Finding Optimal Solutions to Atomix

Falk Hüffner*

Wilhelm-Schickard Institut für Informatik,

Arbeitsbereich Theoretische Informatik/Formale Sprachen,

Universität Tübingen,

Sand 13, D-72076 Tübingen, Fed. Rep. of Germany

`hueffner@informatik.uni-tuebingen.de`

January 9, 2003

Abstract

We present solutions of benchmark instances to the solitaire computer game Atomix found with different heuristic search methods. The problem is PSPACE-complete. An implementation of the heuristic algorithm A* is presented that needs no priority queue, thereby having very low memory overhead. The limited memory algorithm IDA* is handicapped by the fact that, due to move transpositions, duplicates appear very frequently in the problem space; several schemes of using memory to mitigate this weakness are explored, among those, "partial" schemes which trade memory savings for a small probability of not finding an optimal solution. Even though the underlying search graph is directed, backward search is shown to be viable, since the branching factor can be proven to be the same as for forward search.

1 Introduction

Atomix was invented in 1990 by Günter Krämer and first published by Thalion Software for the popular computer systems of that time. The goal is to assemble a given molecule from atoms (see Fig. 1). The player can select an atom at a time and "push" it towards one of the four directions north, south, west, and east; it will keep on moving until it hits
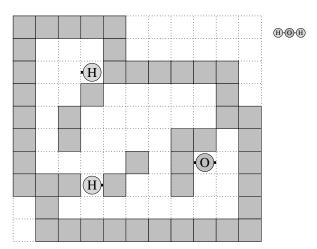

Figure 1: A simple Atomix problem (Atomix 01 in the list of the appendix). It can be solved with the following 13 moves, where the atoms are numbered left-to-right in the molecule: 1 down left, 3 left down right up right down left down right, 2 down, 1 right.

---

*Advisors: Henning Fernau, Klaus-Jörn Lange, Rolf Niedermeier (Universität Tübingen)
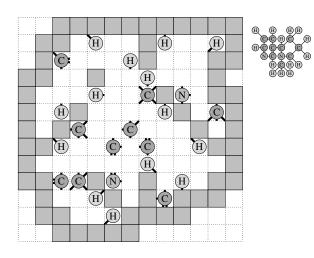
Figure 2: A more complex Atomix problem (level number 43 from the "katomic" implementation). It takes at least 66 moves to solve.

an obstacle or another atom. The game is won when the atoms form the same constellation (the "molecule") as depicted beside the board. A concrete Atomix problem, given by the original atom positions and the goal molecule, is called a *level* of Atomix.

The original game had a time limit and did not count the moves needed; we will instead focus on the analytical aspect and try to minimize the solution length as a goal. Note that we are only interested in *optimal* solutions; in order to just find any solution fast, quite different algorithms would be necessary.

An implementation of this Atomix variation for the X Window System is available as "katomic" from `http://games.kde.org`. A JavaScript version can be played online at `http://www.sect.mce.hw.ac.uk/~peteri/atomix`.

Our solver program written in C++ is able to solve 17 of the 30 problems from the original Atomix and 18 of the 67 problems from katomic optimally. In an appendix, we list a selection of these findings.

2 Heuristic Search

Many common problems and, especially, most solitaire puzzles can be formulated as a *state space search* problem: given are a start state, a set of goal

states and a set of operators to transform one state into another; wanted is a sequence of operators, also simply called a *move sequence*, that transforms the start state into a goal state and that is of minimal length. A state space can be represented as a graph, with nodes representing states and (directed) edges representing moves. That way, well-known graph algorithms can be applied. To emphasize this aspect, states generated in a state space search are often called "nodes".

In the general case, each operator is associated with a cost, and the sum of the costs over the solution sequence is to be minimized. For simplicity, we will assume unit costs for each operator and talk about "number of moves" instead of costs.

For hard combinatorial problems, the use of *heuristics* can often lead to dramatic improvements for a state space search. Many problems would even be unsolvable without them. For a state space search, "heuristic" has a well-defined meaning: an estimate of the moves left from the current state to a goal.

Of special interest are *admissible* heuristics: they never overestimate the number of moves. The well-known algorithms A* and IDA* can be proven to always find an optimal solution when using an admissible heuristic. An admissible heuristic judges the "quality" of a state s: if $g(s)$ is the number of moves already applied, and $h(s)$ is the heuristic estimate, then $f(s) := g(s) + h(s)$ is a lower bound on the total number of moves.

This number, customarily called the "f-value", can be used in two ways: to guide the search and to reduce the effective depth of the search. The first idea naturally leads to the A* algorithm: "promising" states are examined first. The second is applied in the IDA* algorithm: "hopeless" states are not examined at all. As elaborated later, both algorithms can be extended to also take the other aspect into account.

There is no general method of finding admissible heuristics; usually, one tries to examine variations of the original problem with relaxed restrictions, where the solution length can be trivially found.

3 Related Puzzles

Atomix has some similarities to well-known other puzzles, but also some interesting new properties.

Figure 3: The 15-puzzle in its goal state.

Table 1 compares some search space properties of Atomix to other games.

It seems that regarding "difficulty" Atomix lies somewhere in between the 24-Puzzle and Sokoban: Solution lengths are shorter than for the 24-Puzzle or Sokoban, but the branching factor is considerably higher than for the 24-Puzzle, and the state space is considerably larger than for Sokoban.

Due to its close relationships to Atomix (which will become important in the next section), we discuss the 15- and the 24-puzzle as special instances of the $(n^2 - 1)$-puzzle in more details.

3.1 The 15-Puzzle

The 15-puzzle (see Fig. 3) was invented in 1878 by Sam Loyd, and became instantly very popular all over the world [JS79]. It consists of a square tray of size 4×4 with 15 tiles numbered 1 through 15 and one empty square. A move consists of sliding one tile adjacent to the empty square into the empty space. The goal is to obtain the usual ordering of the numbers on the tiles by some move sequence.

The 15-puzzle is likely to be the most thoroughly analyzed puzzle of this kind [Kor85, KT96]. It serves as a kind of "fruit fly" for heuristic search. It is easy to implement, has an obvious heuristic with the "Manhattan distance", and not too large a search space.

The Manhattan distance heuristic can be calculated by summing up the number of turns it would take for a tile to get to its goal position if it was the only tile in the tray. This is obviously a lower bound on the actual number of turns.

Many search methods developed for the 15-puzzle can be easily adapted for Atomix. One important difference is that the underlying search graph for Atomix is directed; not every move can be undone.
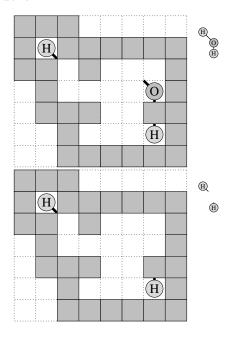

Figure 4: The top problem can be solved in 13 moves. We cannot get a lower bound by leaving out one atom, as in the bottom picture; the problem even becomes unsolvable.

Improved heuristics for the 15-puzzle make it possible to solve even the extended "24-puzzle"-variation [KT96]. Most of them follow the common theme of examining a sub-problem where only a few tiles are regarded and most are ignored. The "linear conflict heuristic" [HMM92], for example, tries to find pairs of tiles in a row or column which need to pass each other to get to the goal position. In such a case, another two moves can be added to the heuristic given by the Manhattan distance, since one tile will have to move out of the way and back.

The work of Culberson and Schaeffer [CS96, CS98] generalizes this idea to "pattern databases": Each possible distribution of the tiles 1–8 on the board is analyzed and solved, yielding a lower bound which is often better than the Manhattan heuristic with the linear conflict heuristic, since there are more tile interactions. The same is done

3

		24-Puzzle	Rubik's Cube	Sokoban	Atomix
Branching factor	Range	2–4	18	0–50	12–40
	Effective	2.3	13.3	10	7
Solution length	Typical	100	18	260	45
	Range	1–112	1–20	97–674	8–70
Search-space size	Upper bound	10^{25}	10^{19}	10^{18}	10^{21}
Underlying graph		Undirected	Undirected	Directed	Directed

Table 1: Search space properties of different games (adapted from Junghanns [JS00]; additional sources are [KT96, Kor97, EK98]). The effective branching factor is the number of children of a state, after applying memory-bounded pruning methods (in particular, not utilizing transposition tables; see Sec. 5.3.1 for the methods applied to Atomix). For Sokoban and Atomix, the numbers are for typical puzzles from the human-made test sets; for Sokoban, those problems are about 20×20 and, for Atomix, about 16×16 fields large.

for the other 7 tiles.

The database takes about half a gigabyte and can be reused for each problem instance. Korf and Felner have improved this technique to yield solutions within seconds [KF01].

Unfortunately, these powerful techniques cannot be directly applied to Atomix, since removing atoms from a state does not necessarily make it easier to solve; in fact, it can even become unsolvable, as is illustrated in Fig. 4.

3.2 Sokoban

Sokoban is a computer solitaire game that was invented in Japan in 1982. It shows some similarities to Atomix: it is played on a grid where identical "stones" have to be pushed to storage positions (see Fig. 5).

The main difference is that, in Sokoban, the player is explicitly represented on the board and occupies one square. He can enter adjacent empty fields or push adjacent stones away from him, thereby entering their previous position. The stone will only move a single field. If there's another stone or a wall behind it, it can't be moved at all. Usually, the objective is to minimize the number of stone pushes; minimizing the number of player moves is considered to be harder since it is not as easy to find a good lower bound.

What makes Sokoban somewhat more difficult than Atomix is the frequent occurrence of dead-
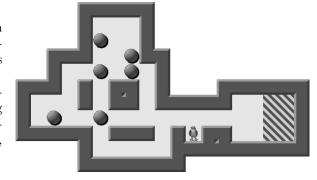

Figure 5: The first one in the collection of "classic" Sokoban problems. The striped area contains the six storage positions.

locks, i.e., states from which no solution can be found. For many human-designed problems, the first few moves have to be done very carefully to keep the puzzle solvable. This makes it hard even to find non-optimal solutions. While for Atomix deadlocks are possible, too, they don't usually occur in human-made problems and would likely be easier to detect.

In addition to that, typical Sokoban solutions are quite long (100–600 moves), and the branching factor can be even larger than for Atomix. Common with Atomix is the difficulty in isolating subgoals.

The best known admissible heuristic for Sokoban needs to perform *minimum cost perfect matching* to

4

assign stones to goal positions optimally [JS00]. It takes $O(\#\text{stones}^2)$ time to calculate per state, even when reusing information from the parent state. Therefore, much less states can be explored; while for Atomix 1,000,000 states per second can be generated, this number is for Sokoban around 10,000.

Sokoban has been shown to be PSPACE-complete by Culberson [Cul98]. Junghanns has analyzed it thoroughly and written the sophisticated solver program *Rolling Stone* [Jun99].

4 Complexity of Atomix

4.1 Complexity of Sliding-Block Puzzles

The time complexity of sliding block puzzles was the subject of intense research in the past. Though seemingly trivial, most variations are at least NP-hard and, some, even PSPACE-complete. Table 2 shows some results. The table was basically taken from Demaine et al. [DDO00], extended by the category of games where the blocks are pushed by an external agent not represented on the board, into which Atomix falls. The columns mean:

1. Are the moves performed by a robot on the board, or by an outside agent?

2. Can the robot pull as well as push?

3. Does each block occupy a unit square, or may there be larger blocks?

4. Are there fixed blocks, or are all blocks movable?

5. How many blocks can be pushed at a time?

6. Does it suffice to move the robot/a special block to a certain target location, instead of pushing *all* blocks into their goal locations?

7. Will the blocks "keep sliding" when pushed until they hit an obstacle?

8. The dimension of the puzzle: is it 2D or 3D?

4.2 A Formal Definition of Atomix

We will now give a formal definition of an Atomix problem instance (*level*).

Definition 1. *An Atomix problem instance consists of:*

- *A finite set A of so-called* atom types.

- *A game* board $B = \{0, \ldots, w-1\} \times \{0, \ldots, h-1\}$.

- *A bit matrix $O = (O[p] \in \{0,1\} \mid p \in B)$ of size $w \times h$ (the* obstacles*). A* position *is simply a tuple $p = (p_x, p_y) \in B$. A* state s *is defined as a subset of $A \times B$. An element of s is also called an* atom. *Note that the same atom type might appear several times in a state.*

  A position $p = (p_x, p_y)$ is said to be empty *for a state s if $O[p] = 0$ and there is no $a \in A$ with $(a, (p_x, p_y)) \in s$.*

  Positions outside of B are assumed not to be empty.

- *A state S (the* start state*), which satisfies that, for all $(a,p) \in S$, $O[p] = 0$.*

- *A state G (the* goal state*). For the problem to be solvable, for all $(a,p) \in G$, $O[p] = 0$ and there must be a bijection between S and G where each atom in S maps onto an atom in G with the same atom type.*

A direction (d_x, d_y) *is a tuple of x and y offsets, i. e., one of $(0,-1)$, $(1,0)$, $(0,1)$ and $(-1,0)$. A* move *is a tuple of a position p and a direction d. For a state s, a move (p,d) is only legal if there is an atom (a,p) in s, and (p_x+d_x, p_y+d_y) is empty.*

Applying a move (p,d) to a state s will yield another state s' in which every atom has the same position, except the atom (a,p): it will be replaced by (a,p') with $p' = (p_x + \delta d_x, p_y + \delta d_y)$, where $(p_x + \delta' d_x, p_y + \delta' d_y)$ is empty for all $0 < \delta' \le \delta$, and $(p_x + (\delta+1)d_x, p_y + (\delta+1)d_y)$ is not empty.

A solution *is a sequence of moves which, incrementally applied to the start state, yields the goal state.*

The main difference between this formal definition and the informal introduction is that the goal positions of the atoms are given explicitly. The reason is that this makes the puzzle both easier to analyze and to implement. Since the number of goal positions is linear in the board size, this difference does not affect the time complexity significantly.

Game	1. Robot	2. Pull	3. Blocks	4. Fixed	5. #	6. Path	7. Slide	8. Dim.	9. Complexity
	+	+	L	+	k	+	−	2D	NP-hard [Wil88]
	+	−	unit	+	k	+	−	2D	NP-hard [DO92]
PushPush3D	+	−	unit	−	1	+	+	3D	NP-hard [OT99]
PushPush	+	−	unit	−	1	+	+	2D	NP-hard [DDO00]
	+	−	unit	−	k	+	−	2D	open [DO92]
	+	−	unit	−	1	−	+	2D	NP-hard [OT99]
Push-∗	+	−	unit	−	k	−	−	2D	NP-hard [Hof00]
Sokoban+	+	−	1×2	+	2	−	−	2D	PSPACE-compl. [DZ99]
Sokoban	+	−	unit	∗	1	−	−	2D	PSPACE-compl. [Cul98]
Sokoban	+	−	unit	+	1	−	−	2D	PSPACE-compl. [Cul98]
15-Puzzle	−		unit	−	1	−	−	2D	NP-hard [RW90]
Rush Hour	−		1×{2,3}	−	1	+	−	2D	PSPACE-compl. [FB02]
Atomix	−		unit	+	1	−	+	2D	PSPACE-compl. [HS01]

Table 2: Time complexity of some sliding-block puzzles.

Our implementation handles different possible goal positions by imposing a move limit and trying all possible goal positions with that limit, and repeating with an incremented move limit until a solution is found.[1]

There are also some conventions for "proper" level design which have been dropped for simplicity: For a good level, the goal constellation should be a chemically valid molecule, and this molecule should be the only possibility to satisfy the bindings of the atoms. The formal definition doesn't have these restrictions; it doesn't even require the goal positions to be adjacent.

4.3 The Hardness of Atomix

Proposition 1. *Atomix on an $n \times n$ board is NP-hard.*

Proof. Any $(n^2 - 1)$-puzzle instance can be transformed into an Atomix instance by replacing the numbered tiles with atoms of unique atom types, as illustrated in Fig. 6 for the special case of the 15-puzzle.

For the $(n^2 - 1)$-puzzle, a legal move consists of sliding a tile into the empty space. In the reduction, those are also the only legal moves, since all atoms
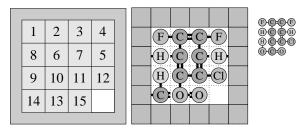

Figure 6: A 15-puzzle instance as Atomix level. It can be solved by our Atomix solver optimally within 34 moves.

not adjacent to the empty square cannot satisfy the move legality condition, and those adjacent to the empty square can only take its place as a move.

As shown by Ratner and Warmuth, the $(n^2 - 1)$-puzzle is NP-complete [RW86, RW90], so Atomix is NP-hard. □

Proposition 2. *Atomix on an $n \times n$ board is in PSPACE.*

Proof. A nondeterministic Turing-machine can solve Atomix by repeatedly applying a legal move from the start state encoded on its tape until a goal is reached. The number of possible Atomix states is limited by $n^2!$; hence, the machine can announce that the puzzle is unsolvable after having applied more moves without finding a solution. Since an

---

[1] As explained later, this incremental approach is already inherent to IDA*, and can be applied to A* with reasonable overhead.

encoding of an Atomix state needs only polynomial space, it follows that Atomix is in NPSPACE, and, by virtue of Savitch's theorem [Sav70], also in PSPACE. □

If one could prove that optimal solution lengths for Atomix are bounded by a polynomial of the problem size, it would follow with the same reasoning that Atomix is in NP.

Considering that Sokoban, which shows some similarities to Atomix, is PSPACE-complete, it doesn't seem unlikely that Atomix falls into this class, too. This would imply the existence of superpolynomially long optimal solutions (unless NP = PSPACE).

After submission of this paper, Holzer and Schwoon [HS01] showed by reduction from *non-empty intersection of finite automata* that Atomix is indeed PSPACE-complete. They also provided a level with an exponentially long optimal solution.

5 Searching the State Space of Atomix

Much progress has been made in the area of heuristic search. This is due to three components:

- faster machines with more memory,
- better heuristics, and
- better search methods.

Of these three, by far, the largest improvements come from better heuristics. The reason for this is, simplified, that a better heuristic has the potential of cutting away several layers of the search tree, which can result in dramatic savings due to the exponential nature of searching (see, for example, [JS00]). On the other hand, better search methods often improve running time just by a constant factor.

5.1 Heuristics for Atomix

As is often the case, a heuristic for Atomix can be devised by examining a model with relaxed restrictions. We drop the condition that an atom slides as far as possible: it may stop at any closer posi-

tion. These moves are called *generalized moves*. [2] In order to obtain an easily computable heuristic, we also allow that an atom may also pass through other atoms or share a place with another atom. The goal distance in this model can be summed up for all atoms to yield an admissible heuristic for the original problem.

The following properties are immediate consequences of the definition.

Property 1. *The heuristic is admissible.*

Proof. Since any Atomix move is also a legal generalized move, every solution with Atomix moves is also a solution in the relaxed model, and, therefore, can't be shorter than the shortest solution of the relaxed model. □

Property 2. *The h-values of child states can only differ from that of the parent state by 0, +1 or −1.*

Proof. The absolute value of the difference cannot be larger than 1, since a single generalized move can be used to transform the parent into the child or the child into the parent. Examples for differences of 0, +1 or −1 can be found easily. □

Property 3. *The heuristic is* monotone (consistent), *i. e., the f-value of a child state cannot be lower than the f-value of the parent state.*

Proof. Follows immediately from the previous property. □

An important property for implementations is that this heuristic can be calculated very efficiently. A table of distances from all board positions to any goal position can be precalculated with breadth-first search. For n different atoms, calculating h then takes n table lookups. [3] By only recalculating the goal distance of the moved atom, it can even be calculated in constant time, which makes a noticeable difference to Sokoban, where a good heuristic takes $O(n^2)$ time [JS00].

Apart from this somewhat obvious heuristic, it proved to be pretty hard to make any improvements. Two ideas were considered, but not implemented due to their limited applicability:

---

[2] The variant of Atomix which uses generalized moves has an undirected search graph. Atomix with generalized moves on an $n \times n$ board is also NP-hard but is in PSPACE.

[3] This is more complicated if there are atoms with identical atom type; see Sec. 6.1.
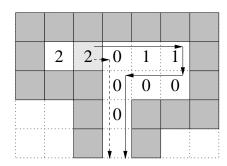
Figure 7: An example for the "cave"-heuristic: if only one atom is in the cave, the number denoted on its square can be added to the heuristic estimate. For example, an atom on the light grey square has to take the path marked with a solid line, instead of the optimal path of generalized moves marked with a dashed line, which is two moves shorter.

- If an atom needs a "stopper" at a certain position to make a turn for each optimal path, but no optimal path of any atom has an intermediate position at the stopper position, h can be incremented by one.

  Unfortunately, gaining more than one move from this "stopper" heuristic isn't easy, since it could happen that by leaving its optimal path one atom can act as stopper for several other atoms on their optimal paths.

- If an atom is alone in a "cave", for some positions, one or two moves can be added to the heuristic (see Fig. 7 for an example). A "cave" is an area that contains no goal position and has only one entry; if an atom is alone in there, it cannot use any stoppers unless another atom leaves its optimal path. This heuristic has a greater potential, since it can be added up admissibly for each cave.

  Unfortunately, only a few levels from our test set contain caves which could yield improved heuristics.

5.2 A*

A* is one of the oldest heuristic search algorithms [HNR68]. It is very time-efficient, but needs an exponential amount of memory. See Fig. 8 for some pseudo-code.

```
list<Move> aStar(State start) {
  // sorting criterion is f
  priority_queue<State> open;
  set<State> states;
  open.push(start);

  while (not open.empty()) {
    State best = open.pop();
    list<Move> moves = best.expand();
    forall (move in moves) {
      State child = best.apply(move);
      if (child.isGoal())
        return solution;
      State cached = states.find(child);
      if (cached == None
          or cached.g > child.g) {
        states.insert(child);
        open.push(child);
      }
    }
  }
}
```

Figure 8: Pseudo-code for A*

A* remembers all states ever encountered in a set states, which is the reason for its exponential space complexity. A priority queue open holds all states that have not yet been expanded. It is sorted by the f-value of the states. Nodes are popped from the queue and expanded afterwards. The children are inserted into the queue or discarded if they were already encountered. Sometimes, the same state is reached with a lower g-value; in that case, its entry in the state table has to be updated and it will be re-inserted into the queue. With an admissible heuristic, A* will always find an optimal solution.

Many implementations of these abstract data structures have been suggested. The state table is usually implemented as a hash table for fast access and low memory overhead. The priority queue can be implemented with a bucket for each f-value, containing all open states with that f-value. In Sect. 6.2, we present an alternative implementation that only needs the state table and does without a priority queue.

```
list<Move> dfs(int bound, State state) {
  if (state.isGoal())
    return solution;
  if (state.g + state.h() > bound)
    return;

  forall (move in state.moves()) {
    State child = state;
    child.apply(move);
    if (idaStar(bound, child))
      return solution;
  }
}

list<Move> idaStar(State start) {
  int bound = 0;
  loop {
    if (dfs(bound, start))
      return solution;
    bound = bound + 1;
  }
}
```

Figure 9: Pseudo-code for IDA*

5.3   IDA*

Iterative Deepening A* (IDA*) was the first algorithm that allowed finding optimal solutions to the 15-puzzle [Kor85]. Figure 9 shows some pseudo-code.

IDA* performs a series of depth-first searches, with an increasing move limit. The heuristic is used to prune subtrees where it is known that the bound will be exceeded, since the f-value is larger than the bound. Each iteration will visit all nodes encountered in the previous iteration again; but, since the majority of nodes will be generated in the last iteration, this does not affect the time complexity.

IDA* uses no memory except for the stack, so its memory use is linear in the search depth. Also, since it needs no intricate data structures, it can be implemented very efficiently. But of course, this comes at a price: IDA* does not detect *transpositions* in the search graph. If a state is encountered that has already been expanded and dismissed, it will be expanded again, possibly resulting in the re-evaluation of a huge subtree.

There are two approaches to mitigate this weak-ness:

- use of problem specific knowledge and
- use of memory.

5.3.1   Pruning the Search Space.

Several techniques are known for pruning the search tree:

- *Predecessor Elimination*, which disallows to take back moves immediately. For games with undirected underlying graphs like the 15-puzzle, this is an obvious optimization. For Atomix, it can still be applied, since pushing an atom into the opposite direction immediately after a move always yields the same state as pushing it in that direction in the first place.

- *Finite-State Machines* [TK93, Ede97]. For the 15-puzzle, many move sequences lead to identical states. With a breadth-first search, such move sequences can be learned and encoded in an FSM, which can then be applied with very low overhead to the depth-first search. This technique cannot be applied to Atomix directly, since the result of a move depends on the position of the other atoms, and thus a transposition cannot be detected by just looking at a sequence of moves.

- *Move Pruning.* When examining a solution move sequence for an Atomix level, one notices that many, though not all moves could be interchanged. Interchanging moves is not possible in four cases, as is explained in Fig. 10.

  The idea is to check if a generated move is independent of the previous move (i. e., applying them in reversed order would yield the same state) and, if they are independent, to impose an arbitrary order (the atom with the lower number must move first). This scheme has proven to be very efficient in avoiding transpositions, reducing running time by several orders of magnitudes.

5.3.2   Move Ordering.

Reinefeld and Marsland [RM94] proposed expanding more "promising" states first in an IDA* search. This can reduce the number of states expanded in
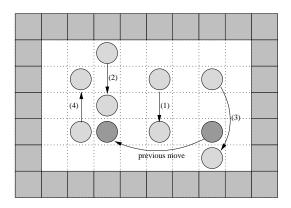
9

Figure 10: There are four cases where two moves are dependent, i.e., their order cannot be interchanged: (1) The current atom would have stopped the previously moved atom earlier. (2) The current atom uses the previously moved atom as a stopper. (3) The current atom would stop earlier if the previously moved atom had not been moved. (4) The current atom was the stopper of the previously moved atom.
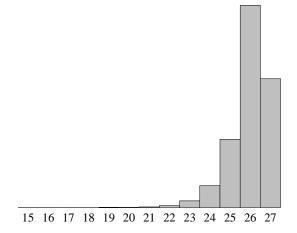

Figure 11: Number of states generated for increasing IDA* search depths of the level "katomic_06" while utilizing move ordering. In total, 553,198,798 states were generated.

the final iteration considerably, since the search will be aborted as soon as a solution is found.

It offers no savings at all for previous iterations; exactly the same states will be examined. Figure 11 illustrates this: The solution in 27 moves is found so early that even less states were generated that in the iteration with maximal 26 moves.

It is an obvious choice to use f as a measure of "hopefulness" for move ordering, but other criteria are possible: For Sokoban, it has been observed that solutions often contain long sequences of pushes of the same stone. Junghanns successfully implemented a move ordering scheme that first expands all moves that push the same stone as the previous move (the *inertia* moves) [JS00].

For Atomix, move ordering that simply orders children by their f-values proved to be very efficient. It is also easy to implement with our heuristic, since as shown in Sec. 5.1, there are only 3 different f-values of child states possible, so the successors can simply be sorted into 3 buckets, which can then be successively expanded.

5.4 Partial IDA*

Analogously to the two-player game search, a *transposition table* can be used to avoid re-expanding states [RM94]. States are inserted into a hash table together with their g-value as they are generated. Then, for each newly generated state, it is looked up whether it has already been expanded with the same or a lower g-value so it can be pruned. If memory was unlimited, this would avoid all possible transpositions. Many schemes have been proposed for proper management of the transposition table with limited memory [ES95]; our implementation simply refuses to insert states into an exhausted table.

A lot of memory can be saved with *Partial IDA** [ELL01, EM01]. This idea originates in the field of *protocol verification*, where the objective is to generate all reachable states and check if they fulfill a certain criterion. A hash table is used to avoid re-expanding states. Just as for a single-agent search, memory is the limiting resource. Therefore, Holzmann suggested *bitstate hashing* [Hol87]: instead of storing the complete state, only a single bit corresponding to the hash value is set, indicating that this state has been visited before. Because of the possibility of hash collisions, states might get pruned erroneously, so this method can give false positives. When applied to IDA*, states on opti-

mal paths could get pruned, so the method looses admissibility, but is still useful to determine upper bounds and likely lower bounds.

For Atomix, initial experiments with Partial IDA* rarely found optimal solutions. The reason is that just knowing a state has been encountered before is not sufficient, because if we encounter it with a lower g-value than previously, it needs to be expanded again. To achieve this, we include g into the hash value and look up with g and $g - 1$. This means transpositions with better g will not be found in the table and expanded, as desired. Transpositions with g worse by 2 or more will also not be detected; experiments showed that they are rare and the resulting subtrees are shallow, though.

By probing twice (with g and $g - 1$), we increase the likelihood of hash collisions. For example, if we declare the table to be full if every 8th bit is set, we have an effective memory usage of 1 byte per state, and a collision probability of $1 - \left(\frac{7}{8}\right)^2 = 23\%$. To improve the collision resistance, one can calculate a second hash value and always set and check two bits, effectively doubling memory usage but lowering collision probability to $1 - \left(\frac{63}{64}\right)^2 = 3\%$.

A related scheme with better memory efficiency and collision resistance is *hash compaction* [WL93]. It utilizes a hash table where, instead of the complete state, only a hash signature is saved. In our implementation, we use 1 byte for the signature, and probe for g and $g - 1$. This way, we have a collision probability of $1 - \left(\frac{255}{256}\right)^2 = 0.8\%$, so even if there is only a single possible solution of length 30, the probability of finding it is $\left(\left(\frac{255}{256}\right)^2\right)^{30} = 79\%$; and in fact, all 47 solutions found this way were optimal.

Different policies are possible in the case of a hash collision detected by differing signatures. Usual hash table techniques like chaining or open addressing can be applied. We tried a much simpler scheme: the old entry gets overwritten. This can be seen as a special case of the *t-limited scheme* proposed by Stern and Dill [SD96] with $t = 1$. One disadvantage of this scheme is that entries will already get overwritten before the table is completely full. Since for the "interesting" (difficult) cases, the state table will fill up soon anyway, this effect is limited.

5.5   Backward Search

Many puzzles are *symmetric*, i.e., the set of children of a state equals the set of possible parents. This is equivalent to the state space graph being undirected. As already mentioned, this is the case for the 15-puzzle, but not for Sokoban or Atomix. For Atomix, it is simple to find all potential parent states, though: they can be found by applying all legal *backward moves*. In a backward move, an atom being pushed may stop moving at any position, but it can only be pushed in a direction if it is adjacent to an obstacle in the *opposite* direction.

Formally defined, a backward move is a triple of a position p, a direction d, and a distance δ. It is legal for a state s if there is an atom (a, p) in s, and $(p_x - d_x, p_y - d_y)$ is *not* empty, and $(p_x + \delta' d_x, p_y + \delta' d_y)$ is empty for all $0 < \delta' \le \delta$. Applying a backward move is analogous to applying a forward move.

It is clear that a sequence of backward moves that transforms the goal state into the start state can be easily converted into a solution for the normal puzzle.

This "backward Atomix" could be clad into the following story: In a space station, robots have held a gathering and now need to get back to their places. Since there is no gravity, wheels would be useless; they can only push themselves off the wall with a kicker. By adjusting the kicking power, they can choose the place where they stop due to friction. Also, they have a device to fix themselves to the floor, so a robot can use another robot to push itself off. Some robots are of equal models and can be arbitrarily assigned to their positions.

To be equivalent to the informal Atomix, where the goal position is not determined, the player would first have to choose the place for the gathering. It would be interesting to see whether human players would consider this puzzle to be easier or harder than "forward Atomix".

Expanding states for backward Atomix is about as easy as for forward Atomix, and the same heuristic can be used, since the generalized moves from Sect. 5.1 comprise backward moves. Hence, the crucial point is the branching factor. At first glance, it seems to be much larger; atoms may stop at any position, so the branching factor is not limited by $4 \cdot \#\text{states}$ like for forward search. However, if an atom stops "in the open", it has no further move option at all, while in forward search an atom usu-
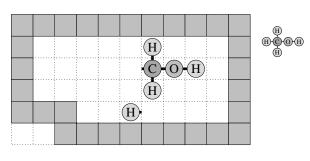
Figure 13: Though only one general move away from the goal, this level takes 14 moves.

ally has at least one move option. Actually, it turns out that both branching factors are identical:

Lemma 1. *The sum of possible forward moves and the sum of possible backward moves of all states of a level are identical and, therefore, the average number of children for backwards expansion is exactly the same as for forward expansion.*

Proof. We first show the equality for a single atom by structural induction. On a board with no empty squares, the equation is trivially true. We show it also remains true when removing an obstacle. The change in the number of moves depends on the pattern of empty squares around the obstacle being removed; we examine all possible patterns (up to symmetry, and omitting the trivial case of 4 obstacles), as illustrated in Fig. 12, with a, b, c and d being the number of empty squares in each direction.

(a) 3 adjacent obstacles: $1 - b + b + 1 = 1 + 1 = 2$.

(b) 2 adjacent obstacles, where the obstacles are diagonally adjacent:
$1 - b + b + d + 2 - d + 1 = 1 + 2 + 1 = 4$.

(c) 2 adjacent obstacles, where the obstacles are opposite:
$c + 2 - b + 0 - c + b + 2 = 1 + 2 + 1 = 4$.

(d) 1 adjacent obstacle: $c + 2 - b + d + 1 - c + b + 2 - d + 1 = 1 + 3 + 1 + 1 = 6$.

(e) no adjacent obstacles:
$d + 2 - a + c + 2 - b + 0 - c + b + 2 - d + a + 2 = 1 + 1 + 4 + 1 + 1 = 8$.

Now, let us consider the contribution of one atom to the possible moves. Each possible distribution of the other atoms can be considered as a pattern of obstacles. With the observation just made, the sum of possible forward and backward moves is the same when summing up over all possible positions of the considered atom; so the sum over all possible distributions of the other atoms is also identical and, since this equality holds for each atom, the lemma is true.

□

In practice, the branching factors can differ substantially, since the generated states are not random; the move operators make certain states more likely than others, and states close to the goal where (by convention) all atoms are close together are much more likely. In our experiments, we observed differences up to 30% in forward and backward branching factors.

5.6 Bidirectional Search

The heuristic is often especially bad for states close to the goal; Figure 13 shows an example where the heuristic estimates 1 move left, but it actually takes at least 14 moves. The idea is to generate a hash table of all states that are at most at distance, say, 6 from the goal by expanding the goal state with reverse move operators. When now the heuristic says "at least 3 moves left", one can look up the exact value in the hash table. If it is not at all in the hash table, it is at least 7 moves away from the goal.

In practice, this scheme failed miserably. It reduced the number of expanded states somewhat, but not enough to even find one new lower bound for any of the about 50 test levels. The reason might be that subtrees close to the goal would get pruned soon anyway due to the maximum move bound (see Sec. 6.2).

6 Implementation

6.1 Identical Atoms

The presence of undistinguishable atoms (i.e., atoms with identical atom types) poses problems for an implementation:

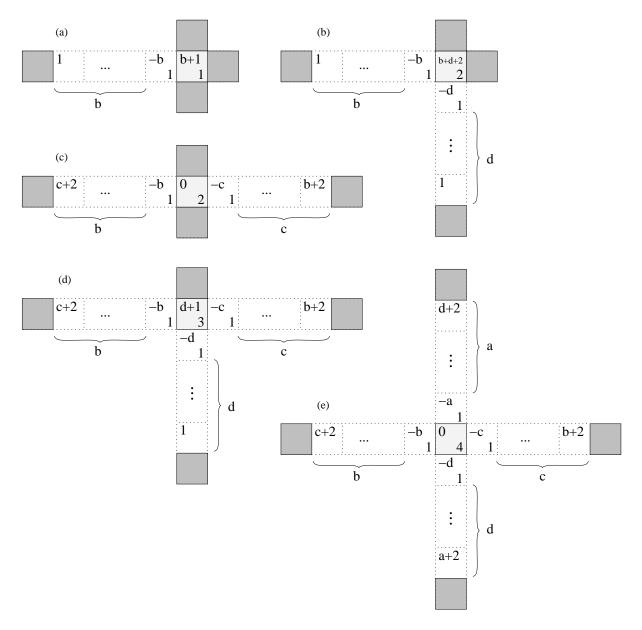Figure 12: The light grey obstacle in the center is being removed. The upper left corner of each square denotes the number of backward moves that are lost or gained by this change for an atom on this square. The lower right corner denotes the number of new forward moves. Squares which are skipped in the sketches (denoted by dots) have zero gain with respect to both forward and backward moves.

13

- The heuristic cannot simply perform a table lookup to find a lower bound for an atom, since it is not clear which atom should go to which goal position. To find a good lower bound, a *minimum cost perfect matching* has to be done for each set of identical atoms to find the cheapest assignment of atoms to goal positions. This is the same problem as for Sokoban, where all stones are identical. Minimum cost perfect matching for a bipartite graph can be solved using minimum cost augmentation in time quadratic in the number of identical atoms [Kuh55].

  Since most of our benchmark problems had only unique atoms or pairs of atoms, we made special cases for those and implemented the matching by examining all possible permutations, which is only feasible for up to 5 or 6 identical atoms.

- States are represented as arrays of positions; the atom type corresponding to a position is not represented explicitly, but determined by its index in the array. Thus, logically identical states have different representations. This is bad because we don't want to waste space with having the same state several times in the hash table. We avoid this by converting a state to a canonical form prior to inserting or lookup: the positions have to appear sorted according to an arbitrary order.

6.2 A*

As sketched in the pseudo code (Fig. 8), an implementation of A* needs the following operations:

- check if a state has been encountered before and with which g-value,

- find an open state with optimal f-value,

- mark an open state as closed, and

- update the g-value of a saved state to a lower value.

This is usually implemented with a hash table and a priority queue which stores all open states. We will show that if the heuristic is monotone, no priority queue is actually needed: an optimal open state can be found efficiently without any additional data structures. Our algorithm is easy to implement and time and space efficient.

Initially, the available memory is allocated for two tables: the *state table* and the *hash table* (see Fig. 14). As states are generated, they are appended to the end of the state table; states never get deleted. The states are tagged with an *open-bit* and with the g-value. The hash table stores a pointer into the state table at the position corresponding to the hash value of the state; this allows a quick lookup of states. A linear displacement scheme is used to resolve hash collisions (in the figure, this happened for the hash value 273).

The monotonicity of the heuristic implies that f_{opt}, the currently optimal f-value of an open state, is also monotone over the run of A*. To find an optimal open state, a linear search on the state table is performed until an open state with $f = f_{\mathrm{opt}}$ is found. At first glance, this seems to be very slow, since finding an optimal open state now could take up to O(#states), whereas a normal priority queue guarantees O(log #states) or even O(1) access. The following proposition shows that this can be done efficiently:

Proposition 3. *In A* with a monotone heuristic with a hash table and no additional data structure, a state with optimal f-value can be found in amortized time O(branching factor).*

Proof. To achieve this, we need to ensure that, for each f_{opt}-value, when we reach the end of the state table, we have expanded all states with $f = f_{\mathrm{opt}}$, so we don't have to go through the table again. This can be ensured by not upgrading a state in place if it is re-encountered with lower g, but to append it at the end like new states. [4] States with $f < f_{\mathrm{opt}}$ will never be reopenened [ES00], so this suffices to ensure the desired property.

Two kinds of states will be skipped because their f-value differs from f_{opt}:

- Closed states with $f < f_{\mathrm{opt}}$. We keep a pointer to the very first open state, so only closed states with $f = f_{\mathrm{opt}} - 1$ or $f = f_{\mathrm{opt}} - 2$ have

---

[4]Our implementation actually doesn't do this, but instead goes through the table repeatedly till no open states with $f = f_{\mathrm{opt}}$ are left. Since re-encountering open states with lower g happens pretty seldom, usually just one or two additional passes per f_{opt}-value are needed.
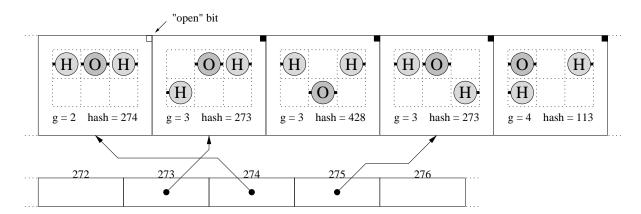
Figure 14: Hash structure used in our A*-implementation.

to be skipped; for any branching factor greater than 1, this can be at most twice as many as states with $f = f_{opt}$ and, with a higher branching factor, their number even becomes negligible.

- Open states with $f > f_{opt}$. They must have been generated by states with $f = f_{opt}$ or $f = f_{opt} - 1$, so their number is linear in the number of states with $f = f_{opt}$ and the branching factor. □

Our implementation with this scheme is several times faster than a naïve implementation using the C++ STL `priority_queue` and `set`, which are based on heaps, resp., binary trees, with a memory overhead of about 30 bytes per state. On a Pentium III with 500 MHz, it can generate around a million states per second.

Our scheme would be especially advantageous with problems where a state can be represented with few bits, like the 15-puzzle. Here, the low memory overhead can make a big difference.

A disadvantage of this scheme is that it is not possible to further discriminate among optimal states. A common idea to speed up A* is to sort among states with equal f-values those closer to the top that are further advanced.

To trade time for memory, the A* implementation works iteratively: similarly to IDA*, an artificial upper bound on the number of moves is applied and, if the f-value of a generated state exceeds this bound, it is pruned. If then the search fails, it is restarted with the bound increased by one. This also allows us to take multiple goal positions into account. Due to the exponential behavior, this slows down the search only by a constant factor.

7 Future Investigations

7.1 Atomix

There remain a lot of open questions about Atomix. For the theoretical part:

- Can some well-known puzzles other than the $(n^2 - 1)$-puzzle, like perhaps Sokoban, be reduced to Atomix? This might give further insight on their complexity, maybe even a proof of PSPACE-completeness.

For practical implementations, the most promising area is better heuristics. Some ideas are mentioned in Sec. 5, but finding good heuristics is an art, and so totally different approaches might be possible.

We sketch further areas of research.

OBDDs [ER98] Ordered Binary Decision Diagrams (OBDDs) provide a means to efficiently encode a set of states in a trie-like structure.

Stochastic Node Caching [MI98] Some states in the transposition table are more "valuable", because they are encountered more often in the state space search, for example if they are close

15

to the start state. By inserting states not always, but only with a fixed probability, one can increase the likeliness that only "valuable" states make it into the hash table. The difficulty lies in tuning the probability to fit the state space size; if the probability is chosen low and the state space is actually small, so that it would fit completely into the memory, a lot of states are expanded needlessly. Perhaps this scheme can also be combined with Partial IDA*.

External memory Edelkamp and Schrödl [ES00] manage to make A* more external memory friendly by grouping "neighbored" states together in memory. Since the search domain was a real 2D-map, it was easy to find an a priori criterion for this; for Atomix, this would be considerably harder.

A* One could try to save memory in the A* hash table by not storing the state itself, but regenerate it on each access from the predecessor pointers. Perhaps, also the move pruning methods from Sec. 5.3 can be applied to A*.

Bidirectional Search More elaborated bidirectional search schemes could prove to be worthwhile [KK97]. A recent option to reduce the memory overhead of A* was proposed by Korf [Kor99]; his method does without a *closed*-list.

Nonoptimal solutions. This field is nearly unexplored currently. It should be easier than for Sokoban to find *any* solution within reasonable time. We found several nonoptimal solutions with WIDA*: in an IDA* search, the h-value is scaled by a constant factor like 1.5. This favors further advanced states, but usually prunes states which could still lead to an optimal solution.

7.2 Search Techniques

The most promising topic seems to be Partial IDA*, especially combined with hash compaction. Experiments with different problems, and better bounds on the error probability would be desirable.

8 Conclusions

Atomix proved itself to be a challenging puzzle; this is corroborated by the recent PSPACE-completeness proof. The classic algorithms A* and IDA* have been implemented and adapted to the problem domain; we have found optimal solutions for many problems from our benchmark set. Our A* implementation with a single data structure for the *open* and *closed* set can solve "smaller" puzzles very efficiently. With Partial IDA* based on hash compaction, we have presented a memory-bounded scheme that makes excellent use of the available memory and has low runtime overhead; improved bounds on the error probability would be useful, though. Further progress is likely to come from improved heuristics rather than from better search methods, since our current heuristic is rather uninformed. We have shown that while the search graph is directed, the backward branching factor does not differ from the forward branching factor; this makes Atomix an interesting testbed for bidirectional algorithms.

References

[CS96]   Joseph C. Culberson and Jonathan Schaeffer. Searching with pattern databases. In Gordon McCalla, editor, *Proceedings of the Eleventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence on Advances in Artificial Intelligence*, volume 1081 of *LNAI*, pages 402–416. Springer, Berlin, May 21–24 1996.

[CS98]   Joseph C. Culberson and Jonathan Schaeffer. Pattern databases. *Computational Intelligence*, 14(3):318–334, 1998.

[Cul98]   Joseph C. Culberson. Sokoban is PSPACE-complete. In Elena Lodi, Linda Pagli, and Nicola Santoro, editors, *Proceedings of the International Conference on Fun with Algorithms (FUN-98)*, pages 65–76. Carleton Scientific, Waterloo, Ontario, June 18–20 1998.

[DDO00]   Erik D. Demaine, Martin L. Demaine, and Joseph O'Rourke. PushPush and Push-1 are NP-hard in 2D. In *Proceedings of the 12th Canadian Conference on Computational Geometry*, pages 211–219, Fredericton, New Brunswick, Canada, August16–18 2000.

[DO92]   A. Dhagat and Joseph O'Rourke. Motion planning amidst movable square blocks. In *Proceedings of the 4th Canadian Conference on Computational Geometry*, pages 188–191, 1992.

[DZ99]   Dorit Dor and Uri Zwick. SOKOBAN and other motion planning problems. *CGTA: Computational Geometry: Theory and Applications*, 13(4):215–228, oct 1999.

[Ede97]   Stefan Edelkamp. Suffix tree automata in state space search. In Gerhard Brewka, Christopher Habel, and Bernhard Nebel, editors, *Proceedings of the 21st Annual German Conference on Artificial Intelligence (KI-97): Advances in Artificial Intelligence*, volume 1303 of *LNAI*, pages 381–384. Springer, Berlin, September 9–12 1997.

[EK98]   Stefan Edelkamp and Richard E. Korf. The branching factor of regular search spaces. In *Proceedings of the 15th National Conference on Artificial Intelligence and the 10th Conference on Innovative Applications of Artificial Intelligence (AAAI-98/IAAI-98)*, pages 299–304. AAAI Press, Menlo Park, CA, USA, July 26–30 1998.

[ELL01]   Stefan Edelkamp, A. L. Lafuente, and S. Leue. Protocol verification with heuristic search. In *AAAI-Spring Symposium on Model-based Validation of Intelligence*, pages 75–83, 2001.

[EM01]   Stefan Edelkamp and Ulrich Meyer. Theory and practice of time-space trade-offs in memory limited search. In *German Conference on Artificial Intelligence (KI-2001)*, Lecture Notes in Computer Science, pages 169–184. Springer, 2001.

[ER98]   Stefan Edelkamp and Frank Reffel. OBDDs in heuristic search. In Otthein Herzog and Andreas Günter, editors, *Proceedings of the 22nd Annual German Conference on Advances in Artificial Intelligence (KI-98)*, volume 1504 of *LNAI*, pages 81–92. Springer, Berlin, September 15–17 1998.

[ES95]   Jürgen Eckerle and Sven Schuierer. Efficient memory-limited graph search. *Lecture Notes in Computer Science*, 981:101–112, 1995.

[ES00]   Stefan Edelkamp and Stefan Schrödl. Localizing A*. In *Proceedings of the 17th National Conference on Artificial Intelligence and the 12th Conference on Innovative Applications of Artificial Intelligence (AAAI-00/IAAI-00)*, pages 885–890. AAAI Press, Menlo Park, CA, USA, July 30– 3 2000.

[FB02]   Gary W. Flake and Eric B. Baum. Rush Hour is PSPACE-complete, or "Why you should generously tip parking lot attendants". *Theoretical Computer Science*, 270:895–911, January 2002.

[HMY92] Othar Hansson, A. E. Mayer, and Moti Yung. Criticizing solutions to relaxed models yields powerful admissible heuristics. *Information Sciences*, 63(3):207–227, September 1992.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, SSC-4(2):100–107, 1968.

[Hof00]   M. Hoffmann. Push-∗ is NP-hard. In *Proceedings of the 12th Canadian Conference on Computational Geometry*, pages 205–209, Fredericton, New Brunswick, Canada, August 2000.

[Hol87]   Gerard J. Holzmann. On limits and possibilities of automated protocol analysis. In H. Rudin and C. West, editors, *Proceedings of the 7th International Conference Protocol Specification, Testing, and Verification*, pages 339–346, Zürich, June 1987. North-Holland Publ. Co., Amsterdam.

[HS01]   Markus Holzer and Stefan Schwoon. Assembling molecules in Atomix is hard. Technical Report TUM-I0101, Institut für Informatik, Technische Universität München, May 2001.

[JS79]   William Woolsey Johnson and William E. Story. Notes on the 15-puzzle. *American Journal of Mathematics*, 2:397–404, 1879.

[JS00]   Andreas Junghanns and Jonathan Schaeffer. Sokoban: A case-study in the application of domain knowledge in general search enhancements to increase efficiency in single-agent search. *Artificial Intelligence, special issue on search*, 2000.

[Jun99]   Andreas Junghanns. *Pushing the Limits: New Developments in Single-Agent Search*. PhD thesis, University of Alberta, 1999.

[KF01]   Richard E. Korf and Ariel Felner. Disjoint pattern database heuristics. *Artificial Intelligence Journal*, 134:9–22, 2001.

[KK97]   Hermann Kaindl and Gerhard Kainz. Bidirectional heuristic search reconsidered. *Journal of Artificial Intelligence Research*, 7:283–317, 1997.

[Kor85]   Richard E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985. Reprinted in Chapter 6 of P.G. Raeth, editors, *Expert Systems, A Software Methodology for Modern Applications*, pages 380–389, 1990. IEEE Computer Society Press, Washington D.C..

[Kor97]   Richard E. Korf. Finding optimal solutions to Rubik's cube using pattern databases. In *Proceedings of the 14th National Conference on Artificial Intelligence and the 9th Conference on Innovative Applications of Artificial Intelligence (AAAI-97/IAAI-97)*, pages 700–705. AAAI Press, Menlo Park, CA, USA, July 27–31 1997.

[Kor99]   Richard E. Korf. Divide-and-conquer bidirectional search: First results. In *IJCAI*, pages 1184–1191, 1999.

[KT96]   Richard E. Korf and Larry A. Taylor. Finding optimal solutions to the Twenty-Four Puzzle. In *Proceedings of the 13th National Conference on Artificial Intelligence and the 8th Conference on Innovative Applications of Artificial Intelligence (AAAI-96/IAAI-96)*, pages 1202–1207. AAAI Press/MIT Press, Menlo Park, CA, USA, August 4–8 1996.

[Kuh55]   Harold W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1):83–98, 1955.

[MI98]   Teruhisa Miura and Toru Ishida. Stochastic node caching for memory-bounded search. In *Proceedings of the 15th National Conference on Artificial Intelligence and the 10th Conference on Innovative Applications of Artificial Intelligence (AAAI-98/IAAI-98)*, pages 450–456. AAAI Press, Menlo Park, CA, USA, July 26–30 1998.

[OT99]   Joseph O'Rourke and The Smith Problem Solving Group. PushPush is NP-hard in 3D. Technical Report 064, Smith College, Northampton, MA, November 1999.

[RM94]   Alexander Reinefeld and T. Anthony Marsland. Enhanced iterative-deepening search. *IEEE Transactions on Pattern Analyis and Machine Intelligence*, 16(7):701–710, July 1994.

[RW86]   Daniel Ratner and Manfred K. Warmuth. Finding a shortest solution for the $n \times n$ extension of the 15-puzzle is intractable. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86)*, volume 1, pages 168–172, Philadelphia, Pennsylvania, August 1986. Morgan Kaufmann.

[RW90]   Daniel Ratner and Manfred K. Warmuth. The (n^2-1)-puzzle and related relocation problems. *Journal of Symbolic Computation*, 10(2):111–137, August 1990.

[Sav70]   Walter J. Savitch. Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192, April 1970.

[SD96]   Ulrich Stern and David L. Dill. Combining state space caching and hash compaction. In Bernd Straube and Jens Schoenherr, editors, *Methoden des Entwurfs und der Verifikation digitaler Systeme, 4. GI/ITG/GME Workshop*, Berichte aus der Informatik, pages 81–90, Kreischa, March 1996. Shaker Verlag, Aachen.

[TK93]   Larry A. Taylor and Richard E. Korf. Pruning duplicate nodes in depth-first search. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 756–761. AAAI Press, Menlo Park, CA, USA, July 1993.

[Wil88]   Gordon T. Wilfong. Motion planning in the presence of movable obstacles. In *Proceedings of the Fourth Annual Symposium on Computational Geometry*, pages 279–288, Urbana-Champaign, IL, June 6–8 1988. ACM Press, New York.

[WL93]   Pierre Wolper and Dennis Leroy. Reliable hashing without collision detection. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV-93)*, volume 7, pages 59–70. Springer, Berlin, 1993.

A Source

The solver is written in ANSI C++ in a fairly low-level style. It does not need any libraries beside the C++ standard libraries; it has been tested on Alpha and ix86 systems and with gcc version 2.95, version 3.0 and the Compaq cxx compiler. The source can be found at `http://www-fs.informatik.uni-tuebingen.de/~hueffner`.

Levels are read in a format taken from katomic:

```
[Level]
Name=Water (Atomix 01)
atom_1=1-c
atom_2=3-cg
atom_3=1-g
feld_00=...............
feld_01=...............
feld_02=...............
feld_03=..#####........
feld_04=..#...#........
feld_05=..#..3######...
feld_06=..#..#.....#...
feld_07=..#.#......##..
feld_08=..#.#..#.##.#..
feld_09=..#....#.#2.#..
feld_10=..###1#..#..#..
feld_11=...#........#..
feld_12=...##########..
feld_13=...............
feld_14=...............
mole_0=123
```

The format is mostly self-explaining; the atom type and binding information (`3-cg`) is ignored by the solver.

The solver is built with several abstract data types, implemented as C++ classes. Each class is is split into two files: the declaration (extension `.hh`) and the implementation (extension `.cc`). Table 3 gives an overview of the source.

The size of the board and the number of Atoms is hardcoded in `Size.hh`. This allows to allocate memory statically, which improves speed and memory efficiency. For convenience, the script `run.sh` is included, which is called with a level file, and will adapt `Size.hh` and recompile. If the board size grows beyond 256 fields, some types would have to be adapted, since currently the type of a field number is a byte.

The maximum amount of memory to use is hardcoded in `parameters.hh` and needs to be adapted to the machine.

While the A* algorithm has no options, a lot of things can be tuned for IDA* in the header file `IDAStar.hh`, as shown in the following table.

Option	Effect
DO_BACKWARD_SEARCH	Use inverse move operator and search a path from the goal state to the start state.
DO_MOVE_PRUNING	Apply move pruning as explained in Sec. 5.3.1.
DO_CACHING	Use a hash table to detect transpositions.
DO_PREHEATING	Only meaningful with DO_CACHING. Run IDA* also with all limits up to the current limit -3. This is supposed to allow the "most useful" states to enter the cache, but potentially slows down the search and skews statistics.
DO_PARTIAL	Partial IDA* with bitstate hashing.
DO_COMPACTION	Partial IDA* with hash compaction.
DO_STOCHASTIC_CACHING	Stochastic state caching. A state will be inserted into the cache with probability CACHE_INSERT_PROBABILITY.

File	Function
AStar.hh/cc	Implementation of A* as described in Sec 6.2.
AStarState.hh	A state for A*. Saves a pointer to the parent state and the open bit. Also the heuristic estimate is cached to avoid recalculating.
Atom.hh/cc	Atom representation including bindings and element. Only used for I/O.
BitVector.hh	A simple bit vector, needed for bitstate hashing.
Board.hh/cc	A complete board representation. Only used for I/O.
CacheState.hh	Derived from State; solves the problem mentioned in Sec. 6.1 that the state representation is unique by sorting identical atoms according to their position.
Dir.hh	A direction (up, down, left or right).
HashTable.hh	A hash table with linear displacement used for the transposition table in IDA*.
IDAStar.hh/cc	The IDA* implementation. Includes transposition tables and partial search. Most enhancements are selectable at compile time via #defines in the header file.
IDAStarCacheState.hh	Like CacheState, but also remembers g and h.
IDAStarState.hh	The state representation used in the IDA* search. Since memory is not a concern, it can cache h and keep a matrix of field content for faster move generation and easier move dependency checking.
Level.hh/cc	Contains two Boards for the start and the goal state. Only relevant for I/O; reaching the goal state is detected by $h = 0$.
Move.hh	Representation of a generalized move.
Pos.hh	A position on the board; mainly used for I/O.
Problem.hh	A helper class that prepares some data structures for the search, e.g. distance tables for the heuristic.
Size.hh	Contains the hardcoded board size and atom number.
State.hh/cc	A basic state representation. A state is compactly encoded as an array of positions. Derived from by AStarState and IDAStarState which add additional functionality.

Table 3: Overview of some source files of the solver.

B Experimental Results

The experiments were performed on a Pentium III with 500 MHz, utilizing 128 MB of main memory and imposing a time limit of one hour. The source can be found at http://www-fs.informatik.uni-tuebingen.de/~hueffner.

Man	Best result found by participants of an online game
IDA*-tt	IDA* with transposition table
IDA*-r	IDA* backward search with transposition table
PIDA*	Partial IDA* with hash compaction to 1 byte

Level	Atoms	Goals	Man	A*	IDA*	IDA*-tt	IDA*-r	PIDA*
Atomix 01	3	17		= 13	= 13	= 13	= 13	= 13
Atomix 02	5	6		= 21	= 21	= 21	= 21	= 21
Atomix 03	6	4		= 16	= 16	= 16	= 16	= 16
Atomix 04	6	2		≥ 23	≥ 22	= 23	= 23	= 23
Atomix 05	9	2		≥ 34	≥ 34	≥ 35	≥ 35	≥ 37
Atomix 06	8	4		= 13	= 13	= 13	= 13	= 13
Atomix 07	9	1		≥ 25	≥ 26	= 27	≥ 25	= 27
Atomix 09	7	1		= 20	= 20	= 20	= 20	= 20
Atomix 10	10	2		≥ 28	≥ 28	≥ 28	≥ 27	≥ 30
Atomix 11	5	14		= 14	= 14	= 14	= 14	= 14
Atomix 12	9	4		= 14	= 14	= 14	= 14	= 14
Atomix 13	8	1		= 28	= 28	= 28	= 28	= 28
Atomix 15	12	1		≥ 35	≥ 36	≥ 37	≥ 37	≥ 37
Atomix 16	9	2		≥ 26	≥ 26	≥ 27	≥ 25	≥ 28
Atomix 18	8	4		= 13	= 13	= 13	= 13	= 13
Atomix 22	8	3		≥ 24	≥ 24	≥ 25	≥ 23	≥ 27
Atomix 23	4	20		= 10	= 10	= 10	= 10	= 10
Atomix 26	4	17		= 14	= 14	= 14	= 14	= 14
Atomix 28	10	1		≥ 28	≥ 29	≥ 29	≥ 26	≥ 29
Atomix 29	8	2		= 22	= 22	= 22	= 22	= 22
Atomix 30	8	4		= 13	= 13	= 13	= 13	= 13
Unitopia 01	3	41	11	= 11	= 11	= 11	= 11	= 11
Unitopia 02	4	5	22	= 22	= 22	= 22	= 22	= 22
Unitopia 03	5	12	16	= 16	= 16	= 16	= 16	= 16
Unitopia 04	6	5	20	= 20	= 20	= 20	= 20	= 20
Unitopia 05	6	7	21	= 20	= 20	= 20	= 20	= 20
Unitopia 06	9	2	33	≥ 29	≥ 30	≥ 30	≥ 30	≥ 31
Unitopia 07	10	1	36	≥ 33	≥ 33	≥ 34	≥ 32	≥ 35
Unitopia 08	7	4	25	= 23	= 23	= 23	= 23	= 23
Unitopia 10	8	2	41	≥ 36	≥ 36	≥ 37	≥ 38	≥ 40

Time performance. A* runs out of memory usually much before a runtime of one hour and, so, can establish less stringent bounds. The advantage of using a transposition table for IDA* outweighs its runtime overhead and yields better results in all cases. Reverse search performs similar to forward search, as founded by the theoretical findings. Partial IDA* consistently beats IDA* with conventional hash tables because of better memory utilization and less runtime overhead. Note that most of these differences are expected to be more significant if the time limit is increased.