (with $f(n_c) \neq f(n)$) are discarded. This is done with the help of a special domain-dependent function called *operator selection function* (OSF). The OSF returns the exact list of operators which will generate nodes with the desired f-value (i.e., $f(n)$). Let Δf be the difference between the f-value of a node and its child, that is $\Delta f = f(n_c) - f(n)$. In order to build an OSF one needs to analyze the domain and see whether the operators have special structure that can be exploited for this purpose. If the Δf of a given operator is always the same, the OSF will exploit this. For example, in a single agent path finding problem on a 4-connected grid if the goal is at the far north, then going north will always have $\Delta f = 0$ when used with Manhattan distance as a heuristic. When having an OSF, when n is expanded, the OSF is consulted and the relevant operators (which yield children with the desired f-value) are applied.

As in PEA*, n is then re-inserted into OPEN with the f-cost of the next best child (this knowledge is also returned by the OSF lookup, see [12]). Unlike PEA*, EPEA* never generates a child and discards it. Thus, while PEA* may generate surplus nodes and discard them, EPEA* will never generate any surplus nodes.

Clearly, EPEA* can have some drawbacks, depending on the efficiency of the OSF and on the specific characteristics of the considered domain. In particular, a detailed time analysis of the different operations is given by Goldenberg et al. [17], with the conclusion that different domains may or may not be suitable for the proposed approach. A general consideration suggests that domains with a large branching factor will typically greatly benefit from EPEA*. Therefore, it is appealing to use it for the permutation domains studied in this paper.

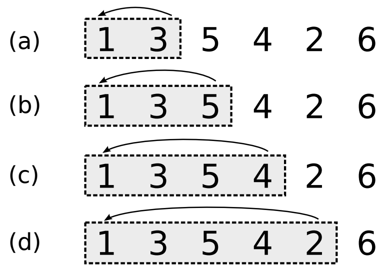

Fig. 3. An example of OSF on a 5-pancake puzzle configuration. For operators (a) and (d) the gap heuristic function does not change (no gap is inserted/removed), heading to a $\Delta f = 1$; operator (b) adds a gap, by breaking the 5-4 adjacency, therefore producing a $\Delta f = 2$; finally, operator (c) removes a gap by making the 1 and 2 pancakes adjacent, therefore reducing the heuristic function by 1, and heading to an overall $\Delta f = 0$.

3.3. OSF for permutation sorting problems

The main challenge within EPEA* is to build an efficient OSF. With permutation sorting problems, it is simple to design efficient OSFs which can compute the Δf of each operator with no need to generate the corresponding child. We note that $\Delta f = \Delta g + \Delta h$. But, in all our problem instances, the cost of applying an operator is constant[3] and thus $\Delta g = 1$. Thus all we need to know for a given operator is its Δh. In our domains the Δh induced by each operator only depends on the boundaries of the moved blocks: e.g., positions $0, k-1$ and k for a k-flip in the (burnt) pancake puzzle. Figure 3 shows an example of OSF for the pancake puzzle: in order to compute Δh^{gap} one only has to consider whether a gap is removed between $k-1$ and k, and/or a gap is inserted by making the pancakes in 0 and k adjacent. Therefore, the OSF for the pancake puzzle iterates over all possible operators and returns only those with the relevant Δf.[4]

For P2 (burnt pancakes) the same consideration holds. For P5 (reversals), the OSF has to take into

[3]In some problem variants the cost of a move could be proportional to the length of the moved blocks, which anyhow can be still easily computed without the need of generating the corresponding child.

[4]Two different types of OSF techniques were discussed by [17]: in the first, called Full-Checking (FC), the OSF needs to iterate through all the possible operators so as to retrieve the list of nodes to be generated, while the second, called Direct-Computation (DC) the OSF allows to directly compute only the operators of the children which will actually be generated. In our experiments below we implemented the FC technique.

account the starting position of the block to be flipped, as well as its length. Thus, the OSF computation has to iterate over the blocks. For P4 (transpositions), the OSF has to consider all the possible combinations of the starting position of the first block, its length, and the length of the second block. For P3 (blocks interchanges) and P6 (translocations), four parameters are needed, adding to the three parameters considered in P3 also the starting position of the second block, which is not necessarily adjacent to the first one. An important general attribute for all these problems is that we do not need to fully apply the operator (and change the location of k elements) in order to know its Δh. It is enough to observe a constant number of elements as just described for the various problem variants.

3.4. Analysis of EPEA*

Goldenberg et al. [17] provides a detailed analysis of the time performance of EPEA*. Basically, the advantages of enhanced partial expansion highly depend on the specific domain. More precisely, two parameters, named α and β, are considered in order to analyze the behavior of the algorithm: α is the average number of times a certain node is expanded (clearly, each time with a different "stored" f-value), while β is the average number of generations per node, that corresponds to effective branching factor. The running time of EPEA* [17] can be computed as:

$$t_{EPEA^*} = \alpha X t'_e + b\alpha X t_{of} + \beta\alpha X t'_m \tag{9}$$

being X the number of unique nodes chosen for expansion, t'_e the time needed for expanding a node, t_{of} the cost for computing a heuristic without generating a node, and t'_m the time needed for generating a node. For comparison, the running time of A* is:

$$t_{A^*} = X t_e + b X t_m \tag{10}$$

where t_e is time spent to retrieve the node with least cost from OPEN, and t_m the time needed for node generation, heuristic computation, and insertion into OPEN. When $\beta \approx b$, being b the average branching factor of A*, then the advantage of EPEA* is clearly not significant, and the same happens with large values of α. If, on the other hand, $\beta << b$ and α is small, the obtained speed-up can be huge. This is the case for our domains and thus EPEA* is a great algorithm for such domains.

3.5. EPEIDA*

EPEIDA* couples the enhanced partial expansion technique with the IDA* algorithm. The algorithm is very simple and elegant. For a given threshold T of an IDA* iteration, through the use of the OSF only children with $f \le T$ are generated. Nodes with $f > T$ will never be generated.

The performance analysis for the running time of EPEIDA* is simpler than for EPEA*. By indicating with t_r the overall time for generating a node, and by t_{of} (as for EPEA*) the cost for computing the heuristic without generating the node, the running time of EPEIDA* is computed as:

$$t_{EPEIDA^*} = bXt_{of} + Xt_r \tag{11}$$

As a comparison, the running time of IDA* is:

$$t_{IDA^*} = bXt_r \tag{12}$$

The speed-up obtained by EPEIDA* with respect to IDA* can thus be rather large.

4. Efficient single-frontier bidirectional heuristic search

Bidirectional search is a long-standing idea, although it has not been used very often with heuristics and when optimality is needed. This is mostly due to the *meet in the middle* problem [26], which is the problem of guaranteeing that the two search frontiers meet and that that the solution returned is optimal. Yet, some progress in this direction has been made where new analysis were presented [3] and a new algorithm that is guaranteed to meet exactly at the solution midpoint was introduced [24] for front-to-end bidirectional heuristic search. In this work we focus only on front-to-front approaches and use the *Single Frontier Bidirectional Search* algorithm [13] desgined for this purpose.

4.1. Single Frontier Bidirectional Search

A novel bidirectional search framework called Single Frontier Bidirectional Search was recently introduced by Felner et al. [13]. It was labeled by the authors as SFBDS but to improve the readability we shorten the notation and denote it by

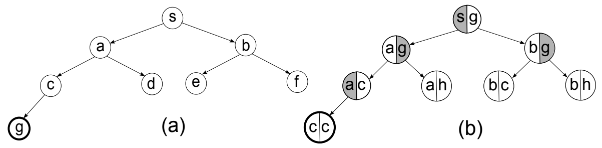

Fig. 4. Unidirectional vs. SBS search trees (expanded states are represented in gray for SBS).

SBS. The driving principle behind SBS is to search through a *double node* search tree, where each *double node* (from now on, simply *node*) N includes two states, one state x from the forward search and one state y from the backward search. In node N, the task is to find the shortest path between x and y. Such a task is recursively decomposed by expanding either x or y, and generating accordingly new nodes between either (1) x and the neighbors of y, or (2) y and the neighbors of x. At every node a *jumping policy* decides which of the two states to expand next, i.e., the search can proceed forward or backward.

Classic unidirectional search and SBS are illustrated in Figure 4. The objective is to find a shortest path from the start state, s, to the goal state, g. Whereas in the unidirectional search tree (Figure 4a) every node implicitly solves the task of getting from the current node to g (the search proceeding across the tree until g is found), in SBS (Figure 4b) nodes are labeled with the shortest-path task that should be solved beneath them. Within each node, the state expanded is shaded in the figure where left (right) means expanding the forward (backward) state.

Each jumping policy induces a tree, which can be searched by employing any admissible search algorithm such as A* (SBS-A*) or IDA* (SBS-IDA*). The search terminates when a *goal node* is expanded, where a goal node is in the form $N(x, x)$.

At any node $N(x, y)$ the remaining search effort can be estimated by a *front-to-front* heuristic function, $h(x, y)$, estimating the distance between x and y. The optimality of the solution is naturally guaranteed if an A*-based or IDA*-based search are activated on such a tree, employing an admissible front-to-front heuristic function.

4.2. Jumping policies

The original SBS paper [13] studied different jumping policies and their influence on the performance. Impressive savings were obtained in many domains, but in some cases a blowup in the search tree made SBS ineffective. For example, if the *never jump* policy is applied, the approach reverts to standard unidirectional search. With the *jump at root* policy, unidirectional backward search (from goal to start) is obtained. Many different intermediate policies may exist. The *alternate* jumping policy repeatedly alternates between the forward and backward sides; at even (odd) levels the forward (backward) side is expanded. The *branching factor* policy expands the side with the smaller branching factor. Similarly, in the case of asymmetric heuristic the *jump if larger* policy[5] for node $N(x, y)$ chooses to expand x (y) if $h(x, y) > h(y, x)$ (and vice versa).

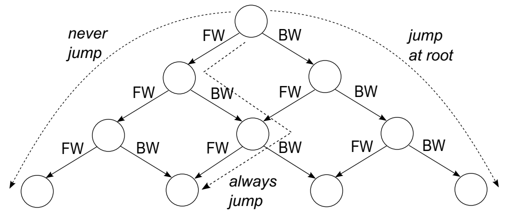

Fig. 5. Some examples of jumping policy: *never jump* (always expand FW direction), *jump at root* (always expand BW direction), *always jump* (alternately expand FW and BW directions).

Figure 5 shows all the policies mentioned above in a *policy space tree*. Nodes of this tree correspond to the different order of expansion actions. At each step, moving left corresponds to a forward expansion and moving right corresponds to a backward expansion. Each path from the root of this tree to a leaf at level d corresponds to a given jumping policy. Regular unidirectional search (never jump) corresponds to always going left. Backwards search (jump only at root) is to always go right. The alternate policy is shown in the middle, where the left and the right children are taken in turn.

4.3. eSBS-A*

SBS was taken several steps forward by Lippi et al. [33], heading to an efficient version named eSBS (efficient SBS). As SBS, eSBS is also a general-purpose search paradigm, which in principle can

[5]The *jump if larger* policy cannot be used in the context of this paper, as all the proposed heuristic functions are symmetric, and thus it would never produce any “jumping” effect.

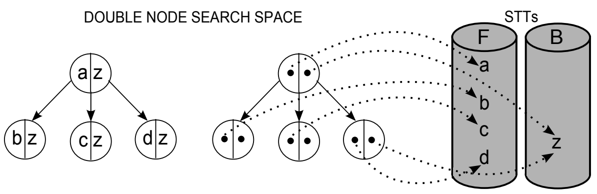

Fig. 6. Representation of nodes in classic SBS (left) and in eSBS (right), using pointers to states stored in the STTs.

be attacked by any search algorithm such as A* or IDA*. With respect to classic unidirectional heuristic search, and also to original SBS, the eSBS paradigm has shown clear advantages in terms of both time and memory requirements, as efficient pruning and caching techniques can be exploited in order to speed up the search and reduce the amount of required memory [33].

We start by describing eSBS-A*, based on four interdependent enhancements to plain SBS-A* [13]. Such enhancements are general enough to be employed also by other implementations of the eSBS framework, which will be covered below.

4.3.1. Referencing states with pointers

The key ingredient of eSBS-A* is based on the simple observation that the number of states seen during the execution of SBS is much smaller than the number of generated nodes, since the same state will be contained in many different nodes. The number of nodes grows quadratically with the number of states as each pair of states from the two frontiers, respectively, can potentially have their own node. Therefore, the first enhancement is not to maintain states inside nodes, but just their references through the use of pointers. This idea is shown in Figure 6 (right).

Basically, two State Transposition Tables (STTs) are employed to store the states visited throughout the search, one for each direction (forward and backward), while nodes of the SBS-A* search tree are stored in OPEN and maintain only references/pointers to the states (one in the FSST and one in BSST). Since states are typically more expensive to store than nodes, the use of references both reduces the amount of memory employed for the algorithm storage and also provides a useful framework for the pruning and caching techniques described below.

4.3.2. Parent pointers and best-g update

An additional reduction in the number of stored pointers can be obtained by moving parent pointers from nodes to states within the STT. In this way, each state will contain a pointer to its predecessor. This might cause a quadratic reduction in the number of pointers, since the same state occurs many times inside nodes. These parent pointers will be used to reconstruct the solution path once the solution has been found. Using such parent pointers introduces a possible drawback, as two nodes sharing the same state will now potentially share the same parent pointer. To guarantee admissibility we must ensure that the pointed parent of each state x lies on the optimal path to x. This can be obtained by providing each state with a variable, called *best-g*, that keeps track of the lowest observed cost, from the corresponding first state of that direction (start or goal, depending on the direction) to the state itself. The parent pointer will be updated whenever *best-g* is updated, as follows. When generating a new node, it is necessary to check whether each of the two states can be found in the appropriate STT: if not, then it is saved in the respective STT and its *best-g* is initialized to the total cost of actions that originated it. Otherwise, it must be checked whether its new g-value (from the side that it was now generated) is smaller than the previously recorded *best-g*. In such case, the state must be updated by resetting both the *best-g* and the pointer to the parent state.

4.3.3. Duplicate detection and best-g pruning

Typically, when search algorithms expand a node N they do not generate the parent of N (this is usually done by keeping the operator that generated N and not applying its inverse to N). As just explained, in eSBS two operators (in the form of pointers) are kept for $N(x, y)$, one for x and one for y. When a node is expanded from its forward (backward) side, the inverse of the operator that was used to first reach x (y) is not applied.

Furthermore, best-first search algorithms like A* store OPEN and CLOSED lists and they usually perform duplicate-detection as follows. In the case of consistent heuristic, CLOSED is employed in order to prune nodes already expanded, while if a newly generated node already exists in OPEN, we keep the copy with the smallest g value and prune the other node. In SBS, a *trivial duplicate detection* technique would check whether the same node already exists in OPEN or CLOSED. However, since there are $O(|V|^2)$ possible nodes that can be created out of all possible pairs of $|V|$ states, this is not enough.

An improved duplicate detection technique called *best-g pruning* [33] is as follows. During node generation, whenever the g-value of an already seen state a (both in the forward or backward direction) is greater than its current *best-g* value, we know that this node lies on a suboptimal path and all its descendants would fail the *best-g* check anyway: thus, this node can be pruned from OPEN. In other words, duplicate pruning is done for each frontier of the search on its own. But, these prunings have mutual effects as duplicate states will never be added to a new generated node. Within eSBS-A* search, including in the OPEN set such a suboptimal successor node, which obviously will not be part of any optimal solution, could lead to a great waste of computational resources. This technique is shown in Figure 7. Suppose the two nodes on the top, (a, b) and (p, q) are both in OPEN. Since they have the same f- and h-values the ordering in which they are visited can change when adopting different jumping policies. If the node on the left (x, b) is generated and added to OPEN first, and the jumping policy requires that the next expansion is in the forward direction, then node (x, q) will later be pruned according to the *best-g* criterion, as node (x, q) will have a g value for state x equal to 4, while it had been previously seen at $g = 3$. On the other hand, if the node on the right (x, q) is added to OPEN first, then also node (x, b) will later be generated.

4.3.4. Successor caching

A powerful caching technique again derives from the consideration that the same state can appear in several nodes. Therefore, if each state in the STTs is attached a list of pointers to its successor states, it will not be necessary to recompute the successors every time the state is reached. When a state s is first generated, its successor pointers are set to NIL. The first time that a node including s is expanded in the direction containing s, the successor function is called on s and all the successor pointers are updated. The next time the state s is encountered, the successors are already available and the cached pointers can be simply returned.

eSBS-A* employs all these four enhancements over basic SBS. Next we describe two more versions of eSBS also presented in [33].

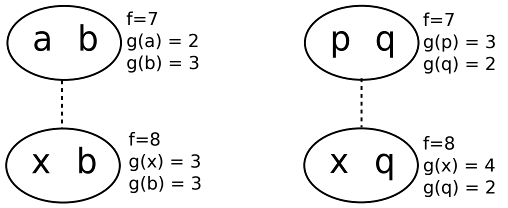

Fig. 7. An example of *best-g* pruning technique. If node (a, b) is expanded before node (p, q), which could depend on the adopted jumping policy, then node (x, q) will be pruned according to the *best-g* criterion. Otherwise, both nodes will be added to OPEN.

4.4. eSBS-A* lite

Experimental results confirm that *best-g* pruning is a powerful technique, which greatly contributes in pruning a significant number of nodes, and therefore in speeding up the whole search process. This consideration heads to the development of a variant of eSBS-A*, named eSBS-A$*$ *lite* [33], which neither stores CLOSED, nor performs duplicate detection within OPEN, but only relies on *best-g* evaluations in order to prune sub-optimal nodes. The resulting algorithm will generate a greater number of nodes with respect to eSBS-A*, owing to the missing duplicate detections in OPEN and CLOSED, but is faster since it does not have to check for such duplicates. In terms of effective memory usage, this variant does not use much more memory than eSBS-A*, since the greater number of generated nodes is somehow balanced by the savings obtained by not storing CLOSED.

4.5. eSBS-H

The combination of SBS with IDA* leads to the algorithm named SBS-IDA* [13], that basically decides, for each expanded (double) node in the iterative deepening framework, which of the two search directions should be chosen according to the given jumping policy. SBS-IDA* does not store OPEN, CLOSED or any of the STTs. Thus, similarly to regular IDA* (and to any other DFS algorithm), its memory requirements are just equal to the depth of the current node in the search tree.

When the eSBS search paradigm is coupled with the Iterative Deepening approach, a new algorithm which is a hybrid between A* and IDA* is obtained. Such algorithm was named as eSBS-H [33], where H stands for hybrid. eSBS-H exploits all the advantages of the eSBS search paradigm, and it is a smart compromise between computational speed and memory consumption. Similarly to IDA*, eSBS-H does not employ OPEN and CLOSED to decide which node to expand next and it uses an iterative deepening approach on the double node search tree. By contrast, like eSBS-A* eSBS-H stores and maintains the STTs for the two search frontiers. Therefore, all the advantages induced by the eSBS search paradigm can be exploited, including in particular *best-g* pruning and successor caching. Basically, eSBS-H acts like IDA* with transposition tables, but its transpositions are not in the form of nodes but in the form of states. It is worth remarking that, due to the use of STTs, the memory requirement of eSBS-H is not linear with the depth of the tree (as plain IDA*). Since the states are stored in the STTs and successors are cached, at each new iteration of the algorithm at depth d the whole subtree of depth $d-1$ is immediately available. Hence, instead of pushing and popping nodes in a stack which is newly regenerated on-the-fly at each iteration, such nodes do not need to be generated, but just accessed via the successor pointers.

4.6. Complexity issues

The memory complexity of the eSBS-based algorithms mainly depends on the STTs. Assuming that the alternate jumping policy is employed, that expands the forward state at one level and the backward at the following one, at each node only the successors of either the forward or the backward state are generated in the relevant STT:

`Level 0` $\rightarrow$ 2 new states
`Level 1` $\rightarrow$ b new states (fw expansion)
`Level 2` $\rightarrow$ b new states (bw expansion)
`Level 3` $\rightarrow$ b $\times$ b new states (fw)
`Level 4` $\rightarrow$ b $\times$ b new states (bw)
...

`Level` d-`1` $\rightarrow \underbrace{b \times b \times \ldots \times b}_{d/2\ times}$ new states (fw)
`Level` $d \rightarrow \underbrace{b \times b \times \ldots \times b}_{d/2\ times}$ new states (bw)

Thus, the overall number of stored states in the STTs will be:

$$O(\sum_{k=1}^{d/2}(b^k + b^k)) = O(2\sum_{k=1}^{d/2} b^k) = O(b^{d/2})$$

Table 1
Memory complexity comparison of SBS and eSBS variants.

Algorithm	Nodes	States	Total
eSBS-A*	$2 \cdot c_p\ b^d$	$c_s \cdot 2 \cdot b^{d/2}$	b^d
eSBS-A* lite	$2 \cdot c_p\ b^d$	$c_s \cdot 2 \cdot b^{d/2}$	b^d
eSBS-H	$2 \cdot c_p\ d$	$c_s \cdot 2 \cdot b^{d/2}$	$b^{d/2}$
SBS-IDA*	$2 \cdot c_p\ d$	-	d

since the search process will store $2 \cdot b^k$ states every two levels.[6]

Assume that c_p is the constant memory allocated for a pointer and c_s is the constant memory needed to store a state. While c_s is mainly problem-dependent, c_p depends primarily on the hardware/software implementation. If $c_s \leq c_p$ eSBS-A* does not reduce memory consumption, since the overhead of maintaining pointers to the states would be greater than the advantage of reducing the number of stored states. However, in most problems it is likely that c_s is much greater than c_p, thus heading to considerable memory savings. It was shown that eSBS-A* has memory complexity of $O(b^d)$, while eSBS-H reduced it to $O(b^{\frac{d}{2}})$ [33]. The memory complexity of these algorithms is summarized in Table 1: basically, eSBS-A* stores both OPEN and CLOSED, and the two STTs, whereas the *lite* version does not store CLOSED; eSBS-H stores neither OPEN nor CLOSED; SBS-IDA* does not even store the STTs, as it happens in classic IDA*.

Results reported in [33] show the advantages of eSBS variants with respect to the original SBS framework in a variety of domains.

5. Efficient Single-frontier Bidirectional Search with Enhanced Partial Expansion (eSBS-EPE)

Both EPE and eSBS exploit smart search strategies and technical solutions in order to build more efficient heuristic search algorithms, capable of both reducing the memory requirements and of greatly speeding up the search process. These improvements are in a sense orthogonal, and can easily be integrated in a unique framework, producing a new family of heuristic search algorithms which combines the benefits of both paradigms. This is the aim of this section.

[6]More precisely, such number would be $b^k_{fw} + b^k_{bw}$, if the branching factors along the two directions were different.

5.1. *Partial expansion vs. successor caching*

As detailed below, SBS and the enhanced version eSBS can all be combined with the EPE paradigm. The only exception is the successor caching technique.

An intrinsic feature of partial expansion is to prevent multiple visits of surplus nodes. By contrast, successor caching exploits the generation of all the children of a certain node (even those which are pruned later on, due to *best-g* pruning). The driving principle in that case is to immediately access the already computed children when the same state is re-expanded: this is particularly useful when generating a successor is an expensive operation which should not be performed multiple times for a single node.

To combine the two opposite techniques, in principle, the successor caching technique could be done *lazily*, which means that successors could be incrementally cached only when the EPE framework generates them. Yet, in this way, when generating a new child, it would be necessary to check whether the corresponding successor had already been cached. We observed experimentally that this process basically balances the advantages introduced by caching, and thus the overall mechanism does not produce any significant speed-up in the search. Therefore, for these reasons, when combining the EPE framework with eSBS, the state successor caching technique will not be used.

The new search paradigm that results from the combination of eSBS and EPE is named efficient Single-frontier Bidirectional Search with Enhanced Partial Expansion (eSBS-EPE). As for eSBS and EPE, clearly it also can be coupled with different heuristic search algorithms. We now cover a few variants of such combinations.

5.2. *eSBS-EPE-A**

Starting from the eSBS-A* algorithm, for example, one could implement enhanced partial expansion in a straightforward way, with the same principle adopted by unidirectional EPEA*. We name the resulting algorithm as eSBS-EPE-A*.

Once a node N is removed from OPEN for expansion, the adopted jumping policy indicates which of the two directions is to be expanded. In either case, the Δf for node N is equal to the Δf induced by the operator on the forward/backward direction chosen by the jumping policy. As already stated in Section 3.3, in case the cost of all operators is always one (as it happens with the permutation sorting problems we consider), for each possible child we have $\Delta f = 1 + \Delta h$, and thus only changes in Δh have to be considered. Following the EPE paradigm, only those states with $\Delta f = C$ will be generated from node N, being C the smallest Δf among those of the children not yet generated (all the children, in the case N is expanded for the first time). As a consequence, only the double nodes containing such states will be generated and stored in OPEN. If some states with a larger Δf are not generated following this principle, then N will be re-inserted into OPEN, with an f-value equal to the smallest among those of the children which were not generated, named f_{next}. Otherwise, N can be moved to CLOSED.

In permutation sorting problems, this mechanism can be easily implemented since the Δh induced by a certain operator can be computed in a straightforward way (without node generation) also in the case of bidirectional search, by exploiting the same OSFs that have been described in Section 3, extended to the considered front-to-front heuristic function.

5.2.1. *Jumping policies*

A special consideration should also be made regarding the link between the jumping policy and the EPE mechanism. In principle, it could happen that a certain node could be re-inserted into OPEN and, the first time it is re-expanded, a different direction (with respect to the one selected at the first expansion) could be chosen. Yet, this cannot happen in unit-cost domains with consistent heuristics and with the *alternate* jumping policy, which is the setting adopted in this paper. In fact, with such policy, a certain node N can be expanded more than once but only at the same depth d within the search tree, otherwise it would have been pruned owing to the *best-g* pruning mechanism. Since with the alternate jumping policy the direction to expand is directly associated to the depth of the node (either odd/even for forward/backward), and we always employ unit costs, this issue is not considered in this paper.

Besides the fact that the *alternate* policy is extremely simple, and that it has shown good results in previous applications of eSBS [33], also note that, in the case of permutation sorting problems, the more sophisticated *jump if larger (JIL)* policy cannot be applied, since all the considered heuristics are symmetric, whereas the JIL policy only works with asymmetric heuristics. In addition, with a given jumping policy, the chosen direction may immediately generate a larger number of states than those which would have been generated by the other direction, due to the EPE constraints on the Δf. This could induce a novel jumping policy, that is a variant of the *branching factor* policy, which takes into account the EPE pruning that happens at the state level. A deeper analysis of jumping policies within the eSBS-EPE framework, and not restricted to the domain of permutation sorting, is left for a future work.

5.2.2. Pseudo code

The pseudocode of eSBS-EPE-A* is given in Algorithm 1. At line 6, the node N with lowest cost is taken from OPEN, and at line 9 the jumping policy indicates the state z to be expanded. Then, at line 10, the OSF is applied to z. Then, each state z_c returned by the OSF is either added to the FSST/BSST (line 15) or considered for *best-g* pruning (lines 12–13). If z_c is not pruned, a new double node is generated which has z_c and the state from N of the other side (which was not chosen for expansion) (line 17), duplicate detection in OPEN and CLOSED is performed (line 18), the *best-g* and parent pointers of its states are updated (line 19), and it is finally put into OPEN (line 20). Lines 22 and 23 decide whether N has to be re-inserted into OPEN according to the EPE mechanism.

5.2.3. Computational issues

This approach hence combines the memory savings induced by eSBS (except the successor caching technique of eSBS, as already stated above) with the speed-up produced by the EPE paradigm. In particular, the time analysis described by Goldenberg et al. [17] for EPEA* is valid for eSBS-EPE-A* too, thus still depending on some domain-dependent parameters. Yet, in this case we must distinguish the re-expansion of *nodes* with the re-expansion of *states*. We can compute the running time of eSBS-EPE-A* as:

$$t_{eSBS-EPE-A^*} = \alpha_N X t_E + b\alpha_S S t_{of} + \beta\alpha_N X t_M \tag{13}$$

where X is the number of unique nodes that are chosen for expansion, S the number of stored states, t_E is the time needed for expanding a (double) node, t_{of} is the time needed for computing the heuristic of a *state* without generating it (i.e., exploiting the OSF), t_M is the time needed for generating a child double node, β is the average effective branching factor, α_N indicates how many times the same node will be expanded, whereas α_S indicates how many times the same state will be expanded. With respect to the running time of EPEA* (see Equation 9), the performance of eSBS-EPE-A* strongly depends on the balancing between α_N and α_S. In fact, if the time needed for heuristic computation via the OSF is lower than the time needed for expanding a node (i.e., if $t_{of} < t_E$), then the speed-up induced by eSBS-EPE-A* will be more evident as long as $\alpha_S > \alpha_N$. This happens when the same *state* is expanded multiple times. As a matter of fact, this condition is typically met in those contexts where even the enhancements of eSBS have more effect, that is where the same state is encountered many times during the search.

Finally, we remark that both the memory and time complexity are the same as those of eSBS-A*.

5.3. eSBS-EPE-A* lite

A *lite* version of eSBS-EPE-A* can be conceived, which is analogous to the eSBS case. Basically, neither CLOSED is employed, nor duplicate detection in OPEN is performed. To avoid the generation of surplus non-optimal nodes, the *best-g* pruning technique is coupled with the partial expansion mechanism, exactly as it happens is eSBS-EPE-A*. Even in this case, an expanded node will be re-inserted in OPEN if and only if there exists some child which was not generated due to the Δf threshold cut. With respect to the pseudo code presented in Algorithm 1, only lines 18 and 23 are removed.

As reported for eSBS [33], the *lite* version of eSBS-A* is not suitable for settings where a large number of duplicate nodes sharing the same g-value exists. In such cases, in fact, the impact of CLOSED on duplicate detection is crucial in order to avoid the re-expansion of many identical nodes. This will happen also in permutation sorting problems, as shown in Section 6.

Algorithm 1 eSBS-EPE-A*

```
 1: Input: start state $s$, goal state $g$
 2: generate the start double node $N_S = (s, g)$
 3: compute $h(N_S)$, set $f(N_S) \leftarrow h(N_S)$
 4: put $N_S$ into OPEN
 5: while OPEN is not empty do
 6:     get $N$ with lowest $f(N)$ from OPEN
 7:     if $N$ is goal then exit
 8:     end if
 9:     choose state $z$ to expand according to the jumping policy
10:     consult $OSF(z)$ and set $(\mathcal{Z}, f_{next}(N))$
11:     for all states $z_c \in \mathcal{Z}$ do
12:         if state $z_c$ already in FSTT or BSTT (according to the jumping policy) then
13:             if $best\text{-}g(z_c) < g(z_c)$ then prune $z_c$
14:             end if
15:         else add $z_c$ to either FSST or BSTT (according to policy)
16:         end if
17:         compute double node child $Z = (m, z_c)$ (or $Z = (z_c, m)$, according to the jumping policy)
18:         check for duplicate nodes in OPEN and CLOSED
19:         update $best\text{-}g(z_c)$ and parent pointers for $z_c$
20:         put $Z$ into OPEN
21:     end for
22:     if $f_{next}(N) < \infty$ then set $f(N) \leftarrow f_{next}(N)$ and re-insert it into OPEN
23:     else put $N$ into CLOSED
24:     end if
25: end while
```

Algorithm 2 DFS procedure for eSBS-EPE-H

```
 1: Input: node $N$, threshold $T$
 2: DFS(N,T)
 3: compute $h(N)$
 4: set $f(N) \leftarrow g(N) + h(N)$
 5: if $f(N) > T$ then return
 6: end if
 7: if $N$ is goal then halt
 8: end if
 9: choose state $z$ to expand according to the jumping policy
10: set $(\mathcal{Z}, f_{next}(N)) \rightarrow OSF(z)$
11: set $T_{next} = \min(T_{next}, f_{next})$
12: for all states $z_c \in \mathcal{Z}$ do
13:     if state $z_c$ already in FSTT or BSTT (according to the jumping policy) then
14:         if $best\text{-}g(z_c) < g(z_c)$ then prune $z_c$
15:         end if
16:     else add $z_c$ to either FSST or BSTT (according to policy)
17:     end if
18:     compute double node child $Z = (m, z_c)$ (or $Z = (z_c, m)$, according to the jumping policy)
19:     set $g(Z) \leftarrow best\text{-}g(m) + best\text{-}g(z_c) + cost(z, z_c)$
20:     update $best\text{-}g$ and parent pointers for $z_c$
21:     call $DFS(Z, T)$
22: end for
```

5.4. eSBS-EPE-H

As described in Section 4.5, when coupling eSBS with iterative deepening A*, we obtain an algorithm that is a hybrid between SBS-A* and SBS-IDA*, since it maintains two transposition tables for state storing, while performing iterative deepening search on nodes. This paradigm remains also when we combine eSBS-H with EPE, just adding to the search framework the mechanism for avoiding the generation of surplus nodes as induced by Δf computations.

As in EPEIDA* and eSBS-H, neither an OPEN nor a CLOSED set are maintained, and the only memory requirements are therefore due to the state transposition tables (similar to eSBS-H). The resulting algorithm is named eSBS-EPE-H, still maintaining in the name the fact that the memory requirements remain a sort of hybrid between A* and IDA*. As a matter of fact, the memory complexity still remains that of eSBS-H, i.e., $O(b^{d/2})$.

In detail, when a node N is expanded, and the child state x (either forward or backward) is chosen according to the adopted jumping policy, only those children y of x having a $\Delta f < T - f(N)$ are effectively generated, being T the current threshold for the iterative deepening process. Such a condition on Δf can be translated in the following condition on the variation of the heuristic function: $\Delta h < T - f(N) - cost(x, y)$. In this way, with respect to eSBS-H, the EPE approach avoids the generation of each surplus node M having $f(M) > T$. The next threshold T_{next} will be equal to the lowest f-value among those of children not generated, named f_{next}.

Algorithm 2 reports the pseudocode of the depth first search (DFS) recursive step of eSBS-EPE-H. First, a state z is chosen for expansion by the jumping policy (line 7). Then, similar to EPEIDA*, the OSF is consulted and retrieves the set of children to be generated (Z) and the lowest cost among the currently unneeded children (f_{next}) (line 8). Such cost is used to update the threshold T_{next} for the next iteration (line 9). As in EPEIDA*, this threshold is initialized to infinity at the beginning of the current iteration. Similar to eSBS-EPE-A*, lines 11–18 consider each state z_c returned by the OSF, whether it has to be added to the FSST/BSST (line 14) or pruned (lines 11–12), the generation of a new double node containing z_c, named Z (line 16), and the update of the *best-g* and parent pointers of its states (line 18).

In domains where the OSFs can be easily computed, eSBS-EPE-H represents a significant step forward with respect to eSBS-H, as it combines the memory savings of single frontier bidirectional search with the efficient node generation induced by the EPE framework.

With respect to the running time, eSBS-EPE-H benefits from the speed-up of partial expansion, at the expense of discarding successor caching. Keeping the same formalism of eSBS-EPE-A*, we compute the running time of eSBS-EPE-H as:

$$t_{eSBS-EPE-H} = bXt_{of} + Xt_{RN} + St_{RS} \quad (14)$$

where t_{RN} is the time needed for generating a node, and t_{RS} the cost for generating a state.

6. Experimental results and discussion

We ran experiments on all the variants of the permutation sorting problems that have been described throughout this paper. In all our experiments we used an Intel Core i7 @3.20 GHz, with 8Gb of RAM.

6.1. Results on small instances

We first performed a set of experiments in order to compare the following ten different algorithms. Two classic unidirectional searches: A* and IDA*. Three single-frontier bidirectional searches: eSBS-A*, eSBS-A*-lite and eSBS-H. Two unidirectional enhanced partial expansion searches: EPEA* and EPEIDA*. Finally, three single-frontier bidirectional partial expansion searches: eSBS-EPE-A*, eSBS-EPE-A* lite and eSBS-EPE-H. The aim of these experiments was to select the most efficient algorithms, to be run on larger problem sizes as many of these 10 algorithms are expected to run out of memory or require too long computational times on large problems. We report representative results obtained by the 10 algorithms on the $M = 18$ burnt-pancake problem (P2) with our new *ogap* heuristic function, and on the $M = 15$ transpositions problem (P4). These two problems were chosen due to their different branching factors of 28 and 560, respectively.[7] Results for the other domains produced similar tendencies and are omitted. Note that, for the burnt pancake puzzle (P2), $M = 18$ represents the largest problem size previously solved by IDA*, with a modified gap heuristic and pattern databases [29]: it is worth mentioning that, with the same heuristic, EPEA*, eSBS-

[7]P3, which has an even larger branching factor than P4, could not be chosen since too many algorithms failed to solve several instances due to memory limitations.

Table 2
Results obtained by ten different competitors on the 18-burnt pancake problem (P2). Time is reported in ms.

	Nodes	States	Time
A*	2,815,908	2,815,908	15,695
IDA*	16,711,388	–	32,988
eSBS-A*	2,384,016	139,278	4,127
eSBS-A* lite	3,009,903	172,640	2,402
eSBS-H	12,643,067	508,981	2,862
EPEA*	175,878	175,878	**399**
EPEIDA*	928,418	–	1,658
eSBS-EPE-A*	**172,601**	**15,930**	1,201
eSBS-EPE-A* lite	176,883	16,034	1,274
eSBS-EPE-H	749,693	48,834	748

Table 3
Results obtained by ten different competitors on the 15-transpositions problem (P4). Time is reported in ms.

	Nodes	States	Time
A*	9,599,941	9,599,941	40,854
IDA*	183,892,146	–	200,757
eSBS-A*	32,926,978	667,533	41,349
eSBS-A* lite	63,862,925	867,933	32,132
eSBS-H	191,164,078	1,649,970	36,724
EPEA*	**141,061**	141,061	**474**
EPEIDA*	1,053,970	–	5,052
eSBS-EPE-A*	244,118	**7,618**	1,330
eSBS-EPE-A* lite	435,259	13,504	3,269
eSBS-EPE-H	933,602	18,779	4,141

EPE-A* and eSBS-EPE-H were able to solve burnt pancake puzzles with $M = 18$ as well, but without the pattern databases. This is due to the enhanced algorithms and the novel *ogap* heuristic.

Tables 2 and 3 present results for which all 10 algorithms could solve 100 random instances. We report the number of nodes, the number of stored states in the STTs (that is equal to nodes for A* and EPEA*, whereas it is not reported for IDA* and EPEIDA* as they do not store transposition tables) and the CPU time in ms. All the measurements are averaged over the 100 instances.

For the burnt pancake problem, EPEA* is the fastest algorithm, with eSBS-EPE-H second best. Yet, the advantage of eSBS-EPE-A* in memory requirement is particularly evident with respect to the other approaches. The *lite* version of the bidirectional algorithm has much less impact when it is coupled with partial expansion: while eSBS-A* lite is much faster than eSBS-A*, this difference is not tangible between eSBS-EPE-A* lite and eSBS-EPE-A*. This result indicates that the overhead due to duplicate detection within OPEN and CLOSED is less significant in eSBS-EPE-A* than in eSBS-A*. The EPE approach showed to be very effective (timewise) for the hybrid algorithm (eSBS-EPE-H) in this domain, due to the combination of not storing OPEN and CLOSED with the property of not generating surplus nodes. Since eSBS-EPE-H does not store OPEN and CLOSED it has a much smaller constant time per node. While its number of nodes and number of states was three times larger than eSBS-EPE-A* it was almost twice as fast in its CPU time.

For the transpositions problem (P4), EPEA* is still the fastest algorithm, but (unlike P2) eSBS-EPE-A* and eSBS-EPE-A* lite are faster than eSBS-EPE-H. This is due to the fact that eSBS-EPE-H generates more nodes than eSBS-EPE-A*, and the cost of OSF is greater for transpositions than for pancakes, because multiple positions have to be checked in order to compute the Δh. For the other algorithms, the trends are very similar to the case of burnt pancakes.

6.2. Results on large problems

Based on these small experiments, we considered the three algorithms which had proven to be the most efficient on the simple problem sets, namely EPEA*, eSBS-EPE-A* and eSBS-EPE-H, and we evaluated them on problems having larger sizes. We also consider eSBS-A* in order to have a full comparison of all A* variants. This time we employed up to 64 Gb of RAM. We tested the four algorithms on 25 random instances of 85-pancake (P1-85), 28-burnt-pancake (P2-28), 15-blocks-interchanges (P3-15), 17-transpositions (P4-17), 17-reversals (P5-17) and 18-translocations (P6-18). These experiments demonstrate the power of heuristic search in general. First, to the best of our knowledge, for blocks interchanges, transpositions, reversals and translocations (P3–P6) these are the first problems optimally solved with heuristic search algorithms. Moreover, for the pancake (P1) and burnt pancake puzzles (P2), which are widely known in the heuristic search literature, these results represent the largest problem size ever solved for the first time. While for the pancake puzzle (P1) the previous best results are for 70 pancakes [17,23] we further pushed the limit and solved the 85 pancake case. Similarly, for the

Table 4
Comparison among EPEA*, eSBS-A*, eSBS-EPE-A* and eSBS-EPE-H on large problem sets. bf is the problem branching factor. g^* and h_0 represent the average solution depth and initial heuristic estimate, respectively. β is the effective branching factor of each algorithm. Time is reported in ms, memory in MB. When results are not available (NA) for an algorithm, the number in parentheses indicates the number of instances on which it runs out of memory. In this case we employed 64G of memory.

Problem	bf	g^*	h_0	Algorithm	β	Nodes	States	Time	Memory
P1-85	84	83.52	83.08	EPEA*	NA (2)	NA (2)	NA (2)	NA (2)	NA (2)
				eSBS-A*	NA (21)	NA (21)	NA (21)	NA (21)	NA (21)
				eSBS-EPE-A*	1.23	**50,381,647**	**2,143,436**	1,050,944	**5,626**
				eSBS-EPE-H	1.24	163,874,256	5,583,024	**274,025**	6,568
P2-28	28	32.72	27.60	EPEA*	NA (1)	NA (1)	NA (1)	NA (1)	NA (1)
				eSBS-A*	NA (7)	NA (7)	NA (7)	NA (7)	NA (7)
				eSBS-EPE-A*	1.66	83,955,659	**2,593,674**	820,757	**9,558**
				eSBS-EPE-H	NA (3)	NA (3)	NA (3)	NA (3)	NA (3)
P3-15	1,365	6.08	3.92	EPEA*	7.43	**7,643,690**	7,643,690	**28,952**	**160**
				eSBS-A*	NA (7)	NA (7)	NA (7)	NA (7)	NA (7)
				eSBS-EPE-A*	8.26	30,516,250	**131,486**	272,854	2,428
				eSBS-EPE-H	NA (2)	NA (2)	NA (2)	NA (2)	NA (2)
P4-17	560	8.08	5.68	EPEA*	4.27	**2,172,773**	2,172,773	41,611	**113**
				eSBS-A*	NA (6)	NA (6)	NA (6)	NA (6)	NA (6)
				eSBS-EPE-A*	4.51	5,007,803	**64,344**	**29,421**	444
				eSBS-EPE-H	5.73	18,607,160	131,205	113,906	976
P5-17	120	10.64	8.12	EPEA*	3.12	**1,750,861**	1,750,861	**23,113**	**208**
				eSBS-A*	NA (2)	NA (2)	NA (2)	NA (2)	NA (2)
				eSBS-EPE-A*	3.29	11,304,659	**120,178**	72,939	1,082
				eSBS-EPE-H	4.28	45,488,349	288,605	118,915	1,952
P6-18	3,060	5.76	4.48	EPEA*	5.98	**415,863**	415,863	**16,645**	**46**
				eSBS-A*	NA (3)	NA (3)	NA (3)	NA (3)	NA (3)
				eSBS-EPE-A*	6.05	529,034	**21,614**	29,222	60
				eSBS-EPE-H	8.53	3,663,731	58,584	281,426	244

burnt-pancake problem the previous largest size was 18 [29] while we solved the 28 version.

Results are shown in Table 4. We report the optimal solution length g^*, the initial heuristic function h_0, an estimate of the effective branching factor β for each algorithm, the number of nodes, the number of stored states in the STTs (that is equal to nodes for EPEA*) and the CPU time in ms. β was estimated as $\sqrt[d]{X}$, being X the number of nodes and d the solution depth. All the measurements are averaged over the solved instances.

Whereas on the previously reported small-size problem instances EPEA* is the fastest algorithm (although not the one consuming the least memory), on larger problem instances the impact of efficient single-frontier bidirectional search becomes evident. First, we can observe that eSBS-EPE-A* is the most robust. It was the only algorithm capable of solving all the instances of these six problem sets. The other three methods ran out of memory on some of the considered instances. eSBS-EPE-A* is also the fastest algorithm on P4-17, and on P2-28, where the other two algorithms cannot solve all the instances. eSBS-EPE-H is the fastest algorithm on the pancake puzzle (P1-85). By contrast, EPEA* is the fastest method on the remaining three problems (P3-15, P5-17, P6-18). In the next paragraphs we cover each of these domains in turn and explain the results.

6.2.1. 85 pancake (P1)

With 85 pancakes EPEA* and eSBS-A* could not solve 2 and 21 of the 25 instances, respectively, given the 64Gb available memory. Only the eSBS-EPE variants could solve all instances. For this problem set, the results show that eSBS-EPE-H is the fastest algorithm, although eSBS-EPE-A* generates fewer nodes. In fact, this is the problem where eSBS-EPE-H has the best constant time per node, with respect to the other domains. This behavior is explained by the fact that the pancake puzzle is the problem where the cost of the OSF computation is the smallest, since it involves the comparison of only three positions $(0, k-1, k)$ in the sequence. According to the analysis in [17], which has been summarized in Section 3, this is the condition that guarantees a large speed-up for EPEIDA* with respect to IDA*, and therefore also for eSBS-EPE-H. The larger running time of eSBS-EPE-A* can be ascribed, in this case, to the computation of the hash functions for each state and node (needed to store nodes in OPEN/CLOSED), that has a significant impact, due to the large size of the puzzles.

Figure 9 (left) shows the memory usage comparison between EPEA*, eSBS-EPE-A* and eSBS-A* on P1-85. The 25 instances are sorted by increasing number of EPEA* nodes, so that the advantage of eSBS-EPE-A* is evident as the number of nodes (i.e., the complexity of the problem) grows. Figure 9 (right) shows time comparison between these three algorithms, highlighting the fact that EPEA* is the fastest, when provided enough memory. For the algorithms which could solve less than 25 instances, only the solved instances are displayed in the plot. Also note that in this domain the heuristic function is extremely informative, with an average difference between solution depth and initial heuristic guess equal to 0.44 on the considered instances. The average effective branching factor β is very small too, as children introducing an additional gap are obviously not generated. For EPEA*, eSBS-EPE-A* and eSBS-EPE-H β is almost identical, being about 1.23.

6.2.2. 28 Burnt pancake (P2)

eSBS-EPE-A* is the only algorithm which could solve all 25 instances of the P2-28, as all the other three competitors run out of memory. Note that this is the problem where the heuristic function (h^{ogap}) is the least informed (see columns g^* and h_0 in Table 4), where we measure informativeness by the difference between g^* and h_0. Nevertheless, results show that this is newly introduced heuristic had a dramatic impact for solving larger instances.

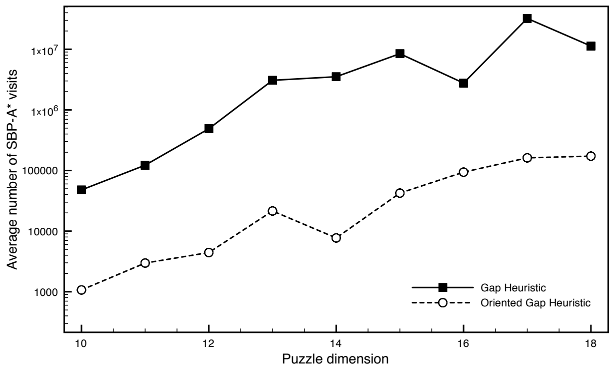

Fig. 8. Impact of the *ogap* heuristic on the number of eSBS-EPE-A* nodes in the burnt pancake puzzle (P2), as a function of the puzzle dimension (y-axis is in log scale).

In fact, until now, the largest solved burnt pancake puzzle had size $M = 18$ [29], solved with a modified gap heuristic and pattern databases. Both our new heuristic (*ogap*) and our new algorithm (eSBS-EPE-A*) made it possible to push the limit and solve the case with 28 pancakes. With our new heuristic, EPEA*, eSBS-EPE-A* and eSBS-EPE-H were able to solve burnt pancake puzzles with $M = 18$ as well, without employing pattern databases (see the results above). Note that, in this problem, the average difference between the goal depth and the initial heuristic estimate is about 5 for $M = 28$, which confirms the highly challenging nature of the problem.

To assess the impact of the *ogap* heuristic, Figure 8 shows the number of eSBS-EPE-A* nodes both with *gap* and *ogap*, as the number of elements grows from 10 to 18 (plotted in log-scale).

It is worth remarking that both P1-85 and P2-28 set new limits for these two permutation sorting problems, as the literature reports no larger problem having been solved before.

6.2.3. 15 Blocks Interchanges (P3)

The blocks interchanges problem has a very large branching factor. The number of nodes grows extremely rapidly with the size of the problem (it was 1,365 for our case), which makes the problem infeasible for the competing algorithms for $M > 15$. The effective branching factors for EPEA* and eSBS-EPE-A* were 7.43 and 8.26, respectively, while eSBS-EPE-H runs out of memory on two instances. This is the most suitable domain for EPEA*, that is almost one order of magnitude faster than eSBS-EPE-A* due to the much lower

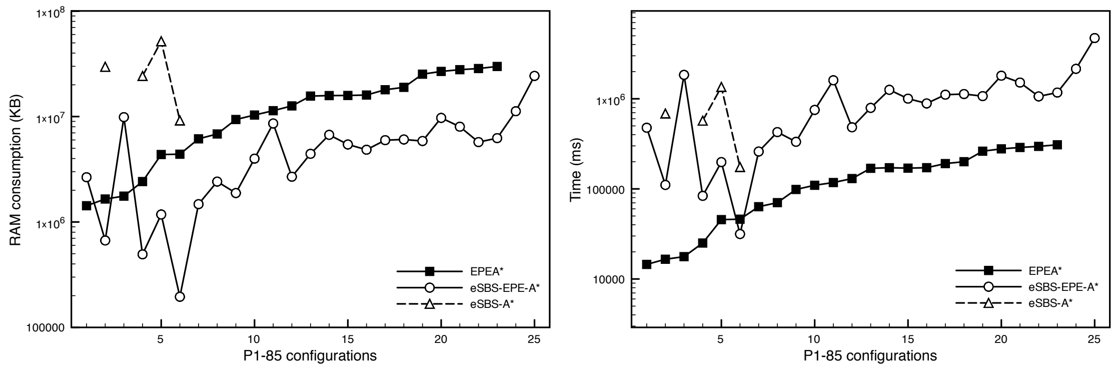

Fig. 9. Memory (left) and time (right) comparison between eSBS-EPE-A*, EPEA* and eSBS-A* on the P1-85 problem, for all 25 instances, ordered by increasing number of EPEA* nodes. Time is plotted in log scale.

number of generated nodes, which was the result of a smaller effective branching factor.

6.2.4. 17 Transpositions (P4)

In transpositions (P4), the space of allowed moves is a subset of blocks interchanges (P3) and, for this reason, the branching factor is smaller (there is one degree of freedom less in specifying moves, given the fact that moved blocks need to be adjacent) than in blocks interchanges. As a consequence, the depth of the solution for the same starting configuration is at least as large as in the blocks interchanges case. In P4-17, eSBS-EPE-A* was faster than EPEA* on average, but here we should remark that eSBS-EPE-A* was faster only in 2 out of 25 instances, which are the most difficult ones and therefore dominate the average. Figure 10 (right) shows the individual instance CPU time of eSBS-EPE-A* and EPEA* as the complexity (expressed as number of nodes) of the P4-17 instances grow (speed is reported in log-scale). This behavior suggests again that the impact of the bidirectional approach grows with the problem size. The comparison on memory usage, reported in Figure 10 (left) highlights the huge memory saving of eSBS-EPE-A* with respect to eSBS-A*, with efficient partial expansion thus playing a crucial role in this domain.

6.2.5. 17 Reversals (P5)

The case of P5-17 is somehow in between the pancake puzzles and the other problems, as it needs to check only four positions (only one more than in P1 and P2), while having a quite large effective branching factor. Here the gap heuristic is still informative, even though not as much as for the pancake puzzle: the average difference between the goal depth and the initial heuristic estimate equals 2.52 for $M = 17$. The effective branching factors for EPEA*, eSBS-EPE-A* and eSBS-EPE-H are 3.12, 3.29 and 4.28, respectively. EPEA* is the fastest algorithm, by generating one order of magnitude less nodes than eSBS-EPE-A*.

6.2.6. 18 Translocations (P6)

Sorting by translocations has a branching factor identical to that of blocks interchanges, with the difference of flipping the moved blocks, as in sorting by reversals. Yet, the heuristic function here is more informative, and thus problems with a larger size can be solved more efficiently. The difference between the goal depth and the initial heuristic function is 1.28 for $M = 18$. Here EPEA* is still the fastest algorithms, but the difference with respect to eSBS-EPE-A* is lower, due to the similar number of generated nodes.

6.3. General observations and guidelines

Overall, some general observations can be made in the light of the experiments described in the previous sections. First, from Tables 2 and 3, it is clear that the eSBS algorithms are constantly outperformed by the corresponding eSBS-EPE versions: eSBS-A* is outperformed by eSBS-EPE-A* (also in the lite versions), as well as eSBS-H by eSBS-EPE-H. Therefore, efficient partial expansion is shown to be a crucial element for these domains, that can be nicely combined with the efficient single frontier bidirectional search framework. As a further proof of this observation, it can be easily observed from Table 4 that EPEA* results to

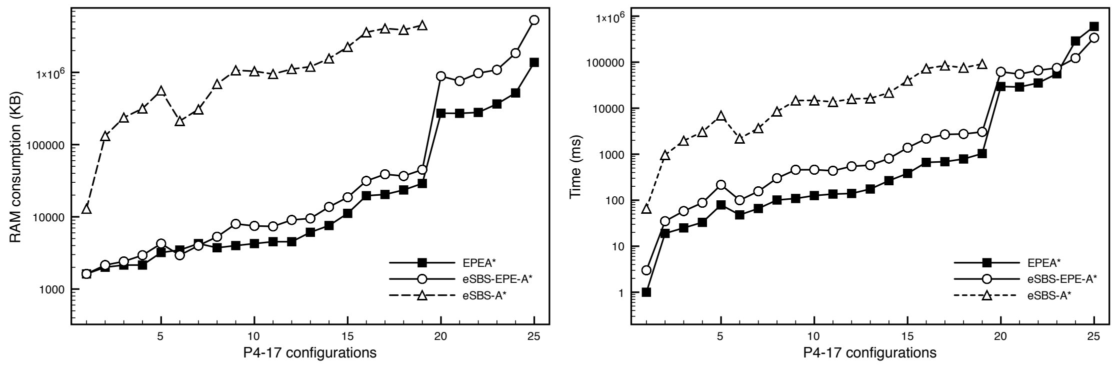

Fig. 10. Memory (left) and time (right) comparison between eSBS-A*, eSBS-EPE-A*, EPEA* and eSBS-A* on the 17-TP problem, for all 25 instances, ordered by increasing number of EPEA* nodes. Time is plotted in log scale.

be the most efficient algorithm only when enough memory is available (P3, P5, P6). Yet, when the complexity of the problem grows, then eSBS-EPE-A* becomes faster, as it is shown by the two most difficult problems of P4-17 (see Figure 10, right). In many cases, eSBS-EPE-A* also requires much less memory, and it is in fact able to solve all the P1 and P2 instances.

When the problems become large EPEA* runs out of memory, one should use alternative algorithms, and the eSBS-EPE framework is perfectly suitable within this context. The novel eSBS-EPE framework combines the advantages of both EPE and eSBS: EPE reduces memory usage by avoiding storing unneeded nodes, while the eSBS does it owing to bidirectional search. These advantages are more evident in those domains where the cost of storing a state is larger than the cost of storing a node ($c_p << c_s$, referring to Section 4.6), which happens in P2-28, but even more in P1-85.

In these domains, also the iterative deepening search becomes a viable alternative. For example, eSBS-EPE-H is the fastest algorithm on P1-85, while suffering in terms of speed when the cost of the OSF computation is not negligible. This clearly happens for P3-15, P4-17, and P6-18, where the nature of the allowed moves requires a check of several gap positions in order to compute the variation of the heuristic function. In addition, on such domains with a very large branching factor, eSBS-EPE-H needs to store many transpositions, thus requiring more memory.

To summarize, the performance of the considered algorithms and their advantages and disadvantages strongly depend on a number of problem-dependent parameters, such as the branching factor, the cost of storing a state, the overhead of applying OSFs, and the informativeness of the heuristic function. Only the net effect of these observation will determine which algorithm should be preferred for a given problem instance. However, we hereby attempt to provide some general rules on the effect of each attribute.

1. One should usually choose either EPEA* or one of the eSBS-EPE variants.
2. When the cost for storing a state is high, EPEA* can suffer from memory requirements. In such cases eSBS-EPE-A* is more efficient. This is confirmed by our results on P1 and P2.
3. When the overhead induced by OSFs is low, and the branching factor is relatively small, then the iterative deepening variant combining eSBS and EPE (eSBS-EPE-H) becomes extremely efficient, as confirmed by our results on P1.
4. When enough memory is available and especially if the cost of storing a state is low, EPEA* is to be preferred: in fact, this is the case where the eSBS framework has fewer advantages. This is confirmed by our results on P3, P5, and P6. Yet, eSBS-EPE-A* is faster than EPEA* on highly complex problems with large branching factor, such as the two hardest instances of P4-17.

Clearly, the development of more sophisticated heuristic functions could have a great impact on the performance of all these algorithms. In particular, the computational time of EPEA* and eSBS-

EPE-A* highly depends on β, the effective branching factor, and on the number of times a given node is expanded (parameter α introduced in Section 3). Both parameters could be reduced with more informed heuristics. If larger problems could be addressed also for the domains with higher branching factor, we argue that eSBS-EPE-A* could outperform EPEA*.

7. Conclusions

In this work we considered the problem of sorting integer permutations with heuristic search algorithms. Several different variants of this kind of problem can be conceived when varying the set of allowed moves. We considered the so-called sorting by transpositions, translocations, reversals, and blocks interchanges, which have recently received great attention in other computer science domains, due to their application to the problem of genome rearrangement in computational biology, but also in logistics and optimization. The pancake and burnt pancake puzzles are the only problem sets, belonging to the family of permutation sorting, which have historically been used in the heuristic search literature. These permutation sorting problems often present very large branching factors and hence they represent extremely challenging tasks for heuristic search algorithms. We believe that these variants of permutation sorting problems could become new challenging benchmarks for heuristic search algorithms.

Sophisticated approaches such as partial expansion or single-frontier bidirectional search are perfectly suitable within this context. Thus, the paper introduced a new family of heuristic search algorithms, named efficient Single-frontier Bidirectional Search with Enhanced Partial Expansion (eSBS-EPE), combining the benefits of partial expansion and single-frontier bidirectional searches.

An extensive experimental evaluation conducted for a wide variety of problem sets highlighted the advantages and limitations of the considered heuristic search algorithms for this kind of task, while showing that the eSBS-EPE approach is a very smart compromise between computational efficiency and memory consumption, often heading to state-of-the-art performance. In general, when increasing the size of the problems, the eSBS-EPE algorithms behave better: while they always employ less memory than the considered competitors, with larger problems they are often even faster. Among the reported experiments, results on the 85-pancake and 28-burnt pancake puzzles pushed the limit and represent the largest problems ever solved for these domains.

Further improvements could be conceived by studying the effect of applying domain-specific jumping policies, or by devising new heuristic functions. In particular, pattern databases could be also employed for this category of problems. Since they are computationally expensive to generate and have large memory requirements, in this work we focused on simpler and more efficient heuristic functions, leaving this interesting research directions for future works.

Acknowledgments

A special dedication of this work is due to Giovanni Soda, who participated to this research with some preliminary discussions, and sadly passed away on July 2nd, 2014. The research was supported by the Israel Science Foundation (ISF) grant number 417/13 to Ariel Felner.

References

[1] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM J. Comput.*, 25(2):272–289, 1996.

[2] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM J. Discrete Math.*, 11(2):224–240, 1998.

[3] J. K. Barker and R. E. Korf. Limitations of front-to-end bidirectional heuristic search. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[4] A. Bergeron, J. Mixtacki, and J. Stoye. On sorting by translocations. In *Research in Computational Molecular Biology*, volume 3500 of *Lecture Notes in Computer Science*, pages 615–629. Springer Berlin Heidelberg, 2005.

[5] L. Bulteau, G. Fertin, and I. Rusu. Pancake flipping is hard. In *Mathematical Foundations of Computer Science (MFCS) 2012*, volume 7464 of *Lecture Notes in Computer Science*, pages 247–258. Springer, 2012.

[6] L. Bulteau, G. Fertin, and I. Rusu. Sorting by transpositions is difficult. *SIAM J. Discrete Math.*, 26(3):1148–1180, 2012.

[7] A. Caprara. Sorting by reversals is difficult. In *First Annual International Conference on Computational Molecular Biology (RECOMB)*, pages 75–83. ACM, 1997.

[8] D. A. Christie. Sorting permutations by block-interchanges. *Information Processing Letters*, 60(4):165 – 169, 1996.

[9] J. Cibulka. On average and highest number of flips in pancake sorting. *Theoretical Computer Science*, 412(8 - 10):822 – 834, 2011.

[10] Á. T. A. de Reyna and C. Linares López. Size-independent additive pattern databases for the pancake problem. In *Fourth Annual Symposium on Combinatorial Search, SoCS 2011*. AAAI Press, 2011.

[11] I. Elias and T. Hartman. A 1.375-approximation algorithm for sorting by transpositions. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 3(4):369–379, 2006.

[12] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. R. Sturtevant, J. Schaeffer, and R. Holte. Partial-expansion A* with selective node generation. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI Press, 2012.

[13] A. Felner, C. Moldenhauer, N. R. Sturtevant, and J. Schaeffer. Single-frontier bidirectional search. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

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

[15] J. Feng and D. Zhu. Faster algorithms for sorting by transpositions and sorting by block interchanges. *ACM Trans. Algorithms*, 3(3), Aug. 2007.

[16] W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27(1):47 – 57, 1979.

[17] M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. R. Sturtevant, R. C. Holte, and J. Schaeffer. Enhanced partial expansion A. *J. Artif. Intell. Res. (JAIR)*, 50:141–187, 2014.

[18] Q.-P. Gu, S. Peng, and H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327 – 339, 1999.

[19] S. Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, 71(1-3):137–151, 1996.

[20] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *J. ACM*, 46(1):1–27, Jan. 1999.

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

[22] L. S. Heath and J. P. C. Vergara. Sorting by bounded block-moves. *Discrete Applied Mathematics*, 88(13):181 – 206, 1998. Computational Molecular Biology DAM - CMB Series.

[23] M. Helmert. Landmark heuristics for the pancake problem. In *SoCS 2010*. AAAI Press, 2010.

[24] R. C. Holte, A. Felner, G. Sharon, and N. R. Sturtevant. Bidirectional search that is guaranteed to meet in the middle. In *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[25] R. C. Holte and I. T. Hernádvölgyi. A space-time tradeoff for memory-based heuristics. In *Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence*, pages 704–709. AAAI Press / The MIT Press, 1999.

[26] H. Kaindl and G. Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[27] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1-2):180–210, 1995.

[28] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13(1-2):180–210, 1995.

[29] M. Keshtkaran, R. Taghizadeh, and K. Ziarati. A novel technique for compressing pattern databases in the pancake sorting problems. In *SOCS*. AAAI Press, 2011.

[30] R. E. Korf. Iterative-deepening-A*: An optimal admissible tree search. In *9th International Joint Conference on Artificial Intelligence*, pages 1034–1036. Morgan Kaufmann, 1985.

[31] A. Labarre and J. Cibulka. Polynomial-time sortable stacks of burnt pancakes. *Theoretical Computer Science*, 412(80):695 – 702, 2011.

[32] Z. Li, L. Wang, and K. Zhang. Algorithmic approaches for genome rearrangement: a review. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 36(5):636–648, Sept 2006.

[33] M. Lippi, M. Ernandes, and A. Felner. Efficient single frontier bidirectional search. In *Fifth Annual Symposium on Combinatorial Search (SoCS) 2012*. AAAI Press, 2012.

[34] K. Qiu, H. Meijer, and S. Akl. Parallel routing and sorting on the pancake network. In *Advances in Computing and Information ICCI '91*, volume 497 of *Lecture Notes in Computer Science*, pages 360–371. Springer Berlin Heidelberg, 1991.

[35] G. Sharon, R. Stern, A. Felner, and N. R. Sturtevant. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.*, 219:40–66, 2015.

[36] A. Solomon, P. Sutcliffe, and R. Lister. Sorting circular permutations by reversal. *WADS*, 2003:319–328, 2003.

[37] G. Tesler. Grimm: genome rearrangements web server. *Bioinformatics*, 18(3):492–493, 2002.

[38] T. Winter. Online and real-time dispatching problems, 1999.

[39] S. Yancopoulos, O. Attie, and R. Friedberg. Efficient sorting of genomic permutations by translocation, inversion and block interchange. *Bioinformatics*, 21(16):3340–3346, 2005.

[40] T. Yoshizumi, T. Miura, and T. Ishida. A* with partial expansion for large branching factor problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 923–929. AAAI Press, 2000.

[41] D. Zhu and L. Wang. On the complexity of unsigned translocation distance. *Theoretical Computer Science*, 352(13):322 – 328, 2006.