Solving Atomix with Pattern Databases

Alex Gliesch and Marcus Ritt
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
Email: {azgliesch, marcus.ritt}@inf.ufrgs.br

***Abstract*—In this paper we study the application of pattern databases (PDBs) to optimally solving Atomix. Atomix is a puzzle, where one has to assemble a molecule from atoms by sliding moves. It is particularly challenging, because the slides makes it hard to create admissible heuristics, and state-of-the-art heuristics are rather uninformed. A pattern database (PDB) stores solutions to an abstract version of a state space problem. An admissible lower bound for a given state is obtained by decomposing it into abstract states and combining their pre-computed solutions. Different from other puzzles a pattern in Atomix cannot be simply obtained by omitting pieces from the puzzle. We also study the search algorithm Partial Expansion A*'s application to Atomix, as a reduced-memory alternative to A*. Experiments show our method solves more instances and significantly improves current lower bounds, running times and node expansions compared to the best solution in the literature.**

I. INTRODUCTION

Atomix is a single player game developed by Günter Krämer and published by Thalion Software in the 1990s [1]. Atomix is PSPACE-complete [2]. The game takes place on an integer board of size $w \times h$, on which are distributed n pieces, called *atoms*. Some board positions are walls through which no atom can pass. Each atom is identified by a label, and multiple atoms have the same label.

Atoms can be moved with *sliding operations*. When an atom is slid in a direction (up, down, left, right), it will move in that direction until hitting an obstacle, which can be a wall or another atom: the atom will then stop at the position just before the obstacle. When sliding, an atom may not stop at any intermediate position between its initial position and its stopping position.

The goal of the game is to assemble the atoms into a specific formation called *molecule*, which is given in the problem statement. There is no restriction on where on the board the molecule must be assembled, as long as the relative positions between the atoms are as required. The molecule may not be mirrored or rotated, and always contains all atoms. Note that atoms that have the same label may have their positions interchanged in the final molecule. In this paper, we are interested in the problem of assembling the final molecule using the minimum number of moves mossible.

A *game instance* defines the board layout (walls, free positions), the number of atoms and their types, the molecule to be assembled, and the initial position of every atom. Figure 1 shows a typical Atomix instance.

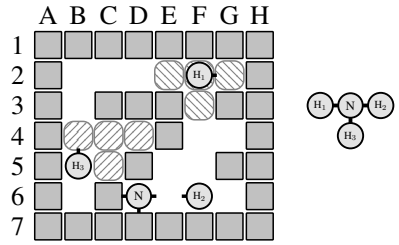

Fig. 1. An example instance. The goal of this instance is to assemble the molecule ammonia, on the right. Gray squares represent walls, and white space represents free positions. The line-patterned squares denote the two possible goal positions of the molecule. The atom label is formed by both the atom type (hydrogen, nitrogen, etc.) and the orientation of its links. In this instance, all atoms are uniquely labeled.

A. Formal definition

An Atomix instance can be represented formally by a tuple (W, L, s_0, F), where $W \in \{0,1\}^{wh}$ is a boolean matrix such that $W_{ij} = 1$ iff the position (i, j) on the board is a wall, $L \subset \mathbb{N}$ is the set of atom labels, s_0 is the initial game state, and F is a set of goal states.

A *game state* s is a set of pairs $(p_i, l_i), i \in [n]$, where $p_i \in \mathbb{N}^2$ is the board position of an atom with label $l_i \in L$. We denote by $S \subseteq \mathbb{N}^2 \times L$ the set of all possible game states. A board position p is said to be *empty* in state s if $W_p = 0$ and $(p, l) \notin s, \forall l \in L$. A position $(r, c) \notin [w] \times [h]$ is said to be *out of bounds*. A *direction* is a coordinate $d \in D$, where $D = \{(0,1), (0,-1), (1,0), (-1,0)\}$ is the set of all directions.

A *move* is a function $move : \mathbb{N}^2 \times L \times D \times S \to S$ that, when applied on an atom (p, l), direction d and state s, yields a state $s' = s \setminus \{(p, l)\} \cup \{(p', l)\}$, where p' is the first position $p + \delta d$ such that $p + \delta' d$ is empty in s for $\delta' \in [0, \delta]$ and $p + (\delta + 1)d$ is not empty. If either $(p, l) \notin s$ or p' is out of bounds the move yields s itself. The neighborhood of a given state s is the set of states $N(s) = \{move(p, l, d, s) \mid d \in D, (p, l) \in s\} \setminus \{s\}$.

The *distance* between two states s and s' is the minimum number of moves necessary to transform s into s'. The goal of the game is to find the minimum distance between the origin state s_0 and any goal state in F.

B. Previous work on Atomix

Hüffner et al. [1] use A* and IDA* to solve Atomix. They propose an admissible lower bound based on relaxed atom moves called *generalized moves*. A generalized move allows atoms to stop at any free space in a given direction, instead of

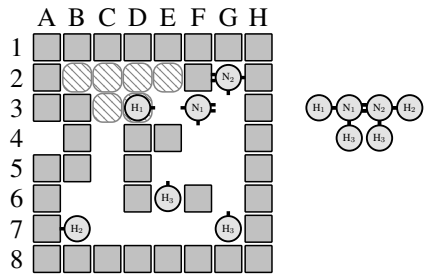

Fig. 2. Example of the computation of the generalized moves heuristic. The generalized distances are: 2 for H_1, 3 for H_2, 2 for N_1, 3 for N_2. Notice that both H_3 have the same label. The possible matchings are {E6–C3, G7–D3} of cost $3 + 2 = 5$, and {E6–D3, G7–C3} of cost $4 + 2 = 6$. The first matching has the minimum cost of 5, thus the final heuristic value is $3 + 2 + 2 + 3 + 5 = 15$.

only at the position just before an obstacle. It also allows for more than one atom to occupy the same space. The *generalized distance* between two positions is the minimum number of generalized moves to move an atom from one position to the other. The lower bound for a state s with respect to a goal state f is the sum of generalized distances of every atom position in s to the corresponding goal position of that atom in f. The generalized distance can be pre-computed for every possible pair of board positions by performing an all-pairs-shortest-paths algorithm on the underlying graph. The paper also proves that this heuristic is admissible and consistent.

If there are multiple atoms with the same label, there exists more than one way to bring these atoms to their goal positions. In order to achieve this efficiently, a *minimum cost perfect matching* is performed on the bipartite graph induced by the generalized distances between the atoms in s and the atoms in f. This problem can be solved in $O(n^3)$ [3]. A similar idea is used in the standard heuristic for Sokoban Junghanns and Schaeffer [4], where the stones must be matched to their goal positions. Figure 2 illustrates the computation of the generalized moves heuristic.

Hüffner et al. deal with multiple goal states in by an iterative deepening strategy similar to IDA*: for an increasing upper bound on the f-values, it applies a depth-first search for each goal state. Alternatively, the search can be an A* search. The latter case has the useful property that it will not generate any states with f-value greater than the optimal solution length, because it discards states with f-value greater than the upper bound, and thus reduces A*'s memory consumption considerably.

The paper also uses the fact that the *generalized moves* heuristic is monotone to propose an efficient open list data structure for A*, which is several times faster than standard implementations of priority queues. The disadvantage of this approach is that it disallows tie-breaking rules to assign further priority to states with the same f-value.

C. Contributions of this paper

In this paper, we propose a different way of handling multiple goal states through a modified generalized moves heuristic. We also present three new heuristics based on additive pattern databases, which improve the state-of-the-art lower bounds achieved by the generalized moves heuristic.

Lastly, we compare the iterative deepening strategy used by Hüffner et al. [1] with a regular A* and the reduced memory approach Partial Expansion A* (PEA*) [5]. We show that PEA*, as a compromise between speed and memory consumption, is able to reduce the number of stored states significantly, as opposed to A*, and is more efficient than the approach using iterative deepening.

The remainder of this paper is organized as follows: in Sections II, III and IV we describe our proposed approaches on multiple goal states, pattern databases and search algorithm, respectively. In Section V we present the computational results of our approaches and compare them with the implementation by Hüffner et al. [1]. We conclude in Section VI.

II. Handling multiple goal states

A simple strategy would be to run an independent search for each goal state, and return the best solution. However, because there is no guarantee that a specific goal state is even reachable, some searches might run until the state space is exhausted. Hüffner et al. [1] solve this by an iterative deepening approach, as discussed above. This method expands every state with an f-value at most the optimal distance once for every goal state and upper bound, but not more than a factor of the problem's effective branching factor [6], compared to a direct approach.

Another possible strategy would be to run A* searches for each goal state in a round-robin fashion, and expand a fixed number of states each time. When a solution is found, the corresponding search stops, and all other searches become bounded to f-values less than that solution's length. The algorithm terminates when all searches stop. The major downside of this approach that is consumes much more memory, since a separate open and closed list must be kept for each search.

Both strategies above expand the same state several times. To solve this problem, we propose to run a single search (either A* or IDA*) guided by all goal states. This means that any admissible heuristic must be a lower bound on the minimum distance of the current state to any of the goal states. Generalized moves can be easily modified to satisfy this, by returning the minimum sum of generalized distances among all goal states.

The main advantage of the strategy that considers all goal states is that a single A* search is enough to find an optimal solution, if it exists. If the heuristic is consistent, states will be expanded only once. Clearly, the heuristic is less informed, but we will show in Section V that this strategy reduces the number of state expansions by a factor of two.

III. Pattern databases

Pattern Databases (PDBs) [7], are one a powerful way to create admissible heuristics. They have been widely used to solve benchmark problems such as the $(n^2 - 1)$-puzzle [7], Sokoban [8], Rubik's Cube [9].

PDBs use an abstraction to reduce the state space problem to a problem with a smaller, abstract state space. The optimal solution of the abstract problem is stored in a look-up table called a PDB, and is used as a heuristic function for the original problem. The abstract state space must be small enough so that it can be exhaustively explored in feasible time and its solution values stored in the PDB. In sliding block puzzles this is normally done by removing some pieces and solving the original puzzle with the remaining pieces (called a *pattern*). In the abstracted problem, a state is identified exclusively by the positions of the pieces. A PDB is constructed by visiting all reachable abstract states with a backward breadth-first search (BFS), starting from the abstract goal state, and storing the distance to every other abstract state.

Of particular interest are *disjoint pattern databases* [10]. In sliding block puzzles, two PDBs are disjoint if their pieces are disjoint. The advantage of disjoint PDBs is that the sum of their heuristic values is an admissible heuristic, whereas non-disjoint PDBs must be combined by taking the maximum among them. In a *statically-partitioned* PDB, the patterns are predefined before the PDB is built. In a *dynamically-partitioned* PDB, a PDB is built for every possible pattern, and the partition is performed at run time, so as to choose the partition that maximizes the sum of the contributions of each disjoint set [11].

A. Constructing PDBs for Atomix

In Atomix, we cannot simply remove a subset of atoms to create a simpler pattern: a heuristic that only removes atoms is not admissible, since sliding atoms may need support from other atoms to reach certain positions. For example, when we remove all atoms except N_2 in Figure 2, N_2 cannot reach its goal any more. We call this kind of interaction a *positive interaction*, since the presence of an atom may shorten the distance to the goals of other atoms. Similarly to other block-moving puzzles, there are also *negative interactions*, where an atom blocks the optimal path of other atoms.

Atomix with the generalized moves relaxation is still too difficult to be solved completely and stored in a PDB. However, in this variant, atoms do not need interactions to achieve an optimal path from their initial to their goal positions, meaning an abstraction that removes atoms is still admissible, enabling us to build admissible additive PDBs on disjoint subsets of atoms. Because it is a relaxation of standard Atomix, any heuristic for generalized Atomix is also admissible for standard Atomix.

We define an atom pattern T as a multiset of atom labels. An abstract state is a state having one atom with label l for every $l \in T$. The neighbourhood of an abstract state is the set of all state obtained by generalized moves of its atoms. Given a partition of the n atoms into disjoint patterns, a PDB for a goal state f is constructed as follows. For every pattern, we perform a backward BFS starting from the set of abstract states that comprise all possible assignment of the pattern's atoms to their positions in f. If an atom in T has multiple goal positions (because there exist more atoms with the same label), there may be more than one possible assignment. The BFS visits all possible abstract states of that pattern, and, for every abstract state, stores in the PDB look-up table the minimum number of movements needed to reach it.

In essence, the PDB we propose is a generalized moves heuristic that also captures negative interactions between atoms in the same pattern. A particular type of negative interactions are *linear conflicts* Hansson et al. [12], which happen when the optimal generalized paths of two atoms meet in opposite directions. Note that if all patterns have size 1 (each atom is in its own pattern) the PDB heuristic is the same as the generalized moves heuristic.

Although capturing interactions helps increase the lower bounds, the heuristic value is upper-bounded by the solution to generalized Atomix. In particular it abstracts the sliding moves, which contributes significantly to the number of moves to solve Atomix.

B. A statically-partitioned disjoint PDB

A statically-partitioned PDB of size k will, in a pre-computing step, partition the n atoms into $\lfloor n/k \rfloor$ patterns of size k and, for each pattern, calculate and store the optimal solution of the abstract problem for all possible configurations. If n is not divisible by k, a PDB of size $n \mod k$ is constructed for the remaining atoms. We build a PDB for each goal state.

For an abstract state, the heuristic value to some goal state is the sum of the distances of all (disjoint) abstract states to the corresponding abstract goal state, and the heuristic value of a state is the minimal heuristic value over all goal states. It can be obtained in time $O(|F| \lceil n/k \rceil)$, and occupies $O(|F| \lfloor n/k \rfloor m^k)$ memory: for each goal state, and for each pattern, we store all possible distributions of the k atoms over the m board positions. Considering current memory sizes, and that we store the PDB's lower bound in one a single byte, a statically-partitioned PDB is only applicable for $k \leq 3$: the maximum memory usage, on all instances, would be 193 MiB (case $k = 4$ would need 17971 MiB).

Since the static partition will stay the same during the search, selecting a good partition is important: partitions that capture more linear conflicts are expected to yield better heuristic values. Figure 3 illustrates this. A good heuristic is to put atoms that are more likely to interact with each other into the same part. A simple but effective heuristic is to choose a partition that minimizes the sum of generalized distances in the final molecule of atoms within each pattern. We obtain this partition by a best-improvement multi-start local search, starting from a random partition, in the neighborhood that swaps all possible pairs of atoms in different patterns.

Because the memory usage of a statically-partitioned PDB with $k = 3$ is rather small, and the calculation of the PDB is fast, we can afford to have multiple statically-partitioned PDBs and take the maximum heuristic among them. In our implementation we use one PDB whose partition is chosen by local search, as described above, and p other PDBs whose partitions are selected randomly. The final heuristic value is the

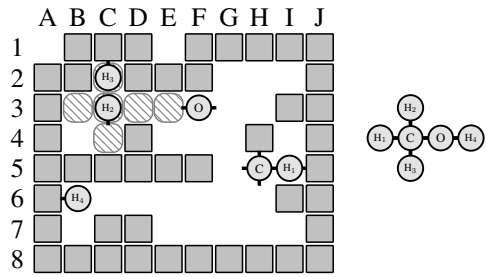

Fig. 3. The PDB partition $\{\{H_2, H_3, H_4\}, \{H_1, C, O\}\}$, for instance, captures 4 linear conflicts (H_2–H_3, H_1–C, H_1–O, C–O) and gives a lower bound of 16 (7+9). The partition $\{\{H_3, H_4, C\}, \{H_1, H_2, O\}\}$, however, captures only one linear conflict (H_1–O), and gives a weaker lower bound of 14 (7+7). The generalized moves heuristic gives a lower bound of 12.

maximum among all those PDBs. Empirically, we have found that generating some partitions randomly leads to better lower bounds, on average, than selecting them all by local search: a possible explanation is because this increases the diversity of atom groupings, and is able to produce better heuristic values on states which are far from the goal state. Higher values of p are more likely to yield better heuristics, but will also occupy more memory and increase the heuristic computation time. In preliminary tests, we have found $p = 5$ to be a good compromise.

C. A dynamically-partitioned PDB

A dynamically-partitioned PDB of size k will, in a pre-computing step, calculate the abstract solution of all $\binom{n}{k}$ partitions of the n atoms into patterns of size k. Similarly to a statically-partitioned PDB, $|F|$ dynamically-partitioned PDBs will be built. For an instance with m free board positions, these PDBs require $O(|F|\binom{n}{k}m^k)$ memory, since each partition has $O(m^k)$ possible configurations. Considering current memory sizes, only dynamically-partitioned PDB of $k \leq 2$ are feasible to construct: the maximum memory usage on all standard instances is 24 MiB (case $k = 3$ would need 16527 MiB).

The dynamically-partitioned PDB is implemented as a look-up table $\text{d-PDB}(f, i, j, p_i, p_j)$ that stores the minimum distance of a pattern with labels i and j in positions p_i and p_j to reach their positions in goal state f. The heuristic of a state s is computed as follows. For every goal state f, we define a complete graph where each vertex represents an atom, and two vertices i and j with $\{(p_i, i), (p_j, j)\} \subseteq s$, are connected by an edge of weight $\text{d-PDB}(f, i, j, p_i, p_j)$. We then compute a maximum weight perfect matching on this graph, which defines the pairs of atoms. This matching can be found in time $O(n^3)$ [13]. If the number of atoms is odd, we add a dummy vertex which connects to every other vertex u with weight equal to the minimum generalized distance between u and any of its goal positions in f. This procedure is repeated for every goal state, and the minimum matching among all goal states is used as a lower bound. Figure 4 illustrates the computation of the heuristic.

The major shortcoming of this approach is that a maximum cost perfect matching must be computed for each node and

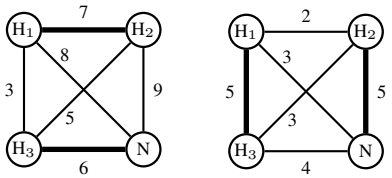

Fig. 4. For the Atomix instance of Fig. 1, the figures below show the matching for the left-most and right-most goal state, with the edges of a perfect matching drawn in bold. The lower bound is given by the minimum matching for all goal states, which is $\{\{H_1, H_3\}, \{H_2, N\}\}$ and has value 10 (5+5).

each goal state. This procedure is significantly more time-consuming than the generalized moves heuristic or a statically-partitioned PDB. One way to improve this would be to store the matched edges together with the state and perform the matching only once every fixed number of movements; however, this would effectively double the memory usage of a state.

Because the dynamically-partitioned PDB always selects, among all partitions, the one that yields the maximum heuristic value, it is always an upper bound on a statically-partitioned PDB for a fixed k. In this paper, however, we use $k = 2$ for the dynamically-partitioned PDB and $k = 3$ for the statically-partitioned PDB, so this is not necessarily true: a statically-partitioned PDB may be able to penalize more interactions.

IV. Reducing the memory consumption of A* / Partial Expansion A*

As a consequence of Atomix's large branching factor, we see a big discrepancy between the amount of states generated and expanded during an A* run. In most instances, over half of A*'s state table is filled with states having an f-value greater than the optimal solution length (f^*). These states occupy a large amount of space, and, by definition, will never be expanded. Because our heuristics are relatively fast to compute, A* is able to generate up to hundreds of thousands of states per second. In practice, the memory limit is typically reached within a few minutes.

Aiming at reducing the number of states generated in problems with large branching factors, Yoshizumi et al. [5] propose a modification to standard A* called Partial Expansion A* (PEA*), which has the property of never adding states with $f > f^*$ to the open list. In PEA*, when a state s is expanded, all of its child states are generated, but only childs c with $f(c) = f(s)$ are added to the open list. After this step, if there are still children of s with $f(c) > f(s)$, $f(s)$ is updated to be the minimum f among those children, and s is reinserted in the open list. Felner et al. [14] propose a modification to PEA* for heuristics that are not consistent: instead of adding to the open list only children with $f(c) = f(s)$, all children with $f(c) \leq f(s)$ are added.

In PEA*, a state can be expanded as many times as there are different f-values among its children. In each expansion, we must generate its children, compute their lower bounds,

TABLE I
CHARACTERISTICS OF THE INSTANCES IN OUR STANDARD TESTBED.

	Min.	Avg.	Max.
Number of atoms (n)	3	10.39	32
Goal states	1	5.57	64
Board size	36	146.78	225
Solution length	7	21.56	47
Best known lower bound	7	30.53	65
Effective branching factor	1.12	9.93	34.26
Search space size	17550	8.25×10^{47}	1.25×10^{50}

and test if they are already in *closed*. These operations can incur a significant time overhead, compared to normal A*.

In this paper, we have implemented PEA* to solve Atomix, as a reduced-memory alternative to A*. As we will show in Section V-C, PEA* is able to reduce memory usage by an order of magnitude, compared to A*, while also being faster than the iterative deepening strategy of Hüffner et al. [1].

V. COMPUTATIONAL RESULTS

A. Experimental setup

All experiments were performed on a PC with an AMD FX-8150 Eight-Core CPU running at 3.6 GHz, with 32 GB of main memory. Each execution was limited to 10 GB of memory, and one hour of computation time. We have implemented the algorithms in C++, and compiled them with GCC 5.3.1 and maximal optimization. For the maximum weighted perfect matching, we used the implementation by Kolmogorov [15]. The standard set contains 155 Atomix instances. Table I summarizes some characteristics of the instances. All summary statistics are reported over all instances, except the solution length, which is reported only for the 88 solved instances.

In preliminary tests, we have found the best configuration of parameters to be the following: PEA* with a heuristic considering all goal states, a statically-partitioned PDB, and breaking ties by goal count. In the following subsections, we compare the different methods using this base configuration.

In each table, we report the number of solved instances ("# solved"), the average relative deviation from the best known lower bound over all instances ("avg. rel. dev."), the number of expanded and opened nodes, and the total solving time in seconds ("time (s)"). The relative deviation is the average over all instances, the remaining quantities only over the instances that have been solved by all variants. Since in PEA* not all generated nodes are put on the open list, we report the number of generated and opened nodes separately. A node is counted as opened if it has not been discarded by an f-value limit and has been added to the open list.

B. Pattern databases

Table II compares the performance of the statically-partitioned and dynamically-partitioned PDBs, as well as the version without a PDB which uses the generalized moves heuristic. It additionally reports the number of instances that have reached the time or memory limit, the average relative deviation from the best lower bound of the initial heuristic value ("avg. initial heuristic"), and the time spent in building and consulting the PDB including the time for computing the matchings ("PDB time (s)"). As discussed in Section III-B, the PDB build time is negligible. Because the performance of the statically-partitioned PDB depends on random partitions, we report in this case averages over 10 replications.

TABLE II
COMPARISON OF THE PDB VARIANTS. THE NUMBER OF INSTANCES SOLVED BY ALL VARIANTS WAS 71.

	Static	Dynamic	No PDB
# Solved	82.8	71	77
Time limit reached	52.5	84	25
Memory limit reached	19.7	0	53
Avg. rel. deviation (%)	0.58	1.72	1.47
Avg. initial heuristic (%)	24.04	23.39	26.23
Time (s)	2,952	17,405	3,420
PDB time (s)	647	15,308	0
Nodes expanded ($\times 10^8$)	3.39	2.39	8.72
Nodes opened ($\times 10^8$)	2.49	1.70	6.27

TABLE III
COMPARISON OF THE DIFFERENT SEARCH METHODS. THE NUMBER OF INSTANCES SOLVED BY ALL VARIANTS WAS 75.

	PEA*	A*	It. Deep. Strategy
# Solved	83	75	80
Avg. rel. deviation (%)	0.62	1.91	0.29
Time (s)	6,381	4,606	7,379
Nodes expanded ($\times 10^8$)	3.99	2.95	9.13
Nodes re-expanded ($\times 10^8$)	3.72	0	0
Nodes opened ($\times 10^8$)	2.92	18.09	15.04
Nodes generated ($\times 10^8$)	45.82	18.09	87.58

We see that the statically-partitioned PDB solved the most instances, 82.8 on average, having also the best overall performance, both in terms of expanded nodes, computation time, and lower bounds. Although its heuristic is slightly more costly to compute than the generalized moves, the improved lower bounds help it expand approximately 2.5 times fewer nodes.

The dynamic PDB solved the least number of instances, 71. It performed even worse than the version without any PDB. The large running time of this variant is due to the high cost of performing a maximum perfect matching at every heuristic call, for every final state: the time used by the PDB amounts to over 85% of the total running time. Because of this, the dynamically-partitioned PDB hits the time limit on all of the unsolved instances. However, it expands the least amount of nodes, since it has the highest lower bounds. We expect that, if the time limit is large enough, the reduced number of expansions of the dynamically-partitioned PDBs could compensate the time for computing it.

C. Search algorithms

Table III compares A*, PEA*, and the iterative deepening strategy, using A*. It additionally report the number of re-expanded nodes. We see that A* has solved the least number of instances, 75. On all unsolved instances, it terminates reaching the memory limit, in average in 7 minutes. As expected, considering only the solved instances A* has the

TABLE IV
COMPARISON OF OUR SOLVER WITH HÜFFNER ET AL. [1]'S SOLVER. THE NUMBER OF INSTANCES SOLVED BY BOTH VARIANTS WAS 75.

	This work	Hüffner et al. [1]
# Solved	82.8	75
Avg. rel. deviation (%)	0.56	1.67
Time (s)	9,211	12,962
Nodes expanded ($\times 10^8$)	6.79	47.18
Nodes opened ($\times 10^8$)	5.03	900.15

best performance, being the fastest and expanding the least number of nodes.

PEA* solved the most instances, 83. It uses a factor of 6.2 less memory compared to A*. The reduction is proportional to the number of opened nodes. Compared to the iterative deepening strategy, PEA* was 15% faster and expanded over two times fewer nodes, although the average relative deviation is higher.

D. Comparison with Hüffner et al. [1]'s solver

Table IV compares the average results of 10 replications of our solver with the implementation by Hüffner et al. [1] using the iterative deepening strategy and the A* algorithm, under the same time and memory limits. Hüffner et al. [1]'s code has been compiled and run in our environment. Hüffner et al. [1]'s program crashed on the execution of instances kai_20, kai_21 and kai_27, so these instances are not taken into account for the calculation of the average relative deviation.

We can see that the approach using PDBs is able to solve on average 7.8 more instances. The number of expanded nodes for solving an instance is almost an order of magnitude smaller, and the running time approximately 40% less. It also reduces the average relative deviation by about 3%.

VI. CONCLUSIONS

In this paper, we have applied pattern databases to create admissible heuristics for Atomix that penalize negative interactions between atoms. We have proposed a statically- and a dynamically-partitioned PDB, and showed that the dynamically-partitioned PDB yields stronger lower bounds leading to fewer node expansions, but requires a substantial amount of time. On the other hand, the statically-partitioned PDB offers a middle ground between stronger heuristics and computational efficiency, achieving better overall results than the generalized moves heuristic proposed by Hüffner et al. [1].

Furthermore, we have shown that, under the imposed memory and time limits, A* consumes available memory rather quickly, mostly due to an inexpensive heuristic. Partial Expansion A* mitigates this problem by not storing states with f-value greater than the solution length.

Lastly, we have compared our experimental results with the state-of-the-art results by Hüffner et al. [1], and shown that the proposed method is able to solve an average of 7.8 extra instances, reducing the number of nodes expanded by an order of magnitude, and improving current known lower bounds.

REFERENCES

[1] F. Hüffner, S. Edelkamp, H. Fernau, and R. Niedermeier, "Finding optimal solutions to Atomix," in *KI 2001: Advances in Artificial Intelligence*, 2001, pp. 229–243.

[2] M. Holzer and S. Schwoon, "Assembling molecules in ATOMIX is hard," *Theor. Comput. Sci.*, vol. 313, no. 3, pp. 447–462, 2004.

[3] J. Munkres, "Algorithms for the assignment and transportation problems," *J. Soc. Ind. and Appl. Math.*, vol. 5, no. 1, pp. 32–38, 1957.

[4] A. Junghanns and J. Schaeffer, "Sokoban: Enhancing general single-agent search methods using domain knowledge," *Artif. Intell.*, vol. 129, no. 1–2, pp. 219–251, 2001.

[5] T. Yoshizumi, T. Miura, and T. Ishida, "A* with partial expansion for large branching factor problems." in *AAAI/IAAI*, 2000, pp. 923–929.

[6] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artif. Intell.*, vol. 27, no. 1, pp. 97–109, 1985.

[7] J. C. Culberson and J. Schaeffer, "Searching with pattern databases," in *Canadian Conference on Artificial Intelligence*, 1996, pp. 402–416.

[8] A. G. Pereira, M. Ritt, and L. S. Buriol, "Finding optimal solutions to Sokoban using instance dependent pattern databases," in *Symposium on Combinatorial Search*, 2013, pp. 141–148.

[9] R. E. Korf, "Finding optimal solutions to Rubik's cube using pattern databases," in *AAAI Conference on Artificial Intelligence*, 1997, pp. 700–705.

[10] R. E. Korf and A. Felner, "Disjoint pattern database heuristics," *Artif. Intell.*, vol. 134, no. 1–2, pp. 9–22, 2002.

[11] A. Felner, R. E. Korf, and S. Hanan, "Additive pattern database heuristics," *J. Artif. Intell. Res.*, vol. 22, pp. 279–318, 2004.

[12] O. Hansson, A. Mayer, and M. Yung, "Criticizing solutions to relaxed models yields powerful admissible heuristics," *Inf. Sci.*, vol. 63, no. 3, pp. 207–227, 1992.

[13] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Courier Corporation, 1982.

[14] A. Felner, M. Goldenberg, G. Sharon, R. Stern, T. Beja, N. R. Sturtevant, J. Schaeffer, and R. C. Holte, "Partial-expansion A* with selective node generation." in *AAAI*, 2012.

[15] V. Kolmogorov, "Blossom V: A new implementation of a minimum cost perfect matching algorithm," *Math. Progr. Comput.*, vol. 1, no. 1, pp. 43–67, 2009.