UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ALEX ZOCH GLIESCH

Solving Atomix Exactly

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Marcus Ritt

Porto Alegre
December 2015

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Graduação: Prof. Sérgio Roberto Kieling Franco
Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb
Coordenador do Curso de Ciência de Computação: Prof. Carlos Arthur Lang Lisbôa
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

“We should not judge people by their peak of excellence,
but by the distance they have traveled from the point where they started.”

— Henry Ward Beecher

ACKNOWLEDGEMENTS

I would like to thank my advisor, my colleagues, my family, and my girlfriend.

ABSTRACT

This work proposes an algorithm based on heuristic search to solve Atomix. Atomix is a video game puzzle developed in the 1990s. It falls under the category of *sliding block puzzles*, which also contains popular games such as Sokoban, Rush Hour, and the $(n^2 - 1)$-puzzle, which have all been well studied in the literature.

The Atomix puzzle takes place on an integer rectangular grid, where pieces (called atoms) can be moved by the player through sliding operations. A sliding operation consists of moving a single atom horizontally or vertically on the grid; once a move is made, the atom will slide over the grid until it reaches an obstacle, which could be another atom or a 'wall' (a static obstacle). The objective of the game is to arrange the atom in a certain configuration called a molecule. Since the place of the molecule is not specified there are often multiple possible goal states.

Atomix's complexity was first studied by Holzer and Schwoon (2004), who have proved it to be PSPACE-complete. Heuristic search methods for Atomix were studied by Hüffner et al. (2001); however, the heuristic proposed by the article is somewhat uninformed, leaving several instances of the standard testbed unsolved.

In this work, we study domain-dependent heuristic functions for Atomix based on pattern databases (CULBERSON; SCHAEFFER, 1996), in the hopes of advancing the contributions made by (HÜFFNER et al., 2001). We also study a number of tie-breaking rules for the A* algorithm, as well as some implementation-specific optimizations. Finally, an improved solution is proposed.

Keywords: Heuristic search. A*. algorithms. Atomix. sliding block puzzles.

Encontrando Soluções Exatas para Atomix.

RESUMO

Este trabalho propõe um algoritmo baseado em busca heurística para resolver Atomix. Atomix é um puzzle de video game desenvolvido nos anos 90. Ele cai na cadegoria de *puzzles de blocos deslizantes*, que também contem jogos populares como Sokoban, Rush Hour, e o $(n^2 - 1) - puzzle$, todos os quais têm sido bem estudados na literatura.

O puzzle Atomix ocorre em uma grade retangular inteira, onde peças (chamadas átomos) podem ser movidas pelo jogador através de operações deslizantes. Uma operações deslizante consiste em mover um único átomo horizontalmente ou verticamente sobre a grade; uma vez que um movimento foi feito, o átomo irá deslizar sobre a grade até que encontre um obstáculo, que pode ser outro átomo ou uma parede (um obstácilo estático). O objetivo do jogo é montar os átomos em uma certa configuração chamada molécula. Como o lugar da molécula não é especificado, é comum haver mais de um estado final.

A complexidade de Atomix foi primeiro estudada por Holzer and Schwoon (2004), que o provou ser PSPACE-completo. Técnicas de busca heurísica para Atomix foram estudadas por Hüffner et al. (2001); porém, a heurística proposta pelo artigo é relativamente desinformada, deixando várias instâncias não resolvidas.

Neste trabalho, nós estudamos heurísticas dependendes de domínio para Atomix baseadas em bancos de dados de padrões (CULBERSON; SCHAEFFER, 1996), na esperança de avançar as contribuições feitas por (HÜFFNER et al., 2001). Nós também estudamos técnicas de desempate para o algoritmo A*, além de algumas otimizações específicas à implementação. Finalmente, uma solução melhorada é proposta.

Palavras-chave: Busca heurística. A* search. Atomix. Puzzles de blocos deslizantes.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

AFS	All Final States
GC	Goal Count
FO	Fill Order
NRP	Number of Realizable Generalized Paths
PS	Perimeter Search
BFS	Breadth First Search
DFS	Depth First Search
PDB	Pattern database

CONTENTS

1 INTRODUCTION

1.1 Structure of This Work

This work is organized as follows. Chapter 1 describes the Atomix puzzle, and briefly discusses what has been previously studied in the literature about Atomix and other sliding block puzzles. Chapter 2 presents a brief overview of the heuristic search methods and the state-of-the-art techniques that were employed in this work. Chapter 3 presents and explains in detail the techniques and heuristics we applied in this work: the standard heuristics, some implementation details, tie-breaking rules and pattern databases. Chapter 4 describes the experiments that were conducted and discusses the results, providing a comprehensive comparison between the methods tested. Finally, Chapter 5 summarizes our contribution, and provides possible ideas on how to further improve our solution.

1.2 The Atomix Puzzle

1.2.1 Origins

The game Atomix was originally developed by Günter Krämer, published by Thalion Software, and released for the Commodore Amiga in 1990. In the late 1990s, it was also published for other computing systems.

1.2.2 The Game Setup

The game takes place on an integer grid of size $w \times h$. Distributed over this grid are n pieces, called *atoms*, which must be assembled together to form a specific molecule. The molecule to be assembled is given by the problem statement. The grid area is surrounded by solid walls: obstacles through which atoms cannot pass. There may also be walls inside the grid area.

A *molecule* is an atom pattern representing the desired final configuration of atom positions. The molecule may be placed anywhere on the board (as long as there is room for it, naturally), so there may be more than one place where one can assemble it. This implies that, when employing a heuristic search algorithm such as A*, there will be not one, but several goal states. The final molecule cannot be mirrored or rotated. A final molecule always contains all of the atoms on the board.

Atoms may be distinct, and must each be represented by an identifying label. For example, “H-” might represent a hydrogen atom with an atomic link to the right, or “=O=” an oxygen atom with links to both right and left directions. Two atoms with different link directions (for example “-H” and “H-”) are considered distinct, i.e., their final positions on the molecule cannot be interchanged. The problem also permits more than one atom to have the same type (or label). For example, a molecule may require two hydrogen atoms with the exact same rotation. We will see in Section 3.1.3 that multiple atoms with the same label make the problem more difficult.

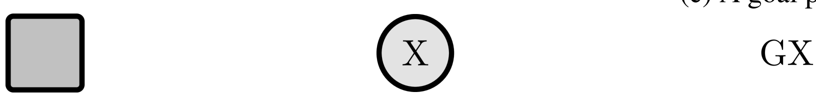
An instance of the Atomix puzzle defines the molecule pattern to be assembled, the grid layout, the number of atoms and their respective types, and the initial position of every atom.

This work provides several graphic examples of Atomix instances, in order to demonstrate peculiar situations that occur in the puzzle. Figure 1.1 shows the basic graphic elements used to describe Atomix: a wall, an atom having label X, and a goal position of the atom with label X. Atoms labels are upper-case letters; if two atoms or two goals have the same label, we differentiate them using subscript indexes, such as X_1 or GX_2.

Figure 1.2 shows an example Atomix instance, instance atomix_03 of the standard testbed. The goal of this instance is to assemble the Methanol molecule. Figure 1.3 shows all the four possible final states, that is, the positions where the final molecule can be placed.

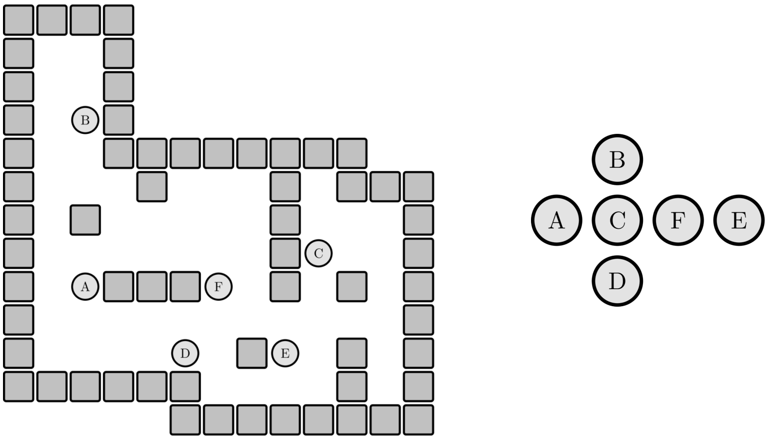
Figure 1.1: The notation used for the examples presented in this work.

(a) A wall. (b) An atom. (c) A goal position.



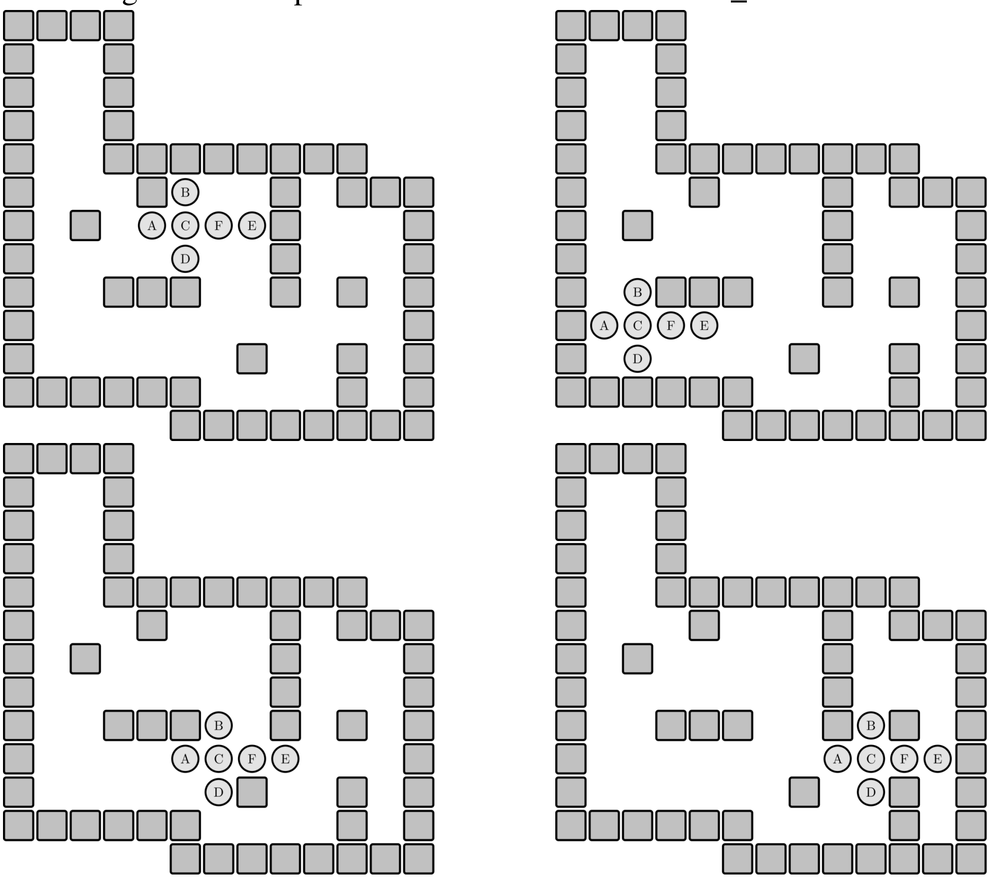
Source: the author.

Figure 1.2: On the left, the initial state for the atomix_03 instance; on the right, the molecule to be assembled.



Source: the author.

Figure 1.3: All possible final states for the atomix_03 instance.



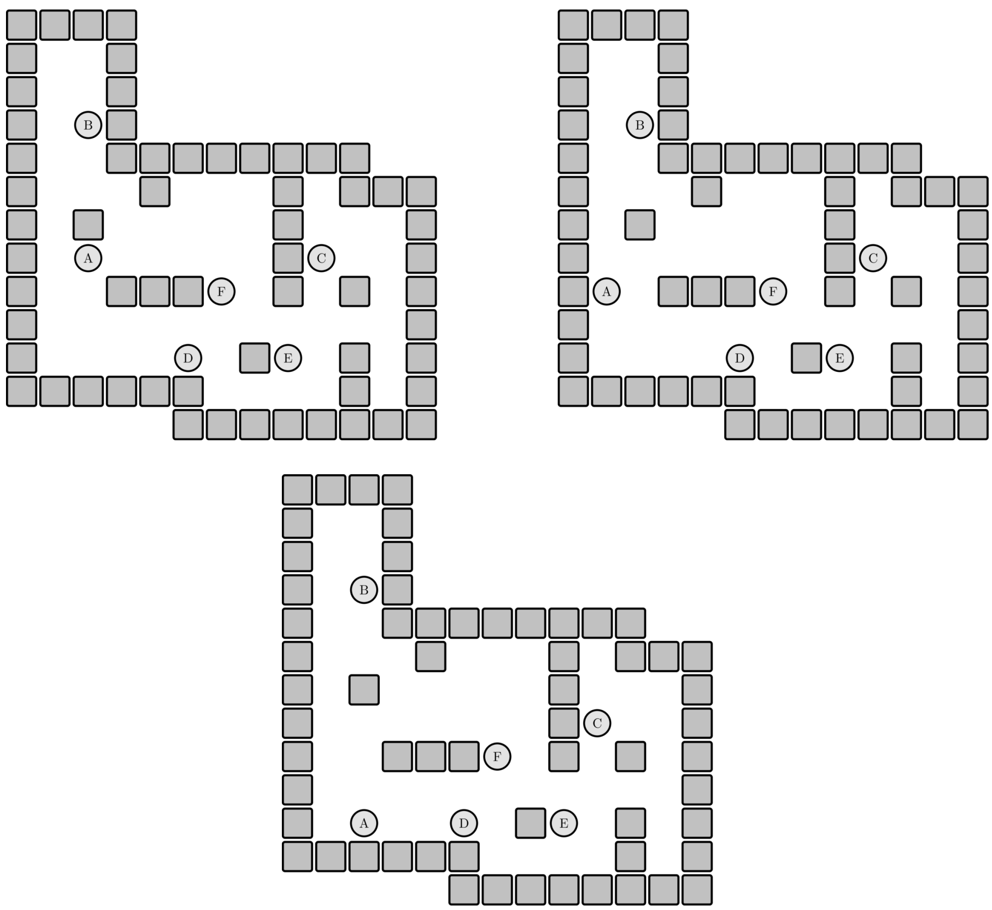
Source: the author.

1.2.3 Moving Atoms

A single atom can be moved with a *sliding operation*. A sliding operation on an atom can be performed in any direction (up, down, left or right), and causes that atom to be moved in the desired direction until an obstacle (another atom, a wall, or an outside border) is reached; the atom will then stop on the position before the obstacle and end its movement. When sliding, an atom may not stop at any intermediate position between its initial position and its stopping point. We can therefore define the concept of *direct neighborhood* of a given game state: it is the set of states generated by moving every single atom on the original configuration in every possible direction.

Figure 1.4 shows the 3 neighboring states achieved by moving the atom A on the initial configuration of atomix_03 instance, described in the previous section. The other 15 neighbors are omitted, for simplicity.

Figure 1.4: Neighboring states achieved by moving the atom A on the initial configuration of atomix_03.



Source: the author.

The atomix_03 example instance can be solved optimally with a sequence of 16 moves: E-up, C-down, C-left, C-up, D-right, F-right, F-down, B-down, B-right, B-down, B-right, B-down, B-left, A-down, A-up, A-right.

1.2.4 Formal Definition

A *game instance* can be represented formally by:

- A boolean matrix $W \in \{0,1\}^{w \times h}$ where $W_{ij} = 1$ if the position (i, j) on the grid is a wall (a static obstacle), and 0 otherwise.
- A set of atom labels $L \subset \mathbb{N}$. Two atoms that have the same label are considered duplicate (they are of the same type), and can be interchanged in a goal state.
- A starting game state S (the definition of game state is given below).
- A set of goal game states G.

A *game state* is represented as set of pairs $\{(p_1, l_1), \ldots, (p_n, l_n)\}$, each tuple representing an atom, where $p_i \in \mathbb{N} \times \mathbb{N}$ is the 2D coordinate (r, c) of that atom, and $l_i \in L$ is a label representing the type of that atom. A *direction* is a coordinate offset (d_x, d_y), which can be down $= (0, 1)$, up $= (0, -1)$, right $= (1, 0)$ or left $= (-1, 0)$. We define the set of all directions to be $D = \{\text{up}, \text{down}, \text{left}, \text{right}\}$. A grid position $p = (r, c)$ is said to be *empty* with respect to a state S if $W_{rc} = 0$ and $(p, l) \notin S$ for any $l \in L$.

A *move* is a function $move(p, d)$ that, when applied on a position p and direction d, yields the first position $p + \delta d$ such that every $p + \delta' d$ is empty for $0 < \delta' \leq \delta$ and $p + (\delta + 1)d$ is not empty. If the position $p + d$ is not empty, $move(p, d)$ yields p.

The neighbors of a given state P form a set of states $N(P) = \{P' \mid P'_i = P_i \ \forall\ i \neq a \text{ and } P'_a = move(P_a, d),\ \ \forall\ a \in \{1, \ldots, n\}\ \ \forall\ d\ \in D\}$.

A state S is considered to be a *solution state* if $S \in G$. The *distance* between states P and Q is the minimum number of neighboring moves needed to transform P into Q.

1.3 Previous Work

1.3.1 On the Complexity of Atomix

Holzer and Schwoon (2004) shows that it is possible to conceive Atomix instances which have optimal solutions that are exponentially long on the board size; however, these instances tend to not contain molecules patterns which are normally found in nature, and most likely would not be present in standard instance sets.

Furthermore, it shows that Atomix is PSPACE-complete with respect to the board size $w \times h$, by reducing the non-emptiness intersection problem for finite automata to Atomix. It also states that the complexity of Atomix is due to the board structure (i.e., the static obstacles on the board), and not from the types of atoms or their distribution on the final molecule.

1.3.2 On Searching the State Space of Atomix

Hüffner et al. (2001) present a technique based on heuristic search to solve Atomix. It uses A* and IDA* to find an optimal sequence of moves to solve the problem.

The paper proposes a relaxed atom movement pattern called *generalized moves*, which allows atoms to stop at any free space in a given direction, instead of only at the position just before an obstacle. It also allows for more than one atom to occupy the same place. The value of the heuristic for a given state P to a goal state G is the sum of the generalized distances of every atom in P to every final position in G. An important advantage of this heuristic function is that single distances can be pre-computed and the total heuristic value can be computed efficiently. However, removing the sliding property of Atomix is a substantial abstraction, and leads to poor lower bounds.

The paper also uses of the fact that the heuristic is monotone (consistent) to propose a very efficient open list data structure for the A* algorithm, which is several times faster than standard implementations such as C++ STL's `priority_queue`. The disadvantage of this approach is that it does not allow tie-breaking techniques to assign further priorities to states with the same f-value.

1.4 Related Puzzles

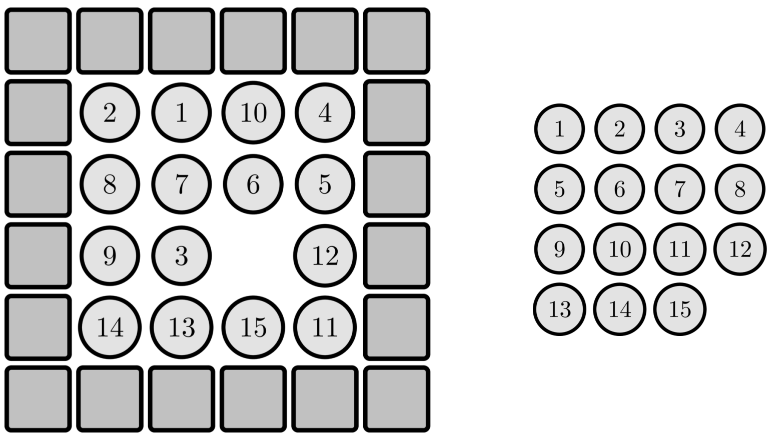
1.4.1 15-puzzle and the $(n^2 - 1)$-puzzle

The 15-puzzle, and the more generic $(n^2 - 1)$-puzzle, are perhaps the most famous benchmark problems used for heuristic search techniques. It consists of a 4×4 board containing a set of 15 numbered tiles and an empty space; tiles that are adjacent to the empty space may slide onto it, occupying its space and leaving their previous position empty. The goal of the puzzle is to arrange the tiles in a pre-defined order.

The advantages of using the $(n^2 - 1)$-puzzle to test new heuristic methods is that it is easy to implement, and has an obvious admissible heuristic: the sum of the Manhattan distances between tiles and their final positions. Also, the 15-puzzle version has a rather small state space and can be solved for many different instances in feasible time. It can also be extended to the 24- or 35-puzzle versions if a harder problem is desired.

Due to its close relation to Atomix, many heuristic search methods developed for the $(n^2 - 1)$-puzzle can also be used in Atomix. In particular, *pattern databases* (CULBERSON; SCHAEFFER, 1996), which were first applied for the 15-puzzle, have proved to be very useful not only for Atomix, but for other sliding block puzzles. Any $(n^2 - 1)$-puzzle can be represented as an Atomix level, as shown in Figure 1.5.

Figure 1.5: The 15-puzzle as an Atomix instance: on the left, the initial state, and, on the right, the molecule to be assembled.



Source: the author.

1.4.2 Sokoban

Sokoban is a classic single-player game taking place in a maze, over which stones (pieces) are scattered. Those stones may be pushed onto adjacent squares by an agent (or *man*) controlled by the player. The objective of the game is to move all stones into a set of goal positions. Sokoban has been shown to be PSPACE-Complete (CULBERSON, 1999).

As a sliding block puzzle, Sokoban bears resemblances to Atomix. It takes place in a maze where pieces are to be moved onto goal positions. This hints that many of the same techniques used to solve Sokoban may be used to our advantage in Atomix. In particular, some methods used by Pereira, Ritt and Buriol (2013) for Sokoban are also employed in this work to improve heuristics for Atomix.

One important difference between the two puzzles is that in Sokoban, the player is represented on a board square as a "man" and may only push stones to which it is adjacent, while in Atomix the player may move any stone at any time, as in a "god mode". Another difference is that, while in Atomix the atoms can all be different, each having one pre-defined goal position, the Sokoban stones are all considered to be the same, so that any matching of stones to goal positions is a viable solution. This reduces the state space considerably, compared to Atomix. Finally, we can also note that solution lengths for Sokoban standard instances are quite long, averaging from about 100-600 movements, whereas, for Atomix, known solution lengths range from 20-60 movements.

Sokoban has an interesting property: it allows for *deadlock* states, that is, states from which no solution can be found. This property might make it easier to solve the problem, since it prunes nodes which will certainly not lead to a solution. In fact, this is also the case for Atomix; however, in the available instances, this kind of situation occurs extremely infrequently, and is not a major problem. One reason for this is that we have no man in Atomix.

1.4.3 Overview of the Complexity of Other Sliding Block Puzzles

Table 1.1 compares Atomix, Sokoban, 15-puzzle, and other sliding block puzzles. The column Move uses the nomenclature *Move-NumPieces-GoalType*, where *Move* can be $Push$, $Pull$ or $PushPull$, *NumPieces* denotes the number of pieces that can be moved at once (1, k, or $*$), and *GoalType* denotes the type of goal of the problem: to move the agent to a final position (P) or to store the pieces in a set of specific position (S). A Move of type $MoveMove$ means that moving pieces will slide until they encounter a goal state. For instance, Atomix would be of type PushPushPullPull, because atoms can be both pushed and pulled by sliding operations. If a result is valid for all variants of *NumPieces* or *GoalType*, the correspondent suffixes are omitted.

Table 1.1: Comparison of the complexity of sliding block puzzles

Game	Move	Complexity	Reference
Sokoban	Push-1-S	PSPACE-comp.	(CULBERSON, 1999)
	Push-1-P	NP-hard	(DEMAINE, 2001)
	Push-k with $k \geq 2$	PSPACE-hard	(DEMAINE; HEARN; HOFFMANN, 2002)
	Push-*	PSPACE-hard	(DEMAINE; HEARN; HOFFMANN, 2002)
	PushPush-1	PSPACE-hard	(DEMAINE; HOFFMANN; HOLZERC, 2004)
	PushPush-k	PSPACE-hard	(DEMAINE; HOFFMANN; HOLZERC, 2004)
	PushPush-*	NP-hard	(DEMAINE; HOFFMANN; HOLZERC, 2004)
	Pull-P	NP-hard	(RITT, 2010)
	Pull-S	PSPACE-hard	(PEREIRA; RITT; BURIOL, 2016)
	PullPull	PSPACE-hard	(PEREIRA; RITT; BURIOL, 2016)
	PushPushPullPull	PSPACE-hard	(PEREIRA; RITT; BURIOL, 2016)
	PushPull	PSPACE-hard	(PEREIRA; RITT; BURIOL, 2016)
15-puzzle		NP-hard	(RATNER; WARMUTH, 1990)
Rush Hour	PushPushPullPull-k-P	PSPACE-comp.	(FLAKE; BAUM, 2002)
Atomix	PushPushPullPull-1-S	PSPACE-comp.	(HOLZER; SCHWOON, 2004)

Source: the author.

2 HEURISTIC SEARCH

2.1 Introduction

Most single-player puzzles can be formulated as a *state space problem*, which consists of a state space S, a set of initial states $I \subseteq S$, a set of goal states $G \subseteq S$, and a set of operators O, where $o \in O$ is a function $S \rightarrow S$ that maps a given state to a neighbor state. In a more general case, a *weighted state space problem* also defines a cost function $w : O \rightarrow \mathbb{R}$ which assigns a cost for every action. In the case of Atomix, all movements have the same cost. The goal of this type of problem is to find an ordered sequence of operators $(o_1, \ldots, o_n) \in O^n$ that, when applied to one of the initial states in I, yields one of the goal states in G, and that minimizes the total cost $\sum_{i=1}^{n} w(o_i)$ of the path taken.

State space problems can be solved by heuristic search algorithms such as A* (HART; NILSSON; RAPHAEL, 1968) and IDA* (KORF, 1985). These algorithms rely on *heuristic functions* to guide the search over the state space. A heuristic function is a function $S \rightarrow \mathbb{R}$ that gives an estimate of the solution path cost for a given current state. In particular, an *admissible* heuristic is one that will never overestimate the actual solution cost. If the heuristic function is admissible, it is proven that A* and IDA* will terminate with an optimal solution; otherwise, that is not guaranteed. A *consistent* or *monotone* heuristic is one where the total estimate solution cost (which is the value of the heuristic plus the total cost accounted so far; also called the *f-value*) is always increasing over any state sequence. A consistent heuristic guarantees that, in A* search, no state will be visited more than once.

Although many implementation-specific optimizations can be made, these will usually increase the performance by only a constant factor. The greatest improvements on A*/IDA* stem from better heuristic functions, i.e., ones that achieve a higher lower bound on the actual solution cost, while still maintaining admissibility. A good heuristic function can be exponentially more efficient than a bad one. This is because, the better the heuristic function, the less of the search space the algorithm will tend to explore.

2.2 The A* Algorithm

A* is one of the most widely used algorithms in heuristic search. Although it is efficient, it requires an amount of memory of the order of the state space size, since it will store every state expanded in memory. Nonetheless, it tends to be much faster than other memory-efficient algorithms such as IDA*.

The A* algorithm ranks visited states based on their f-values $f(s) = g(s) + h(s)$, where $g(s)$ is the number of movements required to reach state s from the start of the search, and $h(s)$, the *heuristic* function, is an estimate on the minimum number of moves required to reach a goal state from s.

The algorithm keeps all states found in a states table, which is usually implemented by a hash table. For each state, we keep its g-value, its h-value, and a pointer or index to its parent state: the state that was visited just before it. States which have been found and not yet expanded are kept in a data structure called the *open list*, which contains, at first, only the starting states. At every iteration, the algorithm selects a state with the lowest f-value for expansion, and removes it from the open list. The neighboring states of the expanded state will then be visited and added to the open list, provided they have not yet been visited; if a neighboring state has been previously visited with a higher g-value, we update its entry in both the states table and the open list. The algorithm terminates when a goal state is removed from the open list, or when there are no more states in the open list. In that case, it means that reaching a goal state is impossible. Algorithm 1 shows the A* algorithm in detail.

In this work, we chose to use A* to implement our solution and test the heuristics proposed in Chapter 3.

2.3 The IDA* Algorithm

Iterative Deepening A* (IDA*) is an alternative search algorithm to A* that uses memory linear on the size of the solution path constructed (which is quite negligible). The main idea behind IDA* is to perform a series of bounded depth-first-searches (DFS) with increasing move limits, until a solution is found. During each DFS, if the current recursion depth plus the heuristic estimate for a node exceeds the move limit, that node is pruned. Like A*, it is proven that, if the heuristic is admissible, IDA* will return an optimal solution. Unlike A*, since it does not keep tab of which nodes were visited, the DFS can end up visiting the same nodes several times.

IDA* can be very useful for cases when we have tight memory constraints, for instance, when A* may consumes all the available memory before a solution is found.

Algorithm 1 The A* algorithm.

Procedure A*
Input: implicit problem graph with start node s, a set of goal nodes T, weight function w, heuristic h, successor generation function $Expand$, and predicate $Goal$.
Output: cost-optimal path from s to $t \in T$, or $\emptyset$ if no such path exists.

```
Closed ← ∅
Open ← {s}
f(s) ← h(s)
while Open ≠ ∅ do
   Remove u from Open with minimum f(u)
   Insert u into Closed
   if Goal(u) then
      return Path(u)
   else
      Succ(u) ← Expand(u)
      for each v in Succ(u) do
         Improve(u, v)
      end for
   end if
end while
```

Procedure Improve
Input: Nodes u and v, v successor of u
Side effects: Update parent of v, $f(v)$, $Open$, and $Closed$

```
if v in Open then
   if g(u) + w(u, v) < g(v) then
      parent(v) ← u
      f(v) ← g(u) + w(u, v) + h(v)
   end if
else
   if v in Closed then
      if g(u) + w(u, v) < g(v) then
         parent(v) ← u
         f(v) ← g(u) + w(u, v) + h(v)
         Remove v from Closed
         Insert v into Open with f(v)
      end if
   else
      parent(v) ← u
      Initialize f(v) ← g(u) + w(u, v) + h(v)
      Insert v into Open with f(v)
   end if
end if
```

Source: Edelkamp and Schroedl (2011), adapted.

2.4 Pattern Databases

Pattern Databases (PDBs), a concept first introduced by Culberson and Schaeffer (1996), are one of the most powerful ways to create admissible heuristics for state space problems. They have been widely used in the last decade to solve benchmark problems such as the $(n^2 - 1)$-puzzle ((CULBERSON; SCHAEFFER, 1996), (FELNER; KORF; HANAN, 2004) and (KORF; FELNER, 2002)), Sokoban (PEREIRA; RITT; BURIOL, 2013), Rubik's Cube (KORF, 1997), and many others.

The most direct definition of a PDB is a look-up table containing all possible values of a heuristic function, which can be accessed in constant time during search. Unfortunately (or fortunately!), the most interesting state space problems have a huge number of possible states, most of them with more states than could probably fit in a computer memory.

The main idea behind PDBs is to use an abstraction to reduce the state space problem to a simpler problem, or pattern, with a smaller search space, which can be fully explored in feasible time. A problem which has been simplified by an abstraction is said to have an *abstract state space*. This abstract state space must be small enough such that its solutions can stored in memory. In sliding block puzzles such as the 15-puzzle and Sokoban, this is normally done by removing some pieces and solving the original puzzle with the remaining pieces, which are called the *pattern*. Another alternative would be to anonymize a set of pieces by removing their labels. In the abstracted problem, a state is identified exclusively by the positions of the pattern pieces (or of all pieces, if we anonymize some of them). A PDB is constructed by visiting all reachable abstract states with a backward breadth-first-search, starting from the goal state, and recording the distance to every other state. If there are multiple goal states (as is the case of Atomix), we may either construct one PDB for every goal state, or a single PDB encompassing all goal states.

Whenever possible, multiple PDBs should be built, in order to better utilize the available memory. Of particular interest are *disjoint pattern databases* (KORF; FELNER, 2002). In sliding block puzzles, two PDBs are disjoint if the sets of pieces used to build them are also disjoint. The key advantage of disjoint pattern databases is that the contribution of multiple disjoint PDBs can be added to make an admissible heuristic, whereas the only obvious way of combining non-disjoint PDBs is by taking the maximum among them. In the literature, Korf (1997) uses the maximum of three overlapping PDBs to compute a heuristic for the Rubik's cube, while Korf and Felner (2002) takes the maximum of the sum of two sets of disjoint PDBs to efficiently solve the 24-puzzle. Holte et al. (2004) explores in depth the use of multiple PDBs and shows that, in some cases, it is more useful to use n (m/n)-sized pattern databases instead of a single m-sized pattern database.

In sliding block puzzles, disjoint PDBs must partition the pieces into disjoint sets, whose respective PDB heuristics will be added. Felner, Korf and Hanan (2004) present two PDB variants that differ on the way of performing the pattern partition: *statically-partitioned* PDBs and *dynamically-partitioned* PDBs. In a statically-partitioned PDB, the disjoint sets are pre-defined according to some criterion before the PDB is actually built. In a dynamically-partitioned PDB, one PDB is built for every possible pattern, and the partition is performed at run-time, so as to choose the partition which maximizes the sum of the contributions of the PDB for each state. Empirically, dynamically-partitioned PDBs are only feasible for small patterns, because the number of possible patterns can be quite large.

In Section 3.4, we explore three different PDB variants for Atomix: a static disjoint PDB of size 3, a dynamically-partitioned PDB of size 2, and a dynamically-partitioned multiple goal PDB of size 2. The three PDB variants are compared experimentally in Section 4.5.

2.5 Hierarchical A*

Hierarchical search (HOLTE et al., 1996) is an A* approach based on a series of increasingly simpler abstractions of the original problem. The heuristic function used for a concrete (not abstracted) A* is the result of a second A* run on an abstracted version of the problem, which in turn, uses as heuristic the result of a third, even more abstract A*, and so on. The argument for hierarchical search is that the large cost of computing the abstracted solutions on the hierarchy ends up being amortized, because it should lead to heuristic functions of much higher quality. In practice, this is not always the case.

In a naïve implementation, multiple A* runs on the same level of abstraction would repeatedly expand a very large number of the same nodes. Holte, Grajkowski and Tanner (2005) present two optimizations for hierarchical search based on caching, which are denominated optimal path caching and P-g caching.

2.6 Perimeter Search

Perimeter search, a concept introduced by Dillenburg and Nelson (1994), attempts to improve heuristics that give poor lower bounds near the vicinity of the goal state. It performs a backward BFS bounded to k moves, starting from the goal state, before the informed search algorithm begins. All nodes in the final perimeter of the BFS (i.e., nodes with distance k from the goal state) are stored, and during the informed search (either A* or IDA*), we compute the heuristic value of a node as the minimum heuristic distance between that state and any of the nodes in the perimeter. As an advantage, it gives better lower bounds, since the distance from the perimeter to the goal is exact, and not an estimate. On the other hand, it makes the heuristic more expensive to compute, since it must be computed for every node in the perimeter. Felner and Ofek (2007) propose a way to improve this by combining perimeter search with pattern database abstractions.

Another advantage is that the forward search can terminate whenever a state in the perimeter is found. Furthermore, if a node's heuristic estimate is smaller than k and that node was not expanded by the perimeter search, we can correct the h-value to be $k+1$; this, however, causes the heuristic to be non-consistent.

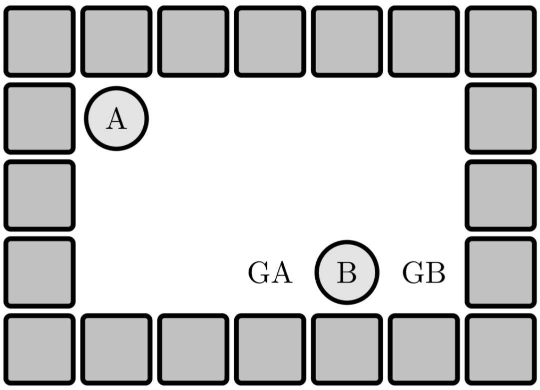
3 SEARCHING THE STATE SPACE OF ATOMIX

3.1 A Standard Heuristic for Atomix

3.1.1 The Idea

In many sliding block puzzles, the most straightforward way to achieve good heuristics is to remove pieces from the board, and solve the abstract problem problem with fewer pieces. This abstract problem is easier, in general. However, the sliding property of Atomix disallows us to do so: interactions between atoms are almost always necessary in order to achieve an optimal solution, and often to achieve any solution at all. An atom on its own may not be able to reach its goal position; in fact, without interactions, the reach of a single sliding atom is often extremely limited. Figure 3.1 exemplifies this: atom A cannot reach its final position GA without the help of atom B, which must act as an obstacle.

Figure 3.1: Example of the reachability problem: A cannot reach GA without the help of B.



Source: the author.

It is clear that any abstraction that only removes atoms is not admissible. In Atomix, before we remove atoms, we must abstract the slide operation.

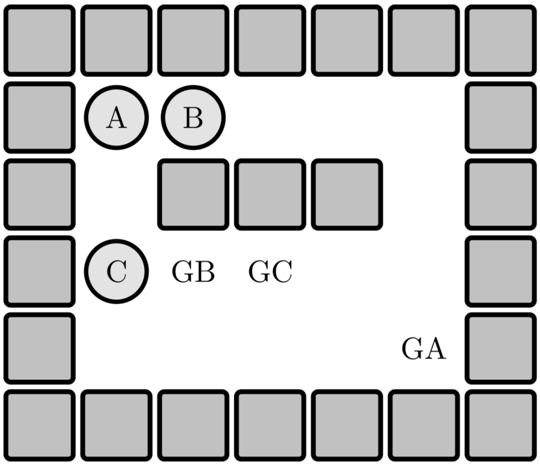
The first heuristic upon which we based this work, which we call the *standard heuristic*, has been proposed by Hüffner et al. (2001). The heuristic is based on abstracted sliding movements called *generalized moves*. It provides two abstractions:

1. Instead of sliding, atoms may stop at any free position between the current position and the end position of the slide. This removes the sliding property and greatly simplifies the problem: when able to stop its slide, an atom does not need other atoms as obstacles to reach a position on the board. However, this increases the branching factor considerably.

2. Interactions between atoms are ignored; two atoms may occupy the same position, and may pass through each other. This amounts to the same as solving the problem separately for every abstracted atom, and adding up all the results. Another way to put this: in the standard game each free cell has a capacity of one. In this version the capacity constraints are relaxed.

The *goal distance* of an atom $P_i = (r, c)$ in a given state P is defined as the distance from any of the goal positions of atom i to (r, c), using generalized moves. Finally, the value of the heuristic function is the sum of the goal distances of all atoms to their final positions. Figure 3.2 shows an example where the standard heuristic would yield the value 6: 2 for atom A, 3 for atom B, and 1 for atom C.

Figure 3.2: Example where the standard heuristic would yield 6: 2 for atom A, 3 for atom B, and 1 for atom C.



Source: the author.

3.1.2 Pre-Computing Relaxed Distances

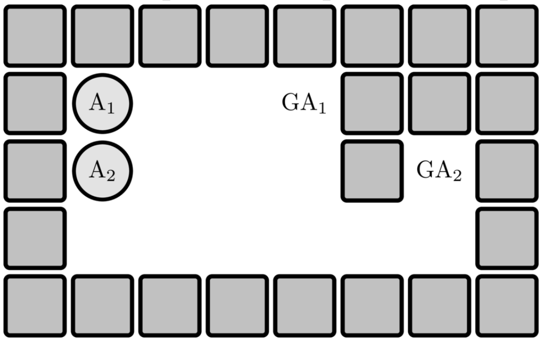
The standard heuristic can be pre-computed for all possible source and target positions before the search algorithm starts. In order to do so, we start a breadth-first-search from every board position, and visit all other positions on the board using generalized moves. The distance vector of the breadth-first-search contains the distance from the source position to all other board positions. Another way to achieve the same result would be to perform the Floyd-Warshall algorithm (FLOYD, 1962) on the induced graph.

The time and memory complexity of this strategy is quadratic in the board size. Since the board size does not exceed 1000 squares in any instance of the standard testbed, the memory and time overheads of this pre-computation are negligible.

3.1.3 Dealing with Duplicate Atoms

The above idea does not work when two atoms have the same label. In this case, there exists more than one final position where they can be placed. Figure 3.3 exemplifies this: both atoms A_1 and A_2 have the same type, and can go to any of their possible final positions GA_1 or GA_2. If we use the same logic presented above, the heuristic will choose both atoms to go to same goal A_1 (the closest one), yielding an h-value of 3 (1 for A_1 and 2 for A_2). By allowing both atoms to go to their closest final position, we lose information, and the heuristic, although still admissible, will be less powerful. It would be better if it chose A_1 to go to GA_1, and A_2 to go to GA_2, thus achieving a heuristic value of 4.

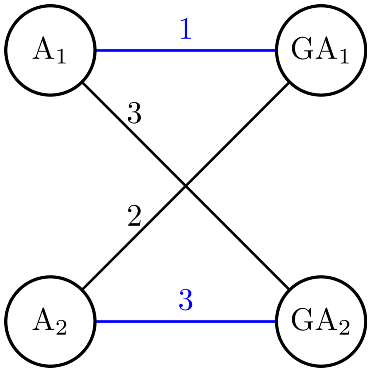
Figure 3.3: Example of the duplicate atoms problem.



Source: the author.

Unfortunately, testing all possible $n!$ combinations of atoms to final positions by brute force would render the heuristic too costly to compute, for some instances. In order to achieve this efficiently, we perform a *minimum cost perfect matching* on the bipartite graph induced by the atom positions and their final position. This problem can be solved in $O(n^3)$ using shortest augmenting paths (MUNKRES, 1957). This is the same idea used for achieving a standard heuristic for Sokoban (PEREIRA; RITT; BURIOL, 2013), where all the stones are considered equal and must be matched to their final positions. Figure 3.4 shows the bipartite graph corresponding to the example in Figure 3.3: the edges marked in blue represent the minimum cost matching for this graph.

Figure 3.4: The bipartite graph induced by the duplicate atoms in the example in Figure 3.3. The edges in blue represent the minimum cost matching.



Source: the author.

In our implementation, if the number of duplicate atoms is equal to 3 or less, a brute force strategy is employed: all possible combinations are tested. Otherwise, a minimum matching is performed. We do this because, for $n \neq 3$, a brute force strategy is easy to implement and requires fewer operations than a minimum matching; for $n > 3$, constant time overheads entailed by the minimum matching, with aspects such as data structure initialization, are justified.

3.1.4 Dealing with Multiple Final States

The fact that Atomix may have multiple positions for the molecule (as discussed in Section 1.2.2) introduces a few problems, as different final states may yield different heuristics. It would be ideal if we knew exactly which of the possible final states will produce the optimal solution, but, unfortunately, it may difficult to show that a given final state even has a feasible solution.

In this section, we present two approaches to handle this problem. The first is used by (HÜFFNER et al., 2001), and the second is proposed in this work.

3.1.4.1 First Approach: Independent Search for All Final States

Hüffner et al. (2001) solves this problem by imposing a move limit and running one A* instance for every final state. The A* search will not add to the open list nodes whose f-values are greater than the move limit; the search will continue until there are no more nodes with an f-value smaller then the limit, or a solution is found. If a solution is not found, the move limit is increased by one, and a new A* search starts. This method is very similar to IDA*, with the exception that a state table is kept, so a state is not visited more than once. However, this means that the search will re-expand many of the same states every time the move limit is increased, which increases the number of nodes expanded, but by not more than a factor of the node branching factor b (KORF, 1985).

The main advantage of this approach is that it allows to compute the heuristic in constant time, instead of linear time in the number of atoms. After a neighboring move, only the contribution of the atom that was actually moved must recomputed: the relaxed distance of that atom on the previous position is subtracted from the current h-value, and the relaxed distance on the new position is added. Another advantage of this method is that, having only one final state, the heuristic will tend to lead the search directly towards the vicinity of that state.

3.1.4.2 Second Approach: Using All Final States

The approach we propose is quite simple: we take the heuristic value to be the minimum sum of generalized distances among all final states. The advantage of this is that it allows us to run a single A* with no move limit, and thus no states have to be expanded more than once (given that the heuristic is consistent); it is introduced in the hope of amortizing the weaker heuristic over the multiple individual searches.

The main disadvantage of this method is that it provides a costlier heuristic, since a standard heuristic is computed for every final state. We are also not able to recompute it in constant time after a move, because the closest final state may have changed, and we would not know the previous h-value for the new closest final state. One way to solve this is by keeping the best h-value for each final state, and updating each of them using only the contribution of the moved atom. However, this will substantially increase the memory usage of a state, which can be a crucial factor for A*. Empirically, the performance gains are insignificant.

3.1.5 Admissibility

The standard heuristic is admissible: the standard Atomix moves are a subset of the generalized moves. This means that the optimal solution is always reproducible by using only generalized moves, and so the heuristic value will be, at most, as long as the optimal solution length.

3.1.6 Consistency

Any standard move can be emulated by a generalized move. This means that, after a neighboring operation that performs one standard move, the total heuristic cost in generalized moves cannot differ by more than one, which is the cost of emulating that standard move by a generalized move. It follows therefore that $h_p \leq h_c + 1$, where h_p is the h-value of the parent state and h_c is the h-value of the child state.

3.2 Implementation Details

3.2.1 Representing States and Positions in Memory

Formally, a position is a pair (r, c) representing the board cell on the i-th row and j-th column. In memory, this can be represented as a single integer, having the value $i \times w + j$, where w is the board width. Unfortunately, since in our testbed instances board sizes have up to 289 positions, we cannot use an 8-bit integer (which holds 256 values) in a generic implementation. In our implementation, a 16-bit integer is used. The major drawback of this approach is that it wastes memory, and causes A* to run out of memory approximately two times faster than when using 8-bit integers. As a future optimization, we could analyze the instance input and re-compile the solution with an integer size suitable to fit the board size.

States are stored as an array of positions. We chose to use static arrays (as opposed to dynamic ones) so as to take advantage of the fact that temporary objects can be placed on the stack, without requiring a heap memory allocation, which usually involves an expensive system call. In order to use a static array, the number of atoms must be known at compile time; for the final tests, we re-compiled the state class for every instance.

3.2.2 An Efficient Bucket-Based Open List for A*

In the A* algorithm, we need an open list implementation that allows efficient access to the element with the lowest f-value, at every state expansion. The basic operations we need to perform are *insert*, *decrease-key* (or update) and *delete-min* (access and remove the smallest element). A data structure that provides those operations is called a *priority queue*. For those means, heap-based data structures such as the binary heap and the Fibonacci heap (FREDMAN;

TARJAN, 1987) are the most obvious choices, as they are powerful, generic, and widely available in data structure libraries. In particular, the Fibonacci heap allows for time complexity $O(\log n)$ for *delete-min* and $O(1)$ for both *insert* and *decrease-key*.

When elements are ranked based on a discrete key which assumes values in a fixed and small range, we can use an open list based on *buckets*. A bucket-based open list consists of an array of k buckets, where k is the maximum f-value (upper bound) that we expect the search to generate. A *bucket* with index i is a dynamic array that stores all open (not expanded) states with f-value equal to i. We also keep, and update, the smallest index $0 \leq \mu \leq k$ for which there is a non-empty bucket. The basic operations are defined as:

insert: add the state to the bucket with index equal to its f-value. We also update $\mu = min(\mu, \text{f-value(state)})$. This is done in $O(1)$ time.

delete-min: while the bucket with μ is empty, increase μ by one. Remove any state in the μ-th bucket and return it. μ will be incremented at most k times, so this is done in $O(k)$. Note that, since k is fixed and does not depend on the number of elements in the open list, this means a constant time. Furthermore, removing any element from a list can be done in $O(1)$ time.

update: to do this, we would have to perform a linear search on the bucket of the states' old f-value, remove it, and re-insert it in the new f-value's bucket. Other alternatives would be for each state to store a pointer to its bucket position, to use a hash table as a bucket, or to store buckets as a linked list of states. As these alternatives would be either too time-expensive, memory-expensive, and/or difficult to implement, we chose to ignore this operation; instead, the state is simply re-inserted into the open list, which is done in $O(1)$ time. To preserve admissibility, when we call *delete-min* and remove a state from the μ-th bucket, we check if that states' f-value is equal to μ; if it is not, we throw this state away and continue expanding the next state. In practice, the effect this has on the number of nodes expanded is very small.

The search ends when a goal state is found, or when $\mu = k$, because then we know that there are no more open states.

For Atomix, out of the 155 instances of our testbed, after one hour of tests, our best solution finds a maximum lower bound of 65; if an instance has a solution length greater than 100, it is unlikely that we will be able to find it in a modest amount of time using the current available heuristics. In our implementation, we set the number of buckets k to be 100, which is easily manageable.

One drawback of this approach is that it may complicate the usage of tie-breaking rules, where we discriminate between states with the same f-value. In Section 3.3, we present three tie-breaking rules and argue that they do not prevent the use of this bucket-based open-list, save for a few small tweaks.

In Section 4.4 we compare experimentally this bucket-based open list implementation with an implementation that utilizes a Fibonacci heap.

3.2.3 Hashing Atomix States

We implemented a hash table as a static sized array. Since the maximum memory available is pre-defined, there is no rehashing: the entire hash table is pre-allocated before A* starts. Each entry in the hash table is an integer which references an index in the states array.

The hashing function used is the same as the one used by Hüffner et al. (2001), shown in Algorithm 2. The ll and gg operators mean left and right shifts, respectively. Compared to the C++'s STL string hashing algorithm and Spooky Hashing (JENKINS, 2012), this seemingly arbitrary hashing function is the one which obtained the best results, in terms of performance.

Algorithm 2 The hash function used for Atomix states.

Parameter S : the input state (an array of n integers)
$h \leftarrow 0$
for $1 \leq i \leq n$ **do**
  $h \leftarrow h + S_i$
  $h \leftarrow h + (h \ll 10)$
  $h \leftarrow h \oplus (h \gg 6)$
end for
$h \leftarrow h + (h \ll 3)$
$h \leftarrow h \oplus (h \gg 11)$
$h \leftarrow h + (h \ll 15)$
return h

Source: Hüffner et al. (2001)

As a future work, an interesting alternative to this hash function would be *Zobrist* hashing (ZOBRIST, 1970), which is a hashing scheme that specializes in abstract board games.

Hash collisions are treated with linear probing, i.e., if the desired table index is occupied, we linearly search the subsequent indexes until a free position is found.

3.3 Tie-Breaking Techniques

In A*, when two states in the open list have the same f-value, it is up to the open list implementation to decide which of those two states will be expanded first. A stable implementation may preserve insertion order, but, in general, the expansion order depends on details of the data structure and its implementation. By adding extra intelligence in choosing which node to expand, we may be able to explore less of the state space and thus find an optimal solution quicker.

Particularly in Atomix, even using the best applicable heuristics that we know of (see Section 3.4), some instances need tens of millions of states expanded before a solution is found; also, solution lengths tend to be smaller than 70. This implies that most of the time, several thousand states in the open list will have the same f-value. It could be beneficial, therefore, if we further discriminate between them.

In this section we present three tie-breaking techniques for Atomix based on domain-dependent knowledge. We compare them experimentally in Section 4.3.

3.3.1 Goal Count

The *goal count* (GC) tie-breaking rule is as simple as the name suggests: it counts the number of atoms already in their goal positions. States with a higher goal count have priority over states with a lower goal count. The tie-breaking value is the maximum goal count among all final states.

This tie-breaking rule is very simple and efficient to compute, requiring only a linear scan on the number of atoms, for every final state. To continue using an open list based on buckets, the same concept used in Section 3.3.2 applies: the amount of buckets is multiplied by the number of atoms.

3.3.2 Number of Realizable Generalized Paths

A *generalized path* between two positions is a sequence of generalized moves (see Section 3.1.1) that brings a single atom from one position to another. A generalized path is said to be *realizable* if it is unobstructed, i.e., there are no atoms blocking its way.

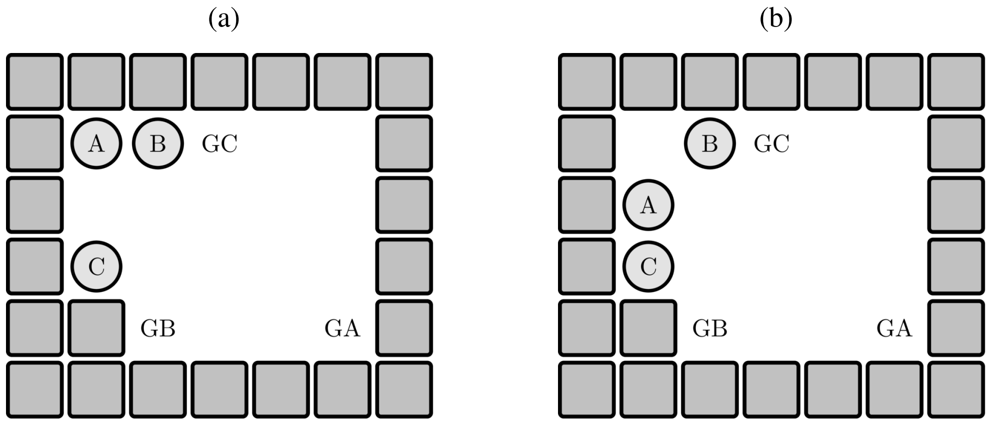
The *number of realizable generalized paths* (NRP) of a given state S with regard to a final state F is the number of atoms $S_i \in S$ which have a realizable generalized path from their current position S_i to any of their goal positions $\{F_j \mid L_j = L_i\}$ (where L is the set of atom labels, as defined in Section 1.2.4). Note that an atom which is already in its goal position counts as a realizable path (of length zero). The final tie-breaking value is the maximum NRP among all final states. States with more realizable paths have priority over states with fewer realizable paths.

The tie-breaking ranks can range from 0 to n, where n is the number of atoms. This allows us to continue using a bucket-based open list, except with number of buckets multiplied by n (since the maximum n over all instances in our testbed is 32, the total number of buckets is still manageable).

There might be more than one optimal generalized path between any two positions. We can generate all those paths by slightly modifying the breadth-first-search that pre-computes the standard heuristic (see Section 3.1.2) to store predecessor nodes, in a way that a state may have several predecessors. For efficient look-up, a generalized path is represented as a boolean array of size equal to the number of board positions, which holds 1 if a position is part of the path, or 0, otherwise.

Figure 3.5 shows a situation where two states with the same h-value (5) yield different tie-breaking values. Suppose both states have the same g-value. In the first example, both B and C have a realizable optimal path to their final position; A's optimal path of length 2, however, is being obstructed by B. All other generalized paths A can make to GA are not optimal. In the second example, an optimal generalized path for all atoms can be realized, thus yielding a tie-breaking value of 3. The state of the second example will therefore have priority over the one on the first.

Figure 3.5: Both situations have the same heuristic value of 5, but, in Figure 3.5a, the number of realizable paths is 2 (atoms B and C), while in Figure 3.5b, it is 3 (atoms A, B and C).



Source: the author.

The biggest downside of this approach is that the tie-breaking computation becomes expensive, as for every possible path we must iterate over all atoms to check for obstructions.

3.3.3 Fill Order

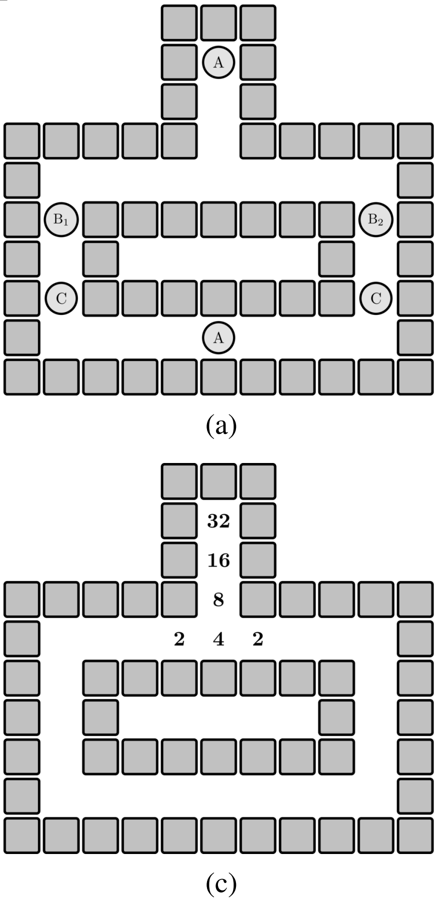
This tie-breaking rule was originally proposed for assembling stones in Sokoban by (PEREIRA; RITT; BURIOL, 2013), and consists of giving higher priority to states which have correctly placed atoms whose placement is more essential, and is more likely to happen first.

A *fill order* (FO) is an ordering of atoms based on a guess of the order in which the atoms would most likely be assembled in an optimal solution. Atoms which should be placed first have higher priority. The FO priorities are computed as follows: starting from the final molecule, we iteratively remove all atoms to which a backward move may be applied. To every atom removed at iteration i will be assigned priority 2^i. The algorithm continues until all atoms have been removed, or until none of the remaining atoms allow a backward move; in the second case, those atoms receive priority 2^{k+1}, where k was the total number of iterations.

Consider Figure 3.6, for example, which depicts the marbles_14 instance of the standard testbed; Figure 3.6a shows the initial state, and Figure 3.6b shows the only possible final state for that instance. From simple inspection, it is visually clear that the most efficient strategy would be to first move atoms C, B_1 and B_2, in this specific order, to their goal positions. Figure 3.6c shows the FO priorities for the marbles_14 instance. Figure 3.6d shows a state that has a fill order rank of 50: 32 for C, 16 for B_1 and 2 for A. Notice that, although there is an atom A in a position with priority 08, that priority is not accounted for, since it is not a goal position for A.

In order to adapt FO to a bucket-based open list, we must multiply the number of buckets by the maximum sum of fill orders among all final states. This increases considerably the number of buckets: for the example on Figure 3.6, it would be 64 times the default number of buckets, instead of only 6. That does not pose a significant performance overhead.

Figure 3.6: A fill order example depicting the marbles_14 instance. 3.6a shows the initial state, 3.6b shows the desired molecule, 3.6c shows the FO priorities for the molecule, and 3.6d shows an example state with FO rank 50.



Source: the author.

3.4 Pattern Databases

3.4.1 Creating Pattern Databases for Atomix

In standard Atomix, it is not so simple to remove atoms to create a simpler pattern, since any heuristic that exclusively removes atoms breaks admissibility. This is because sliding atoms may need support from other atoms to reach certain positions, as shown in Section 3.1.1. On the other hand, we *can* remove atoms in the *generalized* version of Atomix, where atoms may stop at any intermediate position: it does not preclude an atom of reaching its final position in an optimal (generalized) way.

However, since generalized moves allow atoms to pass through each other, it would not make sense to partition the atoms into patterns if the contribution of each atom is computed independently, as it would amount to the same as computing the original heuristic. In order to make a useful PDB, we drop the capacity abstraction: atoms may *not* occupy the same space, or pass through one another. This way, a PDB will capture interaction penalties arising from linear conflicts (when the optimal paths for two atoms are overlapping) between atoms within that pattern. Of course, it may happen be that the optimal generalized path and the actual optimal path are completely disjoint, but, in general, this is not the case.

The way we pre-compute and access our standard heuristic as a look-up table, as defined in Section 3.1.2, can be seen as a special case of PDB, where the pattern size is 1.

3.4.2 A Static Disjoint Pattern Database

Given the dimensions $w \times h$ of an Atomix board, a static pattern database of size k for Atomix would occupy $O((wh)^k)$ memory: all possible distributions of k atoms over the wh positions. Considering that we can store a PDB heuristic value in a single byte, and that the maximum board size ($w \times h$) of all instances in the standard testbed is 289, the maximum expected memory usage for a PDB with size k would be approximately 289^k bytes. Table 3.1 shows the expected memory usage for $k \in \{1, \ldots, 5\}$.

Table 3.1: Expected static PDB memory usages.

k	Memory usage
1	289 bytes
2	81 KB
3	23 MB
4	6.5 GB
5	1.8 TB

Source: the author.

We partition the n atoms into $\lfloor n/k \rfloor$ disjoint groups, and a single PDB is constructed for each k-pattern. Knowing that, and that there are instances with at most 32 atoms, it is feasible to construct static PDBs for Atomix with up to 3 atoms: in the most extreme situation, a set of disjoint PDBs for a single final state will require approximately 230 MB of memory ($\lfloor \frac{32}{3} \rfloor \times 23$ MB). If the number of atoms is not divisible by three, a single smaller PDB (of size 2 or 1) for the remaining atoms is constructed.

It is important to mention that, because Atomix allows multiple goal states, multiple sets of disjoint PDBs might be necessary. In this case, the minimum sum among all sets of disjoint

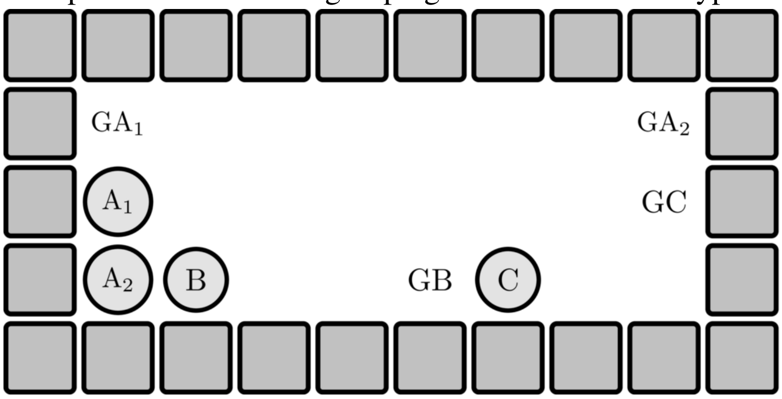
PDBs is taken as heuristic value. The maximum number of goal states for any instance is 64, so, using a *very* pessimistic estimate, we would need 14.7 GB of memory for the PDB, which is still acceptable, considering current main memory sizes.

Being a static PDB, as opposed to a dynamic one, the partition is pre-defined before constructing the PDB. Some partitions will yield overall better heuristic values than others, since some atom groups possess more linear conflicts than others. In our solution, we choose the initial patterns almost arbitrarily: atoms are grouped in alphabetical order, which gives a preference for grouping atoms with the same type in the same pattern.

This preference for grouping atoms with the same type is not unfounded. Consider the example in Figure 3.7. If we were to create two disjoint PDBs of size 2 for it, we would have the following partitions:

- A_1 + A_2 and B + C, yielding a heuristic of 6 (1 for A_1, 2 for A_2, 1 for B and 2 for C)
- A_1 + B and A_2 + C, yielding a heuristic of 5 (1 for A_1, 1 for A_2, 1 for B and 2 for C)
- A_1 + C and A_2 + B, yielding a heuristic of 5 (1 for A_1, 1 for A_2, 1 for B and 2 for C)

Figure 3.7: Example of the benefit of grouping atoms of the same type in a static PDB.



Source: the author.

Notice that, if we put A_1 and A_2 separately, both of them will choose to go to the closest goal, GA_1, which requires only one move. However, since only one of them may actually be placed there, we would lose information. In cases like this, where there is no interaction between atoms in the generalized version, the PDB would be even worse than the standard heuristic. This kind of situation arises very often in Atomix instances with many duplicate atoms. Therefore, in order to keep the PDB competitive and at least as good as the standard heuristic, we take the final heuristic to be the maximum between the standard heuristic and the PDB heuristic.

This very simple partitioning criterion does not guarantee that a good number of linear conflicts will be identified.

3.4.3 A Dynamically-Partitioned Pattern Database

For any given state in the state space, a static partition may not always offer the best heuristic possible among all possible atom partitions. In practice, the difference between the best and the worst partitions can be significant, and lead to poor heuristic values. It would be nice if we could, for any given state, always select the partition which maximizes the heuristic value. Dynamically-partitioned PDBs, a concept introduced by (FELNER; KORF; HANAN, 2004), are a way to achieve this.

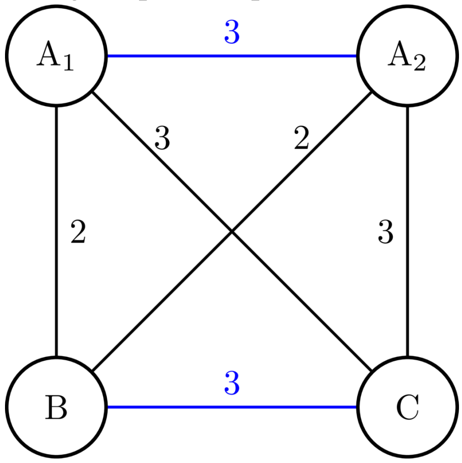
In a dynamic PDB, we store a table (called a k-atom database) which holds, for every possible atom pattern of size k, and for every possible sequence of k board positions, the number of generalized moves necessary to bring those k atoms to their final positions, with interactions between them. In other words, a k-atom database is a set of tuples $(i_1, \dots, i_k, P_1, \dots, P_k, d)$ where $i_1, \dots, i_k$ represent the atom indexes in the pattern, $P_1, \dots, P_k$ their possible positions, and d the number of moves necessary to bring the pattern to its final state. Since there are $\binom{n}{k}$ possible atom patterns of size k and $(w \times h)^k$ possible sequences of k board positions, to construct such PDB would require $O(\binom{n}{k}(w \times h)^k)$ memory. This is only feasible, under the constraints discussed in the previous section, for $k \leq 2$, for which it would require $\binom{32}{2}289^2 \approx 40$MB of memory[1]. In practice, this table can be computed in a matter of milliseconds.

Choosing $k = 2$ rather simplifies computing the heuristic value, which is computed for a state S as follows. We define a complete graph, where each vertex represents an atom. A vertex i is connected to every other vertex j by an edge of weight d, corresponding to the 2-atom database entry (i, j, S_i, S_j, d), where S_i and S_j are the positions of i and j in S. We then compute a *maximum weighted perfect matching* on this graph, which will select a set of edges such that every vertex connects to exactly one edge in the set, and such that the sum of edge weights is maximized. This can be done in $O(n^3)$ time (PAPADIMITRIOU; STEIGLITZ, 1998). In the special case where the number of vertices is odd (and a perfect matching is impossible), we add a "dummy" vertex which connects to every other vertex with weight equal to the minimum generalized distance between that atom and any of its possible final positions. For the maximum cost matching in our implementation, we used the Blossom V library, developed by Kolmogorov (2009).

Figure 3.8 shows the complete graph defined by the example on Figure 3.7. The edges in the maximum matching are marked in blue.

[1]For $k = 3$, it would require over 110GB of memory.

Figure 3.8: The graph representing the possible partitions of the example PDB on Figure 3.7.



Source: the author.

As discussed in the previous section, if there are multiple goal states, multiple PDBs are necessary, and the final heuristic value is the minimum heuristic obtained by performing a maximum matching on all the PDBs' partitions.

The major shortcoming of this approach is that a maximum cost perfect matching must be computed at every heuristic call. Although this is not a big problem when there is only one goal state, it can be very time-consuming in instances that have a large number of goal states: for every goal, there will be one PDB, and consequently one matching. One way to improve the performance would be by storing the matched edges together with the state representation and performing the matching only once every fixed number of neighboring moves; however, this would double the memory usage of a state.

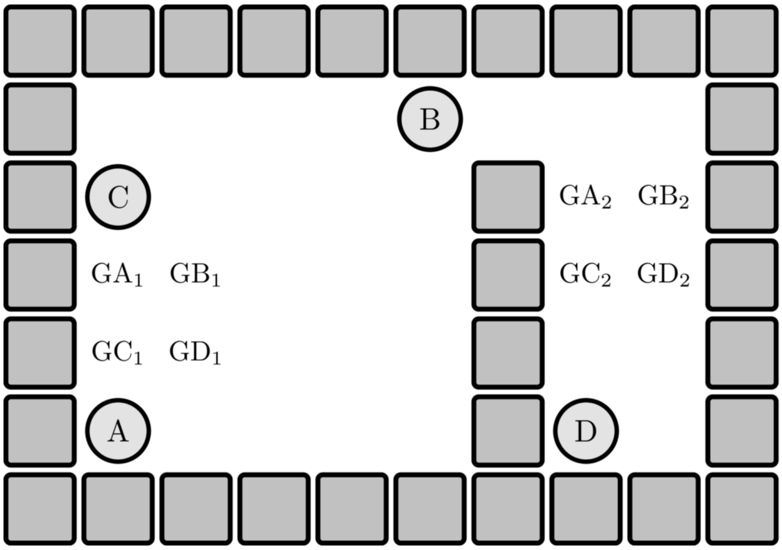
3.4.4 A Multiple Goal Dynamically-Partitioned Pattern Database

The motivation behind a multi-goal PDB is to try to reduce the time lost performing several matchings, by performing only one matching instead.

The main idea of a multi-goal PDB is to combine multiple PDBs with the same characteristic into a single, more generic, PDB. In the literature, Felner and Ofek (2007) use a multi-goal PDB to combine multiple states in the fringe of a perimeter search, and provide a heuristic that gives a lower bound on the distance of a state to any of the states on that fringe. In this work, we use a multiple goal PDB to combine the Atomix goal states into a single PDB.

A multi-goal PDB is represented by the exact same data structure as a single-goal dynamic PDB (i.e., a k-atom database), but is different in the way that it is built. To construct a single-goal PDB, we perform a BFS on every possible pattern, starting from the pattern atoms' positions in the goal state. If there are multiple goal states, multiple breadth-first-searches are required, since we need multiple single-goal PDBs. For the multi-goal PDB, we store a single k-atom database, which is computed with a BFS that, for every possible pattern, starts simultaneously from all possible goal positions. Notice that, when we have one goal state, a multi-goal PDB is the same as a single-goal PDB.

Figure 3.9: An example of when two patterns may choose to go to the different goal positions, on a multi-goal PDB.



Source: the author.

The major problem with a multi-goal PDB is that we lose information as different patterns may "choose to go" to different goal states. Consider the example on Figure 3.9, which has two goal states, one on the left and one on the right. For this example, we examine the partition AC + BD[2]. In a single-goal PDB we would have, for the first goal state, the heuristic $4(\text{AC}) + 5(\text{BD}) = 9$, and for the second goal state, the heuristic $6(\text{AC}) + 4(\text{BD}) = 10$, which yields a minimum of 9. However, when taking into account both goal states, we have that the shortest distance of AC to any of them is 4 (to GA_1 and GC_1, respectively), and the shortest distance for BD to any of them is also 4 (to GB_2 and GD_2, respectively), yielding a total of 8. In this case, we lose information, and, without linear conflicts, the heuristic would be even worse than the standard heuristic. The final heuristic value is taken to be the maximum between the PDB heuristic and the standard heuristic.

[2]This partition was not chosen arbitrarily: it is the one which yields the maximum heuristic, since A and C are the only atoms in linear conflict.

4 EXPERIMENTS AND RESULTS

4.1 Experimental Setup

4.1.1 Platform

The following tests were performed on a AMD FX-8150 Eight-Core Processor CPU with 32 GB of available memory. All tests were run with a time limit of one hour[1], and a memory limit of 22 GB. The programming language used for the implementation was C++, with the compiler GCC 4.7.3 and optimization flag -O3.

4.1.2 Instances

The standard set of instances contains 155 Atomix levels, with number of atoms ranging from 3 to 32, and board sizes ranging from 64 to 289. More details about the instances can be seen in Appendix A, including number of final states, number of free cells, and the length of the best known solution.

For presentation purposes, in this section, we separate the instances into similar-sized groups, based on the number of atoms n. The groups are shown in Table 4.1. We chose n as a grouping factor because, experimentally, it is the input parameter that has the most influence on the difficulty of solving the instance.

In this section, results such as average time, nodes expanded, and lower bounds for each group are computed using the *harmonic mean* over the instances that were solved, because of its stability regarding outliers. The *Total* row on each table contains the sum of times/nodes of all the instances that were solved, plus a penalty for every instance that was not solved by that method but which was solved by another method in the same table. The penalty for every unsolved instance is the smallest upper bound on that parameter that is a multiple of ten; this causes the total result to implicitly show the number of instances that were not solved in its first digits.

[1] With the exception of the tests described in Section 4.2, which were run for a time limit of 10 minutes, because of time constraints for the delivery of this manuscript.

Table 4.1: Instance groups used to present results.

n	# Instances in testbed
≤ 3	8
4	10
5	12
6	13
7	8
8	14
9	11
10 and 11	13
12	21
13 and 14	10
15	10
16	10
≥ 17	15
Total	155

Source: the author.

Table 4.2: A summary of the techinques presented and tested in this work.

Method	First proposed by	Applied to Atomix by
Multiple A*'s with One Final State	(HÜFFNER et al., 2001)	(HÜFFNER et al., 2001)
One A* with All Final States	this work	this work
Goal Count Tie-Breaking		this work
Fill Order Tie-Breaking	(PEREIRA; RITT; BURIOL, 2013)	this work
Number Realizable Paths Tie-Breaking	this work	this work
Fibonacci Heap Open List	(FREDMAN; TARJAN, 1987)	this work
Buckets Open List		this work
Static PDB	(KORF; FELNER, 2002)	this work
Dynamic PDB	(FELNER; KORF; HANAN, 2004)	this work
Multi-goal PDB	(FELNER; OFEK, 2007)	this work

Source: the author.

4.1.3 Techniques Tested

Table 4.2 shows the methods that were tested, and clarifies which methods were developed entirely in this work, or were proposed originally by other authors.

Table 4.3: Comparison between *One Final State* and *All Final States* heuristics.

Instances (n)	One Final State			All Final States		
	# Solved	Time(s)	Nodes Exp.	# Solved	Time(s)	Nodes Exp.
≤ 3	8/8	33.83	1305	8/8	19.51	146
$= 4$	10/10	30.70	12,204	10/10	18.74	3284
$= 5$	12/12	22.93	12,817	12/12	19.26	6002
$= 6$	13/13	26.54	52,185	13/13	22.20	42,001
$= 7$	6/8	31.55	676,157	6/8	28.11	299,065
$= 8$	9/14	30.35	3165	9/14	30.50	3826
$= 9$	3/11	31.91	43,610	3/11	28.61	26,960
$10 \leq n \leq 11$	2/13	36.29	2,451,595	2/13	39.45	2,044,228
$= 12$	0/21	0.00	0	0/21	0.00	0
$13 \leq n \leq 14$	0/10	0.00	0	0/10	0.00	0
$= 15$	1/10	27.19	5,082,501	1/10	19.22	1,449,440
$= 16$	1/10	18.08	87,079	1/10	16.61	58,335
≥ 17	0/15	0.00	0	0/15	0.00	0
Total	65	3657.67	679,552,757	65	2655.89	275,563,258

Source: the author.

4.1.4 Experimental Strategy

In every section of this chapter, we will test a set of related techniques described in Chapter 3, and select the best one. The selected technique will be incorporated into the final solver, and used in the experiments that follow. This assumes that the techniques implemented are somewhat orthogonal, e.g., choosing one tie-breaking rule will not affect too much the performance of the PDBs as opposed to another rule.

4.2 Test A: *One Final State* vs *All Final States* Heuristics

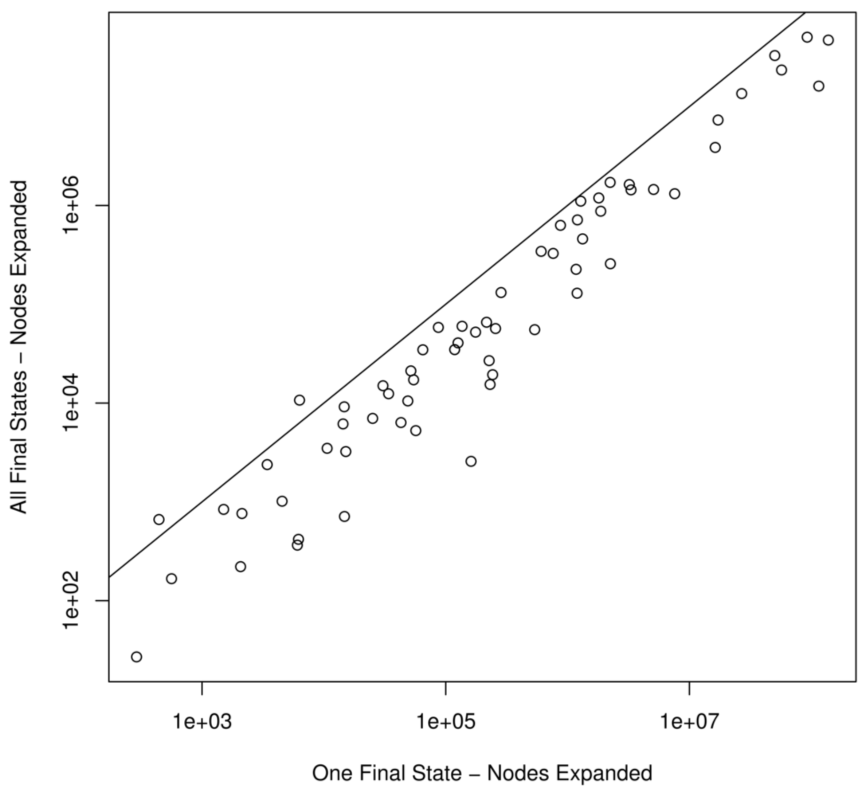
We conducted experiments to test which strategy described in Section 3.1.4 is better: using multiple independent A* runs with *one final state* at a time (OFS), or only one A* considering *all final states* (AFS). Because of time constraints, these tests were run for only 10 minutes, instead of one hour.

Table 4.3 shows the summarized results. The full results can be found in Appendix C. We can see that both versions solved the same number of instances (65). Because of the multiple A* runs, the One Final State version expanded a much larger number of nodes; however, we had originally expected this difference to be much larger. Both these results were a surprise to us: we expected OFS version to do a lot worse, and solve fewer instances.

We believe that these favorable results are due to OFS ending up pruning a large number of states whose f-values are larger than the move limit, and thus exploring a smaller portion of the search space. In particular, for harder instances, the time performance of both versions are similar, which could imply that OFS offers a more scalable solution as the problem difficulty increases.

Figure 4.1 shows the number of nodes expanded by AFS versus OFS. A quick glance at this graph shows a clear preference for AFS, as it expands much fewer nodes in almost all instances.

Figure 4.1: Comparison of nodes expanded between AFS and OFS.



Source: the author.

Considering that All Final States performed better, we decided to choose this version for our final solver. It remains to be studied whether further optimizations to OFS, such as combining it with PDBs or perimeter search, could possibly lead to an even better solver.

4.3 Test B: Tie-Breaking Techniques

Experiments were conducted to test the performance of the three tie-breaking rules described in Section 3.3. All experiments used the standard heuristic considering all final states and a bucket-based open list. The summarized results can be seen in Table 4.4, and the full results can be found in Appendix D.

By analyzing Table 4.4, we observe that all tie-breaking rules presented a good improvement on the version without tie-breaking, solving at least three extra instances. We can see that the NRP (number of realizable paths) rule solved 69 instances, as opposed to GC (goal count) and FO (fill order), which solved 70 instances each. The NRP rule took considerably more time, which was already expected, since it iterates over all the possible paths for every atom. With that in consideration, we conclude that this rule is clearly inferior to the other two.

FO and GC showed very similar results: both solved 70 instances, and had approximately performance in terms of time and expanded nodes, with a slight preference for GC. We argue that, in practice, the FO does not serve much purpose other than a simple goal count, only with weights. This argument was also supported by an informal test we performed using a "reverse" FO: instead of giving priority to nodes on the inside of the molecule, we prioritized nodes on the outside of the molecule. It was expected that, for representative instances such as marbles_14, shown in Section 3.3.3, the reversed FO would fare much worse than normal FO; however, to our surprise, it expanded 22949 nodes, as opposed to 22953 nodes with the normal FO. This kind of behavior was similar in several other instances.

Of the three tie-breaking rules, we declare GC to be the best, since it is simpler to implement, is more scalable than FO (as it requires fewer buckets for the open list) and is slightly faster than FO and much faster than NRP.

Figure 4.2 shows the number of nodes expanded by GC versus the version without tie-breaking, over the instances that both solutions solved. We can see that GC was able to reduce the number of nodes expanded in almost all instances.

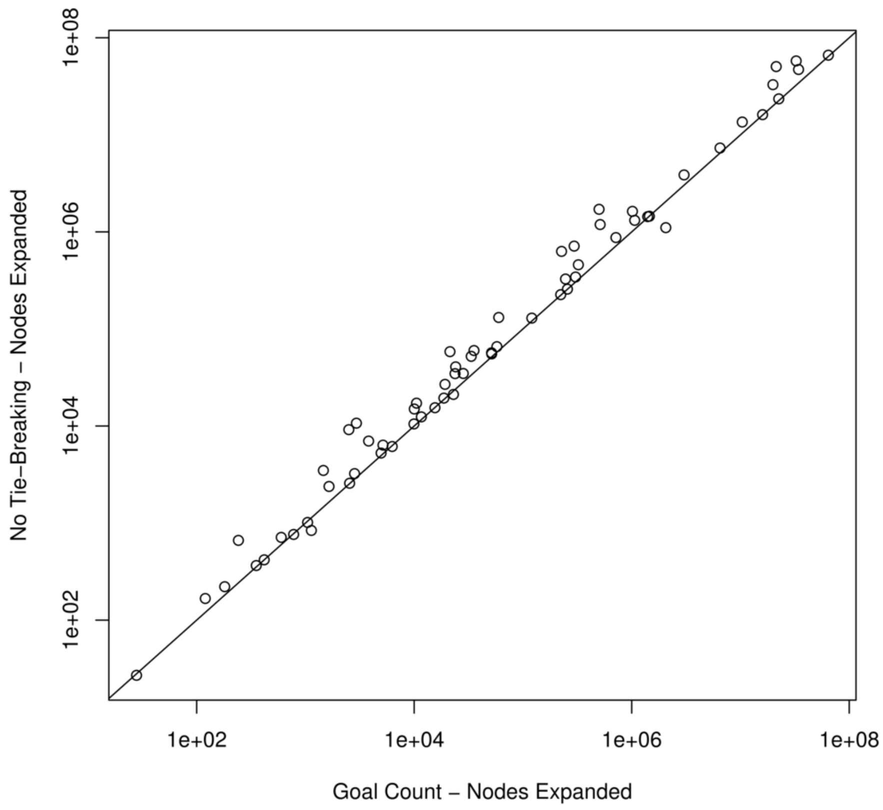
Table 4.4: Comparison between tie-breaking rules.

Instances (n)	No Tie-Break			GC		
	# Solved	Time (s)	Nodes Exp.	# Solved	Time(s)	Nodes Exp.
≤ 3	8/8	18.96	146	8/8	19.06	140
$= 4$	10/10	18.43	3284	10/10	18.53	3233
$= 5$	12/12	18.91	6002	12/12	19.42	5272
$= 6$	13/13	22.16	42,001	13/13	21.47	20,780
$= 7$	7/8	32.89	348,609	7/8	30.38	208,902
$= 8$	9/14	29.15	3826	10/14	31.98	1868
$= 9$	3/11	28.64	26,960	5/11	43.71	12,431
$10 \leq n \leq 11$	2/13	39.67	2,044,228	2/13	47.33	3,435,702
$= 12$	0/21	0.00	0	0/21	0.00	0
$13 \leq n \leq 14$	0/10	0.00	0	1/10	337.59	26,506,951
$= 15$	1/10	18.16	1,449,440	1/10	18.54	1,453,014
$= 16$	1/10	16.48	58,335	1/10	16.06	21,324
≥ 17	0/15	0.00	0	0/15	0.00	0
Total	66	43,482.33	4,333,304,472	70	4477.64	455,192,014

Instances (n)	NRP			FO		
	# Solved	Time(s)	Nodes Exp.	# Solved	Time(s)	Nodes Exp.
≤ 3	8/8	19.20	147	8/8	19.51	142
$= 4$	10/10	18.54	2750	10/10	18.66	3307
$= 5$	12/12	19.12	4354	12/12	19.69	5882
$= 6$	13/13	22.97	22,827	13/13	22.30	21,732
$= 7$	7/8	38.49	255,939	7/8	31.24	234,014
$= 8$	10/14	33.41	2443	10/14	32.26	1535
$= 9$	4/11	38.98	9933	5/11	44.93	12,345
$10 \leq n \leq 11$	2/13	63.89	3,142,103	2/13	47.84	3,435,690
$= 12$	0/21	0.00	0	0/21	0.00	0
$13 \leq n \leq 14$	1/10	762.59	26,516,710	1/10	343.97	26,506,947
$= 15$	1/10	19.98	1,447,183	1/10	18.65	1,441,610
$= 16$	1/10	16.35	21,324	1/10	16.20	21,324
≥ 17	0/15	0.00	0	0/15	0.00	0
Total	69	18,329.68	1,429,385,324	70	4828.03	468,542,161

Source: the author.

Figure 4.2: Comparison of nodes expanded between GC and the version without tie-breaking.



Source: the author.

4.4 Test C: A* Open List Implementations

We conducted two experiments to test the time performance of the bucket-based open list versus the Fibonacci heap-based open list. Both experiments used the standard heuristic considering all final states, and break ties by goal count. For the Fibonacci heap, we used the implementation available with the Boost C++ library (BOOST, 2015). Table 4.5 shows the summarized results for the open list implementations test.

An apparent time difference shown in favor of the Fibonacci heap may lead one to believe that, although that variant solves fewer instances, it is faster. This is not true. The size of the pre-allocated states table for the Fibonacci heap had to be reduced, since the Fibonacci heap implementation we used takes up a considerable amount of memory, and so pre-allocating the states table takes less time. For the buckets version, the pre-allocation of the states table takes approximately 18 seconds, as opposed to 8 seconds for the Fibonacci heap version. This

Table 4.5: Comparison between Fibonacci heap and bucket-based open list.

Instances (n)	Buckets		Fibonacci Heap	
	# Solved	Time(s)	# Solved	Time(s)
≤ 3	8/8	19.06	8/8	8.08
$= 4$	10/10	18.53	10/10	8.31
$= 5$	12/12	19.42	12/12	8.83
$= 6$	13/13	21.47	13/13	11.29
$= 7$	7/8	30.38	6/8	17.23
$= 8$	10/14	31.98	9/14	17.04
$= 9$	5/11	43.71	3/11	15.25
$10 \leq n \leq 11$	2/13	47.33	2/13	62.39
$= 12$	0/21	0.00	0/21	0.00
$13 \leq n \leq 14$	1/10	337.59	0/10	0.00
$= 15$	1/10	18.54	1/10	14.04
$= 16$	1/10	16.06	1/10	9.06
≥ 17	0/15	0.00	0/15	0.00
Total	70	4477.64	65	53,280.97

Source: the author.

constant time overhead gives a disadvantage to the buckets for easier instances ($n \leq 6$), but is compensated by performance improvements in harder instances.

Having solved 70 instances as opposed to the 65 instances solved by the Fibonacci heap, we can declare the bucket-based open list a clear winner. The main reason is that it is able to solve 5 more instances than the Fibonacci heap, and is able to generate significantly more nodes before it hits the memory limit.

The full results can be found in Appendix B. Observing the results shown in the appendix, we can notice a slight difference between the number of nodes expanded on solved instances: this is due to the difference in the open list's *update* methods, where the Fibonacci heap actually updates the state's f-value and the buckets version re-inserts the state. For the unsolved instances, it is interesting to observe the differences in nodes generated, which are due to the Fibonacci heap running out of memory much more quickly.

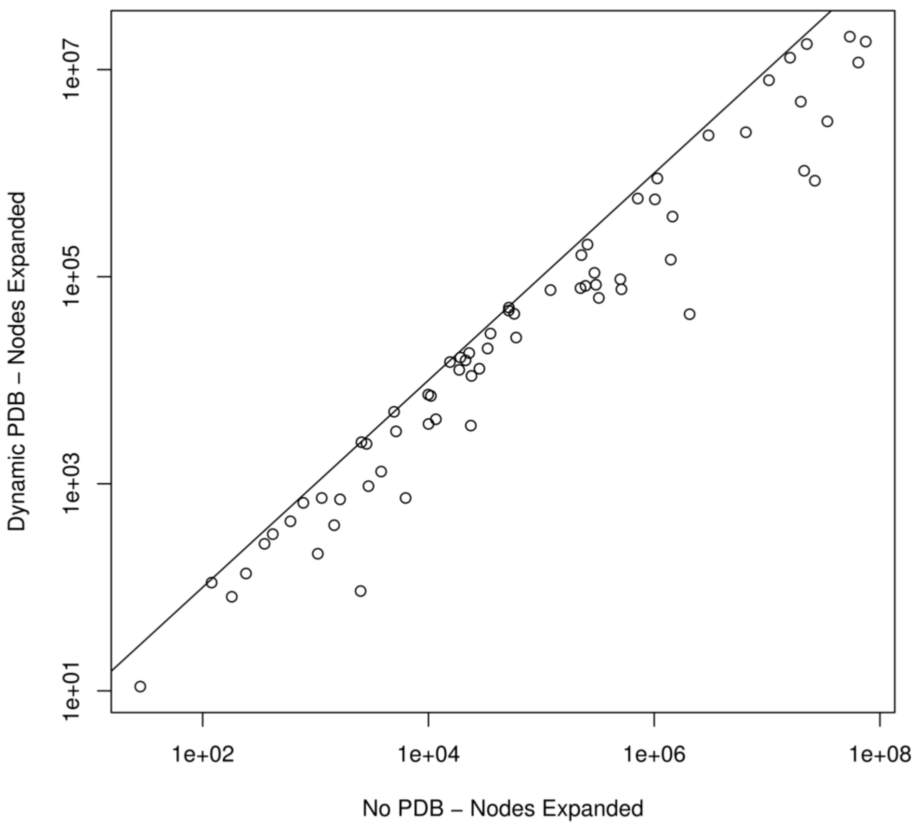
4.5 Test D: Pattern Databases

The summarized results for the PDB experiments can be found in Table 4.6. The full results are found in Appendix E. We can see that all three PDB variants have improved the number of instances solved by at least two. In particular, the static PDB and the multi-goal PDB solved the most number of instances, 73. Out of those two, the static PDB had the best time performance, with a run-time of approximately 4 times faster than the multi-goal PDB.

We can attribute this to the time cost of performing a maximum matching operation at every heuristic call, on the multi-goal PDB.

It can be observed that the static PDB expanded, on average, 68% more nodes than the multi-goal PDB. The results show that, although they might take longer to compute, both the multi-goal and the dynamic PDBs provide a more powerful heuristic than the static PDB. Figure 4.4 shows the number of nodes expanded on the static versus dynamic PDBs, considering the instances that both solutions solved. Although the number of expanded nodes is very similar for a large portion of the instances, we see that, in many cases, the dynamic PDB expanded at least one order of magnitude fewer nodes. Figure 4.3 compares the version without PDB against the dynamic PDB.

Figure 4.3: Comparison of nodes expanded between the version without PDB and the dynamic PDB.



Source: the author.

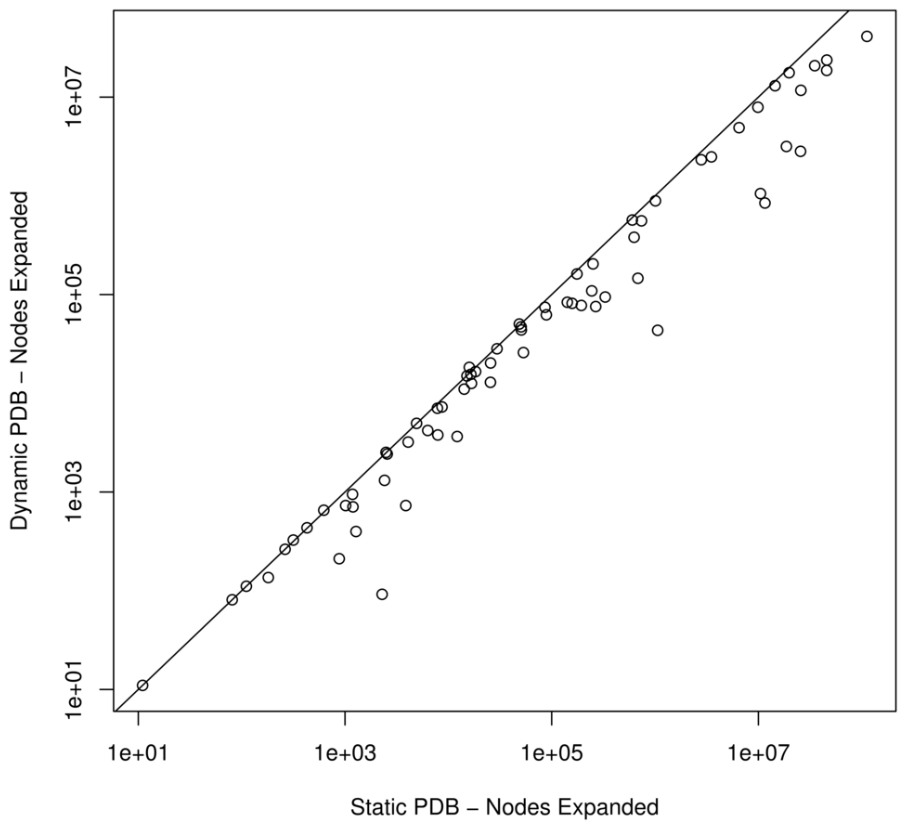
Table 4.6: Comparison between PDB methods.

Instances (n)	No PDB			Static PDB		
	# Solved	Time(s)	Nodes Exp.	# Solved	Time(s)	Nodes Exp.
≤ 3	8/8	19.06	140	8/8	21.74	65
$= 4$	10/10	18.53	3233	10/10	24.77	2692
$= 5$	12/12	19.42	5272	12/12	20.21	4169
$= 6$	13/13	21.47	20,780	13/13	24.37	10,568
$= 7$	7/8	30.38	208,902	8/8	39.01	192,080
$= 8$	10/14	31.98	1868	10/14	29.79	1386
$= 9$	5/11	43.71	12,431	6/11	46.70	13,366
$10 \leq n \leq 11$	2/13	47.33	3,435,702	2/13	40.15	1,915,480
$= 12$	0/21	0.00	0	1/21	709.93	25,582,015
$13 \leq n \leq 14$	1/10	337.59	26,506,951	1/10	160.77	11,574,396
$= 15$	1/10	18.54	1,453,014	1/10	16.89	626,928
$= 16$	1/10	16.06	21,324	1/10	16.39	16,508
≥ 17	0/15	0.00	0	0/15	0.00	0
Total	70	34,477.64	3,455,192,014	73	5827.16	458,051,038

Instances (n)	Dynamic PDB			Multi-Goal PDB		
	# Solved	Time(s)	Nodes Exp.	# Solved	Time(s)	Nodes Exp.
≤ 3	8/8	19.36	65	8/8	19.43	126
$= 4$	10/10	21.60	1355	10/10	19.13	2474
$= 5$	12/12	22.59	1820	12/12	20.28	2539
$= 6$	13/13	30.80	6949	13/13	26.43	10,132
$= 7$	7/8	67.04	131,032	8/8	59.03	181,815
$= 8$	10/14	37.01	979	10/14	37.91	994
$= 9$	5/11	57.85	459	5/11	55.72	724
$10 \leq n \leq 11$	3/13	65.89	129,247	3/13	67.63	129,247
$= 12$	1/21	3360.40	2,823,005	1/21	2650.41	13,447,435
$13 \leq n \leq 14$	1/10	212.09	845,399	1/10	216.52	845,399
$= 15$	1/10	30.14	380,647	1/10	30.53	380,647
$= 16$	1/10	17.35	15,565	1/10	17.85	15,565
≥ 17	0/15	0.00	0	0/15	0.00	0
Total	72	124,033.14	1,192,314,386	73	23,210.46	272,844,946

Source: the author.

Figure 4.4: Comparison of nodes expanded between static and dynamic PDB.



Source: the author.

Considering the results exposed in this section, we decide to choose the static PDB for our final solver: although it expands more nodes, out of the three PDBs, it is the one which solved the most instances in less time.

4.6 Analysis of the Heuristics' Quality

In this section, we compare the lower bounds obtained by different heuristic functions. Table 4.7 shows the average heuristic values for the initial state (relative to the best known lower bounds found by the static PDB) obtained by the standard heuristic, the three PDB versions, the generalized A* solution (which takes into account the interactions between all atoms)[2], and the best known lower bounds. For these results, the arithmetic mean was used. The full results can be found in Appendix F.

[2]This computation was made by running an A* with the difference that, instead of regular moves, generalized moves with capacity constraints were used. A static PDB was used as heuristic.

Table 4.7: Comparison of the initial heuristic of various methods.

Instances (n)	Group Size	Standard Heuristic	Static PDB ($k=3$)	Dynamic PDB ($k=2$)	Multi-Goal PDB ($k=2$)	Generalized A*	Best LB
≤ 3	8	0.52	0.53	0.53	0.52	0.53	1.00
$= 4$	10	0.56	0.57	0.59	0.57	0.61	1.00
$= 5$	12	0.70	0.71	0.73	0.71	0.74	1.00
$= 6$	13	0.68	0.69	0.71	0.69	0.72	1.00
$= 7$	8	0.65	0.66	0.66	0.65	0.68	1.00
$= 8$	14	0.73	0.74	0.76	0.74	0.78	1.00
$= 9$	11	0.70	0.72	0.75	0.73	0.77	1.00
$10 \leq n \leq 11$	13	0.79	0.81	0.82	0.81	0.84	1.00
$= 12$	21	0.86	0.87	0.88	0.87	0.90	1.00
$13 \leq n \leq 14$	10	0.87	0.88	0.91	0.89	0.94	1.00
$= 15$	10	0.81	0.83	0.86	0.86	0.95	1.00
$= 16$	10	0.84	0.85	0.87	0.87	0.96	1.00
≥ 17	15	0.91	0.92	0.95	0.95	0.98	1.00
Average		0.76	0.77	0.79	0.78	0.82	1.00

Source: the author

We observe in Table 4.7 that the average values of the initial heuristic and all the PDBs are very similar, even though the PDBs lead to good in the overall performance of our solution. This hints that even very small improvements to our heuristic function can lead to great improvements in terms of nodes expanded.

We also notice that, even though the static PDB uses a pattern of size $k = 3$ atoms, it is slightly worse than the dynamic and multi-Goal PDBs, that use a pattern of $k = 2$ atoms. This is because both the dynamic and multi-goal PDBs compute the heuristic through a maximum matching operation, and so are always able to select the best partition. Comparing the dynamic and multi-goal PDBs, it was already expected that the dynamic would be better, since the multi-goal PDB is more generalized, as it groups several goal positions into one.

Since the generalized A* captures all linear conflicts, it represents an upper bound on the quality of any PDB we might use. Comparing our best PDB solution with the generalized A*, we can see that they are not that different. The difference between them indicates the amount of information we lose by not capturing some linear conflicts with the PDB.

We can observe an average difference of over 23% between the generalized A* solution and the best lower bounds found so far. This difference accounts for the times when an atom has to deviate from its optimal path in order to provide support for another atom. In other words, it accounts for the sliding property, that we completely abstract in the generalized movement.

Table 4.8: Comparison between our final solution and the implementation by Hüffner et al. (2001).

Instances (n)	Hüffner et al. (2001)			Our Solution		
	# Solved	Time(s)	Nodes Exp.	# Solved	Time(s)	Nodes Exp.
≤ 3	8/8	76.39	1379	8/8	21.74	65
$= 4$	10/10	63.32	8558	10/10	24.77	2692
$= 5$	12/12	21.73	20,319	12/12	20.21	4169
$= 6$	13/13	24.39	86,859	13/13	24.37	10,568
$= 7$	8/8	39.42	1,345,025	8/8	39.01	192,080
$= 8$	11/14	21.16	11,263	10/14	29.79	1386
$= 9$	7/11	43.42	74,267	6/11	46.70	13,366
$10 \leq n \leq 11$	3/13	22.93	7,395,077	2/13	40.15	1,915,480
$= 12$	1/21	2427.92	302,608,420	1/21	709.93	25,582,015
$13 \leq n \leq 14$	1/10	1968.19	187,441,572	1/10	160.77	11,574,396
$= 15$	1/10	33.44	6,009,587	1/10	16.89	626,928
$= 16$	2/10	1155.94	176,592	1/10	16.39	16,508
≥ 17	0/15	0.00	0	0/15	0.00	0
Total	77	22,597.49	6,746,746,521	73	45,827.16	4,458,051,038

4.7 Final Solver

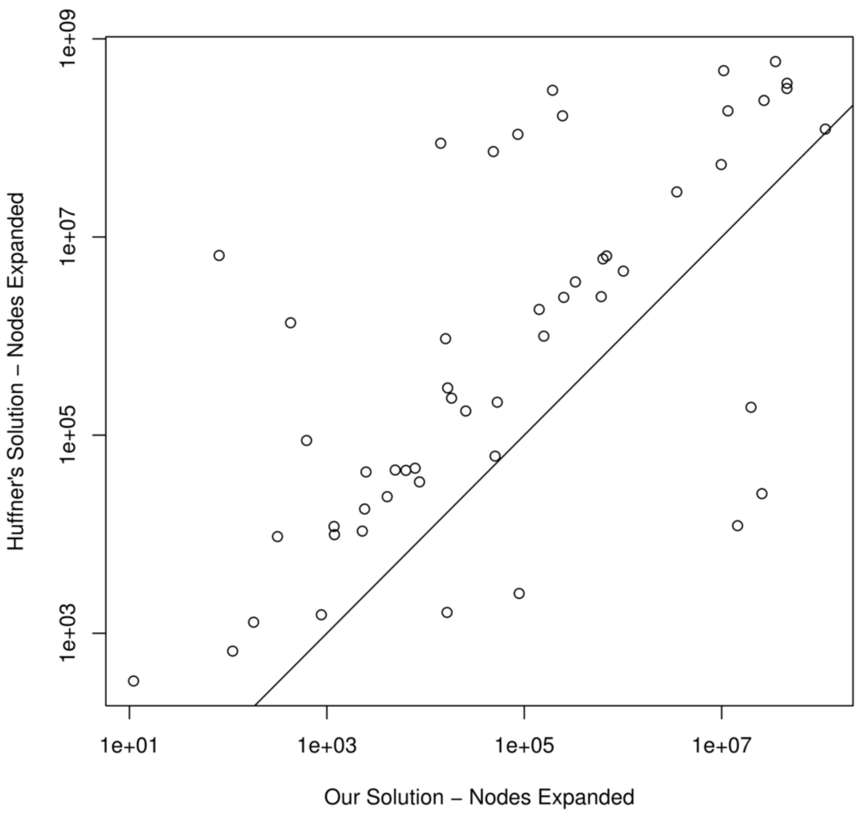
After the experiments conducted in this chapter, we can propose our best final solver, with the following characteristics:

- One A* guided by *All Final States*.
- A *buckets*-based open list.
- *Goal count* tie-breaking.
- A *static PDB* of size 3 as heuristic function.

Table 4.8 compares our solution with the implementation made available from Hüffner et al. (2001). Unfortunately, even using all techniques described in this work, we were still not able to outperform the best solution in the literature, having solved only 73 instances, compared to the 77 instances solved by Hüffner et al. (2001).

However, it can be seen that, even though our implementation did worse in terms of time performance and number of instances solved, it requires a significantly smaller number of node expansions before a solution is found. This is evident in Figure 4.5, which compares the number of nodes expanded of the two solutions, for the instances that were solved by both. Of course, most of this difference comes from the fact that Hüffner et al. (2001)'s solution is based on multiple independent A*'s (or what we call *One Final State*), but it is also due to our more powerful heuristic and tie-breaking criterion. Additionally, we believe that the difference in terms of time performance in favor of Hüffner et al. (2001) is a matter of implementation: their solution uses several low-level context-specific code optimizations, especially regarding memory usage, such as recompiling the code to utilize 8 or 16-bit integers to represent positions, according to the instance board size.

Figure 4.5: Comparison of nodes expanded between our solution and Hüffner et al. (2001).



Source: the author.

One important thing to note is that our solution, which was based on A*, hits the memory barrier too quickly: even though the tests were run for one hour, for most instances the algorithm uses all memory available in less than 10 minutes. This hints that a solution based on a memory-efficient algorithm such as IDA* may be a good option, especially considering the improvements made to the heuristic function. Unfortunately, because of time constraints, we have not tested this option experimentally.

5 CONCLUSION AND FUTURE WORK

In this work, we have studied heuristic search methods to solve Atomix optimally. We have surveyed some of the most important techniques used in state-of-the-art heuristic search, and applied some of them in practice.

The standard heuristic function we presented in this work was based on the *generalized moves* concept, proposed by Hüffner et al. (2001). Based on the standard heuristic, we showed that, for our implementation, an approach which runs a single A* search considering all final states performs better than one which runs multiple A* searches considering one final state each, both in terms of time and nodes.

The three tie-breaking rules we proposed in this work have shown improvements of our solution, as opposed to not using tie-breaking at all. We believe that further research on this topic would be relevant for Atomix, as tie-breaking becomes more important the more powerful the heuristic becomes. It would be interesting to try to combine more than one tie-breaking rule, to better choose between states that have exactly the same f-value and goal count.

Three PDB strategies have been presented: a static PDB, a dynamic PDB and a multi-goal PDB. Even though the dynamic and multi-goal PDBs offer better lower bounds that lead to fewer node expansions, the static PDB has better time performance. Both the dynamic and multi-goal PDBs time performance could be improved by reducing the time cost of computing a max-matching. One interesting way to achieve this could be to find the maximum matching via heuristic methods: even if the absolute maximum is not found, any matching is still admissible. Another alternative would be to implement a max-matching version that is specifically tailored for our purposes, instead of using a generic implementation. However, the usefulness of those PDBs is still limited by the quality of the heuristic function.

Analyzing the quality of the heuristic functions, we conclude that great improvements could be made by giving some sort of penalty (i.e., increasing the h-value) when an atom makes an illegal stop in a generalized move. Unfortunately, this has shown to be not that simple.

We have also observed that the A* algorithm tends to use all available memory very quickly. In the future, we believe that an attempt on memory-efficient algorithms, such as IDA*, combined with the PDB heuristics we developed, could be relevant, because it takes full advantage of the available time. Also, because of IDA*'s very small memory usage, we could possibly use the remaining available memory to construct more powerful PDBs.

Finally, even though our solution did not solve more instances than the best solution in the literature, we have argued that the heuristic functions and tie-breaking methods applied in this work are able reduce node expansions significantly. With that in mind, we think that our contributions for Atomix are valid.

REFERENCES

BOOST. **Boost C++ libraries**. 2015. Available from Internet: <http://www.boost.org/>.

CULBERSON, J. Sokoban is PSPACE-complete. In: **Proceedings in Informatics**. [S.l.: s.n.], 1999. v. 4, p. 65–76.

CULBERSON, J. C.; SCHAEFFER, J. Searching with pattern databases. In: **Advances in Artifical Intelligence**. [S.l.]: Springer, 1996. p. 402–416.

DEMAINE, E. **Playing Games with Algorithms**. [S.l.], 2001.

DEMAINE, E. D.; HEARN, R. A.; HOFFMANN, M. Push-2-f is PSPACE-complete. In: **CCCG**. [S.l.: s.n.], 2002. p. 31–35.

DEMAINE, E. D.; HOFFMANN, M.; HOLZERC, M. Pushpush-k is PSPACE-complete. Citeseer, 2004.

DILLENBURG, J. F.; NELSON, P. C. Perimeter search. **Artificial Intelligence**, Elsevier, v. 65, n. 1, p. 165–178, 1994.

EDELKAMP, S.; SCHROEDL, S. **Heuristic search: theory and applications**. [S.l.]: Elsevier, 2011.

FELNER, A.; KORF, R. E.; HANAN, S. Additive pattern database heuristics. **J. Artif. Intell. Res.**, v. 22, p. 279–318, 2004.

FELNER, A.; OFEK, N. Combining perimeter search and pattern database abstractions. In: **Abstraction, Reformulation, and Approximation**. [S.l.]: Springer, 2007. p. 155–168.

FLAKE, G. W.; BAUM, E. B. Rush hour is PSPACE-complete, or "why you should generously tip parking lot attendants". **Theoretical Computer Science**, Elsevier, v. 270, n. 1, p. 895–911, 2002.

FLOYD, R. W. Algorithm 97: shortest path. **Communications of the ACM**, ACM, v. 5, n. 6, p. 345, 1962.

FREDMAN, M. L.; TARJAN, R. E. Fibonacci heaps and their uses in improved network optimization algorithms. **Journal of the ACM**, ACM, v. 34, n. 3, p. 596–615, 1987.

HART, P. E.; NILSSON, N. J.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **Systems Science and Cybernetics, IEEE Transactions on**, IEEE, v. 4, n. 2, p. 100–107, 1968.

HOLTE, R. C.; GRAJKOWSKI, J.; TANNER, B. Hierarchical heuristic search revisited. In: **Abstraction, Reformulation and Approximation**. [S.l.]: Springer, 2005. p. 121–133.

HOLTE, R. C. et al. Multiple pattern databases. In: **ICAPS**. [S.l.: s.n.], 2004. p. 122–131.

HOLTE, R. C. et al. Hierarchical a*: Searching abstraction hierarchies efficiently. In: CITESEER. **AAAI/IAAI, Vol. 1**. [S.l.], 1996. p. 530–535.

HOLZER, M.; SCHWOON, S. Assembling molecules in atomix is hard. **Theoretical computer science**, Elsevier, v. 313, n. 3, p. 447–462, 2004.

HÜFFNER, F. et al. Finding optimal solutions to atomix. In: **KI 2001: Advances in Artificial Intelligence**. [S.l.]: Springer, 2001. p. 229–243.

JENKINS, B. **Spookyhash: a 128-bit noncryptographic hash**. 2012.

KOLMOGOROV, V. Blossom v: a new implementation of a minimum cost perfect matching algorithm. **Mathematical Programming Computation**, Springer, v. 1, n. 1, p. 43–67, 2009.

KORF, R. E. Depth-first iterative-deepening: An optimal admissible tree search. **Artificial intelligence**, Elsevier, v. 27, n. 1, p. 97–109, 1985.

KORF, R. E. Finding optimal solutions to rubik's cube using pattern databases. In: **AAAI/IAAI**. [S.l.: s.n.], 1997. p. 700–705.

KORF, R. E.; FELNER, A. Disjoint pattern database heuristics. **Artificial intelligence**, Elsevier, v. 134, n. 1, p. 9–22, 2002.

MUNKRES, J. Algorithms for the assignment and transportation problems. **Journal of the Society for Industrial and Applied Mathematics**, SIAM, v. 5, n. 1, p. 32–38, 1957.

PAPADIMITRIOU, C. H.; STEIGLITZ, K. **Combinatorial optimization: algorithms and complexity**. [S.l.]: Courier Corporation, 1998.

PEREIRA, A. G.; RITT, M. R. P.; BURIOL, L. S. Finding optimal solutions to sokoban using instance dependent pattern databases. In: **Sixth Annual Symposium on Combinatorial Search**. [S.l.: s.n.], 2013.

PEREIRA, A. G.; RITT, M. R. P.; BURIOL, L. S. Pull and pushpull are PSPACE-complete. In: **Private Communication.** [S.l.: s.n.], 2016.

RATNER, D.; WARMUTH, M. The (n2-1)-puzzle and related relocation problems. **Journal of Symbolic Computation**, Elsevier, v. 10, n. 2, p. 111–137, 1990.

RITT, M. Motion planning with pull moves. **arXiv preprint arXiv:1008.2952**, 2010.

ZOBRIST, A. L. A new hashing method with application for game playing. **ICCA journal**, v. 13, n. 2, p. 69–73, 1970.

APPENDIX A — INSTANCE DATA

Table A.1: Instance Data 1/4

Instance	n	# Final States	$w \times h$	# Free Positions	Solution Length
adrien_01	3	54	198	77	=7
atomix_01	3	17	156	45	=13
kai_01	3	9	144	39	=9
katomic_01	3	23	143	49	=15
katomic_36	3	21	195	52	=9
marbles_04	3	18	144	49	=22
marbles_13	3	3	100	27	=18
unitopia_01	3	41	180	68	=11
adrienl_05	4	64	260	131	=12
atomix_23	4	20	224	82	=10
atomix_26	4	17	240	102	=14
kai_06	4	16	195	72	=14
kai_19	4	21	210	73	=19
katomic_20	4	16	225	83	=18
katomic_23	4	32	225	93	=18
marbles_01	4	2	100	21	=11
marbles_03	4	5	143	39	=22
unitopia_02	4	5	180	62	=22
adrien_02	5	13	198	81	=17
atomix_02	5	6	208	61	=21
atomix_11	5	14	225	83	=14
kai_02	5	2	182	54	=24
kai_11	5	7	182	52	=15
katomic_02	5	10	195	64	=27
katomic_10	5	8	225	84	=19
katomic_57	5	3	182	45	=21
marbles_02	5	5	132	38	=15
marbles_05	5	2	121	34	=25
marbles_06	5	3	168	41	=14
unitopia_03	5	12	180	56	=16
adrien_03	6	31	198	75	=12
adrien_06	6	15	198	81	=15
atomix_03	6	4	225	65	=16
atomix_04	6	2	195	60	=23
kai_03	6	4	225	65	=16
katomic_03	6	4	210	66	=20
katomic_04	6	8	169	45	=23
katomic_58	6	3	169	60	=17

Source: the author.

Table A.2: Instance Data 2/4

Instance	n	# Final States	$w \times h$	# Free Positions	Solution Length
marbles_08	6	3	144	48	=23
marbles_12	6	3	126	40	=28
marbles_14	6	1	156	27	=22
unitopia_04	6	5	180	59	=20
unitopia_05	6	7	180	68	=20
adrienl_01	7	26	260	122	=20
adrienl_03	7	43	260	133	=22
atomix_09	7	1	156	49	=20
katomic_08	7	1	169	61	=26
katomic_26	7	3	225	81	=36
katomic_46	7	3	195	53	=24
katomic_60	7	4	169	54	=19
unitopia_08	7	4	180	59	=23
adrienl_02	8	7	260	108	≥31
atomix_06	8	4	64	16	=13
atomix_13	8	1	156	49	=28
atomix_18	8	4	64	16	=13
atomix_22	8	3	240	85	≥26
atomix_29	8	2	240	79	=22
atomix_30	8	4	64	16	=13
kai_05	8	2	182	67	=27
kai_17	8	3	225	80	=23
katomic_11	8	4	169	71	=23
katomic_19	8	2	255	103	≥31
katomic_31	8	2	169	54	=29
marbles_11	8	1	225	59	=28
unitopia_10	8	2	180	57	≥39
adrienl_04	9	1	198	68	≥34
atomix_05	9	2	240	80	≥36
atomix_07	9	1	225	79	=27
atomix_12	9	4	64	16	=14
atomix_16	9	2	210	73	≥27
katomic_05	9	2	182	52	=27
katomic_06	9	1	196	50	=27
katomic_14	9	1	225	85	≥28
katomic_32	9	5	121	25	=19
katomic_38	9	1	225	89	≥34
unitopia_06	9	2	180	61	=31
adrien_04	10	16	260	119	≥25

Source: the author.

Table A.3: Instance Data 3/4

Instance	n	# Final States	$w \times h$	# Free Positions	Solution Length
adrien_05	10	13	240	108	≥26
atomix_10	10	2	210	82	≥29
atomix_28	10	1	182	65	=29
kai_09	10	1	238	78	≥35
katomic_09	10	1	225	81	≥31
katomic_25	10	1	195	73	≥35
katomic_33	10	4	225	72	≥50
katomic_35	10	1	143	47	≥34
katomic_61	10	2	165	58	≥53
unitopia_07	10	1	180	60	≥34
katomic_47	11	1	169	60	=29
katomic_66	11	1	225	78	≥31
atomix_08	12	1	210	81	≥34
atomix_14	12	1	224	94	≥35
atomix_15	12	1	210	89	≥36
atomix_21	12	2	240	108	≥31
kai_07	12	1	210	81	≥33
kai_08	12	1	225	72	≥36
kai_18	12	1	240	89	≥34
kai_20	12	1	256	90	≥38
kai_22	12	1	225	85	≥33
katomic_07	12	8	225	68	=24
katomic_12	12	8	225	93	≥36
katomic_13	12	1	225	87	≥41
katomic_18	12	4	225	90	≥46
katomic_27	12	1	225	81	≥46
katomic_28	12	1	225	73	≥37
katomic_42	12	1	169	51	≥34
katomic_62	12	1	289	96	≥51
katomic_63	12	2	195	70	≥41
katomic_67	12	2	169	54	≥32
marbles_15	12	1	225	62	≥37
unitopia_09	12	2	180	60	≥43
katomic_34	13	1	225	96	≥36
atomix_20	14	1	195	68	=29
atomix_25	14	2	240	101	≥37
kai_14	14	2	240	76	≥40
kai_21	14	1	289	101	≥42
kai_24	14	1	272	79	≥40

Source: the author.

Table A.4: Instance Data 4/4

Instance	n	# Final States	$w \times h$	# Free Positions	Solution Length
kai_25	14	1	272	95	≥33
katomic_17	14	2	195	73	≥31
katomic_22	14	4	225	90	≥32
katomic_45	14	1	210	63	≥39
15-puzzle	15	1	64	16	=34
atomix_17	15	1	224	90	≥36
atomix_19	15	1	210	65	≥28
kai_12	15	1	240	97	≥35
katomic_15	15	1	225	87	≥35
katomic_16	15	1	225	69	≥42
katomic_29	15	1	225	71	≥57
katomic_41	15	4	225	81	≥34
katomic_55	15	1	225	83	≥47
katomic_56	15	1	225	79	≥49
atomix_24	16	1	144	32	≥29
kai_28	16	1	289	106	≥46
katomic_21	16	1	196	32	≥26
katomic_40	16	1	169	61	≥56
katomic_51	16	1	225	82	≥39
katomic_53	16	2	99	35	≥25
katomic_54	16	1	225	81	≥35
katomic_59	16	4	169	49	≥27
katomic_64	16	2	255	98	≥53
marbles_10	16	1	80	20	=24
katomic_39	17	1	225	77	≥47
katomic_48	17	1	225	79	≥57
katomic_50	17	2	169	61	≥42
katomic_65	17	1	121	25	≥31
katomic_49	18	1	195	71	≥45
kai_27	19	1	289	112	≥60
katomic_52	19	1	225	84	≥53
atomix_27	20	1	240	84	≥45
katomic_24	20	10	255	115	≥36
kai_29	21	1	272	87	≥63
katomic_30	21	1	225	72	≥51
katomic_44	21	1	210	65	≥48
katomic_37	24	1	289	134	≥54
katomic_43	26	1	225	86	≥65
marbles_20	32	1	100	36	≥37

Source: the author.

APPENDIX B — FIBONACCI HEAP VS. BUCKETS EXPERIMENT RESULTS

Table B.1: Fibonacci Heap vs. Buckets Experiment 1/4

Instance	n	Buckets			Fibonacci Heap		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
adrien_01	3	=7	19	28	=7	8	14
atomix_01	3	=13	19	418	=13	8	473
kai_01	3	=9	19	120	=9	8	146
katomic_01	3	=15	18	599	=15	8	598
katomic_36	3	=9	19	353	=9	8	356
marbles_04	3	=22	18	2548	=22	8	2552
marbles_13	3	=18	19	4963	=18	8	4799
unitopia_01	3	=11	18	181	=11	8	182
adrienl_05	4	=12	18	18,746	=12	8	18,693
atomix_23	4	=10	18	1047	=10	8	1273
atomix_26	4	=14	18	9948	=14	8	10,206
kai_06	4	=14	18	5165	=14	8	5173
kai_19	4	=19	18	19,193	=19	8	19,135
katomic_20	4	=18	18	2829	=18	8	2835
katomic_23	4	=18	18	15,519	=18	8	17,021
marbles_01	4	=11	18	779	=11	8	767
marbles_03	4	=22	18	51,583	=22	8	51,595
unitopia_02	4	=22	18	57,583	=22	8	57,598
adrien_02	5	=17	20	256,410	=17	10	256,195
atomix_02	5	=21	19	10,509	=21	8	10,338
atomix_11	5	=14	19	3811	=14	8	3590
kai_02	5	=24	19	246,130	=24	9	252,201
kai_11	5	=15	19	11,640	=15	8	13,030
katomic_02	5	=27	19	120,615	=27	9	120,481
katomic_10	5	=19	19	6275	=19	8	6243
katomic_57	5	=21	19	33,450	=21	8	33,004
marbles_02	5	=15	19	24,059	=15	8	24,092
marbles_05	5	=25	19	51,427	=25	8	51,433
marbles_06	5	=14	19	1134	=14	8	1232
unitopia_03	5	=16	19	1462	=16	8	1473
adrien_03	6	=12	18	2943	=12	8	2947
adrien_06	6	=15	18	59,843	=15	9	76,529
atomix_03	6	=16	18	28,274	=16	8	28,272
atomix_04	6	=23	42	6,486,774	=23	65	6,486,772
kai_03	6	=16	18	28,274	=16	8	28,272
katomic_03	6	=20	19	295,609	=20	10	295,607
katomic_04	6	=23	19	222,364	=23	9	221,362
katomic_58	6	=17	18	23,748	=17	8	23,794

Source: the author.

Table B.2: Fibonacci Heap vs. Buckets Experiment 2/4

Instance	n	Buckets			Fibonacci Heap		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
marbles_08	6	=23	29	3,027,891	=23	30	3,028,007
marbles_12	6	=28	86	15,969,380	=28	147	15,968,821
marbles_14	6	=22	18	22,953	=22	8	21,664
unitopia_04	6	=20	18	10,017	=20	8	8488
unitopia_05	6	=20	19	226,450	=20	10	226,342
adrienl_01	7	=20	30	1,065,324	=20	26	1,065,004
adrienl_03	7	=22	534	32,511,794	=22	803	32,521,963
atomix_09	7	=20	21	715,535	=20	14	714,799
katomic_08	7	≥25	541	115,111,670	≥24	548	44,994,838
katomic_26	7	=36	287	64,259,387	≥35	712	57,117,594
katomic_46	7	=24	20	512,485	=24	12	512,498
katomic_60	7	=19	18	35,474	=19	9	34,208
unitopia_08	7	=23	23	1,015,270	=23	18	1,015,297
adrienl_02	8	≥31	660	81,215,335	≥30	515	30,570,504
atomix_06	8	=13	18	242	=13	8	200
atomix_13	8	=28	24	1,401,827	=28	23	1,401,772
atomix_18	8	=13	18	1648	=13	9	1648
atomix_22	8	≥25	543	67,584,077	≥25	441	26,255,547
atomix_29	8	=22	20	304,658	=22	12	304,663
atomix_30	8	=13	18	1648	=13	8	1648
kai_05	8	=27	498	74,829,335	≥26	479	30,995,989
kai_17	8	=23	22	500,963	=23	15	500,968
katomic_11	8	=23	159	19,882,485	=23	345	20,674,757
katomic_19	8	≥31	433	58,202,893	≥30	426	26,056,576
katomic_31	8	=29	196	34,194,628	=29	477	34,147,232
marbles_11	8	=28	186	22,485,798	=28	344	22,486,547
unitopia_10	8	≥39	526	103,225,531	≥38	525	41,329,408
adrienl_04	9	≥34	516	99,520,890	≥33	494	38,875,395
atomix_05	9	≥36	455	66,636,140	≥35	442	26,564,437
atomix_07	9	≥26	431	59,906,097	≥26	395	24,460,587
atomix_12	9	=14	18	2506	=14	8	2490
atomix_16	9	≥27	491	59,533,377	≥27	438	25,775,303
katomic_05	9	=27	134	21,309,141	=27	309	21,305,468
katomic_06	9	=27	327	54,058,835	≥26	433	30,108,751
katomic_14	9	≥27	423	61,966,682	≥26	402	25,587,827
katomic_32	9	=19	20	323,260	=19	12	337,712
katomic_38	9	≥33	463	77,742,782	≥32	458	31,369,409
unitopia_06	9	=31	468	57,992,469	≥30	350	20,492,060
adrien_04	10	≥25	1093	43,570,830	≥24	595	16,554,971

Source: the author.

Table B.3: Fibonacci Heap vs. Buckets Experiment 3/4

Instance	n	Buckets			Fibonacci Heap		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
adrien_05	10	≥26	875	35,191,945	≥26	549	15,962,166
atomix_10	10	≥29	416	42,461,881	≥28	341	17,343,008
atomix_28	10	=29	85	10,384,620	=29	168	10,384,624
kai_09	10	≥34	390	48,323,586	≥34	333	18,822,021
katomic_09	10	≥31	356	44,274,522	≥30	348	20,014,442
katomic_25	10	≥33	391	51,777,687	≥33	338	20,675,038
katomic_33	10	≥49	674	63,622,846	≥48	501	27,032,966
katomic_35	10	≥32	379	59,712,760	≥31	369	24,598,238
katomic_61	10	≥53	405	56,297,870	≥52	353	22,349,236
unitopia_07	10	≥34	379	50,456,445	≥33	368	23,403,358
katomic_47	11	=29	32	2,058,349	=29	38	1,831,032
katomic_66	11	≥31	350	35,716,591	≥31	317	16,099,735
atomix_08	12	≥34	362	32,315,494	≥34	337	14,979,838
atomix_14	12	≥35	345	29,642,063	≥35	303	13,193,841
atomix_15	12	≥36	366	31,384,517	≥36	280	12,053,793
atomix_21	12	≥31	786	27,974,963	≥30	512	12,263,540
kai_07	12	≥33	370	34,185,196	≥33	345	15,348,529
kai_08	12	≥35	351	34,248,304	≥35	282	13,506,352
kai_18	12	≥34	271	24,196,706	≥33	266	11,627,576
kai_20	12	≥38	310	30,043,943	≥37	274	13,164,557
kai_22	12	≥33	348	32,573,982	≥32	289	13,472,332
katomic_07	12	≥23	725	33,634,825	≥23	482	15,066,912
katomic_12	12	≥35	822	44,459,538	≥35	589	20,757,046
katomic_13	12	≥41	317	28,162,721	≥41	277	12,599,154
katomic_18	12	≥46	1428	31,716,634	≥46	723	13,847,962
katomic_27	12	≥46	295	30,487,175	≥45	259	12,761,203
katomic_28	12	≥36	373	41,518,698	≥35	369	18,567,374
katomic_42	12	≥34	494	40,758,365	≥33	377	16,920,536
katomic_62	12	≥51	318	32,472,028	≥51	261	13,544,618
katomic_63	12	≥41	408	43,970,240	≥40	349	19,342,172
katomic_67	12	≥30	397	41,205,819	≥30	343	18,361,370
marbles_15	12	≥37	1788	48,437,736	≥36	957	19,853,062
unitopia_09	12	≥43	416	40,493,250	≥42	362	18,763,768
katomic_34	13	≥35	304	24,107,907	≥35	247	10,045,535
atomix_20	14	=29	337	26,506,951	≥28	277	11,846,210
atomix_25	14	≥36	386	22,580,850	≥35	278	9,233,190
kai_14	14	≥40	335	25,311,694	≥40	276	11,235,629
kai_21	14	≥42	338	28,551,322	≥42	289	12,446,221
kai_24	14	≥40	315	21,266,572	≥40	254	9,823,106

Source: the author.

Table B.4: Fibonacci Heap vs. Buckets Experiment 4/4

Instance	n	Buckets			Fibonacci Heap		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
kai_25	14	≥33	330	22,648,412	≥33	271	10,005,478
katomic_17	14	≥31	387	23,867,185	≥30	289	10,261,354
katomic_22	14	≥31	501	24,379,339	≥31	388	12,276,100
katomic_45	14	≥38	266	22,728,680	≥38	220	9,867,074
15-puzzle	15	=34	18	1,453,014	=34	14	1,448,575
atomix_17	15	≥36	654	22,459,789	≥35	400	9,500,351
atomix_19	15	≥27	329	24,576,634	≥27	269	11,099,148
kai_12	15	≥35	314	18,525,081	≥35	251	8,114,402
katomic_15	15	≥35	740	24,666,899	≥35	482	11,548,524
katomic_16	15	≥42	320	25,639,546	≥41	256	11,334,370
katomic_29	15	≥56	287	24,306,760	≥56	217	10,656,680
katomic_41	15	≥34	400	20,762,730	≥33	287	9,288,654
katomic_55	15	≥47	306	25,049,689	≥47	270	11,566,580
katomic_56	15	≥48	283	23,986,260	≥48	235	10,269,943
atomix_24	16	≥29	401	28,506,922	≥28	332	13,628,680
kai_28	16	≥46	254	14,630,723	≥46	216	6,997,644
katomic_21	16	≥25	395	29,326,866	≥25	321	13,993,211
katomic_40	16	≥56	351	32,636,619	≥55	315	14,553,046
katomic_51	16	≥39	316	21,061,341	≥39	262	10,292,715
katomic_53	16	≥25	551	21,595,083	≥24	389	10,143,059
katomic_54	16	≥35	282	21,298,003	≥34	264	10,460,758
katomic_59	16	≥27	326	15,475,405	≥26	243	7,061,313
katomic_64	16	≥53	368	24,516,740	≥53	276	10,702,244
marbles_10	16	=24	16	21,324	=24	9	21,324
katomic_39	17	≥46	272	20,398,950	≥46	226	9,419,227
katomic_48	17	≥56	264	16,139,318	≥55	214	7,717,571
katomic_50	17	≥41	359	24,736,636	≥41	261	11,025,886
katomic_65	17	≥31	302	35,777,006	≥30	286	17,258,307
katomic_49	18	≥45	278	18,385,996	≥44	249	9,222,795
kai_27	19	≥59	242	10,074,866	≥59	179	4,883,797
katomic_52	19	≥53	256	15,094,790	≥52	208	7,532,604
atomix_27	20	≥45	650	10,270,513	≥45	396	5,136,547
katomic_24	20	≥36	3600	8,051,122	≥36	2676	6,024,304
kai_29	21	≥62	264	11,465,703	≥62	191	5,354,661
katomic_30	21	≥51	270	14,001,792	≥51	218	6,919,686
katomic_44	21	≥47	271	16,030,106	≥47	242	8,071,908
katomic_37	24	≥54	270	8,196,266	≥54	203	4,077,825
katomic_43	26	≥64	245	8,184,587	≥64	188	4,520,602
marbles_20	32	≥37	3387	46,133,258	≥37	2135	26,395,956

Source: the author.

APPENDIX C — *ONE FINAL STATE* VS *ALL FINAL STATES* EXPERIMENT RESULTS

Table C.1: *One Final State* vs *All Final States* Experiment 1/4

Instance	n	One Final State			All Final States		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
adrien_01	3	=7	29	290	=7	20	27
atomix_01	3	=13	31	6191	=13	19	418
kai_01	3	=9	25	562	=9	19	167
katomic_01	3	=15	47	14,763	=15	19	712
katomic_36	3	=9	36	6062	=9	19	365
marbles_04	3	=22	67	161,468	=22	19	2572
marbles_13	3	=18	24	56,844	=18	19	5262
unitopia_01	3	=11	37	2072	=11	19	221
adrienl_05	4	=12	72	242,887	=12	19	19,377
atomix_23	4	=10	31	4543	=10	18	1015
atomix_26	4	=14	34	48,911	=14	18	10,501
kai_06	4	=14	28	42,944	=14	18	6351
kai_19	4	=19	34	227,137	=19	18	26,864
katomic_20	4	=18	28	15,145	=18	18	3232
katomic_23	4	=18	63	231,308	=18	18	15,430
marbles_01	4	=11	18	2127	=11	18	763
marbles_03	4	=22	26	537,459	=22	18	55,293
unitopia_02	4	=22	22	216,522	=22	18	65,716
adrien_02	5	=17	34	2,248,373	=17	21	257,127
atomix_02	5	=21	21	54,536	=21	19	17,193
atomix_11	5	=14	25	25,137	=14	19	6996
kai_02	5	=24	20	762,079	=24	19	326,902
kai_11	5	=15	22	33,970	=15	18	12,418
katomic_02	5	=27	32	1,197,850	=27	19	129,319
katomic_10	5	=19	21	14,367	=19	18	6148
katomic_57	5	=21	19	175,810	=21	19	52,206
marbles_02	5	=15	21	126,181	=15	18	40,673
marbles_05	5	=25	20	256,530	=25	18	56,769
marbles_06	5	=14	18	1504	=14	18	839
unitopia_03	5	=16	24	10,634	=16	18	3483
adrien_03	6	=12	29	6316	=12	19	10,694
adrien_06	6	=15	27	284,730	=15	20	131,341
atomix_03	6	=16	19	118,502	=16	19	34,757
atomix_04	6	=23	55	17,189,370	=23	45	7,312,479
kai_03	6	=16	19	118,502	=16	18	34,757
katomic_03	6	=20	22	1,205,458	=20	20	713,597
katomic_04	6	=23	28	1,175,077	=23	18	225,539
katomic_58	6	=17	19	64,919	=17	18	34,633

Source: the author.

Table C.2: *One Final State* vs *All Final States* Experiment 2/4

Instance	n	One Final State			All Final States		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
marbles_08	6	=23	54	16,295,151	=23	31	3,863,347
marbles_12	6	=28	314	115,375,402	=28	83	16,131,164
marbles_14	6	=22	18	51,752	=22	18	21,150
unitopia_04	6	=20	18	30,567	=20	18	14,968
unitopia_05	6	=20	25	874,705	=20	20	629,451
adrienl_01	7	=20	59	7,581,021	=20	33	1,314,800
adrienl_03	7	=22	407	126,250,217	≥22	600	38,773,541
atomix_09	7	=20	23	1,872,983	=20	22	873,796
katomic_08	7	≥26	600	233,702,367	≥25	527	116,241,299
katomic_26	7	≥36	600	259,031,101	=36	286	66,170,428
katomic_46	7	=24	23	1,804,241	=24	22	1,189,955
katomic_60	7	=19	20	136,343	=19	18	59,925
unitopia_08	7	=23	27	3,212,968	=23	26	1,627,424
adrienl_02	8	≥30	510	165,736,520	≥31	600	76,593,585
atomix_06	8	=13	18	443	=13	18	663
atomix_13	8	=28	26	3,316,812	=28	25	1,436,165
atomix_18	8	=13	19	3428	=13	18	2383
atomix_22	8	≥25	445	127,573,489	≥25	516	68,362,973
atomix_29	8	=22	20	607,470	=22	20	342,878
atomix_30	8	=13	19	3428	=13	18	2383
kai_05	8	≥26	470	135,377,447	≥27	531	82,703,697
kai_17	8	=23	25	2,238,382	=23	29	1,709,079
katomic_11	8	=23	162	50,256,808	=23	243	32,867,082
katomic_19	8	≥31	503	153,018,116	≥31	398	56,981,623
katomic_31	8	=29	359	138,260,628	=29	257	47,101,869
marbles_11	8	=28	334	57,111,666	=28	184	23,475,184
unitopia_10	8	≥38	447	165,221,840	≥39	505	101,601,798
adrienl_04	9	≥33	480	134,842,285	≥34	519	99,824,979
atomix_05	9	≥35	353	90,839,699	≥36	472	71,471,615
atomix_07	9	≥26	481	114,981,638	≥26	407	57,354,445
atomix_12	9	=14	19	14,700	=14	18	9168
atomix_16	9	≥28	600	159,464,496	≥27	463	59,565,098
katomic_05	9	=27	276	92,694,848	=27	299	50,476,125
katomic_06	9	≥26	424	113,845,569	≥27	422	71,036,297
katomic_14	9	≥27	600	170,496,312	≥27	401	60,399,949
katomic_32	9	=19	24	1,328,942	=19	20	458,640
katomic_38	9	≥32	410	108,091,942	≥33	457	77,059,936
unitopia_06	9	≥30	404	100,148,214	≥31	440	58,069,889
adrien_04	10	≥25	600	123,727,945	≥24	600	27,529,775

Source: the author.

Table C.3: *One Final State* vs *All Final States* Experiment 3/4

Instance	n	One Final State			All Final States		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
adrien_05	10	≥26	600	135,912,455	≥26	600	29,998,220
atomix_10	10	≥29	600	149,032,555	≥29	387	41,242,166
atomix_28	10	=29	122	26,888,844	=29	101	13,549,793
kai_09	10	≥34	381	86,686,159	≥34	374	47,528,544
katomic_09	10	≥31	586	146,679,694	≥31	346	44,381,988
katomic_25	10	≥33	396	94,841,272	≥33	363	50,187,615
katomic_33	10	≥49	600	154,755,457	≥49	543	62,974,440
katomic_35	10	≥31	339	93,268,395	≥32	364	59,515,565
katomic_61	10	≥53	600	187,921,606	≥53	386	56,884,950
unitopia_07	10	≥35	600	172,441,294	≥34	366	52,329,090
katomic_47	11	=29	21	1,284,348	=29	24	1,105,507
katomic_66	11	≥31	542	108,979,059	≥31	334	36,474,813
atomix_08	12	≥34	600	113,585,841	≥34	367	34,307,518
atomix_14	12	≥35	600	110,173,721	≥35	339	30,216,513
atomix_15	12	≥36	462	78,048,870	≥36	369	33,080,515
atomix_21	12	≥31	600	45,734,879	≥31	600	21,675,143
kai_07	12	≥34	600	109,189,195	≥33	388	37,908,844
kai_08	12	≥35	424	78,627,777	≥35	341	34,578,125
kai_18	12	≥34	600	128,303,194	≥34	277	25,477,485
kai_20	12	≥37	236	42,661,886	≥38	303	29,858,040
kai_22	12	≥32	287	47,977,496	≥33	326	31,608,500
katomic_07	12	≥24	600	124,246,637	≥23	550	32,762,716
katomic_12	12	≥36	600	122,471,146	≥35	600	39,028,978
katomic_13	12	≥41	600	122,441,803	≥41	301	28,343,691
katomic_18	12	≥46	600	57,221,698	≥46	600	13,103,648
katomic_27	12	≥45	211	39,655,836	≥46	292	30,160,378
katomic_28	12	≥35	274	51,228,929	≥36	374	42,901,393
katomic_42	12	≥33	481	60,309,547	≥34	466	40,127,320
katomic_62	12	≥51	389	78,600,703	≥51	304	33,449,681
katomic_63	12	≥41	600	151,007,870	≥41	366	44,101,099
katomic_67	12	≥30	500	110,074,261	≥30	369	42,496,672
marbles_15	12	≥36	600	19,557,606	≥36	600	16,927,977
unitopia_09	12	≥43	439	91,357,210	≥43	379	41,143,697
katomic_34	13	≥35	352	58,442,189	≥35	299	24,464,313
atomix_20	14	≥28	319	46,429,758	≥29	314	26,616,211
atomix_25	14	≥35	296	37,854,184	≥36	339	22,483,883
kai_14	14	≥40	577	101,096,095	≥40	346	28,002,747
kai_21	14	≥43	600	101,287,806	≥42	325	28,005,897
kai_24	14	≥40	600	94,002,953	≥40	310	23,549,989

Source: the author.

Table C.4: *One Final State* vs *All Final States* Experiment 4/4

Instance	n	One Final State			All Final States		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
kai_25	14	≥33	349	45,408,173	≥33	310	22,438,432
katomic_17	14	≥31	600	96,177,743	≥31	337	23,266,052
katomic_22	14	≥32	600	93,082,043	≥31	423	24,670,191
katomic_45	14	≥38	225	36,635,817	≥38	253	22,688,072
15-puzzle	15	=34	27	5,082,501	=34	19	1,449,440
atomix_17	15	≥35	572	27,274,967	≥36	600	21,379,043
atomix_19	15	≥27	324	47,309,683	≥27	297	23,794,653
kai_12	15	≥35	492	62,765,593	≥35	297	18,403,055
katomic_15	15	≥35	600	26,867,264	≥35	600	20,541,058
katomic_16	15	≥41	282	42,887,760	≥42	295	25,550,787
katomic_29	15	≥56	336	54,972,362	≥56	269	24,710,652
katomic_41	15	≥34	600	98,006,917	≥34	362	21,040,786
katomic_55	15	≥48	600	103,569,135	≥47	313	26,756,715
katomic_56	15	≥48	240	39,281,282	≥48	266	23,983,488
atomix_24	16	≥28	341	37,126,276	≥29	369	27,843,517
kai_28	16	≥46	600	88,179,596	≥46	253	14,845,020
katomic_21	16	≥26	600	78,817,174	≥25	357	29,200,930
katomic_40	16	≥55	330	52,943,587	≥56	340	32,406,399
katomic_51	16	≥40	600	81,606,290	≥39	306	22,084,488
katomic_53	16	≥25	600	55,357,841	≥25	486	20,063,361
katomic_54	16	≥34	203	26,637,554	≥35	273	21,403,636
katomic_59	16	≥27	600	98,535,680	≥27	292	15,446,625
katomic_64	16	≥54	600	92,309,368	≥53	364	26,346,409
marbles_10	16	=24	18	87,079	=24	16	58,335
katomic_39	17	≥46	495	69,735,349	≥46	264	20,602,373
katomic_48	17	≥55	188	20,329,327	≥56	239	16,232,317
katomic_50	17	≥41	323	46,888,484	≥41	296	24,622,933
katomic_65	17	≥30	272	53,414,461	≥31	269	35,354,532
katomic_49	18	≥44	216	23,863,533	≥45	267	19,052,593
kai_27	19	≥59	193	15,857,481	≥59	229	9,748,673
katomic_52	19	≥52	157	14,337,795	≥53	246	14,655,126
atomix_27	20	≥45	600	15,179,013	≥45	600	10,083,637
katomic_24	20	≥36	600	14,700,245	≥35	600	1,298,641
kai_29	21	≥62	197	16,775,516	≥62	244	11,409,357
katomic_30	21	≥51	600	66,683,286	≥51	265	14,470,999
katomic_44	21	≥47	349	41,174,257	≥47	248	15,230,213
katomic_37	24	≥54	526	33,791,878	≥54	273	9,208,173
katomic_43	26	≥64	600	46,179,074	≥64	217	7,481,980
marbles_20	32	≥36	600	8,184,118	≥36	600	8,213,768

Source: the author.

APPENDIX D — TIE-BREAKING EXPERIMENT RESULTS

Table D.1: Tie-Breaking Experiment 1/5

Instance	n	No Tie-Break			GC			NRP			FO		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
adrien_01	3	=7	18	27	=7	19	28	=7	19	27	=7	19	28
atomix_01	3	=13	18	418	=13	19	418	=13	19	535	=13	19	381
kai_01	3	=9	19	167	=9	19	120	=9	19	160	=9	19	129
katomic_01	3	=15	19	712	=15	18	599	=15	19	728	=15	19	686
katomic_36	3	=9	19	365	=9	19	353	=9	19	400	=9	19	420
marbles_04	3	=22	18	2572	=22	18	2548	=22	19	2568	=22	19	2619
marbles_13	3	=18	18	5262	=18	19	4963	=18	19	5300	=18	19	4841
unitopia_01	3	=11	18	221	=11	18	181	=11	19	224	=11	19	183
adrienl_05	4	=12	18	19,377	=12	18	18,746	=12	18	19,464	=12	18	20,194
atomix_23	4	=10	18	1015	=10	18	1047	=10	18	649	=10	18	1170
atomix_26	4	=14	18	10,501	=14	18	9948	=14	18	9988	=14	18	9845
kai_06	4	=14	18	6351	=14	18	5165	=14	18	6649	=14	18	5206
kai_19	4	=19	18	26,864	=19	18	19,193	=19	18	21,130	=19	18	19,168
katomic_20	4	=18	18	3232	=18	18	2829	=18	18	3236	=18	18	2849
katomic_23	4	=18	18	15,430	=18	18	15,519	=18	18	14,987	=18	18	16,009
marbles_01	4	=11	18	763	=11	18	779	=11	18	749	=11	18	756
marbles_03	4	=22	18	55,293	=22	18	51,583	=22	18	55,101	=22	18	51,582
unitopia_02	4	=22	18	65,716	=22	18	57,583	=22	18	58,630	=22	18	57,584
adrien_02	5	=17	20	257,127	=17	20	256,410	=17	21	256,100	=17	20	258,277
atomix_02	5	=21	18	17,193	=21	19	10,509	=21	18	11,892	=21	19	10,277
atomix_11	5	=14	18	6996	=14	19	3811	=14	18	3750	=14	19	4001
kai_02	5	=24	19	326,902	=24	19	246,130	=24	19	272,190	=24	20	249,948
kai_11	5	=15	18	12,418	=15	19	11,640	=15	18	11,261	=15	19	11,508
katomic_02	5	=27	19	129,319	=27	19	120,615	=27	19	152,072	=27	19	121,637
katomic_10	5	=19	18	6148	=19	19	6275	=19	18	8520	=19	19	6162
katomic_57	5	=21	18	52,206	=21	19	33,450	=21	18	33,419	=21	19	33,244
marbles_02	5	=15	18	40,673	=15	19	24,059	=15	18	24,569	=15	19	24,062
marbles_05	5	=25	18	56,769	=25	19	51,427	=25	18	51,498	=25	19	51,427
marbles_06	5	=14	18	839	=14	19	1134	=14	18	597	=14	19	1219
unitopia_03	5	=16	18	3483	=16	19	1462	=16	18	2382	=16	19	1938
adrien_03	6	=12	18	10,694	=12	18	2943	=12	18	3028	=12	18	2943

Source: the author.

Table D.2: Tie-Breaking Experiment 2/5

Instance	n	No Tie-Break			GC			NRP			FO		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
adrien_06	6	=15	19	131,341	=15	18	59,843	=15	20	61,445	=15	19	58,095
atomix_03	6	=16	18	34,757	=16	18	28,274	=16	18	28,926	=16	19	33,310
atomix_04	6	=23	44	7,312,479	=23	42	6,486,774	=23	55	6,560,674	=23	44	6,558,705
kai_03	6	=16	18	34,757	=16	18	28,274	=16	18	28,926	=16	19	33,310
katomic_03	6	=20	21	713,597	=20	19	295,609	=20	20	307,547	=20	20	295,602
katomic_04	6	=23	19	225,539	=23	19	222,364	=23	20	222,835	=23	19	224,554
katomic_58	6	=17	18	34,633	=17	18	23,748	=17	18	24,533	=17	19	23,943
marbles_08	6	=23	31	3,863,347	=23	29	3,027,891	=23	35	3,085,903	=23	30	3,091,679
marbles_12	6	=28	83	16,131,164	=28	86	15,969,380	=28	111	15,987,036	=28	90	16,104,076
marbles_14	6	=22	18	21,150	=22	18	22,953	=22	18	23,127	=22	19	22,953
unitopia_04	6	=20	18	14,968	=20	18	10,017	=20	18	16,982	=20	19	12,018
unitopia_05	6	=20	21	629,451	=20	19	226,450	=20	22	377,680	=20	20	236,198
adrienl_01	7	=20	32	1,314,800	=20	30	1,065,324	=20	124	2,221,883	=20	32	1,089,149
adrienl_03	7	=22	857	57,741,214	=22	534	32,511,794	=22	2952	47,253,968	=22	796	40,553,679
atomix_09	7	=20	22	873,796	=20	21	715,535	=20	24	1,049,963	=20	21	758,700
katomic_08	7	≥25	522	116,241,299	≥25	541	115,111,670	≥25	746	122,739,964	≥25	548	115,111,670
katomic_26	7	=36	285	66,170,428	=36	287	64,259,387	=36	496	69,391,517	=36	295	64,305,739
katomic_46	7	=24	23	1,189,955	=24	20	512,485	=24	22	513,200	=24	20	513,245
katomic_60	7	=19	19	59,925	=19	18	35,474	=19	19	43,411	=19	18	40,481
unitopia_08	7	=23	26	1,627,424	=23	23	1,015,270	=23	28	1,077,648	=23	23	1,015,266
adrienl_02	8	≥31	634	81,763,874	≥31	660	81,215,335	≥31	1663	82,315,412	≥31	710	82,240,951
atomix_06	8	=13	17	663	=13	18	242	=13	17	348	=13	18	189
atomix_13	8	=28	23	1,436,165	=28	24	1,401,827	=28	26	1,403,090	=28	25	1,401,794
atomix_18	8	=13	17	2383	=13	18	1648	=13	18	1649	=13	18	1647
atomix_22	8	≥25	502	68,362,973	≥25	543	67,584,077	≥25	1032	70,112,748	≥25	554	67,584,077
atomix_29	8	=22	19	342,878	=22	20	304,658	=22	21	305,644	=22	20	304,658
atomix_30	8	=13	17	2383	=13	18	1648	=13	17	1649	=13	18	1647
kai_05	8	≥27	518	82,703,697	=27	498	74,829,335	=27	778	75,487,839	=27	493	75,097,465
kai_17	8	=23	28	1,709,079	=23	22	500,963	=23	26	516,087	=23	22	500,963
katomic_11	8	=23	239	32,867,082	=23	159	19,882,485	=23	516	28,648,263	=23	165	19,951,930
katomic_19	8	≥31	391	56,981,623	≥31	433	58,202,893	≥31	784	63,798,333	≥31	441	59,008,938
katomic_31	8	=29	252	47,101,869	=29	196	34,194,628	=29	325	35,258,231	=29	224	38,809,491

Source: the author.

Table D.3: Tie-Breaking Experiment 3/5

Instance	n	No Tie-Break			GC			NRP			FO		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
marbles_11	8	=28	181	23,475,184	=28	186	22,485,798	=28	232	22,521,429	=28	186	22,485,852
unitopia_10	8	≥39	492	101,601,798	≥39	526	103,225,531	≥39	787	108,552,885	≥39	535	103,225,531
adrienl_04	9	≥34	516	99,824,979	≥34	516	99,520,890	≥34	768	97,462,646	≥34	525	99,520,890
atomix_05	9	≥36	472	71,471,615	≥36	455	66,636,140	≥36	875	73,081,794	≥36	479	68,100,765
atomix_07	9	≥26	411	57,354,445	≥26	431	59,906,097	≥26	696	65,320,478	≥26	435	59,906,097
atomix_12	9	=14	18	9168	=14	18	2506	=14	18	2503	=14	19	2489
atomix_16	9	≥27	462	59,565,098	≥27	491	59,533,377	≥27	923	60,412,724	≥27	488	59,533,377
katomic_05	9	=27	302	50,476,125	=27	134	21,309,141	=27	242	21,303,377	=27	139	21,303,093
katomic_06	9	≥27	420	71,036,297	=27	327	54,058,835	=27	460	54,077,568	=27	332	54,058,835
katomic_14	9	≥27	405	60,399,949	≥27	423	61,966,682	≥27	669	66,296,048	≥27	440	63,912,542
katomic_32	9	=19	20	458,640	=19	20	323,260	=19	24	324,877	=19	21	316,043
katomic_38	9	≥33	460	77,059,936	≥33	463	77,742,782	≥33	702	79,666,113	≥33	492	81,501,580
unitopia_06	9	≥31	444	58,069,889	=31	468	57,992,469	≥31	878	58,065,813	=31	475	57,952,229
adrien_04	10	≥25	887	42,967,390	≥25	1093	43,570,830	≥24	3600	37,864,309	≥25	1166	43,460,007
adrien_05	10	≥26	667	35,492,297	≥26	875	35,191,945	≥26	3442	34,824,876	≥26	916	34,725,671
atomix_10	10	≥29	371	41,242,166	≥29	416	42,461,881	≥29	843	42,129,792	≥29	432	42,877,148
atomix_28	10	=29	100	13,549,793	=29	85	10,384,620	=29	124	10,387,541	=29	85	10,384,620
kai_09	10	≥34	377	47,528,544	≥34	390	48,323,586	≥34	620	50,105,000	≥34	395	48,323,586
katomic_09	10	≥31	337	44,381,988	≥31	356	44,274,522	≥31	583	45,184,460	≥31	376	45,811,726
katomic_25	10	≥33	360	50,187,615	≥33	391	51,777,687	≥33	607	52,925,362	≥33	395	51,777,687
katomic_33	10	≥49	541	62,974,440	≥49	674	63,622,846	≥49	1620	67,033,985	≥49	643	63,622,712
katomic_35	10	≥32	365	59,515,565	≥32	379	59,712,760	≥32	550	62,122,010	≥32	400	61,216,251
katomic_61	10	≥53	386	56,884,950	≥53	405	56,297,870	≥53	770	57,605,864	≥53	416	56,583,101
unitopia_07	10	≥34	364	52,329,090	≥34	379	50,456,445	≥34	594	54,564,590	≥34	390	50,661,181
katomic_47	11	=29	24	1,105,507	=29	32	2,058,349	=29	43	1,851,005	=29	33	2,058,340
katomic_66	11	≥31	335	36,474,813	≥31	350	35,716,591	≥31	596	37,185,828	≥31	362	36,085,450
atomix_08	12	≥34	362	34,307,518	≥34	362	32,315,494	≥34	668	31,970,919	≥34	369	32,363,925
atomix_14	12	≥35	337	30,216,513	≥35	345	29,642,063	≥35	655	27,886,479	≥35	351	29,660,574
atomix_15	12	≥36	365	33,080,515	≥36	366	31,384,517	≥36	656	30,837,406	≥36	368	31,328,044
atomix_21	12	≥31	751	27,336,892	≥31	786	27,974,963	≥31	1600	26,451,599	≥31	817	28,216,086
kai_07	12	≥33	389	37,908,844	≥33	370	34,185,196	≥33	690	36,024,484	≥33	378	34,322,165
kai_08	12	≥35	336	34,578,125	≥35	351	34,248,304	≥35	642	33,715,466	≥35	358	34,331,230

Source: the author.

Table D.4: Tie-Breaking Experiment 4/5

Instance	n	No Tie-Break			GC			NRP			FO		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
kai_18	12	≥34	273	25,477,485	≥34	271	24,196,706	≥34	503	24,241,430	≥34	282	24,360,143
kai_20	12	≥38	295	29,858,040	≥38	310	30,043,943	≥38	594	30,359,885	≥38	321	30,611,099
kai_22	12	≥33	322	31,608,500	≥33	348	32,573,982	≥33	663	32,359,831	≥33	353	32,368,103
katomic_07	12	≥23	548	32,762,716	≥23	725	33,634,825	≥23	3325	33,405,450	≥23	758	33,688,257
katomic_12	12	≥35	695	45,459,983	≥35	822	44,459,538	≥35	3573	46,459,279	≥35	878	44,544,251
katomic_13	12	≥41	303	28,343,691	≥41	317	28,162,721	≥41	643	27,488,038	≥41	320	28,601,822
katomic_18	12	≥46	1370	29,533,860	≥46	1428	31,716,634	≥46	3600	23,829,382	≥46	1429	31,495,133
katomic_27	12	≥46	288	30,160,378	≥46	295	30,487,175	≥46	528	30,533,865	≥46	303	30,250,141
katomic_28	12	≥36	372	42,901,393	≥36	373	41,518,698	≥36	626	43,515,887	≥36	386	42,263,696
katomic_42	12	≥34	463	40,127,320	≥34	494	40,758,365	≥34	853	41,092,413	≥34	505	40,970,844
katomic_62	12	≥51	302	33,449,681	≥51	318	32,472,028	≥51	664	32,696,500	≥51	320	32,483,619
katomic_63	12	≥41	366	44,101,099	≥41	408	43,970,240	≥41	961	48,144,637	≥41	419	43,970,240
katomic_67	12	≥30	372	42,496,672	≥30	397	41,205,819	≥30	829	42,511,517	≥30	412	41,296,430
marbles_15	12	≥37	1842	50,249,142	≥37	1788	48,437,736	≥37	2286	48,628,518	≥37	1813	48,956,356
unitopia_09	12	≥43	374	41,143,697	≥43	416	40,493,250	≥43	1039	40,702,605	≥43	442	42,914,785
katomic_34	13	≥35	300	24,464,313	≥35	304	24,107,907	≥35	563	23,976,289	≥35	315	24,135,589
atomix_20	14	≥29	304	26,616,211	=29	337	26,506,951	=29	762	26,516,710	=29	343	26,506,947
atomix_25	14	≥36	336	22,483,883	≥36	386	22,580,850	≥36	1042	22,580,050	≥36	395	22,610,437
kai_14	14	≥40	341	28,002,747	≥40	335	25,311,694	≥40	844	24,803,230	≥40	345	24,732,628
kai_21	14	≥42	323	28,005,897	≥42	338	28,551,322	≥42	780	27,516,080	≥42	343	28,585,780
kai_24	14	≥40	311	23,549,989	≥40	315	21,266,572	≥40	704	20,502,802	≥40	313	21,320,267
kai_25	14	≥33	305	22,438,432	≥33	330	22,648,412	≥33	778	23,450,913	≥33	336	22,812,447
katomic_17	14	≥31	335	23,266,052	≥31	387	23,867,185	≥31	1296	23,675,395	≥31	394	23,860,922
katomic_22	14	≥31	423	24,670,191	≥31	501	24,379,339	≥31	2032	26,202,268	≥31	531	24,822,697
katomic_45	14	≥38	252	22,688,072	≥38	266	22,728,680	≥38	494	22,554,830	≥38	275	22,728,680
15-puzzle	15	=34	18	1,449,440	=34	18	1,453,014	=34	19	1,447,183	=34	18	1,441,610
atomix_17	15	≥36	636	23,505,965	≥36	654	22,459,789	≥36	1104	22,712,933	≥36	647	22,434,331
atomix_19	15	≥27	289	23,794,653	≥27	329	24,576,634	≥27	677	24,830,411	≥27	335	24,953,428
kai_12	15	≥35	290	18,403,055	≥35	314	18,525,081	≥35	702	19,302,431	≥35	322	18,871,193
katomic_15	15	≥35	697	23,857,352	≥35	740	24,666,899	≥35	1240	23,957,983	≥35	716	24,110,777
katomic_16	15	≥42	293	25,550,787	≥42	320	25,639,546	≥42	683	25,584,623	≥42	326	25,677,837
katomic_29	15	≥56	268	24,710,652	≥56	287	24,306,760	≥56	776	23,682,186	≥56	289	24,306,760

Source: the author.

Table D.5: Tie-Breaking Experiment 5/5

Instance	n	No Tie-Break			GC			NRP			FO		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
katomic_41	15	≥34	361	21,040,786	≥34	400	20,762,730	≥34	1644	20,275,601	≥34	439	21,216,519
katomic_55	15	≥47	313	26,756,715	≥47	306	25,049,689	≥47	650	26,454,981	≥47	315	25,054,218
katomic_56	15	≥48	267	23,983,488	≥48	283	23,986,260	≥48	535	23,786,199	≥48	289	23,986,260
atomix_24	16	≥29	365	27,843,517	≥29	401	28,506,922	≥29	845	27,975,914	≥29	410	28,409,764
kai_28	16	≥46	245	14,845,020	≥46	254	14,630,723	≥46	511	14,060,661	≥46	265	14,413,095
katomic_21	16	≥25	356	29,200,930	≥25	395	29,326,866	≥25	805	28,998,509	≥25	408	29,965,377
katomic_40	16	≥56	325	32,406,399	≥56	351	32,636,619	≥56	622	32,583,206	≥56	368	32,947,098
katomic_51	16	≥39	299	22,084,488	≥39	316	21,061,341	≥39	702	22,292,128	≥39	322	21,296,663
katomic_53	16	≥25	480	20,063,361	≥25	551	21,595,083	≥25	1515	20,967,929	≥25	571	21,580,462
katomic_54	16	≥35	264	21,403,636	≥35	282	21,298,003	≥35	544	20,946,688	≥35	299	21,718,932
katomic_59	16	≥27	284	15,446,625	≥27	326	15,475,405	≥27	1496	15,778,407	≥27	375	15,430,923
katomic_64	16	≥53	350	26,346,409	≥53	368	24,516,740	≥53	1086	25,471,871	≥53	385	24,105,469
marbles_10	16	=24	16	58,335	=24	16	21,324	=24	16	21,324	=24	16	21,324
katomic_39	17	≥46	270	20,602,373	≥46	272	20,398,950	≥46	564	19,304,958	≥46	289	20,479,863
katomic_48	17	≥56	245	16,232,317	≥56	264	16,139,318	≥56	589	16,113,515	≥56	275	16,199,286
katomic_50	17	≥41	308	24,622,933	≥41	359	24,736,636	≥41	1004	24,658,398	≥41	357	24,510,395
katomic_65	17	≥31	276	35,354,532	≥31	302	35,777,006	≥31	573	36,194,587	≥31	325	36,107,129
katomic_49	18	≥45	267	19,052,593	≥45	278	18,385,996	≥45	691	18,710,046	≥45	303	18,731,380
kai_27	19	≥59	219	9,748,673	≥59	242	10,074,866	≥59	709	10,092,744	≥59	256	10,074,866
katomic_52	19	≥53	240	14,655,126	≥53	256	15,094,790	≥53	643	15,080,106	≥53	271	15,292,856
atomix_27	20	≥45	612	10,554,954	≥45	650	10,270,513	≥45	1464	10,413,119	≥45	660	10,295,784
katomic_24	20	≥36	3600	8,126,193	≥36	3600	8,051,122	≥36	3600	3,134,962	≥36	3600	8,094,698
kai_29	21	≥62	249	11,409,357	≥62	264	11,465,703	≥62	723	11,404,622	≥62	268	11,465,703
katomic_30	21	≥51	264	14,470,999	≥51	270	14,001,792	≥51	676	13,602,781	≥51	282	14,200,689
katomic_44	21	≥47	248	15,230,213	≥47	271	16,030,106	≥47	671	16,151,942	≥47	275	15,961,053
katomic_37	24	≥54	276	9,208,173	≥54	270	8,196,266	≥54	802	8,021,150	≥54	268	8,184,413
katomic_43	26	≥64	216	7,481,980	≥64	245	8,184,587	≥64	917	7,608,668	≥64	249	8,212,056
marbles_20	32	≥37	3122	43,696,363	≥37	3387	46,133,258	≥37	3600	24,788,192	≥37	3449	46,482,389

Source: the author.

APPENDIX E — PDB EXPERIMENT RESULTS

Table E.1: PDB Experiment 1/5

Instance	n	No PDB			Static PDB			Dynamic PDB			Multi-Goal PDB		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
adrien_01	3	=7	19	28	=7	34	11	=7	19	11	=7	19	25
atomix_01	3	=13	19	418	=13	19	316	=13	19	326	=13	19	400
kai_01	3	=9	19	120	=9	19	111	=9	19	111	=9	19	114
katomic_01	3	=15	18	599	=15	20	429	=15	19	434	=15	19	592
katomic_36	3	=9	19	353	=9	20	263	=9	19	263	=9	19	314
marbles_04	3	=22	18	2548	=22	20	2493	=22	19	2520	=22	19	2548
marbles_13	3	=18	19	4963	=18	19	4925	=18	19	4957	=18	19	4962
unitopia_01	3	=11	18	181	=11	25	81	=11	19	81	=11	20	149
adrienl_05	4	=12	18	18,746	=12	130	16,740	=12	34	12,595	=12	19	15,577
atomix_23	4	=10	18	1047	=10	24	879	=10	18	211	=10	18	601
atomix_26	4	=14	18	9948	=14	28	8687	=14	20	7266	=14	19	9106
kai_06	4	=14	18	5165	=14	22	4085	=14	19	3206	=14	18	4335
kai_19	4	=19	18	19,193	=19	23	18,345	=19	23	16,582	=19	19	17,696
katomic_20	4	=18	18	2829	=18	22	2561	=18	19	2427	=18	18	2679
katomic_23	4	=18	18	15,519	=18	30	15,141	=18	25	14,988	=18	19	15,412
marbles_01	4	=11	18	779	=11	19	623	=11	18	654	=11	18	696
marbles_03	4	=22	18	51,583	=22	19	48,587	=22	21	50,300	=22	19	51,263
unitopia_02	4	=22	18	57,583	=22	19	50,822	=22	21	43,792	=22	19	46,992
adrien_02	5	=17	20	256,410	=17	24	251,180	=17	90	204,427	=17	27	234,036
atomix_02	5	=21	19	10,509	=21	19	7835	=21	20	7057	=21	19	8201
atomix_11	5	=14	19	3811	=14	22	2414	=14	19	1312	=14	18	2379
kai_02	5	=24	19	246,130	=24	19	157,627	=24	23	81,490	=24	22	119,221
kai_11	5	=15	19	11,640	=15	19	6319	=15	19	4199	=15	19	5896
katomic_02	5	=27	19	120,615	=27	20	86,113	=27	36	74,283	=27	21	111,265
katomic_10	5	=19	19	6275	=19	20	3874	=19	19	729	=19	19	729
katomic_57	5	=21	19	33,450	=21	19	25,631	=21	20	20,261	=21	19	24,007
marbles_02	5	=15	19	24,059	=15	19	14,232	=15	20	11,015	=15	19	14,807
marbles_05	5	=25	19	51,427	=25	19	50,645	=25	21	47,097	=25	20	48,444
marbles_06	5	=14	19	1134	=14	19	1014	=14	19	729	=14	19	841
unitopia_03	5	=16	19	1462	=16	20	1278	=16	19	398	=16	19	768
adrien_03	6	=12	18	2943	=12	35	1182	=12	20	948	=12	19	1449

Source: the author.

Table E.2: PDB Experiment 2/5

Instance	n	No PDB			Static PDB			Dynamic PDB			Multi-Goal PDB		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
adrien_06	6	=15	18	59,843	=15	29	53,251	=15	33	25,845	=15	20	43,736
atomix_03	6	=16	18	28,274	=16	19	25,551	=16	20	12,918	=16	19	21,263
atomix_04	6	=23	42	6,486,774	=23	33	3,510,934	=23	192	2,480,317	=23	157	3,570,191
kai_03	6	=16	18	28,274	=16	19	25,551	=16	20	12,918	=16	19	21,263
katomic_03	6	=20	19	295,609	=20	21	244,187	=20	32	108,912	=20	25	180,016
katomic_04	6	=23	19	222,364	=23	20	193,498	=23	34	77,567	=23	23	168,230
katomic_58	6	=17	18	23,748	=17	19	12,167	=17	19	3648	=17	19	4069
marbles_08	6	=23	29	3,027,891	=23	30	2,799,525	=23	201	2,319,401	=23	102	2,598,250
marbles_12	6	=28	86	15,969,380	=28	89	14,492,305	=28	1374	13,036,446	=28	631	15,351,723
marbles_14	6	=22	18	22,953	=22	18	15,949	=22	18	18,305	=22	19	18,305
unitopia_04	6	=20	18	10,017	=20	19	7909	=20	19	3781	=20	19	6339
unitopia_05	6	=20	19	226,450	=20	22	175,528	=20	53	161,709	=20	26	180,286
adrienl_01	7	=20	30	1,065,324	=20	95	1,008,911	=20	1077	889,687	=20	98	1,029,247
adrienl_03	7	=22	534	32,511,794	=22	951	26,845,732	≥19	3600	1,868,884	=22	2800	31,004,390
atomix_09	7	=20	21	715,535	=20	21	601,858	=20	51	568,834	=20	51	568,834
katomic_08	7	≥25	541	115,111,670	=26	549	112,281,722	=26	2219	41,225,147	=26	2242	41,225,147
katomic_26	7	=36	287	64,259,387	=36	142	25,770,175	=36	1935	11,712,995	=36	1328	23,712,115
katomic_46	7	=24	20	512,485	=24	20	266,748	=24	29	75,598	=24	26	134,229
katomic_60	7	=19	18	35,474	=19	20	29,577	=19	23	28,240	=19	20	31,135
unitopia_08	7	=23	23	1,015,270	=23	24	739,310	=23	132	558,102	=23	54	623,334
adrienl_02	8	≥31	660	81,215,335	≥31	861	79,504,888	≥30	3600	8,969,574	≥30	3600	51,978,315
atomix_06	8	=13	18	242	=13	18	181	=13	17	136	=13	18	138
atomix_13	8	=28	24	1,401,827	=28	21	682,305	=28	26	146,132	=28	27	146,132
atomix_18	8	=13	18	1648	=13	18	1194	=13	18	707	=13	18	718
atomix_22	8	≥25	543	67,584,077	≥26	584	63,784,653	≥26	3600	18,739,179	≥26	3600	45,384,504
atomix_29	8	=22	20	304,658	=22	20	141,648	=22	28	83,590	=22	28	139,654
atomix_30	8	=13	18	1648	=13	18	1194	=13	18	707	=13	18	718
kai_05	8	=27	498	74,829,335	=27	325	45,821,771	=27	2611	18,634,573	=27	2620	31,317,245
kai_17	8	=23	22	500,963	=23	22	329,671	=23	36	94,425	=23	35	212,829
katomic_11	8	=23	159	19,882,485	=23	75	6,499,337	=23	1223	4,899,164	=23	733	9,060,753
katomic_19	8	≥31	433	58,202,893	≥31	467	57,091,035	≥31	3600	24,388,959	≥31	3600	45,056,314
katomic_31	8	=29	196	34,194,628	=29	124	18,664,928	=29	404	3,161,605	=29	424	6,621,812

Source: the author.

Table E.3: PDB Experiment 3/5

Instance	n	No PDB			Static PDB			Dynamic PDB			Multi-Goal PDB		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
marbles_11	8	=28	186	22,485,798	=28	171	19,824,635	=28	1170	17,658,058	=28	1222	17,658,058
unitopia_10	8	≥39	526	103,225,531	≥39	589	107,715,630	≥39	3600	31,202,456	≥40	3600	59,903,861
adrienl_04	9	≥34	516	99,520,890	≥34	571	101,514,965	≥34	3600	59,478,478	≥34	3600	59,596,102
atomix_05	9	≥36	455	66,636,140	≥36	528	66,867,912	≥36	3600	22,101,948	≥36	3600	39,400,286
atomix_07	9	≥26	431	59,906,097	=27	351	45,931,513	=27	2175	23,719,191	=27	2239	23,719,191
atomix_12	9	=14	18	2506	=14	18	2286	=14	18	92	=14	18	145
atomix_16	9	≥27	491	59,533,377	≥27	563	62,522,384	≥27	3600	21,869,957	≥27	3600	38,056,675
katomic_05	9	=27	134	21,309,141	=27	82	10,481,012	=27	162	1,053,973	=27	280	3,099,254
katomic_06	9	=27	327	54,058,835	=27	229	35,089,002	=27	1785	20,803,147	=27	1818	20,803,147
katomic_14	9	≥27	423	61,966,682	≥28	467	61,297,719	≥28	3600	35,173,244	≥28	3600	34,985,892
katomic_32	9	=19	20	323,260	=19	19	88,760	=19	38	62,346	=19	31	141,615
katomic_38	9	≥33	463	77,742,782	≥34	535	80,862,264	≥34	3600	43,799,916	≥34	3600	44,161,298
unitopia_06	9	=31	468	57,992,469	=31	320	35,851,379	≥30	3600	19,556,181	≥30	3600	29,753,239
adrien_04	10	≥25	1093	43,570,830	≥25	1485	39,699,704	≥24	3600	2,158,023	≥24	3600	23,056,421
adrien_05	10	≥26	875	35,191,945	≥26	1234	36,409,100	≥26	3600	2,544,582	≥26	3600	22,654,999
atomix_10	10	≥29	416	42,461,881	≥29	456	41,882,754	≥28	3600	16,308,937	≥29	3600	29,606,772
atomix_28	10	=29	85	10,384,620	=29	88	9,895,172	=29	706	7,897,803	=29	703	7,897,803
kai_09	10	≥34	390	48,323,586	≥35	416	48,507,793	≥35	3600	33,400,650	≥35	3600	32,978,686
katomic_09	10	≥31	356	44,274,522	≥31	404	47,957,249	=32	1724	16,055,645	=32	1707	16,055,645
katomic_25	10	≥33	391	51,777,687	≥35	395	48,499,480	≥35	3600	34,338,765	≥35	3600	34,567,739
katomic_33	10	≥49	674	63,622,846	≥50	736	59,907,853	≥48	3600	8,777,053	≥49	3600	30,621,261
katomic_35	10	≥32	379	59,712,760	≥34	406	61,331,911	≥35	3600	36,592,724	≥35	3600	35,804,907
katomic_61	10	≥53	405	56,297,870	≥53	448	56,008,022	≥53	3600	17,249,053	≥53	3600	30,739,791
unitopia_07	10	≥34	379	50,456,445	≥34	410	51,661,418	≥34	3600	31,188,574	≥34	3600	35,278,021
katomic_47	11	=29	32	2,058,349	=29	26	1,060,372	=29	22	43,436	=29	23	43,436
katomic_66	11	≥31	350	35,716,591	≥31	389	37,474,626	≥32	3600	23,041,381	≥32	3600	23,221,827
atomix_08	12	≥34	362	32,315,494	≥34	391	32,460,663	≥35	3600	19,535,589	≥35	3600	18,930,874
atomix_14	12	≥35	345	29,642,063	≥35	376	29,700,679	≥35	3600	18,145,551	≥35	3600	18,217,256
atomix_15	12	≥36	366	31,384,517	≥36	409	32,097,828	≥36	3600	21,336,874	≥36	3600	19,645,578
atomix_21	12	≥31	786	27,974,963	≥31	837	26,940,770	≥31	3600	9,445,512	≥31	3600	15,485,877
kai_07	12	≥33	370	34,185,196	≥33	420	36,200,166	≥34	3600	21,506,643	≥34	3600	21,086,518
kai_08	12	≥35	351	34,248,304	≥36	403	36,618,193	≥36	3600	19,415,792	≥36	3600	19,359,370

Source: the author.

Table E.4: PDB Experiment 4/5

Instance	n	No PDB			Static PDB			Dynamic PDB			Multi-Goal PDB		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
kai_18	12	≥34	271	24,196,706	≥34	293	24,108,059	≥35	3600	20,911,390	≥35	3600	20,559,472
kai_20	12	≥38	310	30,043,943	≥38	325	29,234,056	≥39	3600	16,482,745	≥39	3600	16,268,242
kai_22	12	≥33	348	32,573,982	≥33	364	31,456,707	≥34	3600	20,962,127	≥34	3600	20,508,822
katomic_07	12	≥23	725	33,634,825	=24	709	25,582,015	=24	3360	2,823,005	=24	2650	13,447,435
katomic_12	12	≥35	822	44,459,538	≥36	1210	49,793,697	≥34	3600	2,912,660	≥35	3600	20,861,633
katomic_13	12	≥41	317	28,162,721	≥41	345	28,694,961	≥41	3600	20,134,152	≥41	3600	19,898,722
katomic_18	12	≥46	1428	31,716,634	≥46	1551	31,847,353	≥46	3600	4,213,334	≥46	3600	15,145,201
katomic_27	12	≥46	295	30,487,175	≥46	313	30,038,940	≥47	3600	24,640,795	≥47	3600	24,728,554
katomic_28	12	≥36	373	41,518,698	≥37	403	41,135,244	≥38	3600	22,846,027	≥38	3600	22,402,479
katomic_42	12	≥34	494	40,758,365	≥34	519	40,093,468	≥34	3600	19,589,735	≥34	3600	19,742,188
katomic_62	12	≥51	318	32,472,028	≥51	344	33,326,741	≥51	3600	24,926,331	≥51	3600	24,970,426
katomic_63	12	≥41	408	43,970,240	≥41	443	43,479,427	≥40	3600	13,708,735	≥41	3600	25,751,652
katomic_67	12	≥30	397	41,205,819	≥32	475	45,467,272	≥31	3600	14,259,893	≥32	3600	25,234,339
marbles_15	12	≥37	1788	48,437,736	≥37	1853	48,435,965	≥36	3600	10,123,878	≥36	3600	9,929,242
unitopia_09	12	≥43	416	40,493,250	≥43	474	41,406,090	≥43	3600	12,830,440	≥43	3600	24,983,010
katomic_34	13	≥35	304	24,107,907	≥36	335	24,778,262	≥37	3600	17,535,250	≥37	3600	16,984,937
atomix_20	14	=29	337	26,506,951	=29	160	11,574,396	=29	212	845,399	=29	216	845,399
atomix_25	14	≥36	386	22,580,850	≥37	417	20,655,594	≥39	3600	6,573,982	≥38	3600	12,586,821
kai_14	14	≥40	335	25,311,694	≥40	392	25,703,637	≥41	3600	8,293,470	≥41	3600	15,721,962
kai_21	14	≥42	338	28,551,322	≥42	380	29,340,276	≥42	3600	11,934,201	≥42	3600	11,611,828
kai_24	14	≥40	315	21,266,572	≥40	335	21,391,115	≥40	3600	14,817,572	≥40	3600	14,448,487
kai_25	14	≥33	330	22,648,412	≥33	370	23,454,718	≥33	3600	13,022,204	≥33	3600	12,891,890
katomic_17	14	≥31	387	23,867,185	≥31	427	22,860,820	≥34	3600	6,716,724	≥32	3600	12,042,518
katomic_22	14	≥31	501	24,379,339	≥32	609	24,189,444	≥32	3600	3,179,957	≥31	3600	11,513,114
katomic_45	14	≥38	266	22,728,680	≥39	275	22,389,686	≥41	3600	17,171,916	≥41	3600	16,947,551
15-puzzle	15	=34	18	1,453,014	=34	16	626,928	=34	30	380,647	=34	30	380,647
atomix_17	15	≥36	654	22,459,789	≥36	638	21,638,502	≥36	3600	9,452,624	≥36	3600	9,324,463
atomix_19	15	≥27	329	24,576,634	≥28	359	25,412,563	≥30	3600	10,355,621	≥30	3600	10,090,935
kai_12	15	≥35	314	18,525,081	≥35	357	19,434,552	≥36	3600	9,200,169	≥36	3600	9,099,308
katomic_15	15	≥35	740	24,666,899	≥35	751	24,237,434	≥36	3600	7,982,705	≥36	3600	7,933,047
katomic_16	15	≥42	320	25,639,546	≥42	325	23,989,743	≥43	3600	10,807,475	≥43	3600	10,607,996
katomic_29	15	≥56	287	24,306,760	≥57	294	22,728,029	≥58	3600	15,504,711	≥58	3600	15,386,292

Source: the author.

Table E.5: PDB Experiment 5/5

Instance	n	No PDB			Static PDB			Dynamic PDB			Multi-Goal PDB		
		Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.	Moves	Time(s)	Nodes Exp.
katomic_41	15	≥34	400	20,762,730	≥34	506	20,379,180	≥34	3600	3,418,789	≥34	3600	10,999,631
katomic_55	15	≥47	306	25,049,689	≥47	339	26,071,264	≥48	3600	14,881,089	≥48	3600	14,727,947
katomic_56	15	≥48	283	23,986,260	≥49	305	23,882,537	≥50	3600	14,484,614	≥50	3600	14,389,950
atomix_24	16	≥29	401	28,506,922	≥29	414	27,798,912	≥31	3600	12,372,710	≥31	3600	12,316,550
kai_28	16	≥46	254	14,630,723	≥46	295	14,549,114	≥48	3600	10,288,128	≥48	3600	10,202,386
katomic_21	16	≥25	395	29,326,866	≥26	413	28,943,892	≥27	3600	14,343,394	≥27	3600	14,162,018
katomic_40	16	≥56	351	32,636,619	≥56	380	31,444,230	≥58	3600	14,040,485	≥58	3600	13,983,200
katomic_51	16	≥39	316	21,061,341	≥39	381	23,537,979	≥40	3600	11,044,516	≥40	3600	10,858,345
katomic_53	16	≥25	551	21,595,083	≥25	597	21,544,641	≥26	3600	5,037,349	≥26	3600	9,462,184
katomic_54	16	≥35	282	21,298,003	≥35	305	20,794,088	≥36	3600	14,522,724	≥36	3600	14,458,349
katomic_59	16	≥27	326	15,475,405	≥27	414	15,462,612	≥26	3600	3,274,403	≥27	3600	11,122,515
katomic_64	16	≥53	368	24,516,740	≥53	438	24,905,394	≥54	3600	7,201,643	≥53	3600	11,307,986
marbles_10	16	=24	16	21,324	=24	16	16,508	=24	17	15,566	=24	17	15,566
katomic_39	17	≥46	272	20,398,950	≥47	287	19,585,878	≥49	3600	11,887,444	≥49	3600	11,859,533
katomic_48	17	≥56	264	16,139,318	≥57	275	15,904,728	≥59	3600	7,823,579	≥59	3600	7,737,548
katomic_50	17	≥41	359	24,736,636	≥42	381	24,542,647	≥43	3600	7,192,513	≥43	3600	11,574,591
katomic_65	17	≥31	302	35,777,006	≥31	318	36,113,505	≥31	3600	17,879,893	≥31	3600	17,969,184
katomic_49	18	≥45	278	18,385,996	≥45	322	18,230,038	≥46	3600	10,687,009	≥46	3600	10,660,043
kai_27	19	≥59	242	10,074,866	≥60	278	10,241,191	≥61	3600	5,383,429	≥61	3600	5,350,418
katomic_52	19	≥53	256	15,094,790	≥53	294	15,642,556	≥54	3600	8,155,252	≥54	3600	8,075,235
atomix_27	20	≥45	650	10,270,513	≥45	678	10,327,799	≥46	3600	4,160,483	≥46	3600	4,143,178
katomic_24	20	≥36	3600	8,051,122	≥36	3600	7,789,643	≥37	3600	448,884	≥36	3600	2,916,517
kai_29	21	≥62	264	11,465,703	≥63	294	11,375,083	≥65	3600	5,532,461	≥65	3600	5,489,127
katomic_30	21	≥51	270	14,001,792	≥51	297	13,472,166	≥55	3600	6,357,042	≥55	3600	6,284,067
katomic_44	21	≥47	271	16,030,106	≥48	300	15,299,391	≥52	3600	5,997,355	≥52	3600	5,859,866
katomic_37	24	≥54	270	8,196,266	≥54	307	8,455,662	≥56	3600	3,252,668	≥56	3600	3,226,823
katomic_43	26	≥64	245	8,184,587	≥65	266	7,353,377	≥65	3600	3,365,681	≥65	3600	3,292,784
marbles_20	32	≥37	3387	46,133,258	≥37	3546	47,268,499	≥36	3600	7,265,501	≥36	3600	7,089,841

Source: the author.

APPENDIX F — INITIAL HEURISTIC VALUES

Table F.1: Initial Heuristic Values 1/4

Instance	n	Standard Heuristic	Static PDB ($k=3$)	Dynamic PDB ($k=2$)	Multi-Goal PDB ($k=2$)	Generalized A*	Best LB
adrien_01	3	6	6	6	6	6	7
atomix_01	3	8	8	8	8	8	13
kai_01	3	4	4	4	4	4	9
katomic_01	3	8	8	8	8	8	15
katomic_36	3	4	4	4	4	4	9
marbles_04	3	5	5	5	5	5	22
marbles_13	3	6	6	6	6	6	18
unitopia_01	3	8	9	9	8	9	11
adrienl_05	4	6	7	7	6	7	12
atomix_23	4	5	5	7	6	8	10
atomix_26	4	7	7	7	7	7	14
kai_06	4	9	9	9	9	9	14
kai_19	4	13	13	13	13	13	19
katomic_20	4	13	13	13	13	13	18
katomic_23	4	8	8	8	8	8	18
marbles_01	4	6	6	6	6	6	11
marbles_03	4	10	10	10	10	11	22
unitopia_02	4	14	14	14	14	14	22
adrien_02	5	10	10	10	10	10	17
atomix_02	5	16	17	17	16	17	21
atomix_11	5	10	10	10	10	10	14
kai_02	5	15	15	16	15	16	24
kai_11	5	10	11	11	10	11	15
katomic_02	5	18	18	18	18	18	27
katomic_10	5	15	15	16	16	16	19
katomic_57	5	16	16	16	16	16	21
marbles_02	5	9	10	10	10	10	15
marbles_05	5	14	14	14	14	15	25
marbles_06	5	12	12	12	12	13	14
unitopia_03	5	12	12	14	13	14	16
adrien_03	6	9	9	9	9	9	12
adrien_06	6	10	10	11	10	11	15
atomix_03	6	12	12	12	12	12	16
atomix_04	6	14	15	15	15	15	23
kai_03	6	12	12	12	12	12	16
katomic_03	6	14	14	14	14	14	20
katomic_04	6	14	14	16	14	17	23
katomic_58	6	13	13	14	14	14	17

Source: the author.

Table F.2: Initial Heuristic Values 2/4

Instance	n	Standard Heuristic	Static PDB ($k=3$)	Dynamic PDB ($k=2$)	Multi-Goal PDB ($k=2$)	Generalized A*	Best LB
marbles_08	6	12	12	12	12	12	23
marbles_12	6	13	13	13	13	13	28
marbles_14	6	16	16	16	16	18	22
unitopia_04	6	18	18	19	18	19	20
unitopia_05	6	13	13	13	13	13	20
adrienl_01	7	13	13	13	13	13	20
adrienl_03	7	10	10	10	10	10	22
atomix_09	7	11	11	11	11	12	20
katomic_08	7	13	14	14	14	15	26
katomic_26	7	26	27	28	26	29	36
katomic_46	7	19	19	19	19	19	24
katomic_60	7	15	15	15	15	15	19
unitopia_08	7	17	17	17	17	17	23
adrienl_02	8	21	21	22	21	22	-
atomix_06	8	12	12	12	12	12	13
atomix_13	8	23	23	24	24	24	28
atomix_18	8	10	10	10	10	11	13
atomix_22	8	17	18	19	17	19	-
atomix_29	8	17	18	18	17	18	22
atomix_30	8	10	10	10	10	11	13
kai_05	8	18	18	19	19	20	27
kai_17	8	19	19	19	19	19	23
katomic_11	8	15	16	17	15	17	23
katomic_19	8	22	22	22	22	22	-
katomic_31	8	16	18	20	19	20	29
marbles_11	8	19	19	19	19	19	28
unitopia_10	8	28	28	29	29	29	-
adrienl_04	9	24	24	24	24	24	-
atomix_05	9	28	28	28	28	28	-
atomix_07	9	18	19	19	19	20	27
atomix_12	9	11	11	13	12	13	14
atomix_16	9	20	20	20	20	20	-
katomic_05	9	19	19	21	20	22	27
katomic_06	9	15	15	16	16	16	27
katomic_14	9	19	20	20	20	22	-
katomic_32	9	13	15	15	13	15	19
katomic_38	9	22	23	24	24	25	-
unitopia_06	9	24	25	24	24	26	31
adrien_04	10	17	17	18	17	18	-

Source: the author.

Table F.3: Initial Heuristic Values 3/4

Instance	n	Standard Heuristic	Static PDB ($k=3$)	Dynamic PDB ($k=2$)	Multi-Goal PDB ($k=2$)	Generalized A*	Best LB
adrien_05	10	21	21	21	21	22	-
atomix_10	10	22	22	22	22	24	-
atomix_28	10	21	21	21	21	21	29
kai_09	10	29	29	29	29	29	-
katomic_09	10	24	24	25	25	25	-
katomic_25	10	28	29	30	30	31	-
katomic_33	10	38	40	41	40	-	-
katomic_35	10	24	27	27	27	31	-
katomic_61	10	48	48	49	48	49	-
unitopia_07	10	27	27	27	27	28	-
katomic_47	11	27	27	27	27	28	29
katomic_66	11	26	26	26	26	26	-
atomix_08	12	30	30	31	31	32	-
atomix_14	12	31	31	31	31	31	-
atomix_15	12	32	32	32	32	32	-
atomix_21	12	27	27	27	27	28	-
kai_07	12	29	29	29	29	30	-
kai_08	12	32	32	32	32	34	-
kai_18	12	29	30	30	30	31	-
kai_20	12	33	33	34	34	36	-
kai_22	12	29	29	31	31	31	-
katomic_07	12	18	18	19	18	20	24
katomic_12	12	28	28	29	28	32	-
katomic_13	12	38	38	38	38	39	-
katomic_18	12	44	44	44	44	-	-
katomic_27	12	43	43	43	43	43	-
katomic_28	12	31	32	33	33	33	-
katomic_42	12	28	28	29	29	29	-
katomic_62	12	46	46	46	46	-	-
katomic_63	12	33	33	33	33	34	-
katomic_67	12	24	25	25	25	27	-
marbles_15	12	31	31	31	31	31	-
unitopia_09	12	39	39	39	39	39	-
katomic_34	13	30	31	32	32	33	-
atomix_20	14	24	24	25	25	25	29
atomix_25	14	31	33	35	34	-	-
kai_14	14	37	37	38	38	-	-
kai_21	14	39	39	39	39	-	-
kai_24	14	37	37	37	37	37	-

Source: the author.

Table F.4: Initial Heuristic Values 4/4

Instance	n	Standard Heuristic	Static PDB ($k = 3$)	Dynamic PDB ($k = 2$)	Multi-Goal PDB ($k = 2$)	Generalized A*	Best LB
kai_25	14	28	28	28	28	-	-
katomic_17	14	26	26	29	27	-	-
katomic_22	14	25	26	26	25	30	-
katomic_45	14	36	37	38	38	-	-
15-puzzle	15	4	10	10	10	34	34
atomix_17	15	31	31	32	32	33	-
atomix_19	15	22	22	25	25	27	-
kai_12	15	31	31	32	32	32	-
katomic_15	15	31	31	33	33	-	-
katomic_16	15	38	38	39	39	42	-
katomic_29	15	54	55	55	55	-	-
katomic_41	15	30	30	30	30	31	-
katomic_55	15	43	43	44	44	44	-
katomic_56	15	44	44	45	45	45	-
atomix_24	16	24	24	26	26	30	-
kai_28	16	43	44	44	44	-	-
katomic_21	16	20	21	21	21	26	-
katomic_40	16	50	50	53	53	-	-
katomic_51	16	35	36	36	36	36	-
katomic_53	16	20	20	21	21	-	-
katomic_54	16	30	30	31	31	31	-
katomic_59	16	22	22	22	22	23	-
katomic_64	16	50	50	51	50	-	-
marbles_10	16	16	16	16	16	24	24
katomic_39	17	43	43	45	45	-	-
katomic_48	17	53	55	56	56	-	-
katomic_50	17	35	35	37	37	-	-
katomic_65	17	26	26	26	26	31	-
katomic_49	18	41	41	43	43	-	-
kai_27	19	58	58	59	59	-	-
katomic_52	19	51	51	52	52	-	-
atomix_27	20	42	42	43	43	44	-
katomic_24	20	33	33	35	33	-	-
kai_29	21	61	61	62	62	-	-
katomic_30	21	49	50	54	54	-	-
katomic_44	21	44	44	50	50	-	-
katomic_37	24	51	51	53	53	-	-
katomic_43	26	63	64	64	64	-	-
marbles_20	32	28	28	28	28	-	-

Source: the author.

APPENDIX G — FINAL SOLVER RESULTS

Table G.1: Our Final Solution vs. Hüffner et al. (2001)'s 1/4

Instance	n	Hüffner et al. (2001)'s			Our Solution		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
adrien_01	3	=7	97	330	=7	34	11
atomix_01	3	=13	109	9458	=13	19	316
kai_01	3	=9	31	661	=9	19	111
katomic_01	3	=15	123	9344	=15	20	429
katomic_36	3	=9	88	2524	=9	20	263
marbles_04	3	=22	408	130,733	=22	20	2493
marbles_13	3	=18	60	42,481	=18	19	4925
unitopia_01	3	=11	65	1624	=11	25	81
adrienl_05	4	=12	412	299,336	=12	130	16,740
atomix_23	4	=10	46	1538	=10	24	879
atomix_26	4	=14	205	33,721	=14	28	8687
kai_06	4	=14	71	23,909	=14	22	4085
kai_19	4	=19	166	236,139	=19	23	18,345
katomic_20	4	=18	61	16,099	=18	22	2561
katomic_23	4	=18	317	260,856	=18	30	15,141
marbles_01	4	=11	18	2742	=11	19	623
marbles_03	4	=22	90	305,224	=22	19	48,587
unitopia_02	4	=22	42	191,058	=22	19	50,822
adrien_02	5	=17	86	2,456,375	=17	24	251,180
atomix_02	5	=21	19	46,353	=21	19	7835
atomix_11	5	=14	38	17,968	=14	22	2414
kai_02	5	=24	23	998,173	=24	19	157,627
kai_11	5	=15	31	44,096	=15	19	6319
katomic_02	5	=27	86	1,303,898	=27	20	86,113
katomic_10	5	=19	7	25,651	=19	20	3874
katomic_57	5	=21	18	179,752	=21	19	25,631
marbles_02	5	=15	37	171,558	=15	19	14,232
marbles_05	5	=25	30	235,820	=25	19	50,645
marbles_06	5	=14	9	2842	=14	19	1014
unitopia_03	5	=16	37	12,195	=16	20	1278
adrien_03	6	=12	34	11,970	=12	35	1182
adrien_06	6	=15	54	214,855	=15	29	53,251
atomix_03	6	=16	16	175,199	=16	19	25,551
atomix_04	6	=23	59	28,507,754	=23	33	3,510,934
kai_03	6	=16	16	175,199	=16	19	25,551
katomic_03	6	=20	27	1,298,229	=20	21	244,187
katomic_04	6	=23	71	1,361,808	=23	20	193,498
katomic_58	6	=17	12	116,629	=17	19	12,167

Source: the author.

Table G.2: Our Final Solution vs. Hüffner et al. (2001)'s 2/4

Instance	n	Hüffner et al. (2001)'s			Our Solution		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
marbles_08	6	=23	64	15,624,411	=23	30	2,799,525
marbles_12	6	=28	173	72,805,973	=28	89	14,492,305
marbles_14	6	=22	10	61,353	=22	18	15,949
unitopia_04	6	=20	13	44,442	=20	19	7909
unitopia_05	6	=20	35	940,639	=20	22	175,528
adrienl_01	7	=20	133	4,522,601	=20	95	1,008,911
adrienl_03	7	=22	654	238,729,460	=22	951	26,845,732
atomix_09	7	=20	16	2,497,729	=20	21	601,858
katomic_08	7	=26	831	477,625,886	=26	549	112,281,722
katomic_26	7	=36	452	284,211,961	=36	142	25,770,175
katomic_46	7	=24	17	2,294,027	=24	20	266,748
katomic_60	7	=19	22	211,552	=19	20	29,577
unitopia_08	7	=23	35	6,506,879	=23	24	739,310
adrienl_02	8	≥33	3600	1,707,098,508	≥31	861	79,504,888
atomix_06	8	=13	6	1293	=13	18	181
atomix_13	8	=28	16	6,430,548	=28	21	682,305
atomix_18	8	=13	11	9892	=13	18	1194
atomix_22	8	=27	2266	943,223,533	≥26	584	63,784,653
atomix_29	8	=22	15	1,856,294	=22	20	141,648
atomix_30	8	=13	12	9892	=13	18	1194
kai_05	8	=27	544	315,337,836	=27	325	45,821,771
kai_17	8	=23	17	3,518,865	=23	22	329,671
katomic_11	8	=23	210	122,789,263	=23	75	6,499,337
katomic_19	8	-	-	-	≥31	467	57,091,035
katomic_31	8	=29	228	143,112,488	=29	124	18,664,928
marbles_11	8	=28	660	88,325,861	=28	171	19,824,635
unitopia_10	8	≥41	3600	1,317,755,067	≥39	589	107,715,630
adrienl_04	9	≥36	3600	1,106,661,012	≥34	571	101,514,965
atomix_05	9	≥38	3600	1,048,265,144	≥36	528	66,867,912
atomix_07	9	=27	672	356,079,418	=27	351	45,931,513
atomix_12	9	=14	9	10,749	=14	18	2286
atomix_16	9	=29	1850	981,308,861	≥27	563	62,522,384
katomic_05	9	=27	177	108,651,436	=27	82	10,481,012
katomic_06	9	=27	288	166,981,668	=27	229	35,089,002
katomic_14	9	≥29	2668	836,024,185	≥28	467	61,297,719
katomic_32	9	=19	25	833,607	=19	19	88,760
katomic_38	9	≥35	3094	924,033,872	≥34	535	80,862,264
unitopia_06	9	=31	571	328,569,611	=31	320	35,851,379
adrien_04	10	≥26	3600	1,727,133,356	≥25	1485	39,699,704

Source: the author.

Table G.3: Our Final Solution vs. Hüffner et al. (2001)'s 3/4

Instance	n	Hüffner et al. (2001)'s			Our Solution		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
adrien_05	10	≥27	3600	945,686,552	≥26	1234	36,409,100
atomix_10	10	≥31	3600	1,127,126,271	≥29	456	41,882,754
atomix_28	10	=29	98	53,822,181	=29	88	9,895,172
kai_09	10	≥36	2334	551,220,229	≥35	416	48,507,793
katomic_09	10	=32	1176	589,666,142	≥31	404	47,957,249
katomic_25	10	≥35	2076	584,516,478	≥35	395	48,499,480
katomic_33	10	≥51	3504	989,106,558	≥50	736	59,907,853
katomic_35	10	≥34	2745	851,098,956	≥34	406	61,331,911
katomic_61	10	≥55	3600	1,423,335,054	≥53	448	56,008,022
unitopia_07	10	≥36	3359	832,875,517	≥34	410	51,661,418
katomic_47	11	=29	8	2,594,709	=29	26	1,060,372
katomic_66	11	≥33	2314	654,472,530	≥31	389	37,474,626
atomix_08	12	≥35	3600	406,921,253	≥34	391	32,460,663
atomix_14	12	≥36	3600	477,721,036	≥35	376	29,700,679
atomix_15	12	≥38	2778	610,191,870	≥36	409	32,097,828
atomix_21	12	≥32	3600	159,798,083	≥31	837	26,940,770
kai_07	12	≥34	3600	504,455,009	≥33	420	36,200,166
kai_08	12	≥37	2543	721,543,630	≥36	403	36,618,193
kai_18	12	≥36	3458	1,005,810,506	≥34	293	24,108,059
kai_20	12	-	-	-	≥38	325	29,234,056
kai_22	12	≥34	3600	652,733,904	≥33	364	31,456,707
katomic_07	12	=24	2427	302,608,420	=24	709	25,582,015
katomic_12	12	≥37	3600	603,713,028	≥36	1210	49,793,697
katomic_13	12	≥43	3600	964,297,677	≥41	345	28,694,961
katomic_18	12	≥47	3600	177,204,474	≥46	1551	31,847,353
katomic_27	12	≥47	1753	462,753,171	≥46	313	30,038,940
katomic_28	12	≥38	3066	944,475,610	≥37	403	41,135,244
katomic_42	12	≥35	3600	326,807,091	≥34	519	40,093,468
katomic_62	12	-	-	-	≥51	344	33,326,741
katomic_63	12	-	-	-	≥41	443	43,479,427
katomic_67	12	≥32	3600	1,323,375,073	≥32	475	45,467,272
marbles_15	12	≥34	3600	178,748	≥37	1853	48,435,965
unitopia_09	12	≥44	3600	506,812,291	≥43	474	41,406,090
katomic_34	13	≥37	1999	441,614,044	≥36	335	24,778,262
atomix_20	14	=29	1968	187,441,572	=29	160	11,574,396
atomix_25	14	≥37	3600	248,978,222	≥37	417	20,655,594
kai_14	14	≥42	3530	914,325,888	≥40	392	25,703,637
kai_21	14	-	-	-	≥42	380	29,340,276
kai_24	14	≥41	2684	185,944,459	≥40	335	21,391,115

Source: the author.

Table G.4: Our Final Solution vs. Hüffner et al. (2001)'s 4/4

Instance	n	Hüffner et al. (2001)'s			Our Solution		
		# Moves	Time(s)	Nodes Exp.	# Moves	Time(s)	Nodes Exp.
kai_25	14	≥35	3600	317,432,120	≥33	370	23,454,718
katomic_17	14	≥32	3600	301,972,372	≥31	427	22,860,820
katomic_22	14	≥32	3600	340,878,443	≥32	609	24,189,444
katomic_45	14	≥40	1681	464,058,677	≥39	275	22,389,686
15-puzzle	15	=34	33	6,009,587	=34	16	626,928
atomix_17	15	≥36	3600	94,046,263	≥36	638	21,638,502
atomix_19	15	≥29	3600	353,777,325	≥28	359	25,412,563
kai_12	15	≥36	3600	288,303,629	≥35	357	19,434,552
katomic_15	15	≥36	3600	95,662,295	≥35	751	24,237,434
katomic_16	15	≥43	3323	315,685,653	≥42	325	23,989,743
katomic_29	15	≥57	2462	811,894,030	≥57	294	22,728,029
katomic_41	15	≥36	3600	1,248,620,284	≥34	506	20,379,180
katomic_55	15	≥48	2355	663,547,586	≥47	339	26,071,264
katomic_56	15	≥50	1557	430,680,182	≥49	305	23,882,537
atomix_24	16	≥30	3600	262,871,336	≥29	414	27,798,912
kai_28	16	≥47	996	198,517,336	≥46	295	14,549,114
katomic_21	16	≥26	3600	348,297,019	≥26	413	28,943,892
katomic_40	16	-	-	-	≥56	380	31,444,230
katomic_51	16	≥41	2549	726,362,274	≥39	381	23,537,979
katomic_53	16	≥26	3600	242,994,747	≥25	597	21,544,641
katomic_54	16	≥36	1102	264,356,116	≥35	305	20,794,088
katomic_59	16	=28	2987	892,463,476	≥27	414	15,462,612
katomic_64	16	≥55	3550	1,027,710,926	≥53	438	24,905,394
marbles_10	16	=24	716	88,305	=24	16	16,508
katomic_39	17	≥47	1892	617,547,824	≥47	287	19,585,878
katomic_48	17	≥57	1170	297,645,087	≥57	275	15,904,728
katomic_50	17	≥43	1369	457,603,193	≥42	381	24,542,647
katomic_65	17	≥32	1312	460,199,416	≥31	318	36,113,505
katomic_49	18	≥46	2521	268,337,502	≥45	322	18,230,038
kai_27	19	-	-	-	≥60	278	10,241,191
katomic_52	19	≥54	1490	319,862,589	≥53	294	15,642,556
atomix_27	20	≥45	3600	14,295,747	≥45	678	10,327,799
katomic_24	20	≥36	3600	13,039,944	≥36	3600	7,789,643
kai_29	21	≥64	1596	354,301,347	≥63	294	11,375,083
katomic_30	21	≥52	1024	115,621,912	≥51	297	13,472,166
katomic_44	21	≥49	2565	348,829,831	≥48	300	15,299,391
katomic_37	24	≥55	3600	272,524,375	≥54	307	8,455,662
katomic_43	26	≥65	1628	87,029,639	≥65	266	7,353,377
marbles_20	32	=0	3600	0	≥37	3546	47,268,499

Source: the author.