# Solving Sokoban

Timo Virkkala <timo.virkkala@iki.fi>

Helsinki April 12, 2011

Pro gradu -tutkielma – Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

| Tiedekunta — Fakultet — Faculty | Laitos — Institution — Department |
|---|---|
| Faculty of Science | Department of Computer Science |

Tekijä — Författare — Author

Timo Virkkala <timo.virkkala@iki.fi>

Työn nimi — Arbetets titel — Title

Solving Sokoban

Oppiaine — Läroämne — Subject

Computer Science

| Työn laji — Arbetets art — Level | Aika — Datum — Month and year | Sivumäärä — Sidoantal — Number of pages |
|---|---|---|
| Pro gradu -tutkielma – Master's Thesis | April 12, 2011 | 64 pages + 12 appendix pages |

Tiivistelmä — Referat — Abstract

The game of Sokoban is an intriguing platform for algorithm and AI research. While the rules are quite simple, the problem itself is not. The domain has been proven NP-Hard and PSPACE-complete and even simple puzzles require a large amount of computation to solve. This difficulty is caused by long solution depths, a large branching factor and the existence of deadlocks. However, bypassing these complications and finding efficient algorithms for solving Sokoban can have useful implications for real-life scenarios as well as other problem domains in computer science.

In this thesis we present an overview of the techniques that have been applied to the domain of Sokoban. We also explore some of these in more detail and run experiments to see how they perform when applied to different search strategies. Furthermore, by adding a simple modification we are able to significantly improve the results achieved by a previous study.

ACM Computing Classification System (CCS):
A.1 [Introductory and Survey],
I.2.1 [Games],
I.2.8 [Graph and tree search strategies],

Avainsanat — Nyckelord — Keywords

sokoban, graph search, survey, pruning, single-agent, search

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

# Contents

**Appendices**

**1 Result Tables**

# 1 Introduction

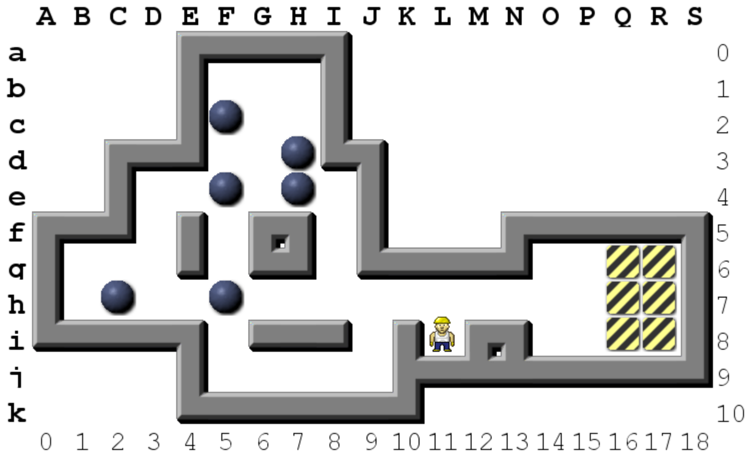


**Figure 1:** Puzzle #1 of the 90-puzzle test set [Mye01]

Sokoban is a game in which the player tries to push all the stones in a maze onto goal squares. Any stone can be placed on any goal square. The stones are moved by pushing them one square at a time by the player character. The player cannot move through walls or stones and can only push the stones along the four cardinal directions, not diagonally. Also, stones cannot go through walls or each other, and only one stone can be pushed at a time. The objective is to place all stones on the goal squares with a minimum number of pushes.

The game of Sokoban is an intriguing platform for algorithm and AI research. While the rules are quite simple, the problem is most definitely not so. As it is NP-Hard [DZ99] and PSPACE-complete [Cul97], even simple levels require quite an amount of computation to solve. This difficulty is caused by the long solution depths (frequently in the hundreds), by the branching factor, which can at times reach values over 100 [JS01], and by the existence of unsolvable positions, *deadlocks*. However, as Sokoban can be seen as a simplification of a robot tending storage units in a warehouse, bypassing these complications and finding efficient algorithms for solving Sokoban can have useful implications for real-life scenarios as well as other problem domains in computer science.

There have been many studies on Sokoban presented in the scientific literature. Various research groups have tried various strategies for creating a Sokoban solver algorithm. So far, none of them have been so successful as to be able to solve any given Sokoban puzzle. The most successful solver presented in scientific studies, Rolling Stone [JS01], is only able to solve two thirds of a challenging 90-puzzle problem set. To be able to achieve better results, one must first know what approaches have already been explored, and what were the results, so as not to be doomed to repeat history.

In this thesis we present an overview of the techniques that have been applied to the domain of Sokoban. We also explore some of these in more detail and run experiments to see how they perform when applied to different search strategies. Furthermore, by adding a simple modification we are able to significantly improve the results achieved by one study.

The rest of this thesis is structured as follows. In section 2 we present an overview on the game of Sokoban, its rules and its challenges as a problem domain. In section 3 we provide an overview of standard, domain-independent graph search techniques and in section 4 we present a survey of Sokoban-specific search enhancements available in the scientific literature. In section 5 we describe a number of experiments to determine the performance of some of those enhancements, in section 6 we discuss some of the details of our implementation and finally in section 7 we provide and discuss the results of those experiments.

## 2 The Game of Sokoban

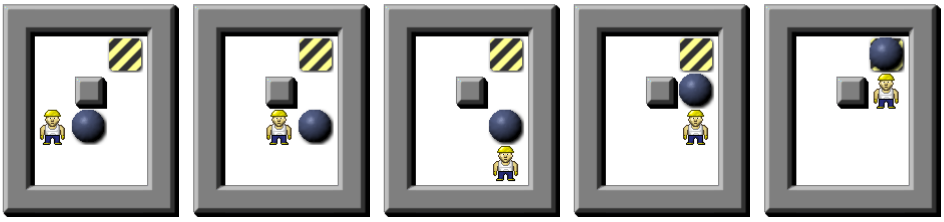
A Sokoban game and playing field consist of a **player** character, a number of **stones**[1], an equal number of **goal** positions and a maze of **floor** positions bounded by **walls**. Figure 1 shows an example of a Sokoban puzzle. The player is at position *Li*, the elements at *Fc*, *Hd*, *Fe*, *He*, *Ch* and *Fh* are stones and the elements at *Qg*, *Rg*, *Qh*, *Rh*, *Qi* and *Ri* are goals[2].

---

[1] Varying terms and metaphors for the pushed objects are used by the many Sokoban implementations and articles out there. Besides the term *stone* used in this thesis, at least box, crate, ball, boulder and money bag have been used. Considering that the word *sokoban* means warehouse keeper in Japanese, boxes or crates would probably be closest to the original. Regardless of the chosen metaphor, the gameplay remains the same.

[2] The notation *Li* means column 11 (L is the 11th letter in the English alphabet) and row 8. This notation is the same as the one used by e.g. [JS01], and was chosen over others (e.g. the one

The rules of the game are simple: the player can move north, south, east and west freely in the floor area of the maze. The player cannot move through walls or stones. If the player tries to move into a position occupied by a stone, that stone is pushed along into the next position – provided that the next position is unoccupied, i.e. it is a floor square and does not contain a stone. The player cannot therefore push more than one stone at a time, nor can he move the stones sideways or pull them. Figure 2 illustrates the various stages of solving a trivially easy Sokoban puzzle.



(a) Starting position (b) 1 push (c) Still 1 push – player moved (d) 2 pushes (e) 3 pushes – finished

**Figure 2:** Solving a trivial 1-stone puzzle

The purpose of the game is to push all the stones into the goal positions. There are always an equal number of stones and goals and any stone can (in principle) be placed on any goal. Scoring can be done by counting either the number of player **moves** or the number of stone **pushes** required to reach the **goal state** (i.e. all the stones are in the goal positions). The total length of the solution in the previous example is 3 pushes – the two player moves from 2(b) to 2(c) do not "cost" anything – if scoring by pushes. If scoring by man moves the solution length is 5 moves (4 of which are shown in the pictures).

There are multiple implementations and puzzle sets of Sokoban. The game was originally created in 1981 by Hiroyuki Imabayashi and published in 1982 by *Thinking Rabbit* [Lis06]. The original game contained only 20 puzzles. After that, several sequels with more puzzles were published, as were several clones with both copied

---

in [Lis06], where it would be *l8*) because it permits us to use two-character notation for all but the largest puzzles (and if extended into, say, the Greek alphabet, even longer), whereas notations that require decimal numbers quickly need a third character. Sequences of stone pushes will be notated *Aa-Ab-Ac Ba-Ca*, which means that the stone on *Aa* was first pushed to *Ab* and then to *Ac*, and after that the stone on *Ba* was pushed to *Ca*.

and original puzzles. Nowadays a quasi-standard puzzle set for Sokoban research is the one provided with the XSokoban implementation [Mye01]. It contains 90 puzzles, all of which are relatively challenging both for human and computer players. One source of easier puzzles are the Microban sets created by David Skinner [Ski00]. Microban1 contains 155 small puzzles which have been designed to illustrate a single game concept each.

As mentioned earlier, Sokoban is made difficult by the large *branching factor* and *solution depth*. In the XSokoban puzzle set the largest encountered branching factor is 136 and the average is 12, while the solution depth ranges from 97 to 674 [JS01]. The puzzle sizes are usually (and always in the XSokoban set) smaller than $20 \times 20$, with walls surrounding the perimeter (so the actual playing area is $18 \times 18$), which would make the search space of all Sokoban problems roughly $10^{98}$ states [Jun99], although the search space of a single Sokoban puzzle is much smaller than that. The median search space size in the XSokoban set is roughly $10^{18}$ [Jun99] states.

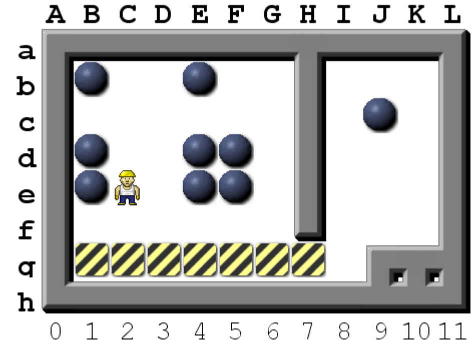
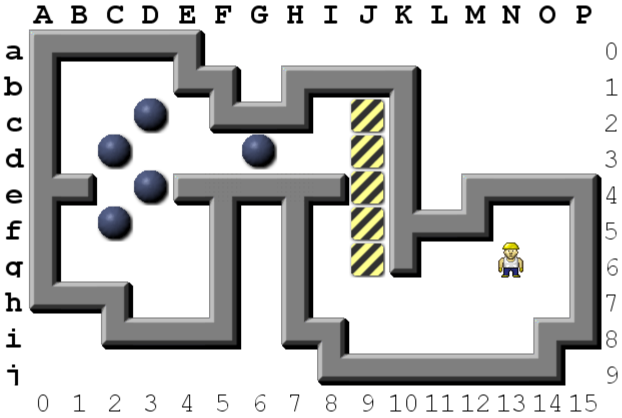
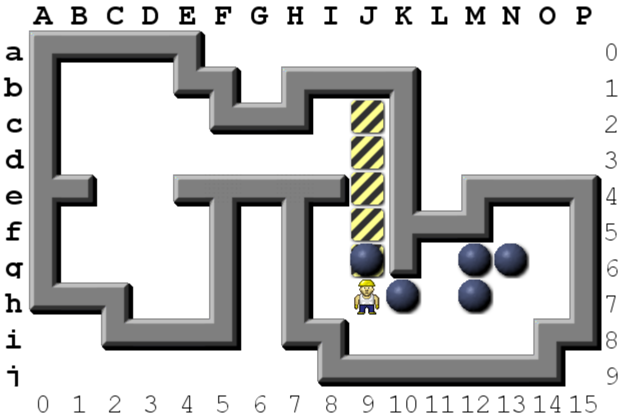


**Figure 3:** Some examples of deadlock situations

Besides the branching factor and solution length, Sokoban is also made difficult by the existence of deadlocks. A **deadlock** is a situation from which the game can no longer be solved. An obvious deadlock situation is one where a stone is pushed into a corner – as the player cannot pull the stones, there is no way of getting the stone out of the corner and therefore the stone can never reach a goal. Some deadlocks, such as this one, are trivial to detect, but others can be more subtle. In extreme cases determining whether a deadlock exists may require actually determining if all stones can in fact be pushed to the goals and thus solving the puzzle. Figure 3 provides some examples of deadlocks. The stone on *Bb* clearly cannot be pushed anywhere. The stone on *Eb* can be pushed, but only along the north wall. The stones in the four stone cluster prevent each other from being pushed, as do the stones on *Bd* and *Be*, and while the stone on *Jc* can move in many directions, it can never leave the room it is in. There are many more possible deadlock configurations. Methods for detecting these are discussed in section 4.4.3.



(a) Initial state



(b) Parking lot full

**Figure 4:** Microban1 puzzle #98 – A good example of a problem that requires pushing all the stones into a parking area

Another difficulty in Sokoban is that in most cases the puzzle cannot be solved by pushing one stone at a time to the goal squares. Most puzzles are constructed in such a way that an initial tangle has to be unraveled before the puzzle becomes straightforward to solve. In some cases the stones have to be actually pushed through the goal area into a *parking area*, from where they can then be pushed to their final positions in the goal. This makes solving such puzzles quite hard (see the results in section 7.1). Figure 4 shows an example of such a situation, while puzzle #50 of the XSokoban set is another, notorious example.

For the purpose of solving Sokoban computationally, the game can be seen as a series of transitions from one state to another. Again, a transition can be either a player move or a stone push. When viewed this way, the game forms a *directed cyclic graph* of states (i.e. transitions from a state to its successor states may be irreversible and there may be transitions that lead to an already encountered state) and the task becomes one of pathfinding, i.e. trying to find a path from the initial state to the goal state.

In addition to pathfinding in this **state space**, we can of course also use pathfinding algorithms in the **game space** itself, i.e. in the actual maze. Operating in the game space is more useful for finding routes for the player and a single stone, while operating in the state space provides access to solving the whole level. Both of these approaches are discussed in more detail later on. Section 4.1 deals with pathfinding in the game space, while section 4.2 discusses pathfinding in the state space.

# 3 A Review of Graph Search Algorithms

Before going into the details of Sokoban solver techniques and algorithms, a brief review of graph search is in order to allow the reader to understand terms such as breadth-first and depth-first search and iterative deepening, which are used often in the following sections. The state space of Sokoban and other single-player games can be seen as a graph, with moves in the game as transitions from state to state. Thus, solving the game usually means searching the graph for a route, preferably an optimal one, from the starting state to the goal state – or *a* goal state if there are more than one.

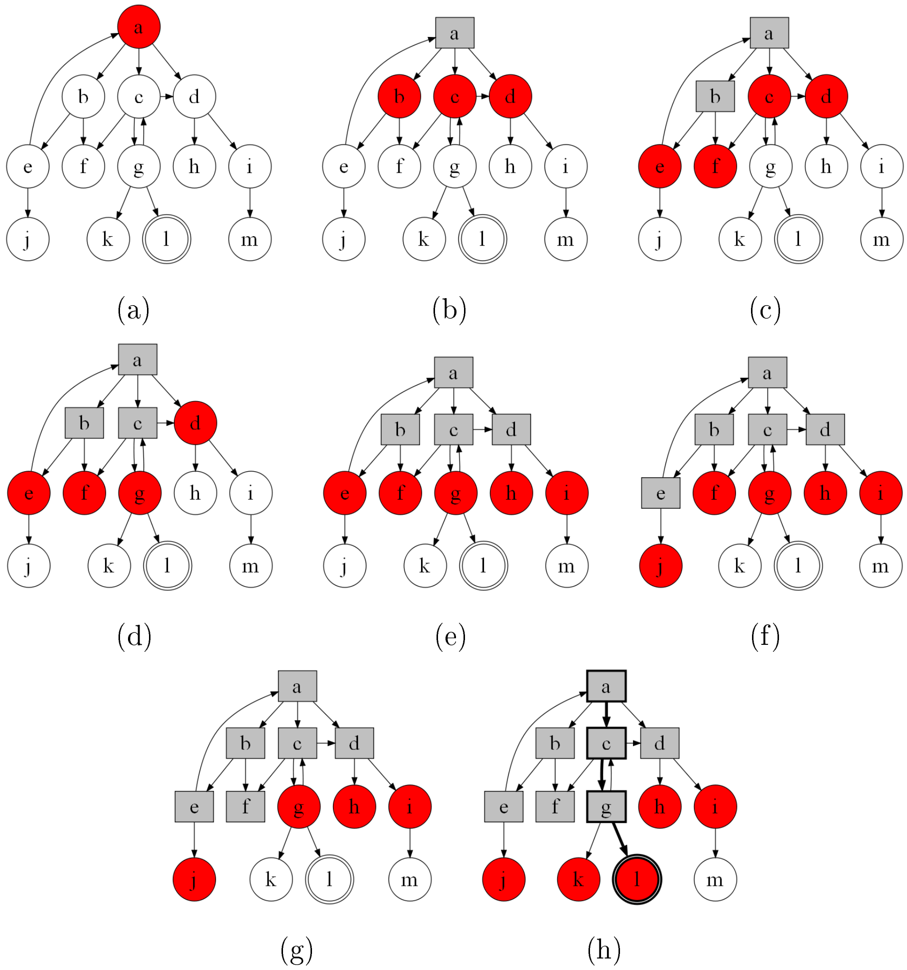
In general, there are two variants of search algorithms: **tree search** and **graph search** [RN09]. The difference between these is that tree search algorithms assume that the searched tree or graph does not contain cycles or multiple routes to any state. Since some states in Sokoban can be reached via multiple routes and a sequence of pushes can lead to back to a state already explored, the search space of Sokoban is clearly a graph instead of a tree. Therefore, we will mostly concern ourselves with graph search algorithms. All of the algorithms below can be either tree search or graph search algorithms depending on how they are implemented.

## 3.1 Uninformed Search: Breadth-First and Depth-First Search

The most basic way of searching for something in a graph (such as a state with certain properties, e.g. a state with all the stones on goal squares in Sokoban) is to check every node until the required node has been found. The most obvious algorithms for this are Breadth-First Search (BFS) and Depth-First Search (DFS) [RN09].

BREADTH-FIRST SEARCH searches through the graph by first visiting the root node, then all the direct successors of the root node, then all the direct successors of those etc. The search visits all the nodes at a given depth before any deeper nodes. To avoid processing the same node more than once (as we are dealing with graphs, not trees, and possibly even cyclical ones) each explored node is stored and for all new nodes a check is performed against this storage. The time complexity of the algorithm is $O(b^d)$, where $b$ is the branching factor of the graph (the number of successors each node has) and $d$ is the depth of the solution. This requires that the nodes are tested for the termination criteria (i.e. whether the node is the one we are looking for) when *generated* rather than when *expanded*; in that case the complexity would be $O(b^{d+1})$ [RN09]. Figure 5 gives an example of how BFS progresses in a graph, while algorithm 1 gives the pseudocode of the algorithm.

Breadth-First Search is **complete** and **optimal** - that is, it is guaranteed to find the solution (given enough time and memory and assuming the solution depth and branching factor are finite) and the solution it finds is the lowest-cost one (if the cost of a path is a non-decreasing function of the solution depth, i.e. all the arcs in the graph have a non-negative cost associated with them). However, for many interesting problems the assumption about enough time and memory is not reasonable. With today's fast processors the main problem is memory - as the search needs to keep every generated node in memory (to check for duplicates and to be able to provide a path to the goal node), at any time there will be $O(b^{d-1})$ nodes in the *explored* set and $O(b^d)$ nodes in the *open* (waiting to be explored) set, also known as the *search*



(a) (b) (c)

(d) (e) (f)

(g) (h)

**Figure 5:** Breadth-First Search searching for a route from node $a$ to node $l$. The white circles are *unexplored* nodes, the red circles are in the *frontier* and the grey rectangles have already been *explored*. In subfigure (h) the route, shown in bold lines, has been found.

*frontier*. Thus, with most interesting problems the search will run out of memory long before processing time becomes an issue.

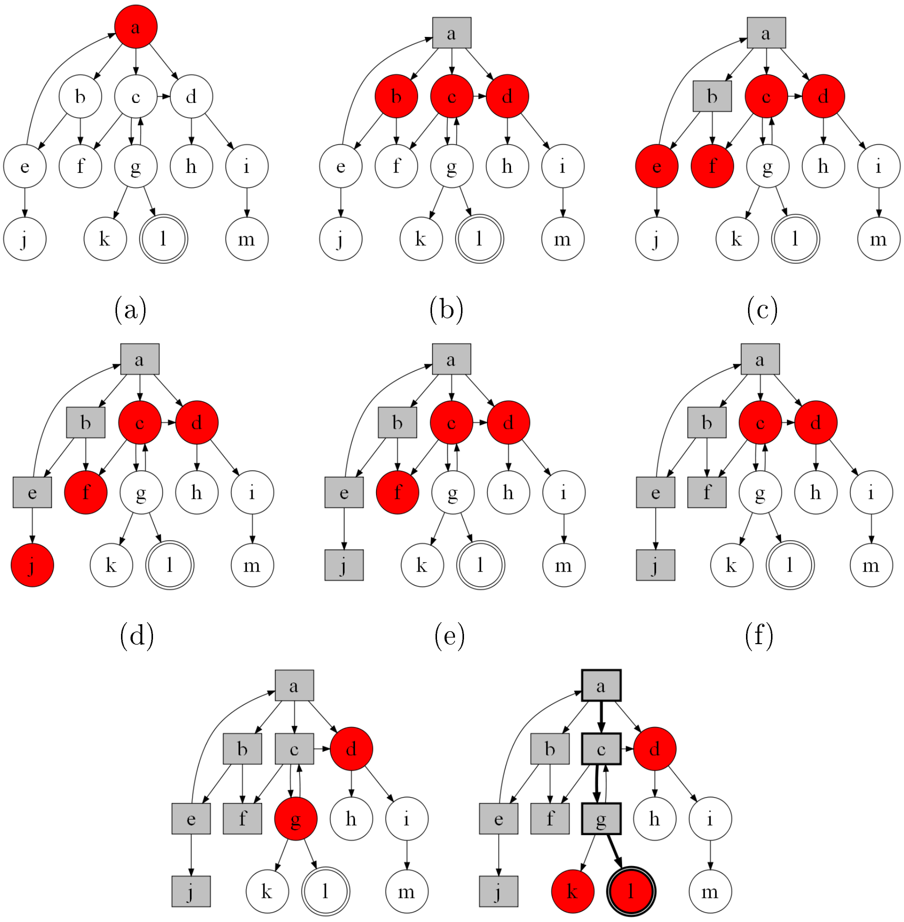
An algorithm that avoids this memory bottleneck is DEPTH-FIRST SEARCH. Instead of progressing all the way through each search depth before moving on to the next, DFS always follows the successors to the maximum depth before moving on to the next successor. So, from the root node it will generate the first successor, then the first successor of that one etc. until it reaches a node which has no successors; a *leaf node*. From there it will backtrack to the deepest node that still has unexplored successors and explore the next successor of that, and so on [RN09]. Figure 6 shows

```
problem – An instance of the graph
node – A node with State =problem.InitialState, PathCost =0
if problem.IsGoalState?(node.State) then
    return Solution(node)
frontier – A FIFO queue with node as the only element
explored – An empty set
while not frontier.IsEmpty?() do
    node := frontier.Pop()/* Returns the shallowest node in frontier */
    explored.Add(node)
    foreach action in problem.Actions(node.State) do
        child := ChildNode(problem, node, action)
        if not (child.State in explored or child.State in frontier)) then
            if problem.IsGoalState?(child.State) then
                return Solution(child)
            frontier.Insert(child)
return failure
```

**Algorithm 1:** Breadth-First Search algorithm [RN09]

how DFS progresses through the same example graph.

As DFS searches everything in a given subtree before moving on to the next, at any time it only needs to keep the current path in memory. This makes it $O(bm)$ in space, $b$ being the branching factor and $m$ the maximum depth. However, while DFS avoids the memory limitations associated with BFS, it also has a number of drawbacks. In finite search spaces DFS is complete, if implemented in a way that it checks if a node already exists in the current path, thus avoiding cycles. It, however, is not optimal. As it explores nodes depth-first, it is quite possible to find longer than optimal paths simply because it might encounter that branch of the path first. If the search space is infinite, the search might not find a solution at all, even if the solution actually exists at a low depth in the graph. The time complexity of DFS is $O(b^m)$, where $b$ is, again, the branching factor and $m$ is the maximum depth of the search space. This is clearly higher - and can be *significantly* higher - than the $O(b^d)$ of BFS. Even worse, if the search space is a true graph with many possible



(a) (b) (c) (d) (e) (f) (g) (h)

**Figure 6:** Depth-First Search searching for a route from node $a$ to node $l$. In this graph the route is found in the same number of iterations as with Breadth-First Search (figure 5), but this is not always the case. The frontier and explored sets are however different, as is the order in which nodes are explored.

routes to a given state, the search will end up searching the same subgraphs over and over again leading to immense duplicated effort. The way to avoid this is to use a **transposition table**, which holds information about already visited nodes and can aid detection of duplicated nodes. This, however, leads quickly to the same memory limitations that BFS suffers from, making DFS an even worse candidate.

```
problem – An instance of the graph
node – A node with State =problem.InitialState, PathCost =0
if problem.IsGoalState?(node.State) then
    return Solution(node)
frontier – A LIFO queue with node as the only element
explored – An empty set
while not frontier.IsEmpty?() do
    node := frontier.Pop()/* Returns the node inserted last */
    explored.Add(node)
    foreach action in problem.Actions(node.State) do
        child := ChildNode(problem, node, action)
        if not (child.State in explored or child.State in frontier)) then
            if problem.IsGoalState?(child.State) then
                return Solution(child)
            frontier.Insert(child)
return failure
```

**Algorithm 2:** Depth-First Search algorithm

## 3.2 Informed Search: A*

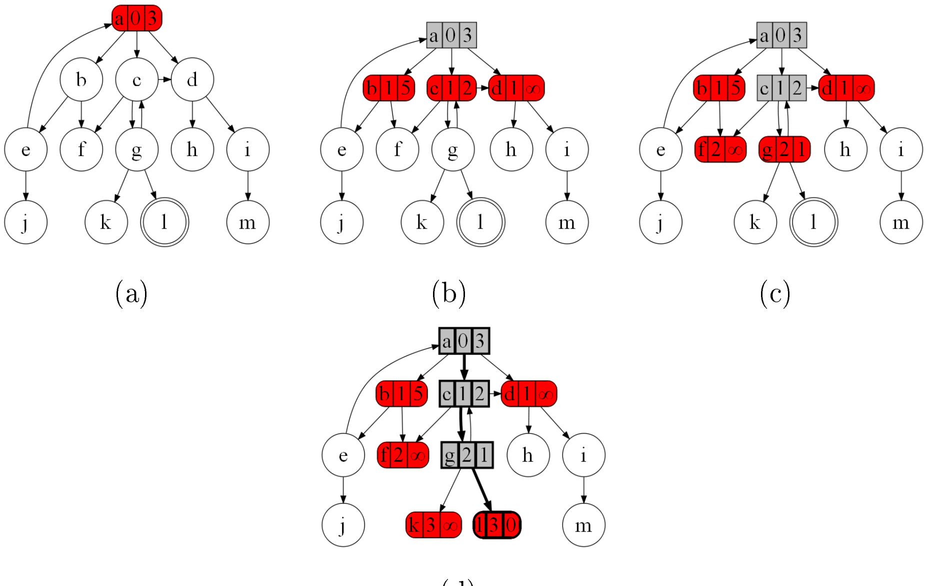
If the search could be guided to seek out the right branch toward the goal node right away, the possibly-never-terminating nature of Depth-First Search would not be nearly as bad a problem. With **informed** or **heuristic** search we have a way to do just that. Of course, being able to always guide the search down the right path would mean we would have to know the path beforehand, but that does not mean that we cannot make educated guesses.

One way to do guide the search is to make a heuristic estimate about the remaining path to the goal along the chosen route. This is precisely what the A* algorithm is about. For each generated node it computes both the path length so far and an estimate of the remaining path length[3] and then proceeds to the node with the

---

[3] Note that when speaking about path lengths, we assume that each arc in the graph has the same cost, and thus cost and length are the same thing. Path length is easier to grasp intuitively than path cost and makes more sense in the context of Sokoban; thus the choice of term.

lowest total estimated path length.



(a) (b) (c) (d)

**Figure 7:** A* searching for a route from node $a$ to node $l$, assuming a *perfect* heuristic (one that always returns the correct remaining path length). Here the frontier and explored nodes are shown with a record shape, with the middle value being the path length so far and the rightmost value being the path length remaining (i.e. the heuristic).

The properties of A* depend heavily on the heuristic function used. If the chosen heuristic function is **admissible** (i.e. it never overestimates) and **consistent** (the estimate for node $n$ is never greater than the cost of reaching $n$'s successor $n'$ from $n$ plus the estimate for node $n'$) then the algorithm is both complete and optimal. For Sokoban, one such possible heuristic would be the sum of the Manhattan distances (distance along the X axis plus distance along the Y axis, disregarding walls and other obstacles) of each stone to some goal - either the nearest goal or, with some more computational effort, an assigned goal for each stone. Another, more accurate heuristic for Sokoban is presented in section 4.2.2.

Furthermore, A* has been proven *optimally efficient* in its category - that is, within the class of search algorithms that search for solutions extending from the root and use the same heuristic information [RN09]. This means that A* is guaranteed to expand at most the same amount of nodes as any other such algorithm. The time and space complexity of A* depend on the heuristic function.

```
problem – An instance of the graph
node – A node with State =problem.InitialState, PathCost =0
node.TotalCost := node.PathCost + Heuristic(node)
frontier – A priority queue ordered by TotalCost, with node as the only element
explored – An empty set
while not frontier.IsEmpty?() do
    node := frontier.Pop()/* Returns the lowest-cost node */
    if problem.IsGoalState?(node.State) then
        return Solution(node)
    explored.Add(node)
    foreach action in problem.Actions(node.State) do
        child := ChildNode(problem, node, action)
        child.TotalCost = child.PathCost + Heuristic(node)
        if not (child.State in explored or child.State in frontier) then
            frontier.Insert(child)
        else if child.State in frontier with higher TotalCost then
            frontier.Replace(child)/* Replace the higher-cost state */
return failure
```

**Algorithm 3:** A* algorithm [RN09]

## 3.3 Depth-Limited Search and Iterative Deepening

Another way to avoid ending up in an infinitely deepening search branch with Depth-First Search is to limit the search depth. This is unsurprisingly called DEPTH-LIMITED SEARCH. It works exactly like DFS, except the search is only allowed to expand nodes up to a given depth. If the depth of a node exceeds the limit, it is treated exactly like a leaf node.

The depth limit brings with it an obvious problem: if the solution is deeper than the limit, the search will never find it. The answer is to start with a conservative limit and, if the search ends without finding a solution, to increase the limit and try again. This is called ITERATIVE DEEPENING. If the search is started with a depth limit of 0 and increased in increments of 1, the search is guaranteed to be complete (with the same assumptions as with BFS). The memory requirement is low, only $O(bd)$ ($d$ being the depth limit; as with DFS, this of course excludes the transposition table, resulting in wasted computation), but the time complexity suffers from having to generate the lower depth nodes multiple times.

## 3.4 Iterative Deepening A* (IDA*)

The ideas of informed search and iterative deepening can of course be combined. The result is ITERATIVE DEEPENING A* or IDA*, which has so far been the most successful search algorithm for Sokoban (albeit with a number of enhancements; see section 4.2). The biggest change from Iterative Deepening Depth-First Search is that rather than using the path length so far as the comparison for the depth limit we rather use the estimated total path length, that is, the path length so far plus the heuristic estimate. In addition, the generated moves are sorted by the estimated total path length and the shortest ones are tried first. This achieves the guiding effect which makes IDA* a guided search algorithm and a relative of A*.

## 3.5 Bidirectional Search

While all the discussion on search algorithms so far has assumed that the root node of the search is the *initial state* of the game, this does not have to be the case. We can indeed reverse the search by starting from the *goal state* and trying to then locate the initial state. All that needs to be changed is the way we generate successors. Depending on the properties of the search space this can lead to better or worse performance. For Sokoban the implications of reverse search are discussed in section 4.2.5. In the context of Sokoban, in reverse search the successor states are generated by *pulling* stones from the goal state towards the initial state.

The change from forward to reverse search soon leads to the idea of bidirectional searching. Instead of searching just from the initial state or the goal state and trying to find the other, we can initiate the search from *both* and try to find the point where the search fronts meet. The rationale for this is that as each depth of the search tends to have more nodes than the preceding depth, combining two shallower search frontiers would result in less wasted search effort. For instance, in a graph with a

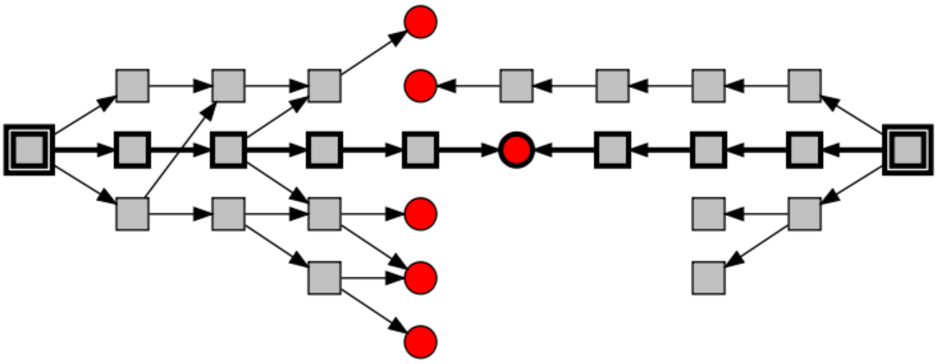


**Figure 8:** Bidirectional search meeting in the middle

branching factor of $b = 4$ and the goal at depth 10, depth 1 would have 4 nodes, depth 2 would have 16 nodes and so on, finally having $4^{10} = 1048576$ nodes at depth 10. This will mean generating a maximum of $\sum_{k=0}^{d} b^k = \sum_{k=0}^{10} 4^k = 1398101$ nodes in the worst case, or $O(b^d)$. But if we search from both directions at the same time, we will only have to search a maximum of $\sum_{k=0}^{d/2} b^k$ nodes from both directions, giving a total of $2\sum_{k=0}^{5} 4^k = 2 \times 1365 = 2730$ nodes, or $O(b^{d/2})$. So, in theory, bidirectional search can give enormous savings. The practical benefits will of course be dependent on the true attributes of the graph.

# 4 Tools for Solving Sokoban

Several different approaches to solving Sokoban have been attempted and documented in the scientific literature. Perhaps the most thoroughly documented is University of Alberta's Sokoban solver Rolling Stone [JS97, JS98a, JS98b, JS98c, JS98d, JS99, Jun99, JS01], which is able to solve 59 of the 90 puzzles in the XSokoban set. While it is based on Iterative Deepening A* search, it contains a number of both domain-independent and Sokoban-specific search enhancements and heuristics, which allow it to perform quite admirably. Many of these are discussed in the following sections. Rolling Stone builds on the success of other search-based solvers [Jun99], but unfortunately little has been published about these earlier efforts.

While the makers of Rolling Stone discovered that a general-purpose planning approach is infeasible for Sokoban [JS01], another team has succesfully applied planning to Sokoban by adding abstraction layers. Their solver Power Plan [BMS02] is able to solve 10 puzzles[4]. Their approach is to treat a Sokoban puzzle as a graph of rooms and tunnels instead of individual positions and thus decompose the initial problem into several simpler sub-problems. This approach is discussed further in section 4.6.

An interesting multi-agent search approach (see section 4.5) was used in the TALKING STONES solver, first introduced in [Lis06] and further discussed in [DLG08]. Their solver is able to solve 54 problems (61 with a little manual help), nearly rivaling the performance of ROLLING STONE. While their multi-agent approach is a refreshing contrast to the single-agent search method of the above solvers, perhaps it is their discovery of an easily-solvable subclass of Sokoban puzzles and/or game states that will prove to be more useful for future Sokoban solver developers. See section 4.5 for details.

Other solvers have been implemented, many with similar techniques, but have not been discussed in scientific literature. The Sokoban Wiki (http://www.sokobano.de/wiki/) provides statistics for many such solvers as well as a description of some of the algorithms used by one of them, the YASS solver (Yet Another Sokoban Solver) [Dam10].

Regardless of the chosen basic solving method, a number of Sokoban-specific issues need to be addressed. The rest of this section provides discussion about the various components of a successful Sokoban solver.

## 4.1 Pathfinding in Game Space

The first step in trying to find a solution to a Sokoban puzzle is to be able to determine if a given stone can reach a given position. A simple way to do this is to adapt a generic pathfinding algorithm, such as A* [HNR68] to be able to account for *pushability*, i.e. to find routes such that the player is always in a position to push the stone in the right direction. This is easily accomplished by giving the algorithm a third dimension to work with: the side of the stone the player is on. So, in addition to the $x$ and $y$ dimensions of the Sokoban puzzle the search space also has four layers in a third dimension – one for each of the four cardinal directions.

---

[4] While they claim this to be only a preliminary result, no further results appear to have been published.

Whereas the cost of moving the stone in the $x$ and $y$ dimensions is always non-zero (as it always requires stone pushes), the cost of moving in the third dimension is zero provided the positions around the stone are reachable, i.e. the player can move around the stone without needing to push any stones. Figure 9 illustrates the concept. For the heuristic function A* needs to estimate the length of the remaining path, something as simple as Manhattan distance from the stone to the target position can be used – or even no heuristic at all, which degenerates the A* algorithm into Dijkstra's algorithm [RN09]. One possible approach for the heuristic is to precalculate walking distances, disregarding pushability, between each pair of points in the maze and then use those as the estimates in later calculations involving pushability and perhaps other stones as obstacles.

In some situations also the player's path from position to position might need to be solved. In most cases the length of the path does not matter (especially when trying to optimize for stone pushes, not player moves) and as the levels are small, it is sufficient to just run a flood fill of all the **accessible** positions from the player's current location when encountering a new game state and then allow the player to teleport to all accessible locations without worrying about the exact path taken. If however an exact path is needed, a standard A* algorithm will provide just that. Of course, doing both will result in wasted effort, as the flood fill would also be able to provide a shortest path with slight modifications, but in practice these two pieces of information (which positions are accessible and what is to shortest path to reach a position) will rarely be needed at the same time. As both algorithms are quite simple, having both in the solver's arsenal should provide useful.

## 4.2 Pathfinding in State Space

If instead of considering pushing the stones in the game space we consider the game as a graph of states and transitions, searching becomes conceptually much clearer. Also, we are much more easily able to search for solutions involving several pushed stones, not just one. Therefore, to search for actual solutions for the puzzle it is more advantageous to operate in the state space.

Any algorithm presented in the section 3 can be used – provided they are the graph search versions. When Sokoban is treated as a state graph it is *directed* and *cyclic*

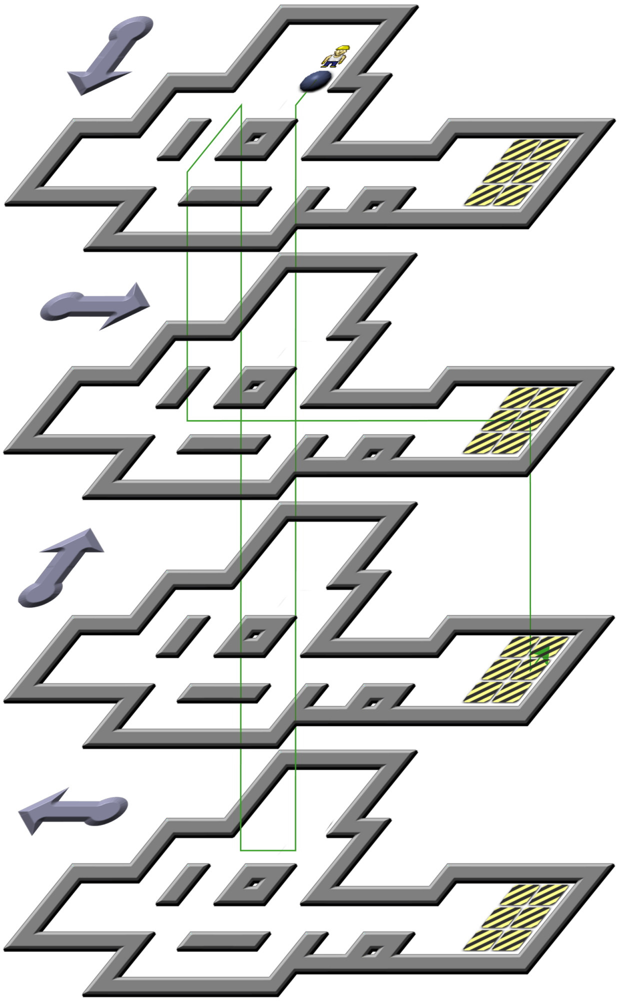


**Figure 9:** The layers of the pushable A* algorithm. The topmost layer shows moves directed south, the next one east, the third north and the fourth layer shows moves west. The cost of all moves from one layer to another are zero when the new player position is accessible.

with multiple possible routes to a given state and thus even simple puzzles become intractable if we choose a tree search algorithm. Unfortunately, this means that we can all but forget about only using a linear amount of memory, as we could do with algorithms like Depth-First Search when operating in an acyclic tree-like environment. After all, Sokoban (when played on an unrestricted-size board) has been proven PSPACE-complete [Cul97], which implies that in most cases the amount of memory required will in fact be polynomial.

As searching in Sokoban is hard, we must find ways of directing the search so that we consider moves leading to the solution as early as possible. Naturally, even considering the inclusion of heuristics leads us to choose an informed algorithm such as A* or its memory bounded variant IDA*. This is precisely what the most successful solver so far, ROLLING STONE, uses [JS01]. It contains a number of enhancements to the basic search, most of which are introduced in the following sections.

### 4.2.1 Transposition Tables

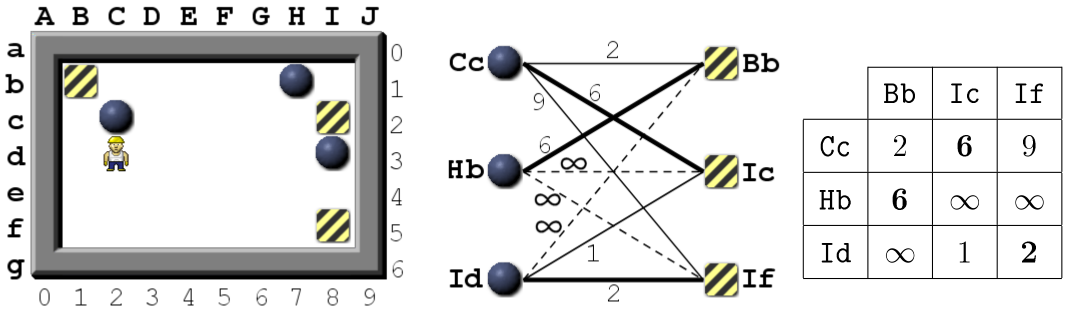
As the search algorithm needs to be a graph search, we need a way to detect if a given state has already been explored. This is the *explored* set in algorithms 2 and 3. A common way to implement it is to use a large *hash table*. In Sokoban the state (from the point of view of transpositions) obviously consists of the positions of the stones and the position of the player. But as the common approach is to optimize for stone pushes, not player moves, storing such a naive representation of state in the transposition table would in fact miss quite a lot of transpositions. After all, moving the player without pushing any stones does not affect the path length and therefore should not affect game state. Therefore two states should be considered equivalent if the stones are at the same positions and the player positions are connected by a legal player path. Thus it is better to consider the state as consisting of the stone positions and the *reachable* area of the player. Since the reachable area is easy to compute, a good way to implement this is to store a *normalized* player position, e.g. the topmost, leftmost reachable position instead of the actual position of the player.

The size of the transposition table can be also limited. This makes the search algorithm a kind of hybrid between tree search and graph search. The advantage is of course the ability to search for solutions to larger puzzles without running out of memory, while an obvious disadvantage is that it may lead to duplicated search effort. Thus, we run into the usual tradeoff between space and time.

### 4.2.2 Lower bound estimation

To be able to use Depth-Limited or Iterative Deepening Search one must be able to estimate the solution depth or waste too much time in an exhaustive search of the lower depths of the search graph. In addition, the ability to estimate a *lower bound* on the solution is useful for the heuristic function used in a guided search like A*.

The developers of Rolling Stone have presented two alternatives for lower bound estimation [JS01]: **Simple Lower Bound** and **Minimum Matching Lower Bound**. The first one, Simple Lower Bound, calculates the sum of the Manhattan distances of each stone to its closest goal. While this can be useful in some situations, in practice it underestimates grossly in most cases. The main reason for this is simple: in Sokoban, only one stone can occupy each goal! By choosing the closest goal for each stone we are clearly overlooking this simple fact.



(a) (b)

| | Bb | Ic | If |
|---|---|---|---|
| Cc | 2 | **6** | 9 |
| Hb | **6** | ∞ | ∞ |
| Id | ∞ | 1 | **2** |

(c)

**Figure 10:** Minimum matching example [JS01]. In this case, the minimum-matching algorithm determines that while stone *Cc* is closest to goal *Bb*, stone *Hb* can never reach any other goals than *Bb*. Similarly, stone *Id* can never reach goal *Bb*. Thus, the algorithm assigns *Cc* to *Ic*, *Hb* to *Bb* and *Id* to *If* and determines the lower bound to be $6 + 6 + 2 = 14$.

The Minimum Matching Lower Bound algorithm [JS01] fares much better. It generates a *minimum-cost*, *perfect bipartite matching* of the stones and goals. Each stone is assigned to a goal so that the total sum of distances (along actual, pushable paths, albeit in an empty maze) is minimized. The actual algorithm used is the **Hungarian method** [Kuh55], which is $O(N^3)$, where $N$ is the number of stones. Clearly, this is an expensive calculation, even with the many possible optimizations [Jun99]. However, it produces much more accurate results than the simple lower bound, and it also provides the *parity* of the final solution – i.e., if the value returned by the algorithm is even, then the number of pushes in the final solution is also even. This makes it possible to skip every other iteration in the iterative deepening search. In some cases the minimum matching algorithm can also detect a deadlock – if the stones were positioned in such a way that some goals were *over-*

*committed*, some stones would be left without goal assignments and so the state would be in deadlock.

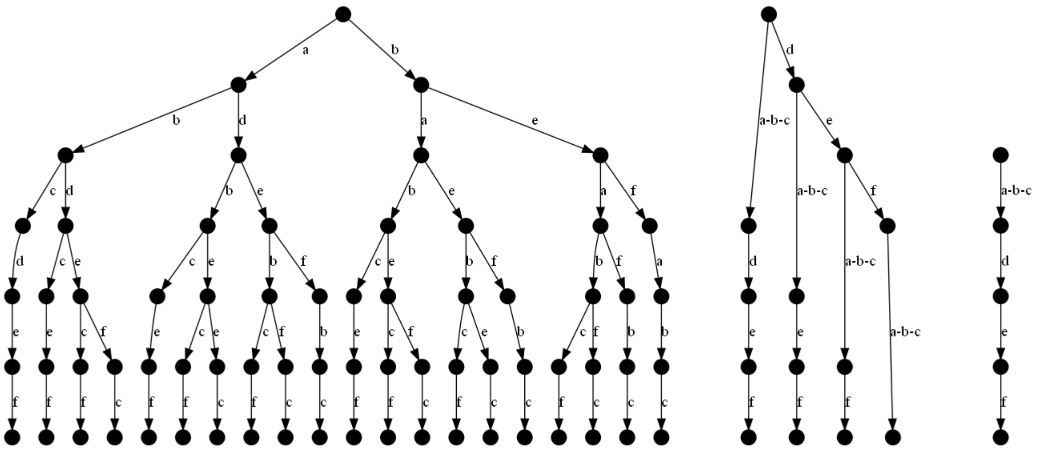
### 4.2.3 Move ordering

While the search effort in informed search methods such as A* is directed toward the solution by estimating the remaining path length, there are still numerous alternatives that have the same estimate. Further direction can be obtained by ordering the available moves by some criteria. When analyzing solutions to Sokoban puzzles the creators of the Rolling Stone solver discovered that the solution paths contain long sequences of pushes targeting a single stone. Therefore, the move ordering scheme used in Rolling Stone is based on *inertia*, i.e. moves which push the stone that was pushed last are tried first [JS01]. Then all the moves that decrease the lower bound, i.e. optimal moves, are tried. The moves are sorted by distance of the pushed stone to its assigned goal. If those prove unsuccessful as well, then the search moves on to the rest of the moves, sorted similarly. In Rolling Stone, this move ordering scheme has proved to be extremely effective – after reaching about 20% of the depth of the search tree the move ordering becomes near perfect [JS01].

### 4.2.4 Macro moves

Because searching in Sokoban is heavily memory-bound, all possible options for reducing the size of the search tree should be exploited. One such option is the utilization of *macro moves*, i.e. collapsing sequences of moves into one move. Obviously the cost of such a macro move is identical to the length of the collapsed subtree.

One possibility for such macro moves are *tunnel macros*. When a stone is pushed into a one-way tunnel (see section 4.3.2), it has to come out from the other end of that tunnel before any other stone can enter that tunnel or before the player can ever reach the other end of it. Therefore the moves that push the stone through the tunnel can be executed right away and no other moves even need to be considered. Thus all other possible moves are discarded and the move sequence is effectively collapsed into a macro move.



(a) No macros (b) Tunnel macro (c) Goal macro

**Figure 11:** The effects of *a-b-c* as a tunnel macro and a goal macro [JS01]

Another macro possibility are *goal macros*. In many Sokoban mazes the goals are grouped in one or more *goal rooms* with usually only one or a few entrances. If a stone is pushed onto such an entrance it can, and should, usually be pushed right through to its final destination. Thus, the move sequence from the entrance to the goal can be replaced with a macro move. In Rolling Stone, no other moves are even considered when a goal macro is present. This is in contrast to tunnel macros – while moves onto a tunnel entrance are substituted with a tunnel macro when they are generated, other moves are still considered alongside that tunnel macro, but when a goal macro is available, all other moves are eliminated. This provides a dramatic reduction in the size of the search space.

Goal macros are only applied when a stone is pushed to the entrance of a goal room. But if a stone elsewhere in the maze can be pushed to its final destination it probably should be pushed there right away. This is the idea behind *goal cuts*, another enhancement in Rolling Stone. It effectively extends goal macros further up the search tree, resulting in even larger reductions in search space.

### 4.2.5 Reversed and Bidirectional Solving

As discussed in section 3.5, starting the search from the initial state and trying to find the goal state does not have to be the only option. As most Sokoban puzzles are designed to provide ample opportunities for deadlocks when pushing, searching for solutions via *pulling* the stones starting from the goal state may be a good technique for avoiding these. Indeed, pulling stones away from the goal state guarantees that we cannot end up in a deadlock in the usual sense. After all, if we can pull stones from the goal state to a given state, we are guaranteed to be able to push the stones to the goal states from that state again. However, when pulling from the goal state we uncover another kind of deadlock. It is possible to end up in a state from where we can no longer reach the *initial* state – that is, we may not be able to pull the stones to their starting positions or if we do, the player may not be able to reach his starting position. However, initial findings suggest that such *pull-deadlocked* states are rarer than *push-deadlocked* ones.

Frank Takes has examined reversed solving of Sokoban in his bachelor thesis [Tak08]. While noting that solving by pulling avoids the usual deadlock states that solving by pushing often runs into, he fails to recognize that solving by pulling creates its own kind of deadlock, although as stated previously this does not seem to be nearly as big a problem as push-deadlocks. In most cases, a pull-deadlock is caused by the player pulling himself into a *corral* (see section 4.4.1) from which it is no longer possible to exit. In such cases the available moves will "dry up" quickly and the deadlock will not cause much lost search effort.

The algorithm used by Takes is simple. It uses two conditions, $X$ and $Y$ to guide the search. While condition $X$ is not satisfied, the stone under consideration is pulled to all unvisited positions. Then, the focus switches to another stone as chosen by condition $Y$. The possible criteria for condition $X$, i.e. when to stop moving a stone, are as follows:

$X_1$ – After each step

$X_2(n)$ – After $n$ steps, for some value of $n$

$X_3$ – When a stone is at a final position (i.e. one of positions of the stones in the initial state of the maze)

$X_4(n)$ – When a stone is $k$ steps away from a final position, with $k$ ranging from 0 to $n$ for some value of $n$

$X_5$ – After a random number of moves

The possible criteria for condition $Y$, i.e. which stone to consider next, are as follows:

$Y_1$ – Every stone. This includes stones that have already been placed.

$Y_2$ – Every unplaced stone.

$Y_3$ – The next stone in *lexicographical* order, meaning an order determined by some numbering given to the stones in advance

$Y_4$ – The next stone as sorted by e.g. each stone's sum of distances to each final position

$Y_5$ – The stone that is currently closest to some final position

$Y_6$ – A random stone

By choosing a different combination of these conditions (and different values for $n$ in $X_2$ and $X_4$) different search behaviors emerge. For instance, choosing $X_1Y_1$ results in a brute-force breadth-first search, examining each possible state and guaranteeing completeness and optimality but gaining little in efficiency, while choosing $X_3Y_2$ efficiently solves puzzles in the Van Lishout subclass (see section 4.5) and choosing $X_4Y_2$ with a sufficient value for $n$ solves puzzles which are nearly in the subclass (note that $X_3$ is the same as $X_4$ with $n = 0$).

Junghanns mentions both reversed and bidirectional approaches in the Failed Ideas section of his thesis [Jun99]. He remarks that while the ideas of backwards and bidirectional search both sound good on paper, and would probably result in a smaller number of nodes searched before finding the solution, they have their problems. For backwards search one large problem is that while in forward search the goals are usually grouped together in just one or two goal areas, in backwards search the "goals", i.e. the starting positions of the stones, are scattered around the maze. This makes it hard to determine the order in which the stones should be positioned. This makes the use of techniques such as goal macros impossible. Another difficulty is presented by deadlocks. While most of the pull-deadlocks result in a situation where the player compresses his own space and soon runs out of available moves, there are also situations where the player can escape the compressed space to work in other areas of the maze, but the stones are in a deadlocked state. This can be hard to detect, especially because pull-deadlocks are harder for humans to visualize and comprehend and are therefore harder to cater for in programming. Junghanns suggests that a deadlock database should be used, similar to what Rolling Stone uses in its forward search but with different patterns for reverse search.

Assuming the problems with reverse search can be solved, for bidirectional search the main problem Junghanns points out is memory consumption[Jun99], specifically that of the search frontiers. In bidirectional search at least one of the search frontiers must be completely maintained in memory so that the search from the other direction can check for matches. This consumes quite a lot of memory. While the concern is still valid, one must take into account that this was written 12 years ago and the available memory in computer systems has grown considerably since then. It is therefore a good idea to investigate if the amount of memory in current computers is sufficient to keep even a large search frontier in memory. In section 5.2 we present an experiment with forward, reverse and bidirectional search where all generated nodes are maintained in memory.

## 4.3 Static Analysis of Puzzle Features

Before even considering any game states, a static analysis of features can reveal crucial information about a Sokoban puzzle. By spending some time on such analysis a large amount of wasted search effort can be avoided. This section introduces a few of such analysis tools.

### 4.3.1 Dead Positions

A position in a Sokoban puzzle is called **dead** if a stone pushed into it can never reach any goal. Such positions can be discovered by a simple algorithm which tries to find a pushable route from all the positions in the puzzle to all the goals. If a stone from that position can be pushed to any goal the position is not dead. For

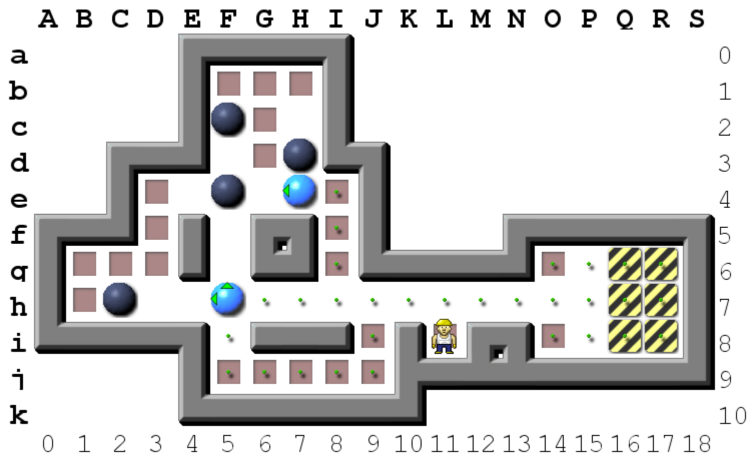


**Figure 12:** The puzzle from figure 1 annotated with some static and dynamic features. The reddish-brown squares show dead positions, green dots show the area accessible to the player at the moment, bright stones are pushable right now and the small arrows on them show the available push directions

reverse solving purposes a position can be also considered dead if a stone at that position could never be pulled to any of the starting positions, i.e. a stone from any of the starting position in the initial state of the puzzle could never be pushed to that position. Figure 12 shows the puzzle in figure 1 annotated with dead positions (the reddish-brown squares) as well as other, dynamic (state-specific) annotations about the available moves. All of the corner positions such as *Fb* are forward-dead (can never reach any goal), while *Gc* and *Gd* are examples of reverse-dead (cannot be reached from the initial state) positions.

While computing dead positions for even a complicated puzzle is a very cheap operation the rewards gained are substantial. On almost any level, in almost any state, knowledge of dead positions allows a number of available moves to be pruned with only a simple lookup.

### 4.3.2 Rooms, Tunnels and Chambers

The algorithms solving Sokoban do not necessarily have to operate only on the level of individual squares; we can also raise the abstraction level. One way to do that is to decompose the puzzle into a graph of rooms and tunnels. A **tunnel** is defined as a part of the maze where the maneuverability of the player is restricted to a width

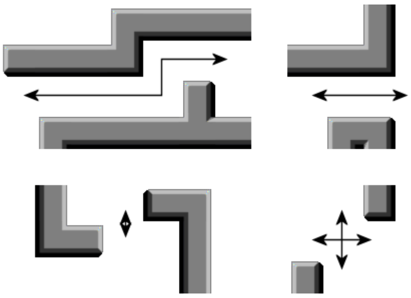


**Figure 13:** Various tunnel types [BMS02]

of one [JS01]. Conversely, a **room** is an area where the player can move more freely. Two points belong to the same room if and only if there is a connection between them that does not cross any tunnel [BMS02]. Figure 13 illustrates various tunnel types, while figure 14 shows a maze decomposed into rooms and tunnels.

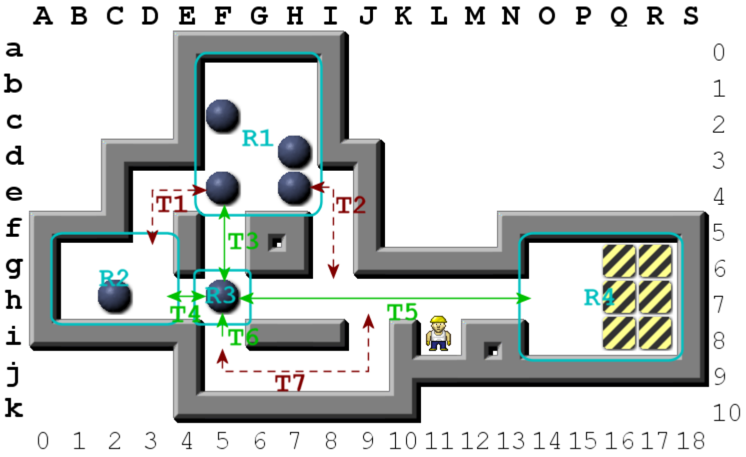


**Figure 14:** Puzzle #1 as a room and tunnel graph. The green, solid-line tunnels are stone tunnels, while the red dashed-line tunnels are only for the player. Note tunnel T6, which is a one-ended tunnel.

A square which, if replaced by a wall, would break the maze into two completely disconnected parts is called an **articulation square**. If a tunnel contains such a square it is a *one-way tunnel* [JS01]. These can be used to e.g. decompose a problem into sub-problems or to implement tunnel macros (see section 4.2.4).

Another way to abstract a Sokoban problem is to decompose it into a graph of **chambers** – areas where each position is *stone-reachable* from each other, i.e. areas where a stone can be pushed from one position to any other position [Sch05]. They can be used for a number of things, such as determining the packing order for goal squares and detecting *structural deadlocks*: chambers that don't have an exit and have less goals than stones. Unfortunately, the only study that discusses them [Sch05] only provides an algorithm for computing them and mentions their potential a few times, but does not actually use them for much.

## 4.4 Dynamic Analysis of Game State

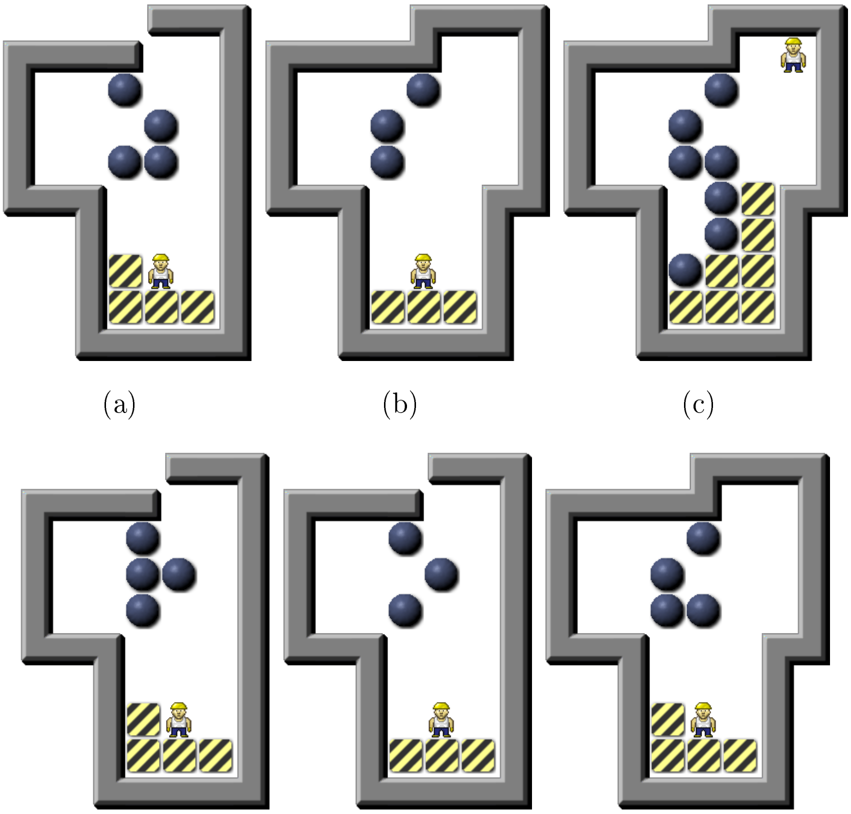
While static analysis is based on the features of the maze created by the walls alone, we can also analyze features created by the stones in the maze. Such analysis is by definition dynamic, since it depends on the positions of the stones in the maze. Analyzing the features of the current game state can reveal crucial information for guiding the search.

### 4.4.1 Zones, Barriers and Corrals

In section 4.1 we already discussed *accessibility* – the area of the maze where the player can move without pushing any stones. This thought can be generalized into zones. A **zone** is an area of the maze floor bounded by stones. The area surrounding the player is the accessible zone while all other zones are by definition inaccessible. A group of stones separating a zone from another is called a **barrier**. Each stone push reshapes one or more zones and may merge or split them. A possible subgoal in some kind of planning-based Sokoban solver could be to join more zones into the accessible zone. This is often a Sokoban player's aim in the beginning stages of a puzzle, though often it is also necessary to push stones in ways that break off zones from the accessible zone.

The YASS solver contains a technique that hasn't yet been documented in scientific literature [Dam10]. They call a zone that the player cannot access a **corral**. If all the stones on the barrier can only be pushed into the zone, the corral is an *I-corral*. Furthermore, if the player can reach all the stones on the barrier and perform the legal pushes inwards the corral is a *PI-corral*.

Now, the key insight with PI-corrals is that if one exists, the player will have to deal with it eventually. This is due to the fact that as the stones on its barrier can only be pushed inwards, one of them will indeed have to be pushed there before any of



(a) (b) (c)

(d) (e) (f)

**Figure 15:** Examples of corral situations. Figures (a) and (b) have a PI-corral to the left of the stones, while figures (d) to (f) do not. Neither of the corrals in figure (c) is alone a PI-corral, but together they do form a combined PI-corral.

the stones in the corral can be pushed elsewhere. So, if the corral has to be dealt with eventually, it is best to deal with it right away and all other moves can be eliminated from consideration. This is called *PI-corral pruning*.

The reason that this works, and that it only applies to PI-corrals and not other corral types is due to *stone influence*. In a PI-corral no other stone can be influenced by the fact that a stone is pushed into the corral, because any stones outside the corral are by definition either accessible or inaccessible regardless of whether the player can walk in the area of the corral, while stones completely inside the corral (not on the boundary) are by definition inaccessible to the player and could not be pushed before the corral is dealt with.

YASS also contains an algorithm for detecting *combined corrals*. When two corrals are separated from each other by stones that are not directly accessible to the player, the stones on their mutual barrier can be considered interior stones and the corrals can be combined. This enables many corrals which would otherwise not be considered PI-corrals to be considered as such, while still preserving the key insight about PI-corrals: that the player has to do something about it eventually, so it's best to resolve the situation right away. Figure 15(c) depicts such a situation. While it is quite rare that a game state contains a PI-corral, combined corrals that together form a PI-corral are quite common. Detecting them should therefore allow PI-corral pruning to achieve impressive savings in the size of the search space. Section 5.3 presents an experiment to determine their usefulness in practice.

### 4.4.2 Doors and One-way Passages

If a stone is positioned in or next to a tunnel in such a way that there is no space to move it away before its surroundings are cleared of stones, it forms a **door**. Doors can be used to form **one-way passages** in puzzles, which restrict the movement of the player to certain areas. Puzzle #29 of the XSokoban set, shown in figure 16 is a good example of a maze which contains multiple doors[5], while figure 17 displays examples of one-way passages.

While no-one has presented a Sokoban solver that uses knowledge of features such as doors to its advantage, Schaul explores the possibility of evolving solver agents that learn such features or *concepts* [Sch05] (see section 4.7 for details). While the reason that doors and other features are rarely used in Solvers might be due to the fact that such features are relatively rare, possibly hard to detect and might just not be useful, it might still be interesting to explore the possibility.

### 4.4.3 Deadlock Detection

Avoiding deadlocks is crucial when trying to solve a Sokoban puzzle. The easiest way to produce a deadlock – and therefore the first one to be avoided – is to push a stone onto a *dead square* (see section 4.3.1). In addition to detecting and avoiding these, other deadlocks produced by the interactions of the stones need to be detected.

---

[5] Incidentally, it is also a great example of a puzzle where the order in which the goal squares should be filled is crucial and extremely hard to determine – and almost certainly requires multiple attempts from a human player.

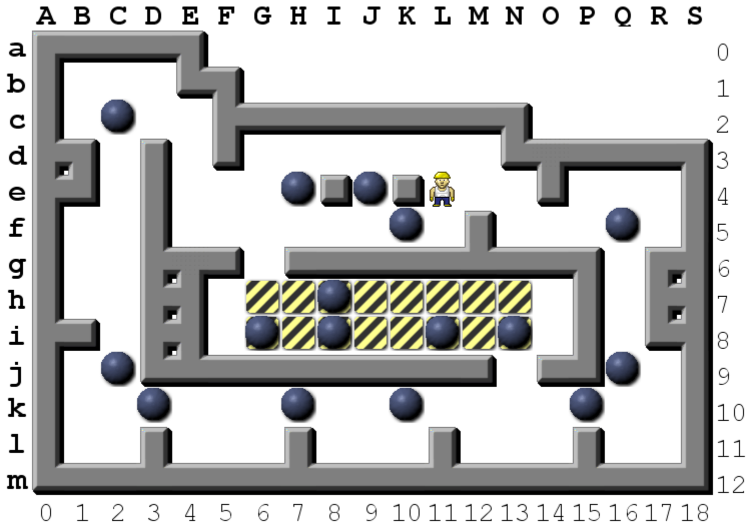


**Figure 16:** Puzzle #29 of the XSokoban set. The stones on *Cc*, *Qf* and *Qj* each form a door, as do all the stones on the *k* row. Also, both of the passages starting from *Cc* and *Qf* and ending at *Nk* are one-way passages for most of the time (until their stones are finally pushed away). Note that contrary to initial impression the stone on *Cj* does not form a door, since it can easily be pushed out of the way to *Cg*.

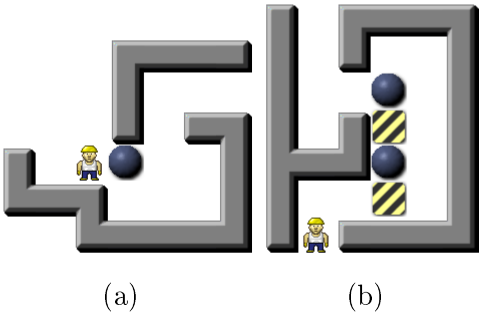

(a) (b)

**Figure 17:** Two examples of one-way tunnels. In figure (b) the player can only pass through once; after that the stones cannot be placed in a way that would allow the player to pass through again.

As mentioned earlier in section 2, in some cases detecting if a position contains a deadlock can mean having to actually solve a puzzle. This does not mean that deadlocks cannot be detected – some common types of deadlocks (such as the 4-stone cluster and the two stones on the west wall in figure 3 on page 4) can be detected with a few trivial lookups, while others take some more computation. This section describes some ways to detect deadlocks.

An obvious way is to hand-code a number of tests for common deadlock positions. However, as Junghanns *et al.* discovered this quickly proves unwieldy and still misses many deadlock positions [JS97]. They instead implemented *deadlock tables* – precomputed tables of all possible stone, goal, wall and player positions in a certain area, with a simple search performed to determine if the area contains a deadlock or not.

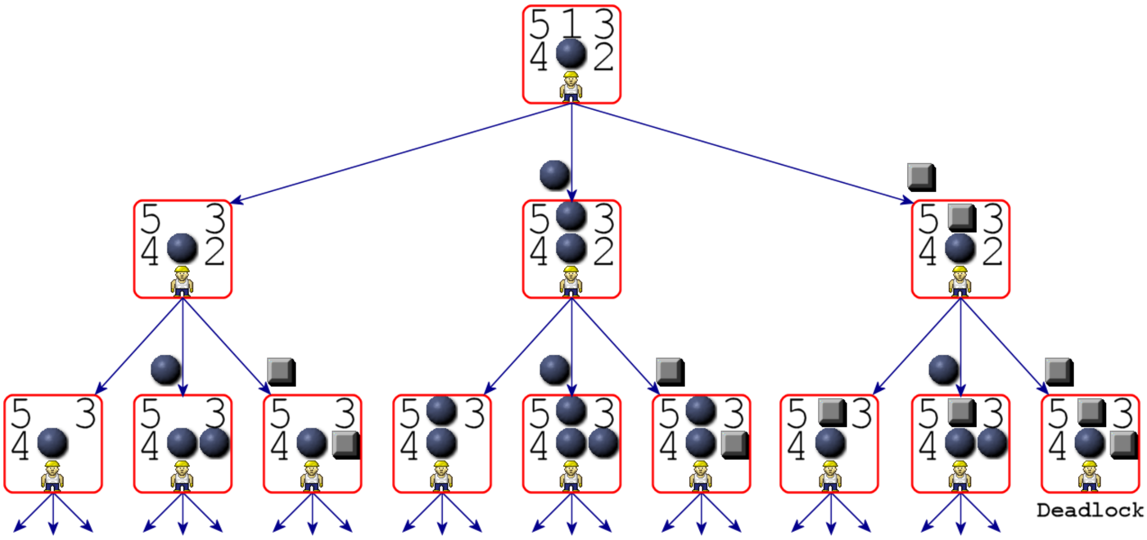


**Figure 18:** Constructing a deadlock pattern database with a 3 × 3-sized pattern [JS01]. Note the deadlock at the lower right corner.

The deadlock tables in Rolling Stone were constructed using an offline search for each possible pattern of stones, walls and empty squares in a 5 × 4 submaze. The search is started with the simplest possible scenario, consisting of only the player and one stone. Since these patterns are designed to be used directly after each push, the first stone is always directly in front of the player (the actual side doesn't matter – the patterns are oriented along the push). Then, three successors for that simple state are generated: one for a stone added behind the first stone, one for a wall and one for an empty space. Next come the successors for those, with the next element placed at the first stone's side. This process, illustrated in figure 18, is continued with a specific order of new squares until all the possible patterns are generated. For each pattern, a search is performed to see whether all the stones in the pattern can be pushed out of the pattern. Various enhancements are used, such as detecting stones on dead squares as immediate deadlocks.

After the deadlock database is computed it is stored and can then be used in all subsequent searches. When a move is considered, the 5 × 4 frame is oriented along the push and overlaid on the maze, as shown in figure 19. Also mirrored and rotated positions of the frame are considered. The deadlock database is then queried for the pattern of walls, stones and floor squares under the frame. If the state is discovered to be a deadlock, the move is pruned from consideration.

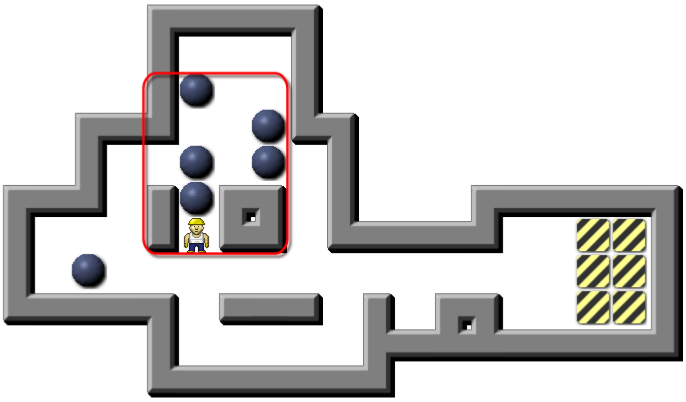


**Figure 19:** An example of applying the deadlock patterns frame in one possible orientation[Jun99]

While Rolling Stone uses precomputed, level-independent deadlock tables, another option is to compute *level-specific* deadlock patterns, as explored by Cazenave *et al.* [CJ10]. Their approach is to use *retrograde analysis* to compute deadlocks. First, all trivial one- and two-stone deadlock patterns are generated – stones in corners, stones on walls between two corners and two adjacent stones on a wall. Then, all possible three-stone configurations for a specific puzzle are generated. In a first pass, for each configuration the algorithm checks for already known deadlocks and then tries all available moves and checks if they all lead to a known deadlock. If this is the case, the configuration is added to the known deadlocks. In subsequent passes the algorithm generates all possible previous states for all known deadlock states. For each generated state all available moves are again checked. When all three-stone configurations have been processed the algorithm can move on to four-stone configurations etc.

## 4.5 Multi-Agent Search and the Van Lishout Subclass

As a game, Sokoban is undeniably a single-player game. However, while the articles on Rolling Stone ([JS97, JS98a, JS98b, JS98c, JS98d, JS99, Jun99, JS01]) exclusively discuss *single-agent search methods*, the game does not necessarily have to be treated as a *single-agent search problem.* If the stones are chosen as the active agents and the player is just a tool to be used by them, then the problem becomes a *multi-agent search problem.* Van Lishout *et al.* have studied Sokoban as such [Lis06]. They allow each stone to consider its own available moves and to call the player character to position himself accordingly. Thus, each stone can be seen as a semi-independent agent in a group of agents working together to find a solution.

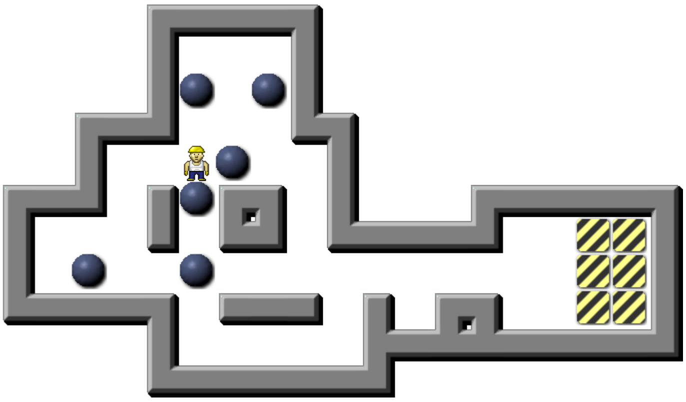
In their study of Sokoban as a multi-agent search problem, Van Lishout *et al.* discovered a subclass of Sokoban puzzles which is almost trivial to solve [Lis06]. A puzzle can be solved *stone-by-stone*, i.e. moving one stone at a time to the goal squares without moving any other stones in between, if it satisfies the following conditions:

1. *Goal-ordering-criterium* – it must be possible to determine the order (or *an* order, as there are usually many possible orders for a given puzzle) in which the goal squares should be filled, regardless of the positions of the stones and the player.
2. *Solvable-stone-existence* – it must be possible to push at least one stone to the first unoccupied goal square without having to move any other stone
3. *Recursive-condition* – for each stone that satisfies the previous condition, the maze obtained by moving that stone to the corresponding goal must also contain at least one stone that satisfies the previous condition

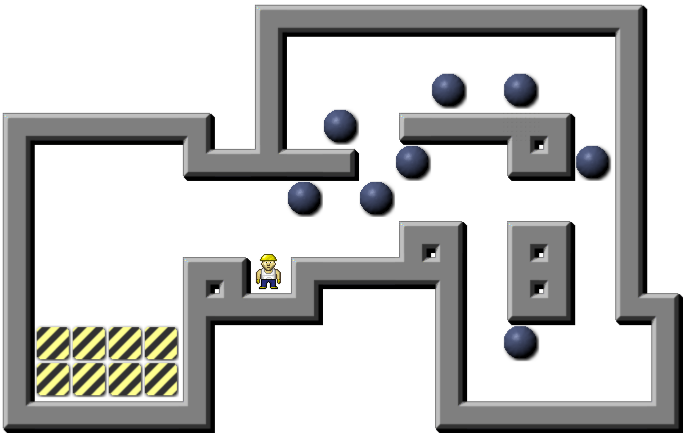
In practice, it is quite rare for a puzzle to be in the Van Lishout subclass in its starting arrangement. In the XSokoban puzzle set only two[6] of the puzzles (puzzles #53 and #78) are in the subclass, while the Microban1 set has four (puzzles #44, #126, #154 and #155 – three of these have only one stone, while puzzle #126 has 7). However, having puzzles be in the subclass after relatively few moves is surprisingly

---

[6] Van Lishout mentions only one, #78, but with a slightly better goal packing order algorithm another, #53, can be found. See sections 6.3 and 7.4 for details.



(a) XSokoban #1 after two pushes



(b) XSokoban #78

**Figure 20:** Two puzzles that are solvable stone-by-stone. Note that the state of puzzle #1 is after three pushes, not the initial state (as shown in figure 1).

common. One of the strengths of human players of Sokoban, when compared to computational approaches, is the ability to recognize early in the solution process that they can reach a state where the maze is solvable stone-by-stone [Lis06]. This can be mimicked by running a search algorithm such as Breadth-First Search for the solution and examining if each encountered node contains a stone that can be pushed to the goal or not. When such a node is discovered, it can be further examined to see if the recursive condition also applies. If not, the search is continued.

It has been noted [Lis06] that while the move ordering scheme of Rolling Stone generates solutions similar to the multi-agent modeling technique of Talking Stones, it wastes quite a lot of processing time doing so. For each node generated by the IDA* search in Rolling Stone a lower bound is computed, all the possible child nodes and their lower bounds are also computed, deadlock patterns are matched etc. and only then the search proceeds to the most promising child, which is usually the one leading the last pushed stone towards the chosen goal. Using the Van Lishout multi-agent modeling algorithm most of the alternative moves will never even have been considered.

One aspect not discussed by Van Lishout *et al.* is that their method is even more closely matched by the goal cut technique of Rolling Stone. Indeed, if a stone is pushable to the next goal in the sequence, then Rolling Stone will try that first and its search will proceed much in the same way as the Talking Stones method. However, if the recursive condition does not apply all the way, i.e. if state is not in fact solvable stone-by-stone but only some stones are pushable to their respective goals, then the algorithm used in Rolling Stone will use the goal cuts as far as it can and continue the search from that state, while the one used in Talking Stones will return to continue the search from the original state from where it first tried to apply the stone-by-stone method. This can be a good or a bad thing, depending on the puzzle.

One can also argue that discussing Sokoban as a multi-agent problem brings nothing new to the table. While considering the game as a sequence of player moves with a maximum branching factor of 4 (north, east, south and west) is clearly a single-agent search problem, one can argue that the mere fact of optimizing for stone pushes instead of player moves and considering the search in terms of stone pushes (with each accessible stone having up to 4 pushable directions) already brings the discussion to the realm of multi-agent search, whether stated explicitly or not. Or rather, if one takes the position that multi-agent operation implies the capability of parallel actions, then Van Lishout's method becomes single-agent as well.

## 4.6 Abstraction and Planning

While the search effort of Rolling Stone operates on the level of individual stone pushes (except in the case of macro moves), the Power Plan solver by Botea *et al.* raises the abstraction level and introduces a planning approach. They present two possible abstraction levels: *tunnel Sokoban* and *Abstract Sokoban*. The first one, tunnel Sokoban, is a partial abstraction where the solver still operates mainly on the level of individual pushes, but where the tunnels (as described in section 4.3.2) present on the level are detected and collapsed into abstract representations with just a few possible states, much like the tunnel macros of Rolling Stone. The other one, Abstract Sokoban, takes the abstraction further and also treats the rooms present in the puzzle as individual entities. The search effort then operates on two levels: on the global graph, which consists of transitions from room to tunnel and tunnel to room, and on individual rooms where the processing operates on individual pushes.

The algorithm used by Power Plan decomposes the Sokoban maze into a graph of rooms and tunnels, where the graph nodes are rooms and the edges are tunnels. This is then used to divide the problem into many local problems (the rooms) as well as the global problem (the whole maze). For each room a *local move graph* is computed. First, the empty state of the room is marked as legal, then all 1-stone configurations are processed, followed by all 2-stone configurations etc. When all the $n$-stone configurations have been marked either as legal or deadlocked, all $(n+1)$-stone configurations are processed. If a path can be found from the current $(n+1)$-stone configuration to a previously known legal combination, the current combination is also marked as legal. Otherwise it is marked as deadlocked. After all the combinations are processed, the graph is analyzed and all strongly connected components are combined into abstract states. All deadlocked combinations become one abstract deadlocked state. For each of the abstract states the values for all predicates (e.g. "can push one more stone inside the room through entrance X") are computed, as well as the resulting states if the corresponding actions would be taken.

For each tunnel, between 1 and 3 abstract states are recognized, depending on the type of the tunnel. A zero-length tunnel cannot have a stone parked inside it and can therefore have only one abstract state: empty. A straight tunnel can either be empty or contain a stone. The same goes for the 4-ended tunnel shown in figure 13. A tunnel having a corner in it can be empty, have a stone parked in its north/west end or have a stone parked in its south/east end[7] (having both would be a deadlock). In both cases the stone can only exit through the entrance it was pushed in from.

The global problem of the whole maze is simplified by mapping it into a graph of the

---

[7] The original article uses the terms *left end* and *right end*, which leaves the obvious question of tunnels that have both ends at the same X coordinate.

local problems, i.e. the rooms, connected by tunnels. The global problem is solved by *planning*, with actions referring to moving a stone from one room or tunnel to another. When an action is taken, the rooms and tunnels involved change their abstract states. To be able to complete actions, stones in the rooms involved in the action may have to be rearranged; this is done by using the local move graphs. To minimize the risk of organizing the stones in such a way that the way to the solution is blocked, the local changes are chosen to minimize the number of local changes and to maximize the number of open entrances.

## 4.7 Evolved Agents

All the solver algorithms and techniques so far have been programmed by hand. A completely different approach is attempted by Schaul [Sch05], who applies an evolutionary system similar to Genetic Programming to Sokoban and thus *evolves* solver *agents*, who participate in a virtual economy by bidding on moves they want to perform. The idea is to evolve agents who recognize and learn to handle different *concepts* and then bid on the moves that are their own specialty. Examples of such concepts include the doors and one-way passages described in section 4.4.2.

The solvers are evolved by training them on simple training puzzles which illustrate a single concept (much like the puzzles of the Microban1 set). Initially, a population of randomly generated agents is created. Each agent consists of a *program tree* which determines its actions. All the agents start out with an initial amount of money. For each game state, a virtual *auction* is held where the population of agents is asked to bid for a move (or a combination of moves) available from the current state. The highest bidder wins the right to perform its move. When the level is eventually solved, the agents that participated in the solution are rewarded by the system. That way the agents can eventually earn more money. This leads irrational agents to go bankrupt (and new, randomly generated agents to take their place) and rational, co-operational agents to stay in the economy.

While the idea of evolving solvers is promising, it has its problems. They include choosing the set of instructions from which the agents are formed and determining the rewards and penalties for good and bad moves respectively. While the system is able to learn to solve simple and medium levels quite quickly, it is only able to solve one puzzle (#1) of the XSokoban puzzle set. This shows that much more work is still required before a hard problem like Sokoban becomes easy to solve by automatic programming.

## 4.8 Other Approaches

In addition to the techniques presented earlier in this section, Rolling Stone also uses further enhancements such as pattern searches, relevance cuts, overestimation and rapid random restarts. The details on these can be found in the numerous publications by Junghanns *et al.* (e.g. [JS01]).

In addition to the enhancements actually used in Rolling Stone, Junghanns also discusses a number of failed ideas [Jun99] that seemed good on paper, but were discarded for some reason. Some of them did decrease the size of the search space but were too costly, while others were only useful on a small number of puzzles and might have harmed performance on others.

# 5 Experiments

As we have seen in the previous section, there is a plethora of possible enhancements available for Sokoban solver algorithms. However, judging their merits and demerits on description alone is difficult and leaves the programmer at a loss for which ones to choose. In this section we aim to ease that task by providing experimental data for the effects of some of the enhancements discussed in the previous section.

## 5.1 Breadth-First vs. Depth-First

While being quite successful, one large problem with the design of the Rolling Stone solver is that a large amount of its running time is spent on maintaining the lower bound [JS01]. As Rolling Stone uses IDA* as its base algorithm, at each node it needs to evaluate the remaining length of the path to the solution. This in fact dominates the algorithm's running time. Therefore, it is an obvious place to begin investigating for improvements, and the best way to minimize the time consumption of an operation is to take it out completely. In this section, we evaluate the performance of two algorithms that don't require lower bound estimation at each node: Breadth-First Search and Depth-First Search.

As noted in earlier sections, using DFS without a depth limit means possibly running off to an infinite branch of the search tree, while determining an upper bound for the solution length in Sokoban is nearly impossible. We therefore chose Iterative Deepening Depth-First Search (IDDFS) as our DFS variant. Note that while it does require a lower-bound estimate at start (to avoid starting at depth limit 1 and effectively degenerating into a badly implemented Breadth-First Search), it does not require one being calculated at each node. For IDDFS, we use the Minimum-Matching Lower Bound described in section 4.2.2.

Before even running the test the first time, we present a hypothesis: as the lower-bound estimate for most puzzles will be lower than the actual solution depth, the IDDFS solver will be disadvantaged by having to regenerate the lower search levels multiple times. Therefore, an enchancement is added to the IDDFS solver: it is allowed to maintain as many nodes as it can in memory and store the over-the-depth-limit nodes in a *postponed* list, from where they will then be moved into the search frontier when the depth limit is increased. Naturally, the comparison will be performed both with and without this enhancement, indicated by PP in the results in section 7.

Another factor affecting the performance of the IDDFS solver is the move ordering. While the BFS solver will search all the direct successors of a node before moving on, the IDDFS solver's performance will vary greatly depending on which child node the solver expands first. Therefore another enhancement will be added to the IDDFS solver: move ordering by inertia. As discussed in section 4.2.3, Sokoban solutions often contain long runs of pushes to the same stone. Therefore the IDDFS solver will first expand all the child nodes that target the stone that was pushed last before moving on to other child nodes. As with the previous enhancement, the comparison will also be performed with and without the move ordering enhancement, shown as IMO in the results.

The performance of the algorithms is measured by memory consumption (how many nodes are explored and therefore stored in memory), processing speed (nodes per second) and solution time. As the solution time will vary from computer to computer

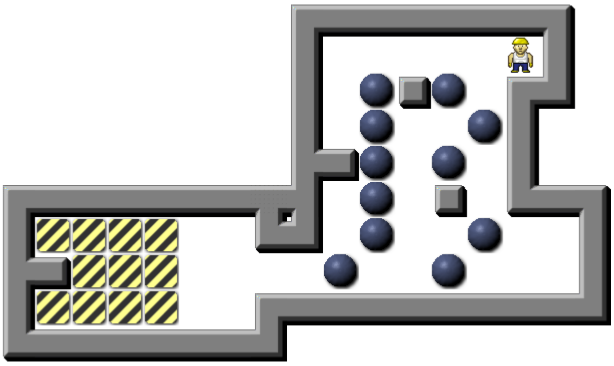


**Figure 21:** Puzzle #3 of the XSokoban set

it should be only regarded as a comparison measure. The test will be run on all of the levels of the Microban1 puzzle set and, to evaluate performance on slightly more complex levels, puzzles #1, #3 and #78 (figures 1, 21 and 20(b), respectively) of the XSokoban set (since these seem simple enough that they might be solved by such naive algorithms[8]). All of the tests will have a 300-second time limit to keep the total running time of the whole test set in a reasonable time frame.

## 5.2 Forward, Reverse and Bidirectional Solving

While Junghanns mentions in the Failed Ideas section of his thesis that backward search and bidirectional search have been tried and found to be challenging [Jun99], and Takes has briefly explored backwards search in his bachelor's thesis [Tak08], no-one has published good comparative results for forwards, backwards and bidirectional search in Sokoban. This section aims to do precisely that.

To determine if reverse and bidirectional solving actually do decrease the search space size, and by how much, they are used to solve the same puzzle set as in the previous experiment (all of Microban1 and puzzles #1, #3 and #78 of XSokoban), recording the same measurements: the number of nodes searched, processing speed and solution time. Both Breadth-First Search and Depth-First Search are be used, as well as combinations of the two[9]. For the IDDFS-based solvers the enhancements that provide the best performance in the previous experiment are used.

---

[8] An assumption that proved to be false.

[9] Bidirectional Depth-First Search was not used, as the nature of DFS makes it improbable for the two search frontiers to meet in any useful way.

## 5.3 PI-corral Pruning

The PI-corral pruning technique presented in section 4.4.1 promises to offer large savings in the size of the search space. After all, if a PI-corral exist on the board, it is always a potential deadlock and, as it by definition cannot affect any stones outside of it, should be dealt with immediately. This excludes all the other stones from consideration, which can possibly have a large pruning effect on the size of the search tree.

In this section we aim to determine just how much of an effect the PI-corral pruning technique has. To do this, we apply it to both Breadth-First Search and Depth-First Search (enhanced with the best-performing enhancements from the first experiment) on the same puzzle set as in the previous experiments. As before, the effects are evaluated on the number of nodes, processing speed and solution time. Again, the time limit is set at 300 seconds. The PI-corral pruning enhancement is shown as PI in the results.

## 5.4 Van Lishout Solving Method

As described in section 4.5, a certain subclass of Sokoban puzzles is almost trivial to solve. Only three things are needed: a predetermined order for goal square packing, a stone that can be pushed to the first goal in that order and, from the resulting state, another stone that can be pushed to the next one. While this is not the initial state of most puzzles, in many cases such a state can be found at a relatively shallow depth in the search tree.

While Van Lishout *et al.* already reported results for their algorithm, claiming to solve 9 problems of the XSokoban puzzle set (#1, #2, #3, #5, #6, #51, #54, #78 and #82), they use an extremely simple method for determining the goal packing order. Because our implementation (described in section 6.3) can generate the goal packing algorithm for more puzzles, we can already detect one more puzzle (#53) that is directly solvable.

To be able to analyze the performance of the Van Lishout stone-by-stone solving enhancement (indicated by VL in the results) on hard problems and to be able to compare results with the original study, the problem set for this fourth experiment is different. Rather than test on the Microban1 set as in the previous experiments, most puzzles of which are easily solvable by trivial algorithms, we are more interested in the performance on hard problems. Therefore the test set for this experiment is the 90-puzzle XSokoban set. We enhance our BFS and IDDFS solvers with the Van Lishout Subclass Detection algorithm, which tries to solve each explored state stone-by-stone. The objective is to determine how many levels can be solved under the usual 300-second time limit, and how much does the VL enhancement slow down the processing.

# 6 Implementation Details

Besides the algorithms themselves, the performance of algorithms depend heavily on the details of their implementation. The search algorithms discussed in section 3 as well as the enhancements presented in section 4 can all be implemented in multiple ways which all have an influence on their performance. In this section we discuss some of the details of our implementation.

The implementation used for all the experiments in this thesis was written in Java. The code is relatively unoptimized with focus on rapid prototyping and development, clarity and ease of modification. Thus, the performance may be severely lacking when compared to solvers that have been in development for several months or years and are usually heavily streamlined and optimized. Nevertheless, the code is fast and robust enough to provide reliable results for comparing the relative performance of various algorithms and enhancements.

## 6.1 BFS and IDDFS Implementations

The implementations for the Breadth-First Search and Iterative Deepening Depth-First Search were written in an obvious way, translating quite directly from the pseudocode descriptions of the algorithms. The BFS implementation uses a linked list for the search frontier and a hash set for the explored set. The IDDFS does not explicitly maintain a data structure for the frontier but instead uses recursive function calls to expand nodes and thus maintains the search frontier in the call stack.

For the reverse and bidirectional searches, various methods were used. The reverse and bidirectional BFS variants (RBFS and BBFS) were implemented as separate algorithms with the BBFS alternating *ticks* in the forward and reverse direction internally, a tick meaning the expansion of one node. The reverse IDDFS (RIDDFS) was similarly implemented as a separate algorithm. For the bidirectional algorithm combinations, IDDFS/RBFS and BFS/RIDDFS, the solvers are given references to one another. The IDDFS or RIDDFS algorithm always asks the accompanying opposite-direction algorithm (RBFS or BFS, respectively) to run one tick of its search and then runs its own. This is done at the beginning of each node expansion, so the end effect is the same as with the BBFS implementation: both search fronts alternate advancing one node at a time. If the IDDFS or RIDDFS algorithm is used in a bidirectional setting, it only allows its search to proceed to the estimated solution depth minus the current opposite-direction search depth.

As the reverse search may start from many different states with a different player position, all the reverse algorithms try such states one after the other if one leads to a dead end. States are considered to be already tried if the player accessible area is identical, disregarding the actual exact player position.

## 6.2 Simple Deadlock Detection

When the move generator generates the moves to successor states from a given state, each of the moves is checked for simple deadlocks. All the deadlock situations shown in figure 22 (as well as other variations of these patterns) are recognized in all orientations. This is done with simple if-structures, actually checking most positions with an `isEmpty?()` check to detect either a wall or a stone (in positions where it does not matter which one it is), and all the necessary checks are implemented as $O(1)$ operations, so the routine is quite fast.

In addition to these patterns, the move generator also recognizes dead squares (computed when the puzzle is loaded) and never even considers moves that would lead a stone onto one of them. While these checks provide far less coverage than the deadlock pattern database of Rolling Stone, they still prune the search space enough for the purposes of these experiments.

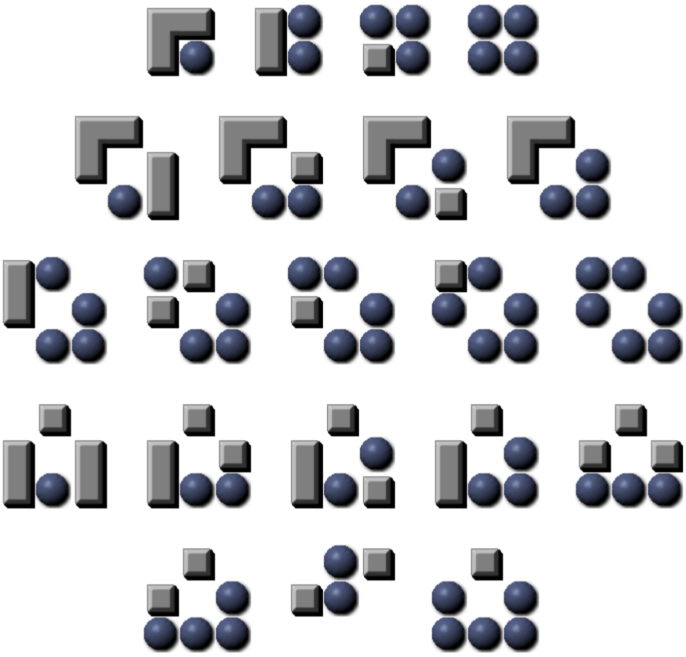


**Figure 22:** Deadlock patterns recognized by the move generator

## 6.3 Goal Packing Order Algorithm

As mentioned in section 4.5, the goal packing order used by Talking Stones, the solver by Van Lishout *et al.*, is extremely simple [Lis06]. They order the goal squares by the number of the walls surrounding the square. The first squares are those which are surrounded on three sides. Next, these are replaced with walls and the surrounding walls are recalculated for all goal squares. When no more goal squares can be found which are surrounded on three sides, then the search fills the squares which are surrounded on two adjacent sides and the search continues.

In our implementation we use this same algorithm as a preprocessing step. After that we identify entrances to the goal area: floor squares that are directly outside the *goal room* (see section 4.3.2). If multiple entrances are found, the current implementation uses the first one found. A better implementation could try to assign entrances to goals using some heuristic. Then we run a search which tries each goal in order (using the ordering made in the preordering step) and checks whether a stone placed on the entrance can be pushed to that goal. If it can, that goal is replaced by a wall and the next goal is tried. If a stone cannot be pushed to any goal the search backtracks.

This method allows us to find a goal order for 48 of the 90 levels in the XSokoban set under a five second time limit, most actually in just a few tenths of a second. Unfortunately, many of these are erroneous in practice, mainly because the levels often require parking stones inside the goal area before moving them to their final locations or pushing stones into the goal area from a different entrance than the one chosen by our simplistic design. Nevertheless, with this algorithm we were able to considerably improve the results achieved by Van Lishout *et al.*

### 6.4 Inertia Move Ordering

While the whole move ordering scheme used in Rolling Stones is called inertia [JS01], the actual inertia scheme is only a part of their whole scheme. In this context, inertia means that the move ordering prefers moves that target the stone that was pushed last. The rest of their move ordering uses their minimum-matching lower bound algorithm to determine which moves decrease the estimated lower bound and arranges those after the inertia moves, sorted by the distance of the chosen stone to its targeted goal, closest first.

We use a simpler scheme, in which only the inertia moves are preferred. The rest of the moves are tried in the order in which they are generated, bypassing the expensive lower bound and stone distance calculations.

## 7 Results and Discussion

The four experiments described in section 5 were executed on a typical home computer with an Intel Core 2 CPU running at 2.13GHz, with 2 gigabytes of memory. In this section, we present and discuss the results of those experiments. The complete result tables of the experiments are included in Appendix 1.

### 7.1 Breadth-First vs. Depth-First

In the first experiment, we evaluated the relative performance of Breadth-First and Iterative-Deepening Depth-First Search. The solvers were tested on all the puzzles of the Microban1 set and puzzles #1, #3 and #78 of the XSokoban set, with

| | BFS | IDDFS | IDDFS+PP | IDDFS+IMO | IDDFS+PP+IMO |
|---|---|---|---|---|---|
| **Solved** *(# of puzzles)* | 151 | 136 | 147 | 136 | 147 |
| **Solved optimally** *(# of puzzles)* | 151 | 136 | 147 | 136 | 147 |
| **Failed** *(# of puzzles)* | 7 | 22 | 11 | 22 | 11 |
| **Processing speed mean** *(1000 nodes/second)* | 31.6 | 25.4 | 18.5 | 17.9 | 17.2 |
| **Processing speed std.dev.** *(1000 nodes/second)* | 19.6 | 15.0 | 8.8 | 9.6 | 8.0 |
| **Processing speed median** *(1000 nodes/second)* | 27.6 | 23.1 | 17.3 | 15.9 | 16.0 |
| **Less nodes than BFS** *(# of puzzles)* | – | 13 | 13 | 12 | 12 |
| **Less nodes than IDDFS** *(# of puzzles)* | 144 | – | 127 | 110 | 145 |
| **Less nodes than IDDFS+PP+IMO** *(# of puzzles)* | 132 | 7 | 40 | 8 | – |
| **Faster than BFS** *(# of puzzles)* | – | 20 | 9 | 19 | 10 |
| **Faster than IDDFS** *(# of puzzles)* | 120 | – | 107 | 16 | 108 |
| **Faster than IDDFS+PP+IMO** *(# of puzzles)* | 130 | 24 | 73 | 16 | – |

**Table 1:** Summary of the results of Experiment #1

IDDFS being tested both with and without the Inertia Move Ordering (IMO) and Postponing (PP) enhancements. With these settings the solvers always solve puzzles optimally if they are able to solve them before the time limit. However, the ability to do so, i.e. the processing speed and the order of nodes searched, does vary.

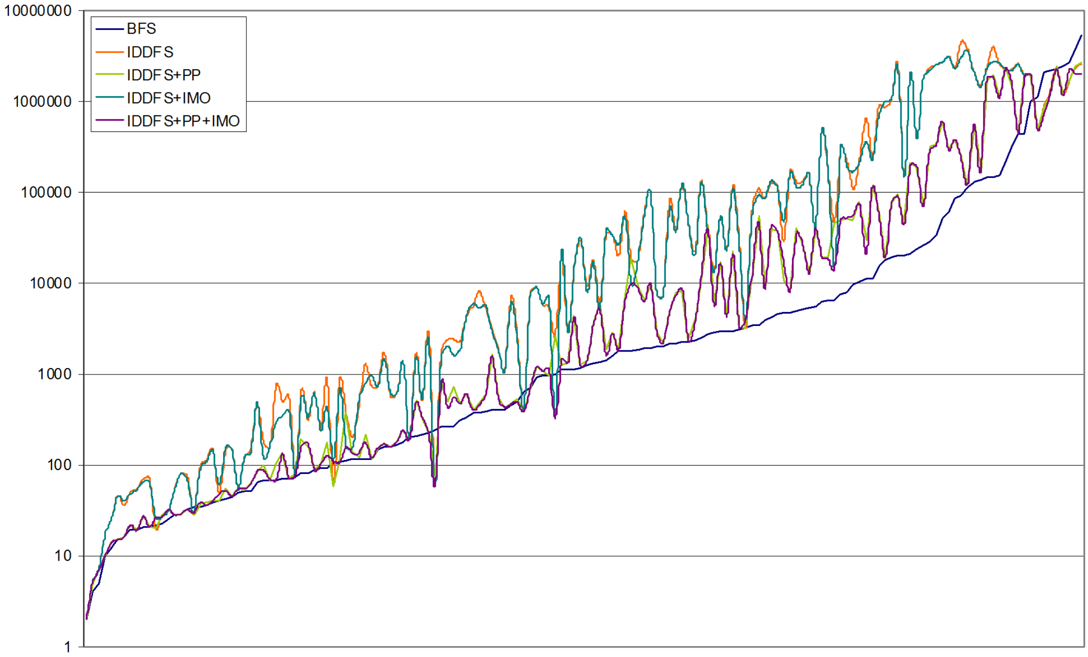


**Figure 23:** Explored nodes for Experiment #1, sorted by the performance of BFS. The Y-axis shows the number of nodes explored. The X-axis indicates puzzles in the test set.

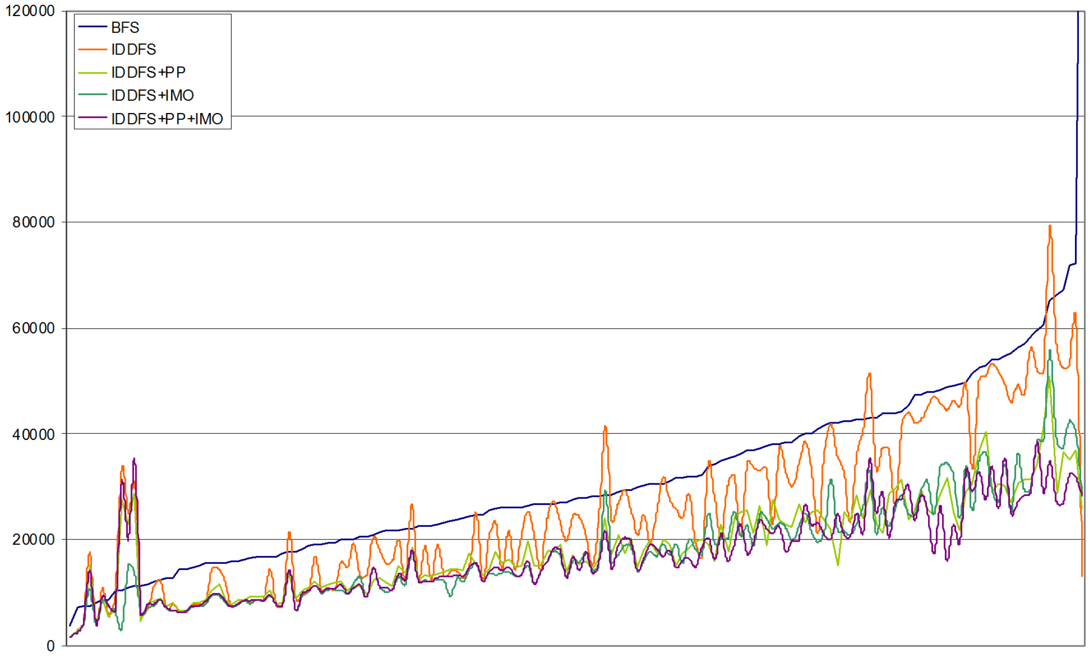


**Figure 24:** Processing speeds for Experiment #1, sorted by BFS. The Y-axis shows the number of nodes processed per second.

The amounts of nodes explored by each of the solvers are plotted[10] in figure 23, the processing speeds in figure 24 and the processing times in figure 25. The most successful of these solvers was BFS, which was able to solve 151 of the 158 tried puzzles. IDDFS with the Postponing enhancement solved 147 puzzles, as did IDDFS with both enhancements, while the IDDFS version with only the Inertia Move Ordering enhancement or without either enhancements only managed 136 puzzles.

An interesting pick in these results is the puzzle M36 (i.e. #36 of the Microban1 set), shown in figure 26. It is quite a simple puzzle with only 5 stones, but it tends to trap naive depth-first solvers. While the puzzle is only a simple room, it actually involves pushing most of the stones through the goal area to the parking area on the bottom right before being able to place the leftmost goal stone and thus the others.

---

[10] All of the charts for the results in this section have the puzzles of the test set as their X-axis. For clarity and ease of comparison the results are sorted by the performance of the BFS solver, with the puzzle having the lowest value (explored nodes, speed or time, depending on the chart) on the left. As the purpose is to compare the relative performance of the solvers, the exact ordering of the X-axis is irrelevant and has therefore been hidden. If needed, the exact data for each individual puzzle is available in the result tables in Appendix 1.

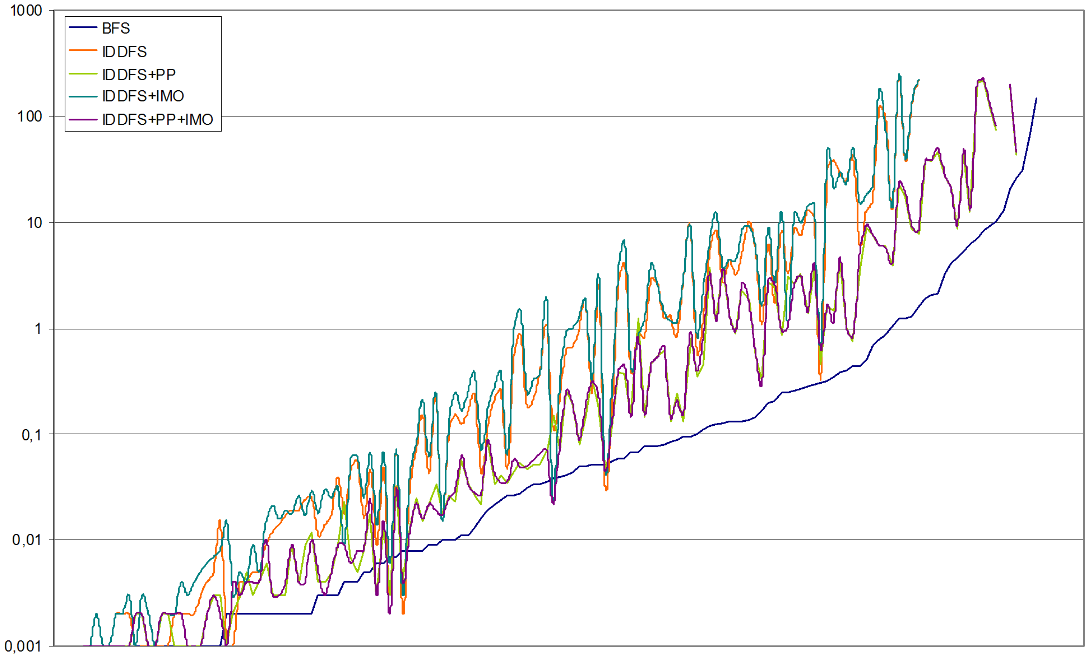


**Figure 25:** Processing times (in seconds) for Experiment #1, sorted by BFS

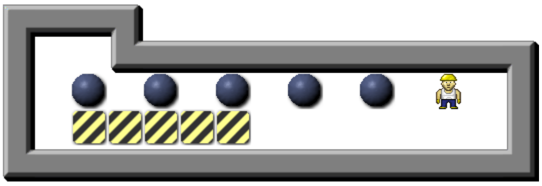


**Figure 26:** Level #36 of the Microban1 set

This situation of having to push stones through the goal is a case in which the Minimum-Matching Lower Bound algorithm fares poorly. Thus, the IDDFS solvers are forced to fully explore the lower depths before determining that the solution is longer than the current lower bound. Indeed, the initial depth limit set by the MMLB algorithm is 35, while the actual solution length is 59 pushes. The IDDFS solver without any enhancements takes 121 seconds to solve this, exploring 2631727 game states – most of them in the lower depths of the search tree examined over and over again. The Postponing enhancement allows it to remember the nodes in the lower depths and only examine the next search depth, which decreases the solution time to 6 seconds and 97372 examined nodes. Inertia Move Ordering still decreases the amount of examined nodes, but the additional move ordering actually causes a slight increase in solution time[11]. Puzzles M98 and M99 is are further examples of a similar situation, where the stones must be pushed through the goal area into a parking area. Only the Postponing enhancement allows the IDDFS solver to solve them under the time limit. In fact, most of the puzzles that are unsolved by some or all of the IDDFS solvers contain at least some stones that must be first pushed through their goals and parked before placing at their final destination.

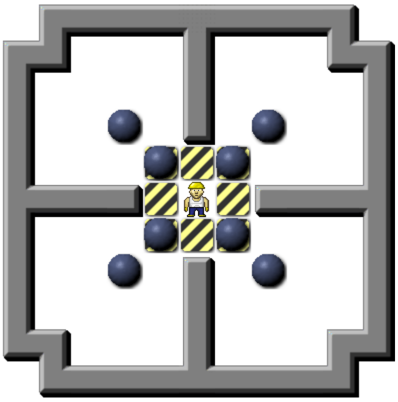


**Figure 27:** Puzzle M93, i.e. #93 of the Microban1 set

Puzzles such as M93 show that a low number of stones – 8 in this case – does not mean that a puzzle is easy. The large amount of moves available at all times (i.e. the branching factor; even the opening state has 8 moves available, and some of the states have up to 20) and the number of potential deadlocks in this puzzle make it extremely hard for automatic solvers. And as a matter of fact, M93 is surprisingly hard for human players too. Some kind of symmetry-detecting algorithm could possibly alleviate the situation considerably. If the solver was able to detect that the puzzle is perfectly symmetrical in both the X and Y axes, it could compare the mirrored and rotated versions of each new state to the transposition table and thus prune the search space considerably. Also, better deadlock detection would surely help here. Levels M144, M145 and M146 suffer from exactly the same symmetry problem and thus are unsolved by all of these solvers.

A different problem affects puzzle M153, shown in figure 28. While in many puzzles there are multiple ways of arriving at the same solution (i.e. different ways to inter-

---

[11] This could actually be avoided by improving the implementation. As we already know that we want the moves targeting the last pushed stone first, the move generator could generate those first and thus the need for an additional sorting step would be removed.

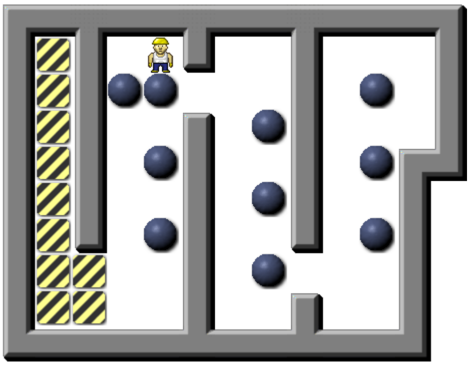


**Figure 28:** Puzzle #153 of the Microban1 set

leave the same pushes or push sequences), in this one the narrow corridors and the relatively large number of stones create a situation where the exact order of pushes is important. An ill-placed stone can result in a state which causes a deadlock to manifest itself much deeper in a search tree. In fact, with this puzzle it takes 90 pushes to reach a state from which the puzzle is easy to solve stone by stone (determined by solving the puzzle by hand). Until this state is reached there is little room for variation within the sequence of pushes and long sequences of pushes to the same stone are rare, making the Inertia Move Ordering scheme actually harmful here (although it would be immensely helpful after this point). To solve such puzzles, deadlock detection is not sufficient. A solver would also needs the ability to *analyze* the deadlock and to backtrack all the way to a state where the preconditions for that deadlock do not exist. Otherwise a depth-first solver will waste precious time and memory in an unfruitful search of a practically deadlocked subtree. While the breadth-first solver does not suffer from this same problem, the number of stones and available moves are again simply too high to solve the puzzle under the time limit.

Overall, these results reveal that in nearly all cases the naive, brute-force Breadth-First Search algorithm will solve puzzles faster and with less memory than a similarly naive Depth-First Search. In many cases, the number of explored nodes with the BFS solver is a full order of magnitude lower than with the IDDFS solver with both enhancements. While this does support the findings of Junghanns *et al.* that domain-dependent search enhancements are needed to solve Sokoban [JS97], it does raise the question that perhaps with better enhancements, a breadth-first approach would be as good or even better than the iterative deepening depth-first approach chosen for Rolling Stone (although Rolling Stone uses an IDA* algorithm as its basis, not the IDDFS used here, the increase in processing speed when leaving out the search-directing heuristic may compensate for the difference). While choosing a breadth-first algorithm as the basis of the search does make it much harder to take advantage of enhancements such as move ordering and macro moves, many pruning-type enhancements are still applicable. Indeed, the other experiments in this section illustrate just that. Furthermore, switching from a depth-first strategy to a breadth-first one might also reveal possibilities for new enhancements which would not be relevant to a depth-first solver.

## 7.2 Forward, Reverse and Bidirectional Solving

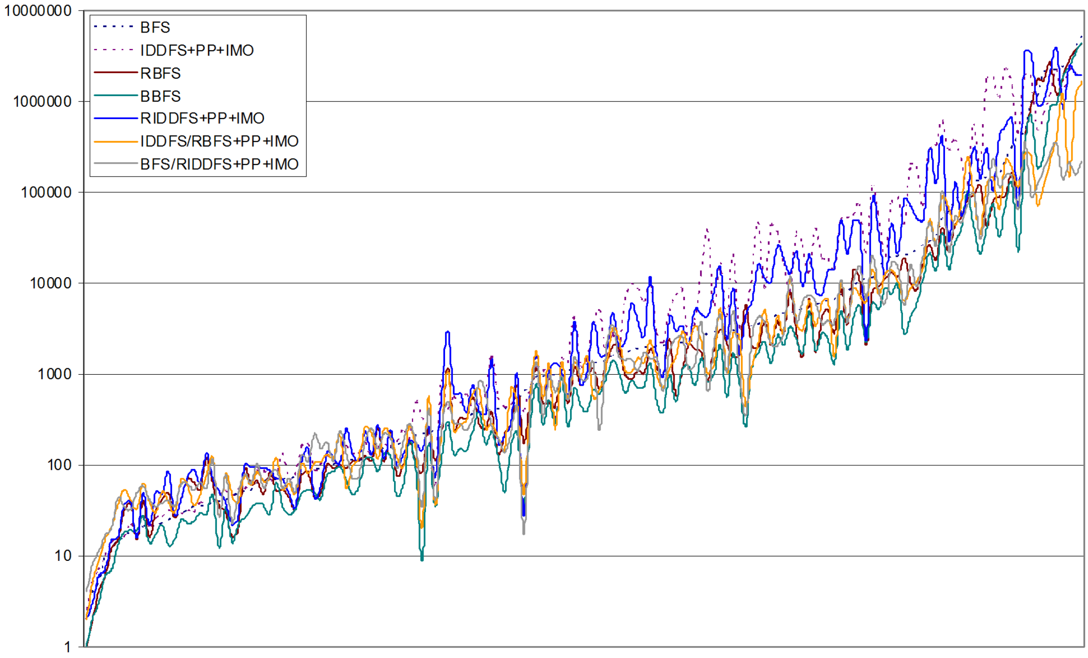


**Figure 29:** Explored nodes for Experiment #2

The results for the second experiment, evaluating reverse and bidirectional solving, reveal a surprise: the solutions found are no longer always optimal. While all puzzles solved by the Reverse and Bidirectional Breadth-First Search solvers (RBFS and BBFS, respectively) solver are always optimal, using Depth-First Search in a bidirectional setting clearly causes the search frontiers to miss each other and thus often produces longer-than-optimal solutions. However, using reverse search does

| | RBFS | BBFS | RIDDFS+PP+IMO | IDDFS/RBFS+PP+IMO | BFS/RIDDFS+PP+IMO |
|---|---|---|---|---|---|
| **Solved** | 151 | 154 | 148 | 148 | 149 |
| **Solved optimally** | 151 | 154 | 145 | 106 | 96 |
| **Failed** | 7 | 4 | 10 | 10 | 9 |
| **Processing speed mean** | 29.0 | 20.6 | 37.1 | 44.6 | 43.7 |
| **Processing speed std.dev.** | 13.9 | 8.8 | 20.6 | 27.3 | 27.4 |
| **Processing speed median** | 28.0 | 19.0 | 34.7 | 42.2 | 43.1 |
| **Less nodes than BFS** | 115 | 150 | 48 | 92 | 87 |
| **Less nodes than IDDFS** | 143 | 153 | 140 | 147 | 148 |
| **Less nodes than IDDFS+PP+IMO** | 126 | 149 | 95 | 109 | 94 |
| **Faster than BFS** | 88 | 98 | 48 | 96 | 94 |
| **Faster than IDDFS** | 117 | 119 | 119 | 125 | 121 |
| **Faster than IDDFS+PP+IMO** | 125 | 132 | 129 | 131 | 128 |

**Table 2:** Summary of the results of Experiment #2

allow many puzzles to be solved considerably faster. For instance, puzzle M36 is solved by all of these solvers, even the iterative-deepening ones, in less time than by any of the forward solvers. Even the forward BFS, which did solve it in under a second, did worse than any of the reverse and bidirectional solvers, and when compared to the forward-searching IDDFS variants the difference in performance is staggering.

While in the forward-solving scenario the BFS solver explored less nodes than the IDDFS solvers on nearly every puzzle, the reverse and bidirectional solvers frequently solve puzzles with a smaller amount of nodes than the forward solvers (figure 29), as expected. Only the RIDDFS solver explores more nodes on average than BFS, but even that one manages to outperform BFS on some puzzles. In addition, the processing speeds and therefore solution times (figures 30 and 31, respectively) are better than with the forward BFS.

As an implementation-specific note, some solvers on some puzzles also ended up running into a memory limit, such as the RIDDFS solver on puzzles M145 and M146. This was to be expected, as there was no limit on the size of the transposition table but all the solvers stored every generated node in memory. Practical solver implementations should of course handle such scenarios gracefully in some way, such as limiting the size of the transposition table or clearing out known deadlocked branches when memory is getting low.

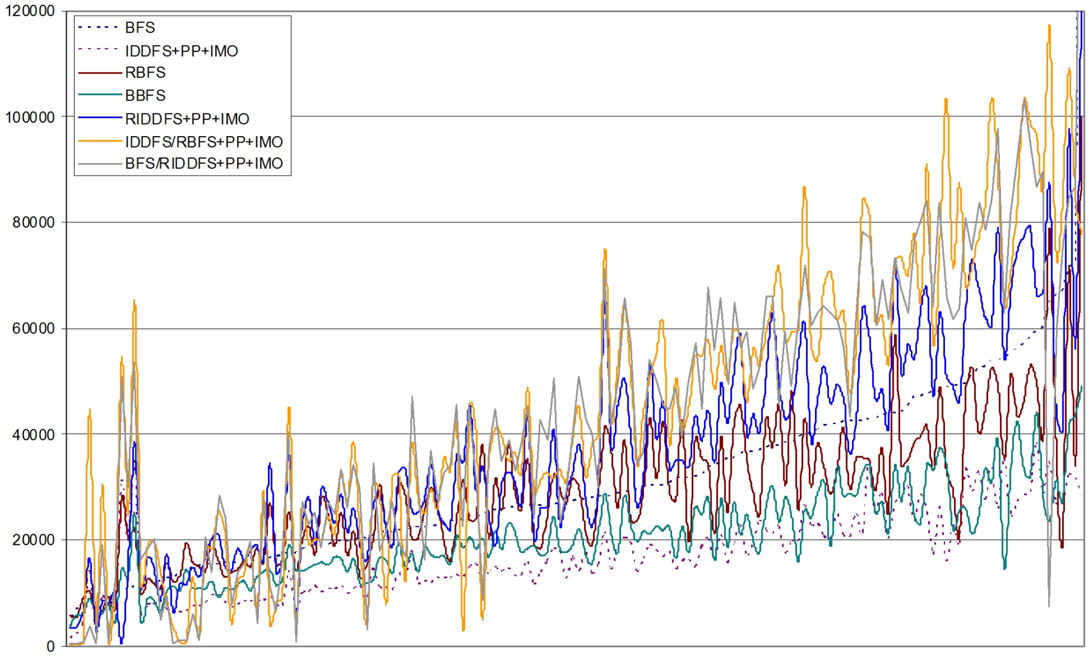


**Figure 30:** Processing speeds for Experiment #2

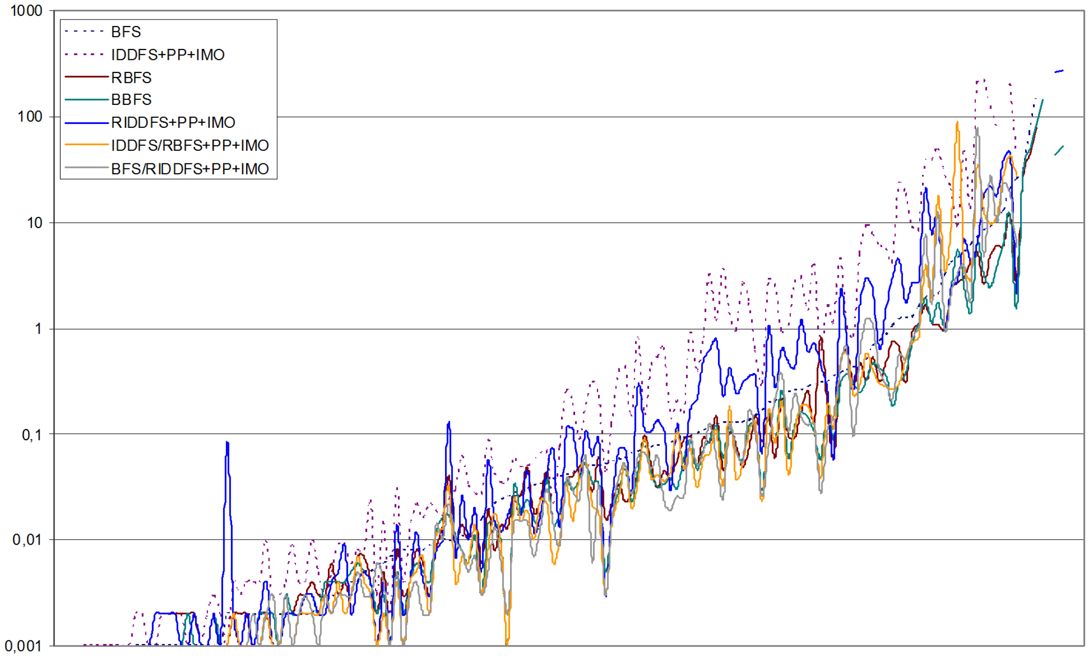


**Figure 31:** Processing times for Experiment #2

## 7.3 PI-corral Pruning

| | BFS+PI | IDDFS+PI+PP+IMO |
|---|---|---|
| **Solved** *(# of puzzles)* | 151 | 148 |
| **Solved optimally** *(# of puzzles)* | 151 | 148 |
| **Failed** *(# of puzzles)* | 7 | 10 |
| **Processing speed mean** *(1000 nodes/second)* | 21.7 | 28.4 |
| **Processing speed std.dev.** *(1000 nodes/second)* | 10.0 | 7.7 |
| **Processing speed median** *(1000 nodes/second)* | 20.9 | 17.7 |
| **Less nodes than BFS** *(# of puzzles)* | 125 | 43 |
| **Less nodes than IDDFS** *(# of puzzles)* | 144 | 142 |
| **Less nodes than IDDFS+PP+IMO** *(# of puzzles)* | 137 | 121 |
| **Faster than BFS** *(# of puzzles)* | 41 | 11 |
| **Faster than IDDFS** *(# of puzzles)* | 115 | 111 |
| **Faster than IDDFS+PP+IMO** *(# of puzzles)* | 121 | 107 |

**Table 3:** Summary of the results of Experiment #3

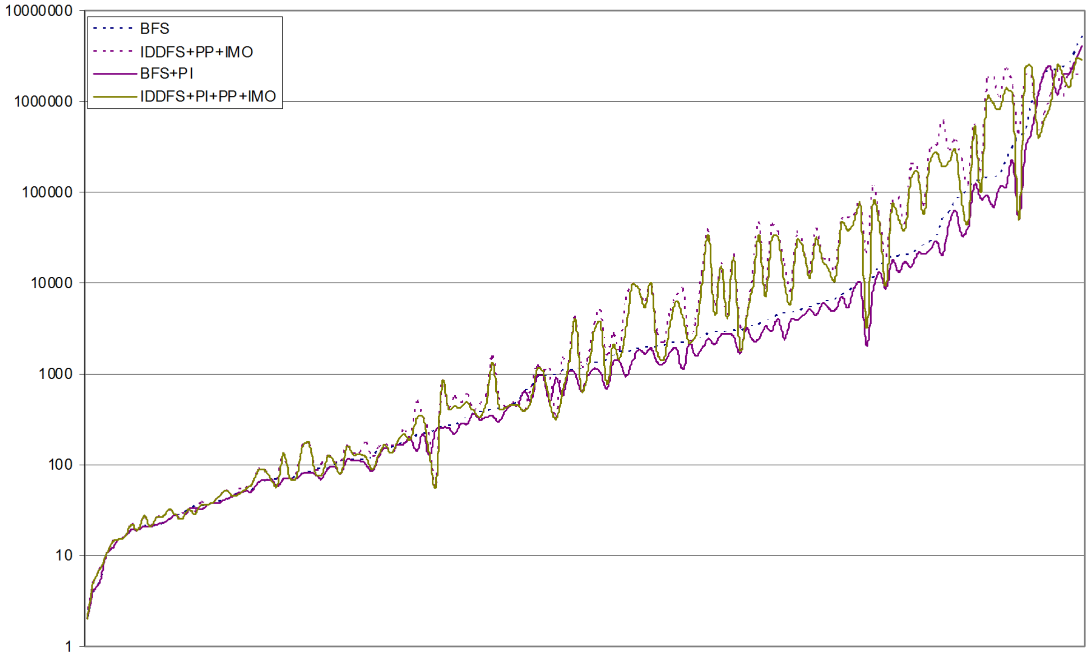


**Figure 32:** Explored nodes for Experiment #3

For the third experiment we studied the effects of PI-corral pruning, i.e. excluding all other moves from consideration when a PI-corral is present. The main focus of this experiment is on the pruning effect itself; to determine how much the pruning does decrease the size of the search space. This is illustrated in figure 32. Of course, adding such an enhancement also has an effect on processing speed and time. These are shown in figures 33 and 34, respectively.

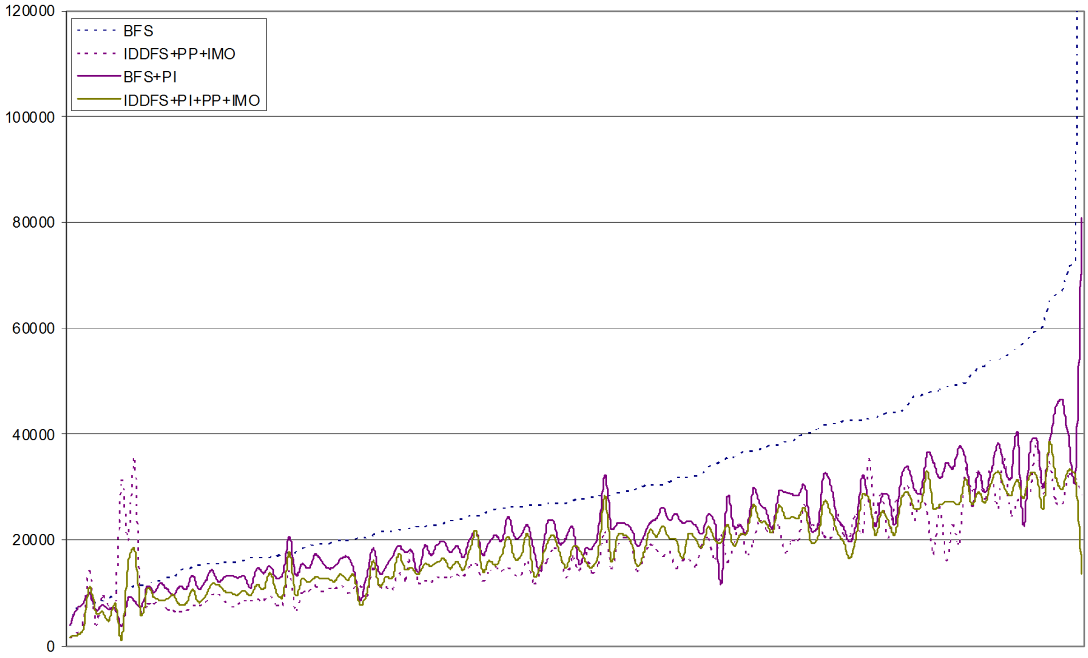


**Figure 33:** Processing speeds for Experiment #3

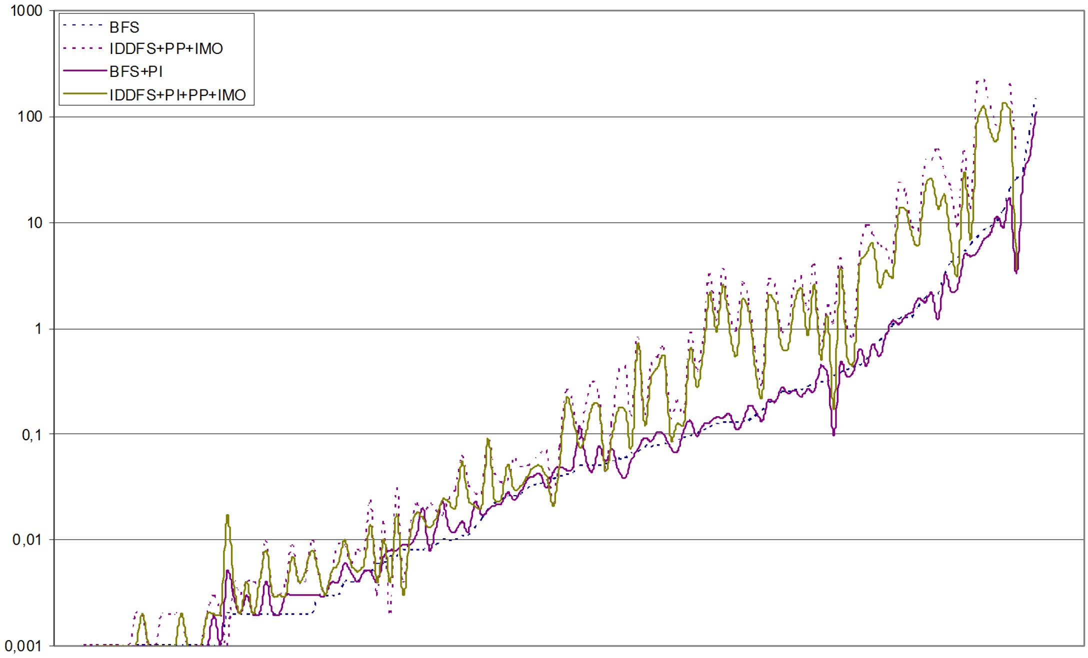


**Figure 34:** Processing times for Experiment #3

As expected, the amount of nodes explored by the PI-corral-pruning-enhanced version of BFS solver is always smaller than that of the unenhanced one. Unfortunately, adding the enhancement slows down the processing speed in nearly every case (puzzles M4, M40, M90, M93, and M146 being the exceptions) and thus increases the solution time. Nevertheless, the BFS+PI solver still solves a third of the puzzles faster than the plain BFS solver.

While the IDDFS solver enhanced with the PI-corral pruning enhancement is much slower than the BFS solvers, it does surprisingly outperform the IDDFS+PP+IMO solver, which does not have the PI-corral pruning enhancement, on the majority of the puzzles. This does indicate that it might be a good addition to an iterative-deepening-based solver such as Rolling Stone.

## 7.4 Van Lishout Solving Method

| | **BFS+VL** | **IDDFS+VL+PP+IMO** |
|---|---|---|
| **Solved** *(# of puzzles)* | 24 | 4 |
| **Solved optimally** *(# of puzzles)* | 4 | 1 |
| **Failed** *(# of puzzles)* | 66 | 86 |
| **Processing speed mean** *(1000 nodes/second)* | 2.7 | 2.7 |
| **Processing speed std.dev.** *(1000 nodes/second)* | 2.8 | 2.5 |
| **Processing speed median** *(1000 nodes/second)* | 1.2 | 1.5 |

**Table 4:** Summary of the results of Experiment #4

With the method for determining the goal packing order used by Van Lishout *et al.* [Lis06], only 9 puzzles of the XSokoban set could be solved with the stone-by-stone solving method. Using a better method for determining the goal packing order improves this result considerably, as can be seen from the results in this section (table 4 and figures 35 and 36). With our goal packing method described in section 6.3 and the stone-by-stone method added to our BFS solver, we were able to solve 24 of the 90 puzzles under the 300 second time limit. In addition, with our method we found another puzzle (X53) that was solvable stone-by-stone right from the initial state.

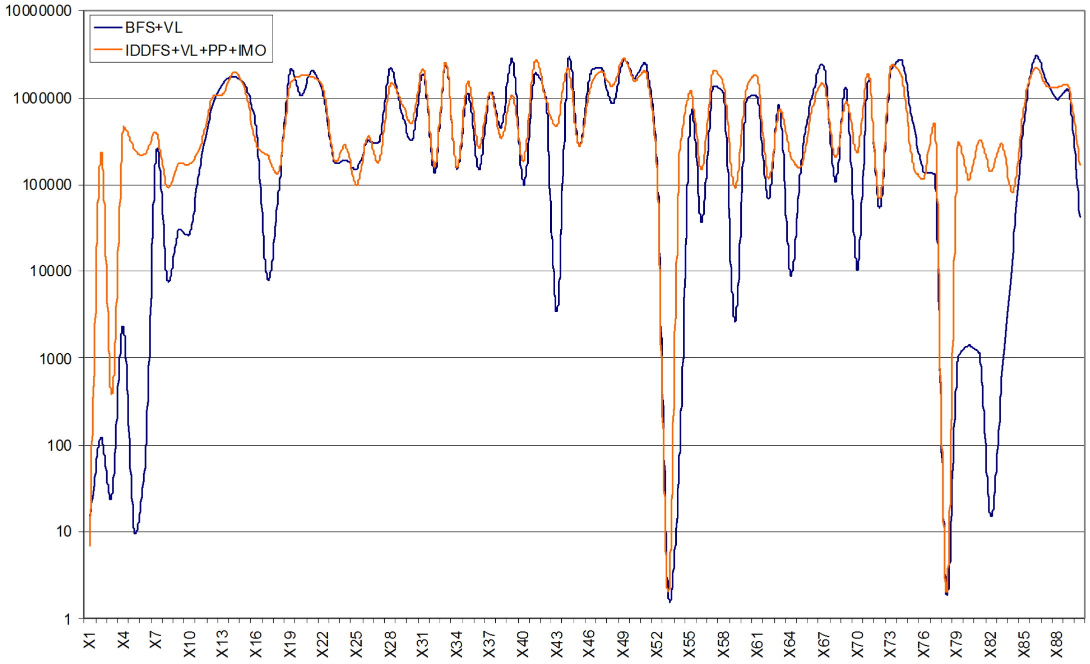


**Figure 35:** Explored nodes for Experiment #4

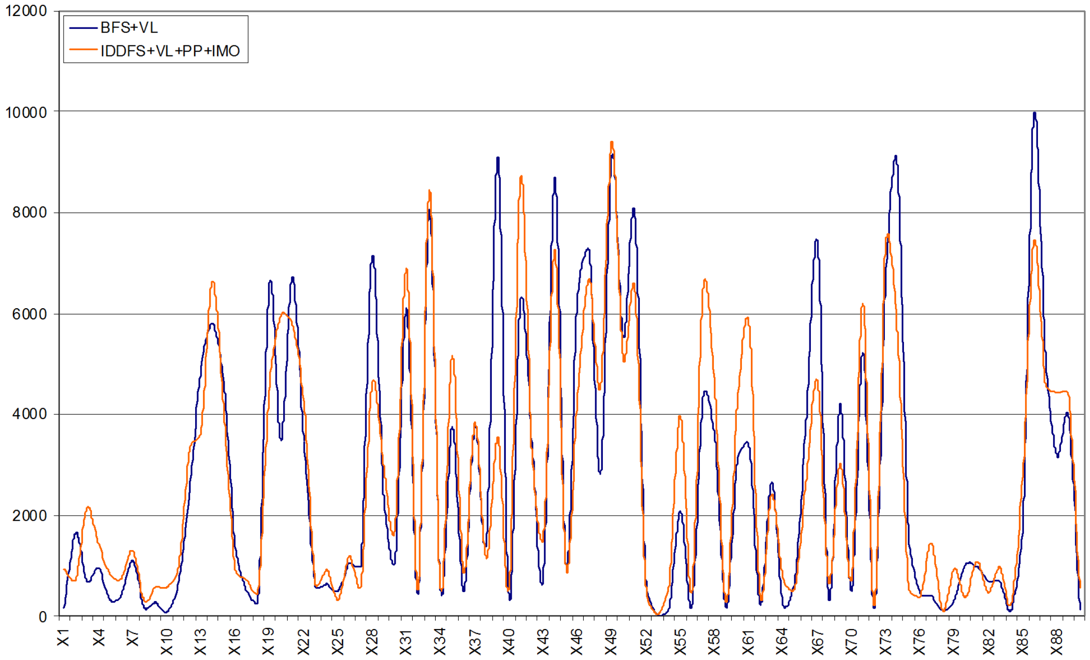


**Figure 36:** Processing speeds for Experiment #4

However, one has to note that of the 24 solved puzzles, only four were solved optimally. This is due to the fact that while a puzzle can be solved stone-by-stone, usually moving some stones slightly out of the way of others can allow the stones to travel via shorter paths and thus achieve a shorter total solution length.

Another fact to note is that when the stone-by-stone solving enhancement was added to the IDDFS+PP+IMO solver, the results were much less impressive. While the processing speeds were slightly higher than with the BFS solver, the amount of puzzles solved was only four. The reason for this is the nature of Depth-First Search: the search is designed to go deep into a search branch and explore that fully before trying other ones. If the state where the puzzle is solvable stone-by-stone is not in that branch, the search will spend too much time before arriving at that state. Clearly, most of the puzzles in the XSokoban set offer too many search branches to make IDDFS and VL a good combination.

Note that while the result graphs for the other experiments were sorted by the performance of the BFS solver, the graphs for this experiment show the results ordered by the puzzle number. Furthermore, as the puzzle set is different than in the other experiments, we are unable to compare the performance figures to the other solvers. However, comparing the processing speeds of these two solvers to the performance of the other solvers on similarly-sized puzzles in the Microban1 set, such as puzzles M144, M145 and M146, which were unsolved by most of the solvers, we can see that the processing speeds here are slightly lower. Only the bidirectional solvers show similarly low processing speeds than these.

## 7.5 Summary

The results in the previous sections show that while it is not easy to achieve results as impressive as those achieved by Rolling Stone and its heavily enhanced IDA* algorithm, the standard Breadth-First Algorithm might also provide a good basis for a Sokoban solver program. In addition, we have shown that with simple enhancements to particular subproblem algorithms (such as the goal packing order algorithm), the performance of existing algorithms can be greatly improved.

# 8 Conclusion

In the previous sections, we have presented an overview of the game of Sokoban, a review of standard graph search algorithms, a survey of ways in which those algorithms have been enhanced to perform better in Sokoban and tested the effects of various enhancements on those algorithms. We have discovered that it is quite possible to solve 24 puzzles of the XSokoban set without the need for the time-consuming lower-bound estimation algorithm that consumes most of the running time of Rolling Stone, albeit at the cost of having to employ another equally time consuming algorithm to do that, namely the Van Lishout stone-by-stone solving algorithm. Nevertheless, we have proven that besides IDA*, other search algorithms such as BFS may also provide a good platform for a Sokoban solver program. Furthermore, employing a breadth-first search strategy instead of a depth-first one may make it possible to develop other enhancements and heuristics that would not be applicable to a depth-first solver.

Despite these findings, based almost purely on intuition and familiarity with the domain, we think that the strategy of creating a solver based on a single search algorithm with various enhancements might not be the best approach for solving Sokoban. If we observe the ways in which a human player tries to solve Sokoban puzzles we can see that they employ different strategies to different puzzles. On some puzzles the player tries to find a stone-by-stone solvable state as quickly as possible, on others the puzzle is decomposed into smaller subproblems which are then solved, while on still others the player tries to identify hazards such as doors and one-way tunnels and adapts their strategy to take those into account. A similarly adaptive strategy might be advantageous to a Sokoban solver. It may be that the multi-layered abstraction approach taken by Botea *et al.* [BMS02] and the evolutionary learning approach of Schaul [Sch05] might indeed be good ways to start.

Whatever the approach, it is clear that there is room for more research on Sokoban. One good way to proceed would be to try to find a better algorithm for solving the goal packing order subproblem in Sokoban. As we have already discovered, a better algorithm for doing that can improve the results achievable by the Van Lishout algorithm considerably. Similarly, developing algorithms for other subproblems in Sokoban would provide progress with other types of Sokoban puzzles. While Sokoban is a game, it is not *just* a game but a hard computer science problem as well. Therefore, such algorithms could also be applicable to other, more practically useful domains.

# References

BMS02 Botea, A., Müller, M. and Schaeffer, J., Using abstraction for planning in sokoban. *Proceedings of the 3rd International Conference on Computers and Games*. Springer, 2002, pages 360–375.

CJ10 Cazenave, T. and Jouandeau, N., Towards deadlock free sokoban. *Board Games Studies Colloquium*, Paris, France, 2010, page 12.

Cul97 Culberson, J. C., Sokoban is PSPACE-complete. Technical Report, Univ of Alberta, 1997.

Dam10 Damgaard, B., Sokoban solver "scribbles"about the YASS solver, 2010. URL `http://www.sokobano.de/wiki/index.php?title=Sokoban_solver_%22scribbles%22_by_Brian_Damgaard_about_the_YASS_solver`.

DLG08 Demaret, J., Lishout, F. V. and Gribomont, P., Hierarchical planning and learning for automatic solving of sokoban problems. *20th Belgium-Netherlands Conference on Artificial Intelligence*, Enschede, The Netherlands, 2008, URL `http://hdl.handle.net/2268/5895`.

DZ99 Dor, D. and Zwick, U., SOKOBAN and other motion planning problems. *Computational Geometry*, 13,4(1999), pages 215–228. URL `http://dx.doi.org/10.1016/s0925-7721(99)00017-6`.

HNR68 Hart, P., Nilsson, N. and Raphael, B., A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4,2(1968), pages 100–107. URL `http://dx.doi.org/10.1109/TSSC.1968.300136`.

JS97 Junghanns, A. and Schaeffer, J., Sokoban: A challenging Single-Agent search problem. *In IJCAI Workshop on Using Games as an Experimental Testbed for AI Reasearch*, (1997), pages 27–36.

JS98a Junghanns, A. and Schaeffer, J., Relevance cuts: Localizing the search. *In The First International Conference on Computers and Games*, 1(1998), pages 1–13.

| | |
|---|---|
| JS98b | Junghanns, A. and Schaeffer, J., Single-Agent search in the presence of deadlocks. *IN AAAI*, (1998), pages 419–424. |
| JS98c | Junghanns, A. and Schaeffer, J., Sokoban: Evaluating standard Single-Agent search techniques in the presence of deadlock. *Advances in Artificial Intelligence*, 1998, pages 1–15. |
| JS98d | Junghanns, A. and Schaeffer, J., Sokoban: Improving the search with relevance cuts. *Journal of Theoretical Computing Science*, 252(1998), pages 1–2. |
| JS99 | Junghanns, A. and Schaeffer, J., Domain-Dependent Single-Agent search enhancements. *IN IJCAI-99*, (1999), pages 570–575. |
| JS01 | Junghanns, A. and Schaeffer, J., Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129,1-2(2001), pages 219–251. |
| Jun99 | Junghanns, A., *Pushing the Limits: New Developments in Single-Agent Search.* Doctoral dissertation, University of Alberta, Edmonton, Alberta, Canada, 1999. URL `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.947`. |
| Kuh55 | Kuhn, H. W., The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2,1-2(1955), pages 83–97. URL `http://onlinelibrary.wiley.com/doi/10.1002/nav.3800020109/abstract`. |
| Lis06 | Lishout, F. V., *Single-player games: introduction to a new solving method.* DEA en sciences appliquées, University of Liège, Liège, France, 2006. URL `http://orbi.ulg.ac.be/handle/2268/28467`. |
| Mye01 | Myers, A., XSokoban home page, 2001. URL `http://www.cs.cornell.edu/andru/xsokoban.html`. |
| RN09 | Russell, S. and Norvig, P., *Artificial Intelligence: A Modern Approach (3rd Edition)*, volume 3. Prentice Hall, 2009. |

Sch05 Schaul, T., *Evolving a compact, concept-based Sokoban solver.* Master's thesis, École Polytechnique Fédérale de Lausanne, May 2005. URL `http://www.whatisthought.com/schaulthesis.pdf`.

Ski00 Skinner, D. W., Microban levels, 2000. URL `http://users.bentonrea.com/~sasquatch/sokoban/`.

Tak08 Takes, F., *Sokoban: Reversed Solving.* Bachelor's thesis, Leiden Institute of Advanced Computer Science, January 2008.

# Appendix 1. Result Tables

This section contains the complete result tables of the experiments descibed in section 5. Discussion and summaries of these results can be found in section 7. For each puzzle, the results for each combination of solver and enhancements are shown. The following abbreviations are used:

**BFS** Breadth-First Search

**RBFS** Reverse Breadth-First Search

**BBFS** Bidirectional Breadth-First Search

**IDDFS** Iterative-Deepening Depth-First Search

**RIDDFS** Reverse IDDFS

**SD** Simple Deadlock Detection enhancement

**PP** Postponing enhancement

**IMO** Inertia Move Ordering enhancement

**PI** PI-corral Pruning enchancement

**VL** Van Lishout Solving enhancement

The name of the puzzle in each table contains the name of the puzzle set (M for Microban 1, X for XSokoban) and the puzzle number in that set. After the puzzle name the number of stones in that puzzle is shown. For each puzzle and solver, the length of the solution (***Path***), number of explored nodes (***Nodes***), processing speed (***Speed***) and solution time (***Time***) are shown. A checkmark symbol (✓) next to the path length means that the level was solved optimally. The processing speed is shown as thousands of nodes, i.e. a 2.5 means that the solver processed an average of 2500 nodes each second.

| | BFS | | | | IDDFS | | | | IDDFS+PP | | | | IDDFS+IMO | | | | IDDFS+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M1 (2) | ✓8 | 19 | 10.5 | 0.002 | ✓8 | 50 | 33.8 | 0.001 | ✓8 | 22 | 27.6 | 0.001 | ✓8 | 48 | 3.3 | 0.015 | ✓8 | 22 | 31.1 | 0.001 |
| M2 (3) | ✓3 | 5 | 11.1 | 0.000 | ✓3 | 7 | 25.0 | 0.000 | ✓3 | 7 | 23.1 | 0.000 | ✓3 | 7 | 15.1 | 0.000 | ✓3 | 7 | 19.9 | 0.000 |
| M3 (2) | ✓13 | 74 | 11.4 | 0.007 | ✓13 | 78 | 30.6 | 0.003 | ✓13 | 78 | 28.9 | 0.003 | ✓13 | 77 | 13.7 | 0.006 | ✓13 | 77 | 35.3 | 0.002 |
| M4 (3) | ✓7 | 44 | 7.6 | 0.006 | ✓7 | 153 | 17.6 | 0.009 | ✓7 | 45 | 15.0 | 0.003 | ✓7 | 150 | 10.8 | 0.014 | ✓7 | 45 | 14.3 | 0.003 |
| M5 (4) | ✓6 | 220 | 8.6 | 0.026 | ✓6 | 515 | 10.9 | 0.047 | ✓6 | 307 | 8.9 | 0.035 | ✓6 | 544 | 8.5 | 0.064 | ✓6 | 338 | 9.4 | 0.036 |
| M6 (3) | ✓29 | 409 | 22.1 | 0.019 | ✓29 | 3145 | 26.9 | 0.117 | ✓29 | 1667 | 18.5 | 0.090 | ✓29 | 3017 | 17.6 | 0.172 | ✓29 | 1614 | 18.1 | 0.089 |
| M7 (6) | ✓6 | 651 | 12.5 | 0.052 | ✓6 | 390 | 12.4 | 0.032 | ✓6 | 390 | 8.9 | 0.044 | ✓6 | 390 | 8.7 | 0.045 | ✓6 | 390 | 8.7 | 0.045 |
| M8 (2) | ✓32 | 118 | 42.9 | 0.003 | ✓32 | 736 | 51.4 | 0.014 | ✓32 | 118 | 29.3 | 0.004 | ✓32 | 975 | 33.0 | 0.030 | ✓32 | 121 | 35.4 | 0.003 |
| M9 (2) | ✓10 | 16 | 65.2 | 0.000 | ✓10 | 36 | 79.4 | 0.000 | ✓10 | 16 | 50.9 | 0.000 | ✓10 | 40 | 56.0 | 0.001 | ✓10 | 16 | 34.9 | 0.000 |
| M10 (3) | ✓21 | 117 | 35.7 | 0.003 | ✓21 | 1280 | 32.1 | 0.040 | ✓21 | 219 | 24.5 | 0.009 | ✓21 | 781 | 25.4 | 0.031 | ✓21 | 178 | 19.2 | 0.009 |
| M11 (2) | ✓16 | 83 | 36.9 | 0.002 | ✓16 | 313 | 34.8 | 0.009 | ✓16 | 162 | 25.4 | 0.006 | ✓16 | 333 | 22.8 | 0.015 | ✓16 | 174 | 17.2 | 0.010 |
| M12 (2) | ✓11 | 22 | 72.2 | 0.000 | ✓11 | 20 | 61.9 | 0.000 | ✓11 | 20 | 36.9 | 0.001 | ✓11 | 26 | 40.3 | 0.001 | ✓11 | 26 | 31.9 | 0.001 |
| M13 (3) | ✓21 | 313 | 37.7 | 0.008 | ✓21 | 2325 | 33.5 | 0.069 | ✓21 | 474 | 18.9 | 0.025 | ✓21 | 1903 | 24.1 | 0.079 | ✓21 | 476 | 22.1 | 0.022 |
| M14 (2) | ✓10 | 15 | 66.3 | 0.000 | ✓10 | 46 | 56.2 | 0.001 | ✓10 | 15 | 28.8 | 0.001 | ✓10 | 45 | 38.7 | 0.001 | ✓10 | 15 | 27.1 | 0.001 |
| M15 (2) | ✓12 | 29 | 71.8 | 0.000 | ✓12 | 82 | 53.1 | 0.002 | ✓12 | 29 | 35.1 | 0.001 | ✓12 | 81 | 42.6 | 0.002 | ✓12 | 29 | 32.3 | 0.001 |
| M16 (3) | ✓39 | 1363 | 26.8 | 0.051 | ✓39 | 5998 | 24.5 | 0.245 | ✓39 | 5998 | 18.0 | 0.333 | ✓39 | 5110 | 16.1 | 0.317 | ✓39 | 5110 | 16.1 | 0.317 |
| M17 (3) | ✓9 | 32 | 42.0 | 0.001 | ✓9 | 74 | 41.9 | 0.002 | ✓9 | 32 | 22.1 | 0.001 | ✓9 | 68 | 31.5 | 0.002 | ✓9 | 32 | 20.1 | 0.002 |
| M18 (2) | ✓13 | 67 | 37.1 | 0.002 | ✓13 | 176 | 33.1 | 0.005 | ✓13 | 100 | 26.6 | 0.004 | ✓13 | 120 | 25.0 | 0.005 | ✓13 | 88 | 23.7 | 0.004 |
| M19 (2) | ✓20 | 68 | 59.6 | 0.001 | ✓20 | 768 | 51.9 | 0.015 | ✓20 | 102 | 34.4 | 0.003 | ✓20 | 303 | 38.9 | 0.008 | ✓20 | 68 | 38.6 | 0.002 |
| M20 (2) | ✓16 | 71 | 47.3 | 0.002 | ✓16 | 583 | 42.2 | 0.014 | ✓16 | 71 | 25.6 | 0.003 | ✓16 | 397 | 24.7 | 0.016 | ✓16 | 71 | 23.7 | 0.003 |
| M21 (2) | ✓5 | 10 | 67.1 | 0.000 | ✓5 | 17 | 52.4 | 0.000 | ✓5 | 10 | 36.7 | 0.000 | ✓5 | 17 | 37.3 | 0.000 | ✓5 | 10 | 26.8 | 0.000 |
| M22 (2) | ✓15 | 70 | 31.7 | 0.002 | ✓15 | 486 | 25.7 | 0.019 | ✓15 | 137 | 15.2 | 0.009 | ✓15 | 347 | 19.1 | 0.018 | ✓15 | 134 | 14.7 | 0.009 |
| M23 (2) | ✓10 | 21 | 45.5 | 0.000 | ✓10 | 70 | 44.1 | 0.002 | ✓10 | 27 | 23.7 | 0.001 | ✓10 | 65 | 24.9 | 0.003 | ✓10 | 27 | 30.4 | 0.001 |
| M24 (2) | ✓9 | 50 | 56.3 | 0.001 | ✓9 | 55 | 49.3 | 0.001 | ✓9 | 55 | 30.9 | 0.002 | ✓9 | 54 | 36.4 | 0.001 | ✓9 | 54 | 27.3 | 0.002 |
| M25 (3) | ✓7 | 35 | 41.7 | 0.001 | ✓7 | 29 | 36.8 | 0.001 | ✓7 | 29 | 23.8 | 0.001 | ✓7 | 30 | 21.6 | 0.001 | ✓7 | 30 | 20.9 | 0.001 |
| M26 (3) | ✓10 | 91 | 54.1 | 0.002 | ✓10 | 254 | 51.8 | 0.005 | ✓10 | 104 | 30.5 | 0.003 | ✓10 | 237 | 27.2 | 0.009 | ✓10 | 102 | 25.9 | 0.004 |
| M27 (2) | ✓10 | 42 | 42.8 | 0.001 | ✓10 | 147 | 40.7 | 0.004 | ✓10 | 56 | 24.2 | 0.002 | ✓10 | 159 | 26.4 | 0.006 | ✓10 | 52 | 21.4 | 0.002 |
| M28 (2) | ✓9 | 21 | 49.0 | 0.000 | ✓9 | 72 | 44.5 | 0.002 | ✓9 | 21 | 31.7 | 0.001 | ✓9 | 65 | 34.5 | 0.002 | ✓9 | 21 | 16.1 | 0.001 |
| M29 (2) | ✓22 | 81 | 34.2 | 0.002 | ✓22 | 672 | 27.4 | 0.025 | ✓22 | 191 | 16.1 | 0.012 | ✓22 | 561 | 19.3 | 0.029 | ✓22 | 158 | 16.3 | 0.010 |
| M30 (3) | ✓5 | 25 | 35.5 | 0.001 | ✓5 | 32 | 30.4 | 0.001 | ✓5 | 32 | 17.8 | 0.002 | ✓5 | 32 | 20.4 | 0.002 | ✓5 | 32 | 15.9 | 0.002 |
| M31 (3) | ✓6 | 40 | 38.0 | 0.001 | ✓6 | 48 | 37.3 | 0.001 | ✓6 | 41 | 23.7 | 0.002 | ✓6 | 60 | 23.2 | 0.003 | ✓6 | 47 | 22.6 | 0.002 |
| M32 (3) | ✓9 | 39 | 58.6 | 0.001 | ✓9 | 149 | 56.5 | 0.003 | ✓9 | 40 | 31.2 | 0.001 | ✓9 | 145 | 29.7 | 0.005 | ✓9 | 40 | 29.0 | 0.001 |
| M33 (3) | ✓10 | 243 | 30.5 | 0.008 | ✓10 | 75 | 31.8 | 0.002 | ✓10 | 75 | 20.0 | 0.004 | ✓10 | 60 | 19.1 | 0.003 | ✓10 | 60 | 16.9 | 0.004 |
| M34 (4) | ✓8 | 166 | 20.9 | 0.008 | ✓8 | 609 | 20.6 | 0.030 | ✓8 | 182 | 12.6 | 0.014 | ✓8 | 606 | 14.6 | 0.041 | ✓8 | 181 | 14.8 | 0.012 |
| M35 (5) | ✓31 | 5440 | 21.8 | 0.249 | ✓31 | 164233 | 19.9 | 8.240 | ✓31 | 12905 | 15.2 | 0.848 | ✓31 | 161445 | 12.9 | 12.486 | ✓31 | 12887 | 13.7 | 0.939 |
| M36 (5) | ✓59 | 20133 | 26.2 | 0.769 | ✓59 | 2631727 | 21.7 | 121.449 | ✓59 | 97372 | 16.4 | 5.942 | ✓59 | 2438114 | 13.8 | 176.302 | ✓59 | 90368 | 14.8 | 6.086 |
| M37 (3) | ✓23 | 266 | 38.5 | 0.007 | ✓23 | 2402 | 32.8 | 0.073 | ✓23 | 727 | 23.0 | 0.032 | ✓23 | 1620 | 22.4 | 0.072 | ✓23 | 557 | 17.7 | 0.031 |
| M38 (3) | ✓8 | 23 | 24.6 | 0.001 | ✓8 | 28 | 25.1 | 0.001 | ✓8 | 28 | 15.0 | 0.002 | ✓8 | 27 | 15.4 | 0.002 | ✓8 | 27 | 15.7 | 0.002 |
| M39 (2) | ✓27 | 107 | 52.9 | 0.002 | ✓27 | 883 | 50.9 | 0.017 | ✓27 | 107 | 40.4 | 0.003 | ✓27 | 709 | 36.6 | 0.019 | ✓27 | 107 | 27.7 | 0.004 |
| M40 (3) | ✓7 | 68 | 28.4 | 0.002 | ✓7 | 159 | 41.7 | 0.004 | ✓7 | 68 | 24.1 | 0.003 | ✓7 | 161 | 29.4 | 0.005 | ✓7 | 70 | 21.7 | 0.003 |
| M41 (3) | ✓13 | 52 | 36.9 | 0.001 | ✓13 | 154 | 33.8 | 0.005 | ✓13 | 64 | 21.1 | 0.003 | ✓13 | 139 | 18.8 | 0.007 | ✓13 | 62 | 19.5 | 0.003 |
| M42 (3) | ✓15 | 157 | 40.2 | 0.004 | ✓15 | 1747 | 38.6 | 0.045 | ✓15 | 172 | 23.3 | 0.007 | ✓15 | 1474 | 25.1 | 0.059 | ✓15 | 169 | 26.6 | 0.006 |
| M43 (3) | ✓22 | 336 | 30.5 | 0.011 | ✓22 | 4224 | 25.3 | 0.167 | ✓22 | 591 | 17.3 | 0.034 | ✓22 | 4381 | 17.0 | 0.258 | ✓22 | 607 | 18.3 | 0.033 |
| M44 (1) | ✓1 | 2 | 190.9 | 0.000 | ✓1 | 2 | 13.1 | 0.000 | ✓1 | 2 | 25.8 | 0.000 | ✓1 | 2 | 28.3 | 0.000 | ✓1 | 2 | 28.4 | 0.000 |
| M45 (3) | ✓11 | 116 | 49.7 | 0.002 | ✓11 | 202 | 49.4 | 0.004 | ✓11 | 140 | 29.0 | 0.005 | ✓11 | 152 | 34.2 | 0.004 | ✓11 | 136 | 33.2 | 0.004 |
| M46 (2) | ✓8 | 19 | 48.1 | 0.000 | ✓8 | 55 | 47.2 | 0.001 | ✓8 | 19 | 24.8 | 0.001 | ✓8 | 54 | 24.8 | 0.002 | ✓8 | 19 | 17.3 | 0.001 |
| M47 (2) | ✓22 | 92 | 43.8 | 0.002 | ✓22 | 909 | 37.5 | 0.024 | ✓22 | 182 | 21.3 | 0.009 | ✓22 | 442 | 26.0 | 0.017 | ✓22 | 127 | 28.9 | 0.004 |
| M48 (3) | ✓14 | 176 | 31.9 | 0.006 | ✓14 | 1360 | 28.7 | 0.047 | ✓14 | 242 | 18.2 | 0.013 | ✓14 | 1356 | 20.1 | 0.067 | ✓14 | 241 | 16.4 | 0.015 |
| M49 (3) | ✓21 | 210 | 44.0 | 0.005 | ✓21 | 1690 | 37.1 | 0.046 | ✓21 | 505 | 28.7 | 0.018 | ✓21 | 1528 | 22.5 | 0.068 | ✓21 | 489 | 20.4 | 0.024 |
| M50 (2) | ✓17 | 64 | 29.2 | 0.002 | ✓17 | 486 | 25.7 | 0.019 | ✓17 | 86 | 20.4 | 0.004 | ✓17 | 491 | 18.7 | 0.026 | ✓17 | 86 | 19.7 | 0.004 |
| M51 (2) | ✓8 | 28 | 49.1 | 0.001 | ✓8 | 55 | 46.3 | 0.001 | ✓8 | 28 | 25.2 | 0.001 | ✓8 | 55 | 32.4 | 0.002 | ✓8 | 28 | 22.9 | 0.001 |
| M52 (4) | ✓8 | 414 | 26.5 | 0.016 | ✓8 | 1094 | 25.4 | 0.043 | ✓8 | 431 | 19.9 | 0.022 | ✓8 | 1088 | 15.1 | 0.072 | ✓8 | 430 | 16.0 | 0.027 |
| M53 (4) | ✓12 | 149 | 47.3 | 0.003 | ✓12 | 736 | 42.7 | 0.017 | ✓12 | 156 | 29.8 | 0.005 | ✓12 | 727 | 29.1 | 0.025 | ✓12 | 156 | 28.6 | 0.005 |
| M54 (4) | ✓30 | 3473 | 26.3 | 0.132 | ✓30 | 66382 | 20.8 | 3.184 | ✓30 | 13468 | 15.2 | 0.889 | ✓30 | 58208 | 13.4 | 4.352 | ✓30 | 12682 | 13.4 | 0.948 |
| M55 (2) | ✓27 | 103 | 48.3 | 0.002 | ✓27 | 59 | 45.9 | 0.001 | ✓27 | 59 | 28.3 | 0.002 | ✓27 | 110 | 33.8 | 0.003 | ✓27 | 110 | 26.4 | 0.004 |
| M56 (2) | ✓6 | 12 | 49.5 | 0.000 | ✓6 | 26 | 45.1 | 0.001 | ✓6 | 14 | 21.7 | 0.001 | ✓6 | 26 | 24.0 | 0.001 | ✓6 | 14 | 19.6 | 0.001 |
| M57 (2) | ✓23 | 88 | 54.1 | 0.002 | ✓23 | 640 | 53.2 | 0.012 | ✓23 | 89 | 29.2 | 0.003 | ✓23 | 634 | 29.5 | 0.021 | ✓23 | 89 | 33.9 | 0.003 |
| M58 (3) | ✓11 | 52 | 52.5 | 0.001 | ✓11 | 123 | 50.3 | 0.002 | ✓11 | 54 | 37.2 | 0.001 | ✓11 | 125 | 35.0 | 0.004 | ✓11 | 55 | 32.9 | 0.002 |
| M59 (3) | ✓50 | 1972 | 15.7 | 0.125 | ✓50 | 102020 | 12.2 | 8.378 | ✓50 | 9868 | 8.6 | 1.143 | ✓50 | 100644 | 8.1 | 12.392 | ✓50 | 9751 | 8.5 | 1.145 |
| M60 (4) | ✓44 | 2181 | 42.6 | 0.051 | ✓44 | 82760 | 32.9 | 2.513 | ✓44 | 4856 | 25.2 | 0.193 | ✓44 | 68738 | 21.7 | 3.166 | ✓44 | 4902 | 21.2 | 0.232 |

**Table 5:** Results of experiment 1 *(part 1)*

| | BFS | | | | IDDFS | | | | IDDFS+PP | | | | IDDFS+IMO | | | | IDDFS+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M61 (4) | ✓21 | 266 | 26.9 | 0.010 | ✓21 | 2398 | 23.5 | 0.102 | ✓21 | 492 | 19.1 | 0.026 | ✓21 | 2035 | 17.1 | 0.119 | ✓21 | 425 | 18.0 | 0.024 |
| M62 (4) | ✓30 | 704 | 29.2 | 0.024 | ✓30 | 7539 | 29.5 | 0.256 | ✓30 | 708 | 17.4 | 0.041 | ✓30 | 7248 | 18.9 | 0.383 | ✓30 | 708 | 20.5 | 0.035 |
| M63 (2) | ✓50 | 223 | 60.4 | 0.004 | ✓50 | 2910 | 51.7 | 0.056 | ✓50 | 223 | 40.9 | 0.005 | ✓50 | 2397 | 38.9 | 0.062 | ✓50 | 223 | 28.7 | 0.008 |
| M64 (4) | ✓30 | 379 | 42.0 | 0.009 | ✓30 | 8175 | 35.8 | 0.228 | ✓30 | 502 | 15.2 | 0.033 | ✓30 | 5348 | 21.9 | 0.245 | ✓30 | 464 | 24.8 | 0.019 |
| M65 (4) | ✓41 | 4520 | 22.8 | 0.198 | ✓41 | 118735 | 19.1 | 6.213 | ✓41 | 37049 | 13.7 | 2.698 | ✓41 | 110368 | 12.5 | 8.855 | ✓41 | 36866 | 12.9 | 2.868 |
| M66 (3) | ✓15 | 489 | 15.6 | 0.031 | ✓15 | 2640 | 14.5 | 0.182 | ✓15 | 544 | 11.6 | 0.047 | ✓15 | 2337 | 9.5 | 0.245 | ✓15 | 493 | 9.9 | 0.050 |
| M67 (3) | ✓8 | 35 | 54.6 | 0.001 | ✓8 | 100 | 49.6 | 0.002 | ✓8 | 38 | 30.1 | 0.001 | ✓8 | 94 | 34.2 | 0.003 | ✓8 | 39 | 35.6 | 0.001 |
| M68 (3) | ✓28 | 394 | 39.7 | 0.010 | ✓28 | 5271 | 34.4 | 0.153 | ✓28 | 601 | 26.7 | 0.023 | ✓28 | 5688 | 22.6 | 0.252 | ✓28 | 589 | 19.8 | 0.030 |
| M69 (3) | ✓37 | 1958 | 25.6 | 0.077 | ✓37 | 59041 | 20.6 | 2.869 | ✓37 | 6552 | 14.6 | 0.448 | ✓37 | 54972 | 13.6 | 4.032 | ✓37 | 6257 | 14.0 | 0.448 |
| M70 (4) | ✓26 | 2364 | 26.9 | 0.088 | ✓26 | 23106 | 27.3 | 0.845 | ✓26 | 4303 | 17.8 | 0.241 | ✓26 | 20587 | 17.9 | 1.152 | ✓26 | 3861 | 18.4 | 0.210 |
| M71 (2) | ✓21 | 114 | 27.9 | 0.004 | ✓21 | 375 | 20.7 | 0.018 | ✓21 | 375 | 16.0 | 0.023 | ✓21 | 160 | 17.6 | 0.009 | ✓21 | 160 | 17.7 | 0.009 |
| M72 (3) | ✓40 | 1104 | 42.6 | 0.026 | ✓40 | 19311 | 36.2 | 0.534 | ✓40 | 1268 | 28.6 | 0.044 | ✓40 | 22759 | 23.0 | 0.991 | ✓40 | 1461 | 25.0 | 0.058 |
| M73 (3) | ✓25 | 986 | 25.8 | 0.038 | ✓25 | 2669 | 23.3 | 0.114 | ✓25 | 2669 | 17.9 | 0.149 | ✓25 | 323 | 13.5 | 0.024 | ✓25 | 323 | 14.9 | 0.022 |
| M74 (4) | ✓34 | 2905 | 28.5 | 0.102 | ✓34 | 14182 | 23.7 | 0.598 | ✓34 | 6004 | 17.2 | 0.350 | ✓34 | 13172 | 15.9 | 0.828 | ✓34 | 5713 | 14.6 | 0.393 |
| M75 (4) | ✓34 | 1323 | 31.0 | 0.043 | ✓34 | 18163 | 27.6 | 0.659 | ✓34 | 3779 | 19.3 | 0.196 | ✓34 | 17157 | 17.2 | 0.998 | ✓34 | 3522 | 18.0 | 0.195 |
| M76 (3) | ✓56 | 1820 | 26.6 | 0.068 | ✓56 | 18784 | 20.7 | 0.907 | ✓56 | 18784 | 15.0 | 1.248 | ✓56 | 9732 | 11.6 | 0.839 | ✓56 | 9732 | 11.7 | 0.835 |
| M77 (4) | ✓55 | 4179 | 31.7 | 0.132 | ✓55 | 126600 | 24.1 | 5.246 | ✓55 | 39988 | 17.4 | 2.303 | ✓55 | 134813 | 15.7 | 8.577 | ✓55 | 42878 | 16.1 | 2.666 |
| M78 (5) | ✓33 | 10617 | 15.6 | 0.679 | ✓33 | 234657 | 14.5 | 16.187 | ✓33 | 78138 | 10.5 | 7.459 | ✓33 | 216230 | 9.5 | 22.713 | ✓33 | 74604 | 9.6 | 7.756 |
| M79 (3) | ✓18 | 159 | 33.9 | 0.005 | ✓18 | 567 | 34.7 | 0.016 | ✓18 | 159 | 19.1 | 0.008 | ✓18 | 625 | 25.0 | 0.025 | ✓18 | 160 | 20.4 | 0.008 |
| M80 (4) | ✓38 | 1775 | 30.2 | 0.059 | ✓38 | 61588 | 24.8 | 2.488 | ✓38 | 6889 | 18.0 | 0.382 | ✓38 | 53812 | 16.1 | 3.346 | ✓38 | 6463 | 16.5 | 0.392 |
| M81 (3) | ✓12 | 116 | 44.2 | 0.003 | ✓12 | 438 | 41.7 | 0.011 | ✓12 | 120 | 31.4 | 0.004 | ✓12 | 509 | 28.5 | 0.018 | ✓12 | 135 | 27.7 | 0.005 |
| M82 (3) | ✓14 | 36 | 55.4 | 0.001 | ✓14 | 112 | 46.1 | 0.002 | ✓14 | 39 | 27.0 | 0.001 | ✓14 | 108 | 25.1 | 0.004 | ✓14 | 36 | 24.8 | 0.001 |
| M83 (4) | ✓47 | 6219 | 19.9 | 0.312 | ✓47 | 498640 | 15.9 | 31.435 | ✓47 | 19154 | 12.2 | 1.566 | ✓47 | 496625 | 10.4 | 47.529 | ✓47 | 19163 | 11.6 | 1.659 |
| M84 (3) | ✓68 | 3524 | 29.9 | 0.118 | ✓68 | 111321 | 20.8 | 5.339 | ✓68 | 55363 | 14.8 | 3.730 | ✓68 | 94132 | 14.1 | 6.660 | ✓68 | 47453 | 14.2 | 3.352 |
| M85 (3) | ✓51 | 5016 | 19.1 | 0.262 | ✓51 | 128701 | 17.0 | 7.590 | ✓51 | 40299 | 12.1 | 3.320 | ✓51 | 113012 | 11.4 | 9.901 | ✓51 | 35485 | 11.4 | 3.106 |
| M86 (4) | ✓25 | 1115 | 27.5 | 0.041 | ✓25 | 15480 | 24.6 | 0.629 | ✓25 | 4386 | 17.2 | 0.255 | ✓25 | 14863 | 16.1 | 0.924 | ✓25 | 4319 | 16.9 | 0.256 |
| M87 (4) | ✓53 | 7980 | 27.0 | 0.295 | ✓53 | 214447 | 19.7 | 10.891 | ✓53 | 52500 | 14.0 | 3.737 | ✓53 | 189033 | 12.8 | 14.814 | ✓53 | 52184 | 12.7 | 4.100 |
| M88 (3) | ✓63 | 2803 | 21.8 | 0.129 | ✓63 | 43258 | 16.2 | 2.674 | ✓63 | 43258 | 11.4 | 3.802 | ✓63 | 39181 | 10.7 | 3.668 | ✓63 | 39181 | 10.6 | 3.685 |
| M89 (4) | ✓35 | 5677 | 28.2 | 0.201 | ✓35 | 40744 | 23.4 | 1.741 | ✓35 | 40744 | 16.4 | 2.490 | ✓35 | 40599 | 15.0 | 2.699 | ✓35 | 40599 | 15.4 | 2.630 |
| M90 (4) | ✓16 | 2978 | 17.6 | 0.169 | ✓16 | 23369 | 21.5 | 1.088 | ✓16 | 4360 | 13.6 | 0.322 | ✓16 | 22981 | 14.1 | 1.628 | ✓16 | 4318 | 14.3 | 0.302 |
| M91 (4) | ✓14 | 410 | 47.9 | 0.009 | ✓14 | 1933 | 45.1 | 0.043 | ✓14 | 555 | 25.8 | 0.022 | ✓14 | 1885 | 31.1 | 0.061 | ✓14 | 553 | 25.3 | 0.022 |
| M92 (3) | ✓48 | 1436 | 40.3 | 0.036 | ✓48 | 35661 | 33.8 | 1.056 | ✓48 | 1854 | 25.3 | 0.073 | ✓48 | 39681 | 21.7 | 1.831 | ✓48 | 1617 | 22.9 | 0.071 |
| M93 (8) | – | 2215888 | 7.4 | – | – | 1258376 | 4.2 | – | – | 1196007 | 3.9 | – | – | 1179131 | 3.9 | – | – | 1190688 | 3.9 | – |
| M94 (3) | ✓29 | 464 | 57.1 | 0.008 | ✓29 | 7183 | 47.5 | 0.151 | ✓29 | 466 | 31.4 | 0.015 | ✓29 | 6198 | 29.3 | 0.212 | ✓29 | 466 | 28.5 | 0.016 |
| M95 (8) | ✓8 | 3191 | 10.4 | 0.306 | ✓8 | 3216 | 9.8 | 0.328 | ✓8 | 3216 | 7.0 | 0.460 | ✓8 | 4043 | 6.4 | 0.631 | ✓8 | 4043 | 6.6 | 0.615 |
| M96 (3) | ✓37 | 1162 | 43.0 | 0.027 | ✓37 | 29549 | 33.4 | 0.884 | ✓37 | 1247 | 22.9 | 0.054 | ✓37 | 31752 | 21.1 | 1.504 | ✓37 | 1240 | 25.1 | 0.049 |
| M97 (5) | ✓41 | 9555 | 21.5 | 0.444 | ✓41 | 109055 | 18.1 | 6.021 | ✓41 | 48731 | 12.8 | 3.815 | ✓41 | 171016 | 11.1 | 15.410 | ✓41 | 56343 | 11.2 | 5.023 |
| M98 (5) | ✓110 | 91208 | 21.7 | 4.196 | – | 4628281 | 15.4 | – | ✓110 | 251723 | 11.9 | 21.229 | – | 3060061 | 10.2 | – | ✓110 | 234094 | 11.1 | 21.105 |
| M99 (4) | ✓131 | 147878 | 20.6 | 7.170 | – | 4035368 | 13.5 | – | ✓131 | 1984661 | 9.5 | 209.076 | – | 2744506 | 9.1 | – | ✓131 | 1896584 | 9.2 | 206.149 |
| M100 (4) | ✓52 | 2262 | 38.5 | 0.059 | ✓52 | 120683 | 30.0 | 4.021 | ✓52 | 8366 | 22.4 | 0.373 | ✓52 | 128134 | 19.7 | 6.513 | ✓52 | 8716 | 19.9 | 0.439 |
| M101 (5) | ✓15 | 937 | 12.2 | 0.077 | ✓15 | 9140 | 11.2 | 0.816 | ✓15 | 1220 | 8.5 | 0.144 | ✓15 | 9105 | 7.6 | 1.191 | ✓15 | 1203 | 7.9 | 0.152 |
| M102 (4) | ✓44 | 5222 | 20.0 | 0.261 | ✓44 | 129328 | 14.8 | 8.736 | ✓44 | 27273 | 10.5 | 2.606 | ✓44 | 121248 | 9.9 | 12.298 | ✓44 | 26429 | 9.9 | 2.682 |
| M103 (4) | ✓12 | 956 | 28.9 | 0.033 | ✓12 | 5777 | 26.8 | 0.215 | ✓12 | 1069 | 20.9 | 0.051 | ✓12 | 5761 | 17.3 | 0.334 | ✓12 | 1067 | 18.9 | 0.057 |
| M104 (3) | ✓27 | 373 | 27.8 | 0.013 | ✓27 | 5648 | 24.3 | 0.233 | ✓27 | 411 | 15.5 | 0.026 | ✓27 | 5919 | 15.8 | 0.375 | ✓27 | 403 | 14.2 | 0.028 |
| M105 (8) | ✓24 | 436579 | 16.8 | 25.912 | ✓24 | 2613388 | 14.5 | 180.068 | ✓24 | 444714 | 10.3 | 43.159 | ✓24 | 2558259 | 9.4 | 270.746 | ✓24 | 444747 | 9.6 | 46.560 |
| M106 (5) | ✓50 | 11347 | 22.6 | 0.501 | ✓50 | 243876 | 18.9 | 12.911 | ✓50 | 122038 | 13.4 | 9.076 | ✓50 | 227074 | 12.4 | 18.254 | ✓50 | 115990 | 12.3 | 9.417 |
| M107 (11) | ✓10 | 1108 | 20.1 | 0.055 | ✓10 | 2937 | 19.3 | 0.152 | ✓10 | 1339 | 11.3 | 0.119 | ✓10 | 2887 | 11.9 | 0.242 | ✓10 | 1327 | 10.9 | 0.122 |
| M108 (4) | ✓68 | 33235 | 16.1 | 2.062 | – | 2551819 | 8.5 | – | ✓68 | 330996 | 8.8 | 37.497 | – | 2547979 | 8.5 | – | ✓68 | 329067 | 8.5 | 38.586 |
| M109 (5) | ✓42 | 60730 | 18.8 | 3.231 | – | 3140865 | 10.5 | – | ✓42 | 294699 | 11.1 | 26.611 | – | 3085209 | 10.3 | – | ✓42 | 292367 | 10.4 | 28.196 |
| M110 (4) | ✓14 | 2038 | 51.4 | 0.040 | ✓14 | 7226 | 33.4 | 0.217 | ✓14 | 2151 | 30.4 | 0.071 | ✓14 | 7210 | 25.5 | 0.283 | ✓14 | 2210 | 29.2 | 0.076 |
| M111 (6) | ✓61 | 327238 | 15.9 | 20.566 | – | 2246474 | 7.5 | – | ✓61 | 1511888 | 7.8 | 192.800 | – | 2204238 | 7.3 | – | ✓61 | 1507651 | 7.4 | 204.505 |
| M112 (5) | ✓94 | 221577 | 17.0 | 13.066 | – | 2260546 | 7.5 | – | – | 2376893 | 7.9 | – | – | 2345133 | 7.8 | – | – | 2345006 | 7.8 | – |
| M113 (4) | ✓51 | 26751 | 16.7 | 1.603 | ✓51 | 1789598 | 8.1 | 219.982 | ✓51 | 71042 | 9.2 | 7.749 | ✓51 | 1784328 | 8.0 | 224.393 | ✓51 | 71110 | 8.5 | 8.339 |
| M114 (6) | ✓60 | 51733 | 24.7 | 2.093 | ✓60 | 2703820 | 12.5 | 215.813 | ✓60 | 615087 | 13.4 | 45.798 | ✓60 | 2709746 | 12.1 | 223.339 | ✓60 | 614915 | 12.2 | 50.335 |
| M115 (5) | ✓29 | 20549 | 19.4 | 1.062 | ✓29 | 150496 | 11.1 | 13.581 | ✓29 | 45566 | 11.6 | 3.913 | ✓29 | 148522 | 10.6 | 14.053 | ✓29 | 45053 | 10.8 | 4.171 |
| M116 (5) | ✓14 | 2029 | 30.4 | 0.067 | ✓14 | 7417 | 19.4 | 0.383 | ✓14 | 2919 | 19.6 | 0.149 | ✓14 | 7451 | 17.8 | 0.419 | ✓14 | 2828 | 19.1 | 0.148 |
| M117 (5) | ✓47 | 131805 | 12.8 | 10.318 | – | 2125024 | 7.1 | – | ✓47 | 569288 | 7.6 | 74.517 | – | 2091505 | 7.0 | – | ✓47 | 569843 | 7.0 | 80.933 |
| M118 (4) | ✓44 | 7522 | 19.6 | 0.384 | ✓44 | 310713 | 10.6 | 29.234 | ✓44 | 49571 | 11.8 | 4.207 | ✓44 | 316722 | 10.6 | 29.922 | ✓44 | 50240 | 10.7 | 4.699 |
| M119 (3) | ✓18 | 264 | 23.7 | 0.011 | ✓18 | 1755 | 14.0 | 0.125 | ✓18 | 829 | 14.6 | 0.057 | ✓18 | 1546 | 9.2 | 0.168 | ✓18 | 820 | 13.0 | 0.063 |
| M120 (4) | ✓64 | 4675 | 42.6 | 0.110 | ✓64 | 28633 | 23.4 | 1.225 | ✓64 | 10631 | 23.6 | 0.450 | ✓64 | 47978 | 20.8 | 2.304 | ✓64 | 16388 | 20.6 | 0.795 |

**Table 6:** Results of experiment 1 *(part 2)*

| | BFS | | | | IDDFS | | | | IDDFS+PP | | | | IDDFS+IMO | | | | IDDFS+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M121 (5) | ✓47 | 11340 | 32.3 | 0.351 | ✓47 | 652936 | 16.9 | 38.578 | ✓47 | 29345 | 20.0 | 1.466 | ✓47 | 367330 | 17.4 | 21.100 | ✓47 | 21371 | 18.7 | 1.141 |
| M122 (5) | ✓90 | 147440 | 17.5 | 8.444 | – | 2285519 | 7.6 | – | ✓90 | 1767347 | 8.1 | 217.268 | – | 2295136 | 7.7 | – | ✓90 | 1747889 | 7.7 | 226.929 |
| M123 (5) | ✓101 | 447578 | 14.5 | 30.952 | – | 1938926 | 6.5 | – | – | 2008320 | 6.7 | – | – | 1916955 | 6.4 | – | – | 1911401 | 6.4 | – |
| M124 (3) | ✓39 | 2199 | 28.2 | 0.078 | ✓39 | 40222 | 15.1 | 2.661 | ✓39 | 7819 | 14.6 | 0.535 | ✓39 | 37163 | 14.3 | 2.597 | ✓39 | 7293 | 13.7 | 0.533 |
| M125 (4) | ✓38 | 6578 | 26.3 | 0.250 | ✓38 | 46520 | 14.3 | 3.256 | ✓38 | 46520 | 14.9 | 3.119 | ✓38 | 14234 | 13.2 | 1.078 | ✓38 | 14234 | 13.4 | 1.066 |
| M126 (7) | ✓23 | 137431 | 22.1 | 6.221 | ✓23 | 1504857 | 12.5 | 120.171 | ✓23 | 169283 | 13.6 | 12.434 | ✓23 | 1454611 | 11.6 | 125.226 | ✓23 | 166664 | 12.5 | 13.365 |
| M127 (4) | ✓32 | 1761 | 34.9 | 0.050 | ✓32 | 20140 | 20.1 | 1.003 | ✓32 | 1857 | 23.0 | 0.081 | ✓32 | 26769 | 20.2 | 1.325 | ✓32 | 1853 | 21.1 | 0.088 |
| M128 (4) | ✓19 | 1280 | 38.0 | 0.034 | ✓19 | 9244 | 22.9 | 0.404 | ✓19 | 1408 | 27.7 | 0.051 | ✓19 | 8125 | 22.1 | 0.368 | ✓19 | 1409 | 21.2 | 0.066 |
| M129 (5) | ✓22 | 3055 | 36.5 | 0.084 | ✓22 | 26985 | 20.6 | 1.308 | ✓22 | 3264 | 25.2 | 0.130 | ✓22 | 24286 | 20.7 | 1.174 | ✓22 | 3323 | 23.1 | 0.144 |
| M130 (4) | ✓36 | 6544 | 23.2 | 0.283 | ✓36 | 170915 | 13.1 | 13.075 | ✓36 | 19509 | 14.1 | 1.385 | ✓36 | 174352 | 12.3 | 14.231 | ✓36 | 18783 | 13.0 | 1.448 |
| M131 (4) | ✓31 | 3859 | 26.1 | 0.148 | ✓31 | 84340 | 14.4 | 5.860 | ✓31 | 8733 | 15.8 | 0.552 | ✓31 | 85219 | 13.7 | 6.209 | ✓31 | 8690 | 14.2 | 0.612 |
| M132 (4) | ✓37 | 1596 | 32.1 | 0.050 | ✓37 | 34421 | 19.2 | 1.795 | ✓37 | 2919 | 19.7 | 0.148 | ✓37 | 33570 | 18.2 | 1.847 | ✓37 | 2802 | 14.9 | 0.188 |
| M133 (5) | ✓39 | 24133 | 19.2 | 1.259 | ✓39 | 395237 | 10.5 | 37.602 | ✓39 | 194016 | 11.0 | 17.643 | ✓39 | 395084 | 9.9 | 39.833 | ✓39 | 194143 | 10.2 | 19.075 |
| M134 (4) | ✓76 | 29065 | 14.9 | 1.957 | – | 2334974 | 7.8 | – | ✓76 | 323385 | 8.2 | 39.388 | – | 2185009 | 7.3 | – | ✓76 | 293720 | 7.6 | 38.636 |
| M135 (4) | ✓36 | 2949 | 22.8 | 0.130 | ✓36 | 56221 | 12.5 | 4.492 | ✓36 | 17390 | 13.0 | 1.335 | ✓36 | 55048 | 12.2 | 4.505 | ✓36 | 16968 | 12.1 | 1.405 |
| M136 (4) | ✓25 | 1903 | 23.9 | 0.080 | ✓25 | 17745 | 14.1 | 1.259 | ✓25 | 8766 | 14.4 | 0.609 | ✓25 | 18061 | 12.8 | 1.411 | ✓25 | 8903 | 13.5 | 0.660 |
| M137 (4) | ✓46 | 19753 | 15.5 | 1.271 | ✓46 | 1061641 | 8.0 | 132.029 | ✓46 | 76056 | 8.8 | 8.640 | ✓46 | 1074538 | 7.8 | 137.204 | ✓46 | 77069 | 8.3 | 9.329 |
| M138 (5) | ✓54 | 84769 | 16.0 | 5.307 | – | 2396382 | 8.0 | – | ✓54 | 382205 | 8.6 | 44.319 | – | 2307280 | 7.7 | – | ✓54 | 384666 | 7.9 | 48.747 |
| M139 (6) | ✓106 | 2263798 | 15.1 | 149.494 | – | 2360259 | 7.9 | – | – | 2442854 | 8.1 | – | – | 2278839 | 7.6 | – | – | 2285731 | 7.6 | – |
| M140 (4) | ✓80 | 20706 | 16.8 | 1.230 | ✓80 | 2116445 | 8.7 | 243.683 | ✓80 | 207027 | 9.2 | 22.550 | ✓80 | 2101278 | 8.4 | 251.319 | ✓80 | 200482 | 8.5 | 23.513 |
| M141 (6) | ✓52 | 17865 | 41.0 | 0.436 | ✓52 | 857228 | 21.1 | 40.694 | ✓52 | 18974 | 25.6 | 0.741 | ✓52 | 999642 | 19.5 | 51.315 | ✓52 | 19301 | 23.2 | 0.832 |
| M142 (4) | ✓20 | 2287 | 24.5 | 0.093 | ✓20 | 45344 | 15.5 | 2.924 | ✓20 | 2323 | 17.3 | 0.134 | ✓20 | 44310 | 14.5 | 3.065 | ✓20 | 2326 | 14.9 | 0.156 |
| M143 (6) | ✓65 | 111182 | 24.0 | 4.623 | – | 3836075 | 12.8 | – | ✓65 | 121688 | 14.2 | 8.570 | – | 3624073 | 12.1 | – | ✓65 | 123078 | 13.2 | 9.351 |
| M144 (16) | – | 2644328 | 8.8 | – | – | 1674304 | 5.5 | – | – | 1681647 | 5.5 | – | – | 2243823 | 7.4 | – | – | 2257824 | 7.5 | – |
| M145 (12) | – | 2138993 | 7.1 | – | – | 875699 | 2.9 | – | – | 876024 | 2.9 | – | – | 731175 | 2.4 | – | – | 731382 | 2.4 | – |
| M146 (12) | – | 1144297 | 3.8 | – | – | 495639 | 1.6 | – | – | 495809 | 1.6 | – | – | 495157 | 1.6 | – | – | 495440 | 1.6 | – |
| M147 (3) | ✓50 | 2557 | 26.7 | 0.096 | ✓50 | 134060 | 14.2 | 9.419 | ✓50 | 12186 | 15.2 | 0.800 | ✓50 | 130209 | 13.8 | 9.436 | ✓50 | 12713 | 14.0 | 0.906 |
| M148 (4) | ✓49 | 3031 | 22.5 | 0.135 | ✓49 | 122690 | 12.1 | 10.168 | ✓49 | 22916 | 12.5 | 1.840 | ✓49 | 111095 | 12.1 | 9.184 | ✓49 | 21109 | 12.0 | 1.757 |
| M149 (4) | ✓35 | 962 | 44.0 | 0.022 | ✓35 | 5497 | 26.3 | 0.209 | ✓35 | 965 | 29.6 | 0.033 | ✓35 | 7255 | 26.0 | 0.279 | ✓35 | 1158 | 27.2 | 0.043 |
| M150 (5) | ✓43 | 153953 | 16.8 | 9.173 | – | 2646043 | 8.8 | – | ✓43 | 1128222 | 9.2 | 122.737 | – | 2626001 | 8.8 | – | ✓43 | 1104320 | 8.8 | 125.072 |
| M151 (4) | ✓50 | 15657 | 18.3 | 0.857 | ✓50 | 886327 | 9.4 | 93.908 | ✓50 | 61307 | 10.3 | 5.955 | ✓50 | 700155 | 9.6 | 73.225 | ✓50 | 57221 | 10.0 | 5.748 |
| M152 (4) | ✓35 | 4685 | 11.7 | 0.400 | ✓35 | 176120 | 7.0 | 25.086 | ✓35 | 8199 | 8.1 | 1.018 | ✓35 | 166122 | 7.2 | 23.165 | ✓35 | 8228 | 7.9 | 1.045 |
| M153 (10) | – | 5302141 | 17.7 | – | – | 2605291 | 8.7 | – | – | 2695954 | 9.0 | – | – | 1998455 | 6.7 | – | – | 2010842 | 6.7 | – |
| M154 (1) | ✓2 | 4 | 11.4 | 0.000 | ✓2 | 5 | 6.1 | 0.001 | ✓2 | 5 | 4.7 | 0.001 | ✓2 | 5 | 6.0 | 0.001 | ✓2 | 5 | 6.0 | 0.001 |
| M155 (11) | ✓175 | 199 | 20.5 | 0.010 | ✓175 | 190 | 13.0 | 0.015 | ✓175 | 190 | 11.3 | 0.017 | ✓175 | 190 | 12.8 | 0.015 | ✓175 | 190 | 11.5 | 0.017 |
| X1 (6) | ✓97 | 997833 | 14.6 | 68.534 | – | 1905752 | 6.4 | – | – | 1988150 | 6.6 | – | – | 1984535 | 6.6 | – | – | 1983284 | 6.6 | – |
| X3 (11) | – | 3852051 | 12.8 | – | – | 2337432 | 7.8 | – | – | 2460857 | 8.2 | – | – | 2013045 | 6.7 | – | – | 2018672 | 6.7 | – |
| X78 (8) | – | 2435883 | 8.1 | – | – | 1179439 | 3.9 | – | – | 1222628 | 4.1 | – | – | 1152209 | 3.8 | – | – | 1154789 | 3.8 | – |

**Table 7:** Results of experiment 1 *(part 3)*

| | RBFS | | | | BBFS | | | | RIDDFS+PP+IMO | | | | IDDFS/RBFS+PP+IMO | | | | BFS/RIDDFS+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M1 (2) | ✓8 | 37 | 28.2 | 0.001 | ✓8 | 19 | 14.8 | 0.001 | ✓8 | 39 | 0.5 | 0.085 | ✓8 | 42 | 54.6 | 0.001 | ✓8 | 32 | 50.9 | 0.001 |
| M2 (3) | ✓3 | 4 | 20.2 | 0.000 | ✓3 | 3 | 12.7 | 0.000 | ✓3 | 6 | 15.2 | 0.000 | ✓3 | 9 | 24.7 | 0.000 | ✓3 | 11 | 29.8 | 0.000 |
| M3 (2) | ✓13 | 32 | 22.1 | 0.001 | ✓13 | 32 | 25.0 | 0.001 | ✓13 | 34 | 38.8 | 0.001 | 15 | 48 | 65.4 | 0.001 | ✓13 | 50 | 53.5 | 0.001 |
| M4 (3) | ✓7 | 16 | 10.5 | 0.002 | ✓7 | 14 | 9.0 | 0.002 | ✓7 | 22 | 16.7 | 0.001 | ✓7 | 47 | 44.7 | 0.001 | ✓7 | 23 | 3.7 | 0.006 |
| M5 (4) | ✓6 | 85 | 6.7 | 0.013 | ✓6 | 9 | 6.2 | 0.001 | ✓6 | 146 | 8.6 | 0.017 | ✓6 | 21 | 30.4 | 0.001 | ✓6 | 35 | 19.1 | 0.002 |
| M6 (3) | ✓29 | 377 | 20.2 | 0.019 | ✓29 | 237 | 17.5 | 0.014 | ✓29 | 1443 | 25.5 | 0.057 | 31 | 291 | 38.1 | 0.008 | 31 | 254 | 47.2 | 0.005 |
| M7 (6) | ✓6 | 172 | 10.8 | 0.016 | ✓6 | 35 | 6.4 | 0.005 | ✓6 | 27 | 8.5 | 0.003 | ✓6 | 48 | 14.5 | 0.003 | ✓6 | 17 | 5.0 | 0.003 |
| M8 (2) | ✓32 | 113 | 35.0 | 0.003 | ✓32 | 121 | 34.2 | 0.004 | ✓32 | 130 | 58.1 | 0.002 | 34 | 225 | 81.6 | 0.003 | 34 | 257 | 77.1 | 0.003 |
| M9 (2) | ✓10 | 31 | 78.9 | 0.000 | ✓10 | 18 | 23.5 | 0.001 | ✓10 | 36 | 87.1 | 0.000 | ✓10 | 54 | 117.2 | 0.000 | ✓10 | 35 | 7.5 | 0.005 |
| M10 (3) | ✓21 | 125 | 42.2 | 0.003 | ✓21 | 115 | 27.2 | 0.004 | ✓21 | 236 | 52.2 | 0.005 | ✓21 | 261 | 59.9 | 0.004 | ✓21 | 188 | 65.0 | 0.003 |
| M11 (2) | ✓16 | 84 | 35.1 | 0.002 | ✓16 | 53 | 24.9 | 0.002 | ✓16 | 156 | 39.7 | 0.004 | 18 | 81 | 46.0 | 0.002 | ✓16 | 110 | 59.3 | 0.002 |
| M12 (2) | ✓11 | 31 | 34.4 | 0.001 | ✓11 | 17 | 43.6 | 0.000 | ✓11 | 51 | 56.6 | 0.001 | ✓11 | 30 | 84.8 | 0.000 | ✓11 | 35 | 86.9 | 0.000 |
| M13 (3) | ✓21 | 334 | 43.1 | 0.008 | ✓21 | 154 | 26.2 | 0.006 | ✓21 | 618 | 51.0 | 0.012 | ✓21 | 279 | 59.3 | 0.005 | ✓21 | 284 | 66.0 | 0.004 |
| M14 (2) | ✓10 | 15 | 38.0 | 0.000 | ✓10 | 13 | 31.3 | 0.000 | ✓10 | 16 | 44.7 | 0.000 | ✓10 | 36 | 73.8 | 0.000 | ✓10 | 43 | 60.1 | 0.001 |
| M15 (2) | ✓12 | 42 | 71.8 | 0.001 | ✓12 | 25 | 41.6 | 0.001 | ✓12 | 42 | 97.5 | 0.000 | ✓12 | 69 | 109.2 | 0.001 | ✓12 | 62 | 84.8 | 0.001 |
| M16 (3) | ✓39 | 688 | 24.4 | 0.028 | ✓39 | 607 | 17.7 | 0.034 | ✓39 | 1607 | 26.4 | 0.061 | 45 | 672 | 32.8 | 0.021 | 41 | 242 | 38.9 | 0.006 |
| M17 (3) | ✓9 | 70 | 28.8 | 0.002 | ✓9 | 23 | 18.9 | 0.001 | ✓9 | 71 | 45.8 | 0.002 | ✓9 | 60 | 70.7 | 0.001 | ✓9 | 62 | 63.1 | 0.001 |
| M18 (2) | ✓13 | 48 | 24.9 | 0.002 | ✓13 | 36 | 17.8 | 0.002 | ✓13 | 91 | 40.3 | 0.002 | ✓13 | 66 | 52.7 | 0.001 | ✓13 | 56 | 52.2 | 0.001 |
| M19 (2) | ✓20 | 52 | 49.7 | 0.001 | ✓20 | 64 | 44.2 | 0.001 | ✓20 | 71 | 66.1 | 0.001 | ✓20 | 120 | 96.4 | 0.001 | ✓20 | 89 | 86.7 | 0.001 |
| M20 (2) | ✓16 | 44 | 38.5 | 0.001 | ✓16 | 29 | 24.6 | 0.001 | ✓16 | 49 | 54.3 | 0.001 | ✓16 | 63 | 78.0 | 0.001 | ✓16 | 71 | 76.8 | 0.001 |
| M21 (2) | ✓5 | 6 | 19.7 | 0.000 | ✓5 | 6 | 27.4 | 0.000 | ✓5 | 7 | 40.9 | 0.000 | ✓5 | 15 | 83.8 | 0.000 | ✓5 | 17 | 75.5 | 0.000 |
| M22 (2) | ✓15 | 54 | 27.6 | 0.002 | ✓15 | 35 | 16.7 | 0.002 | ✓15 | 70 | 35.2 | 0.002 | ✓15 | 54 | 50.7 | 0.001 | ✓15 | 56 | 48.9 | 0.001 |
| M23 (2) | ✓10 | 40 | 35.2 | 0.001 | ✓10 | 27 | 34.0 | 0.001 | ✓10 | 49 | 57.1 | 0.001 | ✓10 | 63 | 69.8 | 0.001 | ✓10 | 59 | 62.7 | 0.001 |
| M24 (2) | ✓9 | 20 | 43.4 | 0.000 | ✓9 | 22 | 42.4 | 0.001 | ✓9 | 27 | 74.8 | 0.000 | ✓9 | 39 | 83.9 | 0.000 | ✓9 | 47 | 93.4 | 0.001 |
| M25 (3) | ✓7 | 64 | 33.7 | 0.002 | ✓7 | 24 | 31.2 | 0.001 | ✓7 | 90 | 52.9 | 0.002 | ✓7 | 36 | 68.1 | 0.001 | ✓7 | 53 | 64.4 | 0.001 |
| M26 (3) | ✓10 | 50 | 49.0 | 0.001 | ✓10 | 42 | 38.9 | 0.001 | ✓10 | 58 | 79.1 | 0.001 | ✓10 | 108 | 88.5 | 0.001 | ✓10 | 152 | 97.9 | 0.002 |
| M27 (2) | ✓10 | 31 | 35.6 | 0.001 | ✓10 | 35 | 32.8 | 0.001 | ✓10 | 39 | 63.8 | 0.001 | ✓10 | 81 | 84.2 | 0.001 | ✓10 | 80 | 78.3 | 0.001 |
| M28 (2) | ✓9 | 16 | 33.0 | 0.000 | ✓9 | 14 | 34.2 | 0.000 | ✓9 | 22 | 51.0 | 0.000 | ✓9 | 39 | 103.4 | 0.000 | ✓9 | 41 | 65.6 | 0.001 |
| M29 (2) | ✓22 | 75 | 21.3 | 0.004 | ✓22 | 48 | 16.1 | 0.003 | ✓22 | 102 | 34.8 | 0.003 | ✓22 | 102 | 48.5 | 0.002 | ✓22 | 128 | 56.0 | 0.002 |
| M30 (3) | ✓5 | 49 | 25.2 | 0.002 | ✓5 | 13 | 18.7 | 0.001 | ✓5 | 85 | 42.1 | 0.002 | ✓5 | 39 | 50.5 | 0.001 | ✓5 | 41 | 49.4 | 0.001 |
| M31 (3) | ✓6 | 35 | 45.8 | 0.001 | ✓6 | 12 | 23.3 | 0.001 | ✓6 | 51 | 49.5 | 0.001 | ✓6 | 27 | 71.7 | 0.000 | ✓6 | 26 | 45.6 | 0.001 |
| M32 (3) | ✓9 | 63 | 53.1 | 0.001 | ✓9 | 46 | 32.1 | 0.001 | ✓9 | 71 | 79.3 | 0.001 | ✓9 | 120 | 98.6 | 0.001 | ✓9 | 115 | 94.8 | 0.001 |
| M33 (3) | ✓10 | 114 | 42.4 | 0.003 | ✓10 | 35 | 21.9 | 0.002 | ✓10 | 73 | 46.2 | 0.002 | ✓10 | 36 | 61.1 | 0.001 | ✓10 | 41 | 45.4 | 0.001 |
| M34 (4) | ✓8 | 78 | 20.2 | 0.004 | ✓8 | 46 | 12.6 | 0.004 | ✓8 | 100 | 26.6 | 0.004 | 10 | 123 | 30.9 | 0.004 | ✓8 | 92 | 34.6 | 0.003 |
| M35 (5) | ✓31 | 6691 | 30.6 | 0.218 | ✓31 | 5041 | 19.5 | 0.259 | ✓31 | 21069 | 32.8 | 0.641 | ✓31 | 6585 | 32.2 | 0.204 | ✓31 | 7352 | 19.8 | 0.372 |
| M36 (5) | ✓59 | 11919 | 38.0 | 0.314 | ✓59 | 9652 | 23.2 | 0.415 | ✓59 | 20891 | 33.0 | 0.634 | ✓59 | 10599 | 35.0 | 0.302 | 79 | 15368 | 38.8 | 0.396 |
| M37 (3) | ✓23 | 249 | 30.1 | 0.008 | ✓23 | 131 | 28.2 | 0.005 | ✓23 | 612 | 42.9 | 0.014 | ✓23 | 237 | 57.6 | 0.004 | ✓23 | 296 | 59.4 | 0.005 |
| M38 (3) | ✓8 | 42 | 24.0 | 0.002 | ✓8 | 22 | 18.4 | 0.001 | ✓8 | 48 | 29.3 | 0.002 | ✓8 | 51 | 40.7 | 0.001 | 10 | 38 | 30.3 | 0.001 |
| M39 (2) | ✓27 | 102 | 44.4 | 0.002 | ✓27 | 97 | 33.5 | 0.003 | ✓27 | 103 | 62.1 | 0.002 | ✓27 | 213 | 82.9 | 0.003 | ✓27 | 233 | 78.5 | 0.003 |
| M40 (3) | ✓7 | 84 | 41.3 | 0.002 | ✓7 | 29 | 28.9 | 0.001 | ✓7 | 88 | 65.1 | 0.001 | ✓7 | 72 | 75.2 | 0.001 | ✓7 | 77 | 71.3 | 0.001 |
| M41 (3) | ✓13 | 60 | 29.0 | 0.002 | ✓13 | 34 | 19.8 | 0.002 | ✓13 | 95 | 43.9 | 0.002 | ✓13 | 60 | 56.4 | 0.001 | ✓13 | 62 | 48.6 | 0.001 |
| M42 (3) | ✓15 | 110 | 43.0 | 0.003 | ✓15 | 133 | 25.7 | 0.005 | ✓15 | 118 | 60.8 | 0.002 | 17 | 249 | 86.7 | 0.003 | 17 | 224 | 71.9 | 0.003 |
| M43 (3) | ✓22 | 324 | 31.7 | 0.010 | ✓22 | 142 | 22.6 | 0.006 | ✓22 | 379 | 39.2 | 0.010 | 24 | 315 | 55.7 | 0.006 | ✓22 | 242 | 49.9 | 0.005 |
| M44 (1) | ✓1 | 1 | 100.1 | 0.000 | ✓1 | 1 | 49.0 | 0.000 | ✓1 | 2 | 128.4 | 0.000 | ✓1 | 2 | 77.8 | 0.000 | ✓1 | 4 | 167.5 | 0.000 |
| M45 (3) | ✓11 | 110 | 47.9 | 0.002 | ✓11 | 47 | 26.1 | 0.002 | ✓11 | 134 | 63.7 | 0.002 | ✓11 | 96 | 68.0 | 0.001 | ✓11 | 71 | 80.8 | 0.001 |
| M46 (2) | ✓8 | 15 | 33.2 | 0.000 | ✓8 | 19 | 33.2 | 0.001 | ✓8 | 16 | 47.3 | 0.000 | ✓8 | 33 | 57.0 | 0.001 | ✓8 | 50 | 64.1 | 0.001 |
| M47 (2) | ✓22 | 102 | 37.5 | 0.003 | ✓22 | 77 | 27.2 | 0.003 | 26 | 139 | 48.5 | 0.003 | ✓22 | 132 | 62.6 | 0.002 | 26 | 182 | 69.3 | 0.003 |
| M48 (3) | ✓14 | 100 | 19.8 | 0.005 | ✓14 | 59 | 17.7 | 0.003 | ✓14 | 129 | 33.8 | 0.004 | ✓14 | 96 | 45.2 | 0.002 | ✓14 | 179 | 49.3 | 0.004 |
| M49 (3) | ✓21 | 101 | 25.5 | 0.004 | ✓21 | 82 | 21.2 | 0.004 | ✓21 | 172 | 41.6 | 0.004 | ✓21 | 141 | 53.2 | 0.003 | ✓21 | 170 | 61.7 | 0.003 |
| M50 (2) | ✓17 | 67 | 23.7 | 0.003 | ✓17 | 38 | 22.9 | 0.002 | ✓17 | 96 | 40.1 | 0.002 | ✓17 | 84 | 51.6 | 0.002 | ✓17 | 104 | 56.9 | 0.002 |
| M51 (2) | ✓8 | 26 | 29.6 | 0.001 | ✓8 | 15 | 22.2 | 0.001 | ✓8 | 28 | 49.2 | 0.001 | ✓8 | 30 | 71.5 | 0.000 | ✓8 | 35 | 61.7 | 0.001 |
| M52 (4) | ✓8 | 227 | 35.3 | 0.006 | ✓8 | 50 | 18.5 | 0.003 | ✓8 | 215 | 43.5 | 0.005 | ✓8 | 150 | 48.8 | 0.003 | ✓8 | 137 | 45.4 | 0.003 |
| M53 (4) | ✓12 | 255 | 39.6 | 0.006 | ✓12 | 86 | 23.2 | 0.004 | ✓12 | 274 | 63.5 | 0.004 | 14 | 117 | 64.7 | 0.002 | ✓12 | 152 | 80.3 | 0.002 |
| M54 (4) | ✓30 | 2046 | 25.6 | 0.080 | ✓30 | 1219 | 17.7 | 0.069 | ✓30 | 6508 | 26.8 | 0.243 | ✓30 | 1254 | 32.6 | 0.038 | 32 | 2684 | 38.3 | 0.070 |
| M55 (2) | ✓27 | 91 | 48.9 | 0.002 | ✓27 | 85 | 37.4 | 0.002 | ✓27 | 120 | 63.2 | 0.002 | ✓27 | 120 | 77.0 | 0.002 | ✓27 | 110 | 83.7 | 0.001 |
| M56 (2) | ✓6 | 12 | 21.0 | 0.001 | ✓6 | 7 | 25.6 | 0.000 | ✓6 | 15 | 46.2 | 0.000 | ✓6 | 21 | 87.4 | 0.000 | ✓6 | 20 | 63.7 | 0.000 |
| M57 (2) | ✓23 | 43 | 52.6 | 0.001 | ✓23 | 50 | 30.8 | 0.002 | ✓23 | 44 | 60.5 | 0.001 | ✓23 | 105 | 103.5 | 0.001 | ✓23 | 221 | 84.5 | 0.003 |
| M58 (3) | ✓11 | 89 | 40.4 | 0.002 | ✓11 | 26 | 24.7 | 0.001 | ✓11 | 101 | 67.6 | 0.001 | ✓11 | 75 | 77.2 | 0.001 | ✓11 | 80 | 83.9 | 0.001 |
| M59 (3) | ✓50 | 1907 | 13.0 | 0.147 | ✓50 | 1314 | 11.0 | 0.120 | ✓50 | 11650 | 15.0 | 0.774 | 54 | 2349 | 21.5 | 0.109 | 52 | 1457 | 24.2 | 0.060 |
| M60 (4) | ✓44 | 2465 | 41.2 | 0.060 | ✓44 | 984 | 28.3 | 0.035 | ✓44 | 4291 | 47.0 | 0.091 | ✓44 | 1164 | 63.2 | 0.018 | ✓44 | 1835 | 56.3 | 0.033 |

**Table 8:** Results of experiment 2 *(part 1)*

| | RBFS | | | | BBFS | | | | RIDDFS+PP+IMO | | | | IDDFS/RBFS+PP+IMO | | | | BFS/RIDDFS+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M61 (4) | ✓21 | 1184 | 29.7 | 0.040 | ✓21 | 302 | 18.2 | 0.017 | ✓21 | 3033 | 22.8 | 0.133 | ✓21 | 1083 | 33.3 | 0.032 | ✓21 | 503 | 23.1 | 0.022 |
| M62 (4) | ✓30 | 551 | 39.0 | 0.014 | ✓30 | 269 | 28.5 | 0.009 | ✓30 | 589 | 50.5 | 0.012 | ✓30 | 552 | 64.3 | 0.009 | ✓30 | 404 | 65.7 | 0.006 |
| M63 (2) | ✓50 | 262 | 39.9 | 0.007 | ✓50 | 170 | 28.3 | 0.006 | ✓50 | 260 | 67.0 | 0.004 | ✓50 | 570 | 86.3 | 0.007 | ✓50 | 425 | 89.7 | 0.005 |
| M64 (4) | ✓30 | 274 | 33.4 | 0.008 | ✓30 | 390 | 33.9 | 0.012 | ✓30 | 384 | 49.4 | 0.008 | ✓30 | 675 | 61.8 | 0.011 | 36 | 854 | 61.5 | 0.014 |
| M65 (4) | ✓41 | 3851 | 26.3 | 0.147 | ✓41 | 2689 | 16.9 | 0.159 | ✓41 | 26367 | 25.8 | 1.021 | ✓41 | 4389 | 26.0 | 0.169 | 45 | 2066 | 27.7 | 0.075 |
| M66 (3) | ✓15 | 770 | 15.9 | 0.048 | ✓15 | 224 | 9.2 | 0.024 | ✓15 | 953 | 21.0 | 0.045 | ✓15 | 486 | 25.5 | 0.019 | ✓15 | 422 | 28.6 | 0.015 |
| M67 (3) | ✓8 | 55 | 35.5 | 0.002 | ✓8 | 30 | 14.6 | 0.002 | ✓8 | 64 | 54.5 | 0.001 | ✓8 | 48 | 64.5 | 0.001 | ✓8 | 65 | 62.8 | 0.001 |
| M68 (3) | ✓28 | 252 | 25.7 | 0.010 | ✓28 | 169 | 15.9 | 0.011 | ✓28 | 376 | 54.5 | 0.007 | ✓28 | 252 | 59.6 | 0.004 | 30 | 608 | 62.2 | 0.010 |
| M69 (3) | ✓37 | 1031 | 20.5 | 0.050 | ✓37 | 739 | 16.7 | 0.044 | ✓37 | 2650 | 24.7 | 0.107 | ✓37 | 1341 | 35.0 | 0.038 | ✓37 | 1739 | 37.1 | 0.047 |
| M70 (4) | ✓26 | 1887 | 32.3 | 0.058 | ✓26 | 762 | 24.5 | 0.031 | ✓26 | 5138 | 41.0 | 0.125 | 34 | 3405 | 31.6 | 0.108 | ✓26 | 1322 | 50.7 | 0.026 |
| M71 (2) | ✓21 | 92 | 21.6 | 0.004 | ✓21 | 69 | 17.1 | 0.004 | ✓21 | 252 | 27.9 | 0.009 | ✓21 | 57 | 32.0 | 0.002 | ✓21 | 140 | 42.8 | 0.003 |
| M72 (3) | ✓40 | 996 | 35.6 | 0.028 | ✓40 | 908 | 28.5 | 0.032 | ✓40 | 1058 | 43.0 | 0.025 | ✓40 | 1362 | 57.0 | 0.024 | ✓40 | 962 | 62.3 | 0.015 |
| M73 (3) | ✓25 | 439 | 31.5 | 0.014 | ✓25 | 294 | 21.7 | 0.014 | ✓25 | 1333 | 19.0 | 0.070 | ✓25 | 246 | 41.1 | 0.006 | ✓25 | 632 | 44.7 | 0.014 |
| M74 (4) | ✓34 | 1880 | 36.9 | 0.051 | ✓34 | 986 | 21.3 | 0.046 | ✓34 | 8041 | 37.7 | 0.213 | ✓34 | 1308 | 42.7 | 0.031 | 36 | 2144 | 41.7 | 0.051 |
| M75 (4) | ✓34 | 1166 | 29.0 | 0.040 | ✓34 | 667 | 22.7 | 0.029 | ✓34 | 3746 | 33.4 | 0.112 | ✓34 | 564 | 37.8 | 0.015 | ✓34 | 1328 | 44.8 | 0.030 |
| M76 (3) | ✓56 | 889 | 19.3 | 0.046 | ✓56 | 848 | 18.9 | 0.045 | ✓56 | 5963 | 20.4 | 0.292 | 58 | 1047 | 28.9 | 0.036 | 68 | 1469 | 24.0 | 0.061 |
| M77 (4) | ✓55 | 2067 | 42.8 | 0.048 | ✓55 | 1308 | 22.6 | 0.058 | ✓55 | 10403 | 35.1 | 0.296 | 61 | 2121 | 41.0 | 0.052 | 63 | 2177 | 39.3 | 0.055 |
| M78 (5) | ✓33 | 10758 | 20.2 | 0.531 | ✓33 | 5708 | 12.3 | 0.465 | ✓33 | 46981 | 20.4 | 2.303 | ✓33 | 7359 | 18.3 | 0.401 | ✓33 | 15419 | 14.0 | 1.103 |
| M79 (3) | ✓18 | 237 | 34.5 | 0.007 | ✓18 | 126 | 27.9 | 0.005 | ✓18 | 245 | 44.6 | 0.005 | ✓18 | 204 | 57.9 | 0.004 | ✓18 | 200 | 67.8 | 0.003 |
| M80 (4) | ✓38 | 977 | 29.2 | 0.033 | ✓38 | 622 | 21.7 | 0.029 | ✓38 | 2213 | 31.7 | 0.070 | ✓38 | 1089 | 44.2 | 0.025 | ✓38 | 1445 | 36.5 | 0.040 |
| M81 (3) | ✓12 | 113 | 34.1 | 0.003 | ✓12 | 57 | 27.3 | 0.002 | ✓12 | 116 | 51.4 | 0.002 | 14 | 120 | 73.6 | 0.002 | ✓12 | 74 | 67.5 | 0.001 |
| M82 (3) | ✓14 | 119 | 51.4 | 0.002 | ✓14 | 29 | 33.1 | 0.001 | ✓14 | 138 | 66.9 | 0.002 | ✓14 | 78 | 70.6 | 0.001 | ✓14 | 56 | 81.3 | 0.001 |
| M83 (4) | ✓47 | 4207 | 25.2 | 0.167 | ✓47 | 2830 | 16.6 | 0.171 | ✓47 | 7388 | 28.7 | 0.258 | 75 | 6174 | 32.5 | 0.190 | ✓47 | 3578 | 33.6 | 0.107 |
| M84 (3) | ✓68 | 1991 | 23.6 | 0.084 | ✓68 | 1739 | 19.8 | 0.088 | ✓68 | 16332 | 26.2 | 0.624 | 74 | 2265 | 34.1 | 0.066 | 80 | 4340 | 34.2 | 0.127 |
| M85 (3) | ✓51 | 2804 | 17.2 | 0.163 | ✓51 | 2434 | 15.3 | 0.159 | ✓51 | 22727 | 19.1 | 1.191 | 53 | 3684 | 20.2 | 0.182 | 55 | 3701 | 25.4 | 0.146 |
| M86 (4) | ✓25 | 1181 | 31.7 | 0.037 | ✓25 | 709 | 19.1 | 0.037 | ✓25 | 3740 | 32.8 | 0.114 | ✓25 | 1377 | 39.2 | 0.035 | ✓25 | 1508 | 39.8 | 0.038 |
| M87 (4) | ✓53 | 3466 | 26.9 | 0.129 | ✓53 | 1857 | 17.7 | 0.105 | ✓53 | 21262 | 29.1 | 0.731 | ✓53 | 3711 | 31.1 | 0.119 | 59 | 3686 | 29.3 | 0.126 |
| M88 (3) | ✓63 | 869 | 19.0 | 0.046 | ✓63 | 830 | 13.8 | 0.060 | ✓63 | 4339 | 18.9 | 0.230 | 67 | 1026 | 32.0 | 0.032 | 67 | 647 | 27.2 | 0.024 |
| M89 (4) | ✓35 | 1714 | 28.5 | 0.060 | ✓35 | 1807 | 21.3 | 0.085 | ✓35 | 8444 | 30.4 | 0.277 | 39 | 3300 | 40.2 | 0.082 | ✓35 | 6350 | 29.9 | 0.212 |
| M90 (4) | ✓16 | 2160 | 25.4 | 0.085 | ✓16 | 562 | 19.2 | 0.029 | ✓16 | 2411 | 36.2 | 0.067 | ✓16 | 1041 | 45.1 | 0.023 | ✓16 | 1037 | 40.3 | 0.026 |
| M91 (4) | ✓14 | 129 | 41.5 | 0.003 | ✓14 | 124 | 34.3 | 0.004 | ✓14 | 171 | 67.4 | 0.003 | ✓14 | 201 | 90.9 | 0.002 | ✓14 | 200 | 84.2 | 0.002 |
| M92 (3) | ✓48 | 1318 | 30.0 | 0.044 | ✓48 | 909 | 24.1 | 0.038 | ✓48 | 1723 | 38.5 | 0.045 | ✓48 | 1221 | 56.6 | 0.022 | ✓48 | 1838 | 60.4 | 0.030 |
| M93 (8) | − | 2706616 | 9.0 | − | ✓34 | 872784 | 6.0 | 144.853 | − | 2027953 | 6.7 | − | − | 217820 | 0.7 | − | − | 267073 | 0.9 | − |
| M94 (3) | ✓29 | 281 | 46.3 | 0.006 | ✓29 | 164 | 33.4 | 0.005 | ✓29 | 289 | 77.4 | 0.004 | ✓29 | 711 | 103.2 | 0.007 | ✓29 | 227 | 103.3 | 0.002 |
| M95 (8) | ✓8 | 5848 | 7.0 | 0.839 | ✓8 | 262 | 4.6 | 0.058 | ✓8 | 4476 | 9.9 | 0.450 | ✓8 | 441 | 11.7 | 0.038 | ✓8 | 353 | 13.2 | 0.027 |
| M96 (3) | ✓37 | 776 | 29.5 | 0.026 | ✓37 | 451 | 25.1 | 0.018 | ✓37 | 779 | 46.4 | 0.017 | ✓37 | 924 | 60.8 | 0.015 | 39 | 914 | 60.4 | 0.015 |
| M97 (5) | ✓41 | 14097 | 30.4 | 0.464 | ✓41 | 4097 | 16.6 | 0.247 | ✓41 | 49560 | 29.2 | 1.698 | ✓41 | 8724 | 29.0 | 0.301 | 45 | 10319 | 18.0 | 0.573 |
| M98 (5) | ✓110 | 52224 | 22.1 | 2.362 | ✓110 | 39298 | 16.3 | 2.414 | ✓110 | 56176 | 22.6 | 2.486 | 112 | 73608 | 7.8 | 9.389 | 114 | 46298 | 21.4 | 2.167 |
| M99 (4) | ✓131 | 80393 | 15.4 | 5.237 | ✓131 | 75113 | 11.9 | 6.310 | ✓131 | 104666 | 16.3 | 6.411 | 149 | 146022 | 4.3 | 33.789 | ✓131 | 238805 | 3.0 | 78.799 |
| M100 (4) | ✓52 | 1100 | 48.1 | 0.023 | ✓52 | 1142 | 24.4 | 0.047 | ✓52 | 3345 | 45.3 | 0.074 | 62 | 2988 | 59.4 | 0.050 | ✓52 | 2585 | 49.2 | 0.053 |
| M101 (5) | ✓15 | 1150 | 11.9 | 0.097 | ✓15 | 777 | 8.7 | 0.089 | ✓15 | 1614 | 14.8 | 0.109 | 17 | 1791 | 20.0 | 0.090 | 19 | 1373 | 20.4 | 0.067 |
| M102 (4) | ✓44 | 1565 | 17.1 | 0.091 | ✓44 | 1732 | 14.1 | 0.123 | ✓44 | 9206 | 21.5 | 0.428 | ✓44 | 3066 | 27.5 | 0.112 | ✓44 | 6368 | 26.1 | 0.244 |
| M103 (4) | ✓12 | 479 | 26.1 | 0.018 | ✓12 | 276 | 17.7 | 0.016 | ✓12 | 561 | 45.5 | 0.012 | ✓12 | 543 | 53.0 | 0.010 | ✓12 | 341 | 50.6 | 0.007 |
| M104 (3) | ✓27 | 560 | 29.7 | 0.019 | ✓27 | 243 | 22.2 | 0.011 | ✓27 | 751 | 38.1 | 0.020 | ✓27 | 627 | 45.1 | 0.014 | ✓27 | 338 | 51.0 | 0.007 |
| M105 (8) | ✓24 | 74795 | 27.1 | 2.758 | ✓24 | 22285 | 14.5 | 1.540 | 26 | 73010 | 34.7 | 2.104 | 36 | 115356 | 4.2 | 27.685 | 38 | 65465 | 11.1 | 5.872 |
| M106 (5) | ✓50 | 8283 | 25.8 | 0.321 | ✓50 | 6050 | 18.7 | 0.323 | ✓50 | 83554 | 27.9 | 2.993 | 56 | 14472 | 25.1 | 0.576 | ✓50 | 20216 | 16.3 | 1.242 |
| M107 (11) | ✓10 | 480 | 21.6 | 0.022 | ✓10 | 264 | 16.5 | 0.016 | ✓10 | 577 | 26.1 | 0.022 | ✓10 | 537 | 38.5 | 0.014 | ✓10 | 527 | 34.4 | 0.015 |
| M108 (4) | ✓68 | 18144 | 16.1 | 1.124 | ✓68 | 13752 | 12.4 | 1.112 | ✓68 | 124699 | 16.2 | 7.689 | ✓68 | 27483 | 13.6 | 2.022 | 70 | 25877 | 15.5 | 1.665 |
| M109 (5) | ✓42 | 22175 | 22.6 | 0.979 | ✓42 | 14182 | 14.7 | 0.965 | ✓42 | 29427 | 28.1 | 1.048 | ✓42 | 66524 | 19.3 | 3.450 | 46 | 22262 | 22.8 | 0.975 |
| M110 (4) | ✓14 | 965 | 52.7 | 0.018 | ✓14 | 377 | 21.2 | 0.018 | ✓14 | 950 | 72.8 | 0.013 | ✓14 | 717 | 71.8 | 0.010 | 16 | 656 | 74.9 | 0.009 |
| M111 (6) | ✓61 | 167347 | 13.9 | 12.034 | ✓61 | 125722 | 10.6 | 11.874 | ✓61 | 649790 | 14.2 | 45.767 | 63 | 184290 | 4.3 | 42.627 | 69 | 150620 | 7.9 | 19.075 |
| M112 (5) | ✓94 | 93008 | 15.5 | 6.007 | ✓94 | 76908 | 11.5 | 6.705 | ✓94 | 536460 | 13.9 | 38.707 | ✓94 | 229029 | 8.2 | 27.929 | ✓94 | 154883 | 6.6 | 23.487 |
| M113 (4) | ✓51 | 16674 | 15.0 | 1.110 | ✓51 | 11514 | 10.4 | 1.105 | ✓51 | 49392 | 17.8 | 2.775 | ✓51 | 15471 | 18.8 | 0.823 | 53 | 20474 | 19.8 | 1.037 |
| M114 (6) | ✓60 | 41191 | 38.0 | 1.084 | ✓60 | 35657 | 20.4 | 1.747 | ✓60 | 406120 | 33.8 | 12.002 | ✓60 | 89862 | 5.0 | 17.890 | ✓60 | 105632 | 8.6 | 12.329 |
| M115 (5) | ✓29 | 18779 | 25.0 | 0.750 | ✓29 | 2812 | 15.3 | 0.184 | ✓29 | 83925 | 27.2 | 3.080 | ✓29 | 6405 | 24.4 | 0.262 | ✓29 | 5777 | 26.9 | 0.215 |
| M116 (5) | ✓14 | 1185 | 43.1 | 0.027 | ✓14 | 490 | 21.1 | 0.023 | ✓14 | 1543 | 53.2 | 0.029 | ✓14 | 987 | 50.3 | 0.020 | ✓14 | 1487 | 54.2 | 0.027 |
| M117 (5) | ✓47 | 94431 | 15.8 | 5.983 | ✓47 | 40084 | 10.5 | 3.810 | ✓47 | 317224 | 17.4 | 18.190 | 63 | 89055 | 7.8 | 11.373 | 55 | 97391 | 9.5 | 10.293 |
| M118 (4) | ✓44 | 8722 | 19.0 | 0.460 | ✓44 | 5002 | 16.9 | 0.297 | ✓44 | 50448 | 23.3 | 2.163 | ✓44 | 10455 | 21.3 | 0.491 | 48 | 10940 | 24.9 | 0.440 |
| M119 (3) | ✓18 | 218 | 16.1 | 0.014 | ✓18 | 121 | 15.4 | 0.008 | ✓18 | 585 | 22.1 | 0.026 | ✓18 | 240 | 32.9 | 0.007 | ✓18 | 260 | 34.1 | 0.008 |
| M120 (4) | ✓64 | 2769 | 29.7 | 0.093 | ✓64 | 2080 | 28.9 | 0.072 | ✓64 | 18077 | 36.3 | 0.498 | 66 | 2829 | 47.7 | 0.059 | 78 | 2900 | 43.3 | 0.067 |

**Table 9:** Results of experiment 2 *(part 2)*

| | RBFS | | | | BBFS | | | | RIDDFS+PP+IMO | | | | IDDFS/RBFS+PP+IMO | | | | BFS/RIDDFS+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M121 (5) | ✓47 | 2087 | 35.4 | 0.059 | ✓47 | 2236 | 24.8 | 0.090 | ✓47 | 2345 | 38.7 | 0.061 | ✓47 | 6258 | 54.8 | 0.114 | ✓47 | 6833 | 44.9 | 0.152 |
| M122 (5) | ✓90 | 43023 | 16.5 | 2.608 | ✓90 | 41434 | 12.6 | 3.286 | ✓90 | 305844 | 16.9 | 18.124 | 94 | 124287 | 9.7 | 12.856 | 98 | 77117 | 16.2 | 4.773 |
| M123 (5) | ✓101 | 353856 | 13.2 | 26.825 | ✓101 | 312742 | 10.4 | 30.010 | − | 3511938 | 11.7 | − | − | 267989 | 0.9 | − | − | 363313 | 1.2 | − |
| M124 (3) | ✓39 | 580 | 19.0 | 0.031 | ✓39 | 496 | 15.6 | 0.032 | ✓39 | 3026 | 22.3 | 0.135 | 41 | 1569 | 38.2 | 0.041 | 55 | 2468 | 40.3 | 0.061 |
| M125 (4) | ✓38 | 2825 | 29.1 | 0.097 | ✓38 | 1301 | 21.7 | 0.060 | ✓38 | 14877 | 31.1 | 0.478 | ✓38 | 1521 | 36.8 | 0.041 | 40 | 3527 | 33.2 | 0.106 |
| M126 (7) | ✓23 | 119036 | 28.4 | 4.192 | ✓23 | 21033 | 15.1 | 1.393 | ✓23 | 143666 | 33.5 | 4.290 | 27 | 35538 | 12.3 | 2.882 | 25 | 31550 | 16.8 | 1.882 |
| M127 (4) | ✓32 | 1993 | 39.7 | 0.050 | ✓32 | 1080 | 28.0 | 0.039 | ✓32 | 2045 | 49.8 | 0.041 | ✓32 | 2514 | 56.7 | 0.044 | 38 | 1559 | 65.8 | 0.024 |
| M128 (4) | ✓19 | 915 | 37.5 | 0.024 | ✓19 | 394 | 30.3 | 0.013 | 21 | 1062 | 62.9 | 0.017 | ✓19 | 1581 | 64.8 | 0.024 | 23 | 887 | 66.1 | 0.013 |
| M129 (5) | ✓22 | 1667 | 45.5 | 0.037 | ✓22 | 795 | 21.1 | 0.038 | ✓22 | 1707 | 58.7 | 0.029 | ✓22 | 2562 | 57.9 | 0.044 | ✓22 | 1091 | 56.5 | 0.019 |
| M130 (4) | ✓36 | 5146 | 20.0 | 0.258 | ✓36 | 2441 | 17.0 | 0.143 | ✓36 | 13801 | 23.2 | 0.594 | 40 | 6537 | 35.5 | 0.184 | 38 | 3941 | 32.8 | 0.120 |
| M131 (4) | ✓31 | 3778 | 25.6 | 0.148 | ✓31 | 2306 | 18.2 | 0.126 | ✓31 | 11030 | 30.8 | 0.358 | ✓31 | 5139 | 39.5 | 0.130 | 33 | 3563 | 34.9 | 0.102 |
| M132 (4) | ✓37 | 2031 | 39.2 | 0.052 | ✓37 | 1413 | 26.4 | 0.054 | ✓37 | 4731 | 43.6 | 0.109 | 39 | 3162 | 54.9 | 0.058 | 47 | 3533 | 57.2 | 0.062 |
| M133 (5) | ✓39 | 8454 | 27.9 | 0.303 | ✓39 | 5954 | 15.9 | 0.375 | ✓39 | 52127 | 29.8 | 1.752 | 41 | 10395 | 20.1 | 0.516 | 41 | 9287 | 18.5 | 0.503 |
| M134 (4) | ✓76 | 26459 | 15.9 | 1.664 | ✓76 | 21488 | 11.0 | 1.949 | ✓76 | 312910 | 14.9 | 20.972 | 82 | 50733 | 12.9 | 3.918 | 102 | 47342 | 6.1 | 7.819 |
| M135 (4) | ✓36 | 3132 | 30.1 | 0.104 | ✓36 | 2110 | 17.2 | 0.122 | ✓36 | 14749 | 34.4 | 0.428 | ✓36 | 5409 | 29.6 | 0.183 | 38 | 4469 | 36.8 | 0.121 |
| M136 (4) | ✓25 | 1095 | 24.4 | 0.045 | ✓25 | 713 | 20.8 | 0.034 | ✓25 | 3804 | 36.2 | 0.105 | ✓25 | 1509 | 41.6 | 0.036 | 27 | 1070 | 45.7 | 0.023 |
| M137 (4) | ✓46 | 13091 | 14.3 | 0.918 | ✓46 | 7627 | 10.9 | 0.697 | ✓46 | 45446 | 17.0 | 2.675 | 52 | 13890 | 18.8 | 0.737 | 54 | 16895 | 20.8 | 0.814 |
| M138 (5) | ✓54 | 43973 | 14.5 | 3.026 | ✓54 | 28668 | 11.5 | 2.489 | ✓54 | 128030 | 18.4 | 6.945 | 56 | 43851 | 12.1 | 3.611 | 56 | 54830 | 14.0 | 3.912 |
| M139 (6) | ✓106 | 1182247 | 15.0 | 78.594 | ✓106 | 930534 | 10.9 | 85.358 | − | 3924707 | 13.1 | − | − | 417092 | 1.4 | − | − | 345967 | 1.2 | − |
| M140 (4) | ✓80 | 10410 | 15.8 | 0.660 | ✓80 | 4232 | 14.0 | 0.302 | ✓80 | 71814 | 16.2 | 4.439 | ✓80 | 8721 | 29.2 | 0.299 | ✓80 | 15914 | 27.8 | 0.572 |
| M141 (6) | ✓52 | 11285 | 38.4 | 0.294 | ✓52 | 8888 | 29.0 | 0.306 | ✓52 | 12080 | 46.7 | 0.259 | 54 | 12675 | 53.8 | 0.235 | 54 | 5993 | 62.7 | 0.096 |
| M142 (4) | ✓20 | 2101 | 23.9 | 0.088 | ✓20 | 1227 | 20.7 | 0.059 | ✓20 | 2114 | 45.5 | 0.046 | 24 | 2115 | 45.5 | 0.047 | 26 | 1127 | 45.8 | 0.025 |
| M143 (6) | ✓65 | 84377 | 31.3 | 2.694 | ✓65 | 103662 | 18.5 | 5.613 | ✓65 | 96442 | 34.6 | 2.787 | 81 | 250485 | 2.8 | 89.821 | 73 | 73784 | 22.6 | 3.270 |
| M144 (16) | − | 2860221 | 9.3 | − | − | 2358661 | 7.9 | − | − | 2462593 | 8.0 | − | − | 146465 | 0.5 | − | − | 216358 | 0.7 | − |
| M145 (12) | − | 1688897 | 5.6 | − | ✓18 | 256187 | 5.9 | 43.722 | − | 991280 | 3.7 | 264.507 | − | 123629 | 0.4 | − | − | 184120 | 0.6 | − |
| M146 (12) | − | 1762037 | 5.7 | − | ✓14 | 188724 | 3.6 | 52.040 | − | 917818 | 3.3 | 274.823 | − | 73418 | 0.2 | − | − | 122887 | 0.4 | − |
| M147 (3) | ✓50 | 1727 | 18.5 | 0.093 | ✓50 | 1512 | 17.2 | 0.088 | ✓50 | 4539 | 26.1 | 0.174 | 52 | 2238 | 31.3 | 0.072 | ✓50 | 3806 | 42.6 | 0.089 |
| M148 (4) | ✓49 | 1538 | 20.6 | 0.075 | ✓49 | 1696 | 14.0 | 0.121 | ✓49 | 8903 | 25.3 | 0.352 | ✓49 | 2850 | 27.4 | 0.104 | ✓49 | 5042 | 29.8 | 0.169 |
| M149 (4) | ✓35 | 488 | 58.8 | 0.008 | ✓35 | 522 | 34.3 | 0.015 | ✓35 | 1058 | 72.6 | 0.015 | ✓35 | 1320 | 72.7 | 0.018 | ✓35 | 977 | 73.3 | 0.013 |
| M150 (5) | ✓43 | 90817 | 18.6 | 4.874 | ✓43 | 32159 | 13.1 | 2.463 | ✓43 | 420441 | 19.1 | 21.997 | ✓43 | 66159 | 6.9 | 9.642 | ✓43 | 115727 | 4.2 | 27.507 |
| M151 (4) | ✓50 | 9319 | 19.9 | 0.469 | ✓50 | 5123 | 14.2 | 0.362 | ✓50 | 34494 | 22.8 | 1.515 | ✓50 | 7572 | 26.5 | 0.286 | 52 | 8162 | 27.4 | 0.298 |
| M152 (4) | ✓35 | 7969 | 12.6 | 0.630 | ✓35 | 3344 | 9.0 | 0.373 | ✓35 | 12763 | 16.8 | 0.760 | 37 | 11154 | 18.7 | 0.596 | 43 | 12080 | 18.6 | 0.649 |
| M153 (10) | − | 4270841 | 14.2 | − | − | 4378037 | 14.5 | − | − | 1931712 | 6.4 | − | − | 1653554 | 5.5 | − | − | 221338 | 0.7 | − |
| M154 (1) | ✓2 | 2 | 10.1 | 0.000 | ✓2 | 2 | 4.8 | 0.000 | ✓2 | 3 | 11.2 | 0.000 | ✓2 | 5 | 11.8 | 0.000 | ✓2 | 8 | 16.4 | 0.000 |
| M155 (11) | ✓175 | 186 | 13.0 | 0.014 | ✓175 | 188 | 11.9 | 0.016 | ✓175 | 184 | 18.8 | 0.010 | ✓175 | 276 | 20.5 | 0.013 | ✓175 | 284 | 29.8 | 0.010 |
| X1 (6) | ✓97 | 859364 | 19.5 | 43.976 | ✓97 | 727504 | 14.5 | 50.171 | − | 3495467 | 11.7 | − | − | 259370 | 0.9 | − | 99 | 92540 | 1.1 | 85.040 |
| X3 (11) | − | 3630364 | 12.1 | − | − | 3450373 | 11.4 | − | − | 1985701 | 6.6 | − | − | 1138976 | 3.8 | − | − | 157207 | 0.5 | − |
| X78 (8) | − | 1867255 | 6.2 | − | − | 1914264 | 6.4 | − | − | 837196 | 2.8 | − | − | 1213442 | 4.0 | − | − | 136516 | 0.5 | − |

**Table 10:** Results of experiment 2 *(part 3)*

| | BFS+PI | | | | IDDFS+PI+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M1 (2) | ✓8 | 19 | 3.8 | 0.005 | ✓8 | 22 | 1.3 | 0.017 |
| M2 (3) | ✓3 | 5 | 9.2 | 0.001 | ✓3 | 7 | 15.8 | 0.000 |
| M3 (2) | ✓13 | 69 | 8.5 | 0.008 | ✓13 | 70 | 18.5 | 0.004 |
| M4 (3) | ✓7 | 44 | 10.1 | 0.004 | ✓7 | 45 | 11.4 | 0.004 |
| M5 (4) | ✓6 | 220 | 8.0 | 0.028 | ✓6 | 338 | 6.5 | 0.052 |
| M6 (3) | ✓29 | 342 | 18.0 | 0.019 | ✓29 | 1331 | 14.8 | 0.090 |
| M7 (6) | ✓6 | 651 | 11.8 | 0.055 | ✓6 | 390 | 8.7 | 0.045 |
| M8 (2) | ✓32 | 85 | 27.1 | 0.003 | ✓32 | 88 | 27.5 | 0.003 |
| M9 (2) | ✓10 | 16 | 38.3 | 0.000 | ✓10 | 16 | 38.6 | 0.000 |
| M10 (3) | ✓21 | 99 | 22.3 | 0.004 | ✓21 | 120 | 18.8 | 0.006 |
| M11 (2) | ✓16 | 83 | 21.4 | 0.004 | ✓16 | 174 | 21.3 | 0.008 |
| M12 (2) | ✓11 | 22 | 31.4 | 0.001 | ✓11 | 26 | 31.9 | 0.001 |
| M13 (3) | ✓21 | 287 | 25.6 | 0.011 | ✓21 | 428 | 23.3 | 0.018 |
| M14 (2) | ✓10 | 15 | 45.4 | 0.000 | ✓10 | 15 | 31.5 | 0.000 |
| M15 (2) | ✓12 | 25 | 38.5 | 0.001 | ✓12 | 25 | 33.5 | 0.001 |
| M16 (3) | ✓39 | 1036 | 23.4 | 0.044 | ✓39 | 3693 | 19.5 | 0.189 |
| M17 (3) | ✓9 | 32 | 30.4 | 0.001 | ✓9 | 32 | 25.4 | 0.001 |
| M18 (2) | ✓13 | 67 | 27.1 | 0.002 | ✓13 | 88 | 23.6 | 0.004 |
| M19 (2) | ✓20 | 58 | 39.3 | 0.001 | ✓20 | 58 | 32.2 | 0.002 |
| M20 (2) | ✓16 | 71 | 29.6 | 0.002 | ✓16 | 71 | 25.9 | 0.003 |
| M21 (2) | ✓5 | 10 | 46.3 | 0.000 | ✓5 | 10 | 29.7 | 0.000 |
| M22 (2) | ✓15 | 70 | 25.1 | 0.003 | ✓15 | 134 | 20.2 | 0.007 |
| M23 (2) | ✓10 | 21 | 34.0 | 0.001 | ✓10 | 27 | 29.1 | 0.001 |
| M24 (2) | ✓9 | 49 | 40.4 | 0.001 | ✓9 | 47 | 31.4 | 0.001 |
| M25 (3) | ✓7 | 34 | 32.4 | 0.001 | ✓7 | 29 | 27.3 | 0.001 |
| M26 (3) | ✓10 | 69 | 38.3 | 0.002 | ✓10 | 76 | 33.0 | 0.002 |
| M27 (2) | ✓10 | 42 | 32.1 | 0.001 | ✓10 | 52 | 28.5 | 0.002 |
| M28 (2) | ✓9 | 21 | 34.6 | 0.001 | ✓9 | 21 | 27.3 | 0.001 |
| M29 (2) | ✓22 | 79 | 23.5 | 0.003 | ✓22 | 155 | 20.2 | 0.008 |
| M30 (3) | ✓5 | 25 | 28.3 | 0.001 | ✓5 | 32 | 22.9 | 0.001 |
| M31 (3) | ✓6 | 38 | 29.1 | 0.001 | ✓6 | 45 | 26.6 | 0.002 |
| M32 (3) | ✓9 | 38 | 38.3 | 0.001 | ✓9 | 39 | 32.8 | 0.001 |
| M33 (3) | ✓10 | 238 | 26.2 | 0.009 | ✓10 | 60 | 22.7 | 0.003 |
| M34 (4) | ✓8 | 166 | 18.5 | 0.009 | ✓8 | 181 | 16.1 | 0.011 |
| M35 (5) | ✓31 | 5207 | 19.0 | 0.275 | ✓31 | 11147 | 17.4 | 0.640 |
| M36 (5) | ✓59 | 13252 | 24.3 | 0.544 | ✓59 | 51262 | 20.7 | 2.473 |
| M37 (3) | ✓23 | 219 | 29.0 | 0.008 | ✓23 | 437 | 24.2 | 0.018 |
| M38 (3) | ✓8 | 23 | 21.7 | 0.001 | ✓8 | 27 | 21.8 | 0.001 |
| M39 (2) | ✓27 | 79 | 29.1 | 0.003 | ✓27 | 79 | 27.1 | 0.003 |
| M40 (3) | ✓7 | 68 | 32.4 | 0.002 | ✓7 | 70 | 28.4 | 0.002 |
| M41 (3) | ✓13 | 51 | 29.9 | 0.002 | ✓13 | 61 | 26.8 | 0.002 |
| M42 (3) | ✓15 | 154 | 30.3 | 0.005 | ✓15 | 165 | 26.0 | 0.006 |
| M43 (3) | ✓22 | 282 | 24.1 | 0.012 | ✓22 | 490 | 20.7 | 0.024 |
| M44 (1) | ✓1 | 2 | 80.9 | 0.000 | ✓1 | 2 | 13.5 | 0.000 |
| M45 (3) | ✓11 | 111 | 35.4 | 0.003 | ✓11 | 129 | 31.6 | 0.004 |
| M46 (2) | ✓8 | 19 | 34.7 | 0.001 | ✓8 | 19 | 26.1 | 0.001 |
| M47 (2) | ✓22 | 90 | 28.5 | 0.003 | ✓22 | 123 | 25.5 | 0.005 |
| M48 (3) | ✓14 | 170 | 23.5 | 0.007 | ✓14 | 215 | 21.1 | 0.010 |
| M49 (3) | ✓21 | 141 | 28.5 | 0.005 | ✓21 | 328 | 23.6 | 0.014 |
| M50 (2) | ✓17 | 64 | 22.4 | 0.003 | ✓17 | 86 | 19.9 | 0.004 |
| M51 (2) | ✓8 | 28 | 33.6 | 0.001 | ✓8 | 28 | 27.2 | 0.001 |
| M52 (4) | ✓8 | 396 | 22.8 | 0.017 | ✓8 | 424 | 21.2 | 0.020 |
| M53 (4) | ✓12 | 120 | 28.9 | 0.004 | ✓12 | 127 | 26.3 | 0.005 |
| M54 (4) | ✓30 | 2332 | 20.9 | 0.112 | ✓30 | 9175 | 17.0 | 0.538 |
| M55 (2) | ✓27 | 97 | 31.7 | 0.003 | ✓27 | 106 | 26.7 | 0.004 |
| M56 (2) | ✓6 | 12 | 37.8 | 0.000 | ✓6 | 14 | 27.0 | 0.001 |
| M57 (2) | ✓23 | 78 | 33.2 | 0.002 | ✓23 | 81 | 31.1 | 0.003 |
| M58 (3) | ✓11 | 51 | 32.8 | 0.002 | ✓11 | 54 | 29.1 | 0.002 |
| M59 (3) | ✓50 | 1900 | 13.1 | 0.144 | ✓50 | 9575 | 10.3 | 0.927 |
| M60 (4) | ✓44 | 1744 | 22.6 | 0.077 | ✓44 | 3717 | 19.8 | 0.188 |

**Table 11:** Results of experiment 3 *(part 1)*

| | BFS+PI | | | | IDDFS+PI+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M61 (4) | ✓21 | 255 | 19.3 | 0.013 | ✓21 | 411 | 17.7 | 0.023 |
| M62 (4) | ✓30 | 501 | 23.2 | 0.022 | ✓30 | 501 | 21.0 | 0.024 |
| M63 (2) | ✓50 | 127 | 29.8 | 0.004 | ✓50 | 127 | 25.9 | 0.005 |
| M64 (4) | ✓30 | 312 | 24.3 | 0.013 | ✓30 | 339 | 21.6 | 0.016 |
| M65 (4) | ✓41 | 4038 | 19.2 | 0.210 | ✓41 | 31899 | 15.8 | 2.014 |
| M66 (3) | ✓15 | 456 | 12.2 | 0.037 | ✓15 | 456 | 11.5 | 0.040 |
| M67 (3) | ✓8 | 32 | 33.8 | 0.001 | ✓8 | 36 | 29.9 | 0.001 |
| M68 (3) | ✓28 | 338 | 28.4 | 0.012 | ✓28 | 479 | 24.2 | 0.020 |
| M69 (3) | ✓37 | 1670 | 19.5 | 0.086 | ✓37 | 5455 | 16.1 | 0.338 |
| M70 (4) | ✓26 | 1596 | 23.6 | 0.068 | ✓26 | 2675 | 20.9 | 0.128 |
| M71 (2) | ✓21 | 114 | 18.5 | 0.006 | ✓21 | 160 | 15.8 | 0.010 |
| M72 (3) | ✓40 | 571 | 23.4 | 0.024 | ✓40 | 639 | 21.1 | 0.030 |
| M73 (3) | ✓25 | 918 | 21.0 | 0.044 | ✓25 | 315 | 15.3 | 0.021 |
| M74 (4) | ✓34 | 2117 | 22.3 | 0.095 | ✓34 | 4451 | 16.1 | 0.276 |
| M75 (4) | ✓34 | 1143 | 23.7 | 0.048 | ✓34 | 2950 | 20.2 | 0.146 |
| M76 (3) | ✓56 | 1346 | 17.8 | 0.075 | ✓56 | 9591 | 13.2 | 0.725 |
| M77 (4) | ✓55 | 2972 | 23.3 | 0.128 | ✓55 | 31001 | 16.6 | 1.867 |
| M78 (5) | ✓33 | 10242 | 14.4 | 0.713 | ✓33 | 73764 | 11.7 | 6.295 |
| M79 (3) | ✓18 | 135 | 24.9 | 0.005 | ✓18 | 134 | 22.5 | 0.006 |
| M80 (4) | ✓38 | 934 | 20.6 | 0.045 | ✓38 | 2997 | 17.3 | 0.173 |
| M81 (3) | ✓12 | 111 | 31.9 | 0.003 | ✓12 | 131 | 27.8 | 0.005 |
| M82 (3) | ✓14 | 36 | 31.7 | 0.001 | ✓14 | 36 | 28.5 | 0.001 |
| M83 (4) | ✓47 | 6035 | 16.5 | 0.367 | ✓47 | 17873 | 13.7 | 1.309 |
| M84 (3) | ✓68 | 2470 | 18.9 | 0.130 | ✓68 | 33441 | 15.0 | 2.224 |
| M85 (3) | ✓51 | 3921 | 17.3 | 0.227 | ✓51 | 30154 | 12.9 | 2.338 |
| M86 (4) | ✓25 | 1026 | 22.4 | 0.046 | ✓25 | 4093 | 18.7 | 0.219 |
| M87 (4) | ✓53 | 5282 | 20.7 | 0.256 | ✓53 | 38207 | 14.8 | 2.577 |
| M88 (3) | ✓63 | 2420 | 16.9 | 0.143 | ✓63 | 33391 | 12.9 | 2.582 |
| M89 (4) | ✓35 | 4344 | 21.6 | 0.201 | ✓35 | 31598 | 17.7 | 1.783 |
| M90 (4) | ✓16 | 2786 | 20.7 | 0.134 | ✓16 | 4059 | 17.7 | 0.230 |
| M91 (4) | ✓14 | 301 | 36.4 | 0.008 | ✓14 | 424 | 33.2 | 0.013 |
| M92 (3) | ✓48 | 686 | 21.8 | 0.031 | ✓48 | 772 | 19.6 | 0.039 |
| M93 (8) | – | 2404906 | 8.0 | – | – | 921403 | 3.1 | – |
| M94 (3) | ✓29 | 454 | 22.6 | 0.020 | ✓29 | 456 | 27.9 | 0.016 |
| M95 (8) | ✓8 | 3191 | 7.1 | 0.448 | ✓8 | 4042 | 8.1 | 0.502 |
| M96 (3) | ✓37 | 646 | 22.8 | 0.028 | ✓37 | 672 | 20.9 | 0.032 |
| M97 (5) | ✓41 | 8876 | 13.7 | 0.647 | ✓41 | 46450 | 11.4 | 4.083 |
| M98 (5) | ✓110 | 33183 | 15.1 | 2.201 | ✓110 | 78671 | 13.1 | 6.003 |
| M99 (4) | ✓131 | 69149 | 13.1 | 5.260 | ✓131 | 944352 | 10.0 | 94.062 |
| M100 (4) | ✓52 | 1106 | 28.7 | 0.038 | ✓52 | 4155 | 24.3 | 0.171 |
| M101 (5) | ✓15 | 936 | 10.1 | 0.092 | ✓15 | 1130 | 9.3 | 0.121 |
| M102 (4) | ✓44 | 4402 | 16.9 | 0.261 | ✓44 | 22624 | 12.5 | 1.809 |
| M103 (4) | ✓12 | 921 | 23.2 | 0.040 | ✓12 | 1033 | 20.9 | 0.049 |
| M104 (3) | ✓27 | 363 | 15.5 | 0.023 | ✓27 | 389 | 18.3 | 0.021 |
| M105 (8) | ✓24 | 49822 | 15.1 | 3.295 | ✓24 | 50385 | 13.8 | 3.639 |
| M106 (5) | ✓50 | 8432 | 19.0 | 0.444 | ✓50 | 77994 | 15.5 | 5.038 |
| M107 (11) | ✓10 | 1058 | 14.6 | 0.073 | ✓10 | 1271 | 13.5 | 0.094 |
| M108 (4) | ✓68 | 28714 | 13.3 | 2.153 | ✓68 | 275499 | 10.6 | 26.107 |
| M109 (5) | ✓42 | 47698 | 14.7 | 3.239 | ✓42 | 218966 | 12.1 | 18.028 |
| M110 (4) | ✓14 | 1301 | 26.4 | 0.049 | ✓14 | 1421 | 26.8 | 0.053 |
| M111 (6) | ✓61 | 223970 | 13.4 | 16.722 | ✓61 | 1136649 | 10.0 | 113.471 |
| M112 (5) | ✓94 | 116624 | 12.9 | 9.053 | ✓94 | 1414171 | 10.4 | 136.283 |
| M113 (4) | ✓51 | 20967 | 10.9 | 1.917 | ✓51 | 57753 | 9.5 | 6.103 |
| M114 (6) | ✓60 | 20694 | 17.2 | 1.205 | ✓60 | 193454 | 14.3 | 13.569 |
| M115 (5) | ✓29 | 17281 | 14.7 | 1.175 | ✓29 | 38990 | 12.7 | 3.071 |
| M116 (5) | ✓14 | 1331 | 23.2 | 0.057 | ✓14 | 1618 | 21.9 | 0.074 |
| M117 (5) | ✓47 | 123684 | 10.8 | 11.445 | ✓47 | 528661 | 8.9 | 59.491 |
| M118 (4) | ✓44 | 7133 | 15.1 | 0.473 | ✓44 | 46027 | 12.3 | 3.734 |
| M119 (3) | ✓18 | 258 | 17.7 | 0.015 | ✓18 | 809 | 14.6 | 0.055 |
| M120 (4) | ✓64 | 2401 | 19.7 | 0.122 | ✓64 | 9720 | 16.6 | 0.587 |

**Table 12:** Results of experiment 3 *(part 2)*

| | BFS+PI | | | | IDDFS+PI+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| M121 (5) | ✓47 | 2056 | 21.1 | 0.097 | ✓47 | 3184 | 18.5 | 0.172 |
| M122 (5) | ✓90 | 91700 | 13.4 | 6.851 | ✓90 | 1150064 | 9.2 | 124.536 |
| M123 (5) | ✓101 | 289557 | 11.2 | 25.817 | – | 2348142 | 7.8 | – |
| M124 (3) | ✓39 | 1905 | 18.4 | 0.104 | ✓39 | 6295 | 14.9 | 0.424 |
| M125 (4) | ✓38 | 4951 | 20.5 | 0.242 | ✓38 | 10663 | 16.4 | 0.650 |
| M126 (7) | ✓23 | 84210 | 17.6 | 4.772 | ✓23 | 100007 | 14.5 | 6.880 |
| M127 (4) | ✓32 | 1380 | 11.6 | 0.119 | ✓32 | 1457 | 19.5 | 0.075 |
| M128 (4) | ✓19 | 942 | 22.2 | 0.042 | ✓19 | 1070 | 21.5 | 0.050 |
| M129 (5) | ✓22 | 1683 | 22.8 | 0.074 | ✓22 | 1853 | 21.1 | 0.088 |
| M130 (4) | ✓36 | 5308 | 19.8 | 0.268 | ✓36 | 14320 | 16.5 | 0.867 |
| M131 (4) | ✓31 | 3325 | 19.9 | 0.167 | ✓31 | 6964 | 17.3 | 0.404 |
| M132 (4) | ✓37 | 1341 | 22.8 | 0.059 | ✓37 | 2057 | 20.3 | 0.101 |
| M133 (5) | ✓39 | 21166 | 16.1 | 1.312 | ✓39 | 171492 | 12.8 | 13.372 |
| M134 (4) | ✓76 | 23424 | 13.2 | 1.768 | ✓76 | 231417 | 10.6 | 21.792 |
| M135 (4) | ✓36 | 2697 | 17.2 | 0.157 | ✓36 | 15565 | 15.2 | 1.026 |
| M136 (4) | ✓25 | 1866 | 19.0 | 0.098 | ✓25 | 8699 | 16.0 | 0.543 |
| M137 (4) | ✓46 | 17646 | 12.4 | 1.428 | ✓46 | 72090 | 9.6 | 7.532 |
| M138 (5) | ✓54 | 62789 | 12.8 | 4.909 | ✓54 | 290066 | 9.6 | 30.079 |
| M139 (6) | ✓106 | 1199460 | 10.8 | 111.527 | – | 2476347 | 8.3 | – |
| M140 (4) | ✓80 | 15084 | 13.8 | 1.097 | ✓80 | 143238 | 10.7 | 13.399 |
| M141 (6) | ✓52 | 8726 | 22.5 | 0.389 | ✓52 | 9226 | 20.3 | 0.455 |
| M142 (4) | ✓20 | 2170 | 20.1 | 0.108 | ✓20 | 2200 | 18.1 | 0.122 |
| M143 (6) | ✓65 | 40629 | 16.8 | 2.424 | ✓65 | 46919 | 14.4 | 3.259 |
| M144 (16) | – | 2076256 | 6.9 | – | – | 1435313 | 4.7 | – |
| M145 (12) | – | 2106661 | 7.0 | – | – | 641497 | 2.1 | – |
| M146 (12) | – | 1167216 | 3.9 | – | – | 423009 | 1.6 | 260.774 |
| M147 (3) | ✓50 | 1925 | 14.3 | 0.134 | ✓50 | 9881 | 15.2 | 0.649 |
| M148 (4) | ✓49 | 2556 | 13.8 | 0.185 | ✓49 | 19013 | 13.6 | 1.399 |
| M149 (4) | ✓35 | 487 | 22.9 | 0.021 | ✓35 | 500 | 21.1 | 0.024 |
| M150 (5) | ✓43 | 115591 | 14.7 | 7.869 | ✓43 | 825377 | 11.6 | 71.316 |
| M151 (4) | ✓50 | 13366 | 15.6 | 0.859 | ✓50 | 45217 | 12.7 | 3.555 |
| M152 (4) | ✓35 | 3943 | 11.4 | 0.346 | ✓35 | 6070 | 10.9 | 0.558 |
| M153 (10) | – | 4033426 | 13.4 | – | – | 2880753 | 9.6 | – |
| M154 (1) | ✓2 | 4 | 7.5 | 0.001 | ✓2 | 5 | 5.9 | 0.001 |
| M155 (11) | ✓175 | 199 | 8.6 | 0.023 | ✓175 | 190 | 7.8 | 0.024 |
| X1 (6) | ✓97 | 468316 | 10.8 | 43.344 | – | 2471062 | 8.2 | – |
| X3 (11) | – | 2887739 | 9.6 | – | – | 2910080 | 9.7 | – |
| X78 (8) | – | 2007114 | 6.7 | – | – | 1841518 | 6.1 | – |

**Table 13:** Results of experiment 3 *(part 3)*

| | BFS+VL | | | | IDDFS+VL+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| X1 (6) | ✓97 | 13 | 0.2 | 0.082 | ✓97 | 7 | 0.9 | 0.007 |
| X2 (10) | 133 | 120 | 1.6 | 0.073 | – | 221457 | 0.7 | – |
| X3 (11) | 148 | 26 | 0.7 | 0.037 | 152 | 388 | 2.2 | 0.180 |
| X4 (20) | 357 | 2330 | 1.0 | 2.417 | – | 431309 | 1.4 | – |
| X5 (12) | ✓143 | 10 | 0.3 | 0.030 | – | 253830 | 0.8 | – |
| X6 (10) | ✓110 | 66 | 0.4 | 0.156 | – | 224920 | 0.7 | – |
| X7 (11) | 106 | 234778 | 1.1 | 213.472 | – | 389755 | 1.3 | – |
| X8 (18) | 280 | 7988 | 0.2 | 45.313 | – | 94078 | 0.3 | – |
| X9 (14) | 239 | 30045 | 0.3 | 110.093 | – | 173381 | 0.6 | – |
| X10 (32) | – | 27303 | 0.1 | – | – | 173304 | 0.6 | – |
| X11 (14) | – | 191167 | 0.6 | – | – | 282623 | 0.9 | – |
| X12 (15) | – | 757441 | 2.5 | – | – | 998349 | 3.3 | – |
| X13 (16) | – | 1476974 | 4.9 | – | – | 1100406 | 3.7 | – |
| X14 (18) | – | 1739252 | 5.8 | – | – | 1992512 | 6.6 | – |
| X15 (15) | – | 1387719 | 4.6 | – | – | 1208609 | 4.0 | – |
| X16 (15) | – | 412637 | 1.4 | – | – | 280793 | 0.9 | – |
| X17 (6) | 217 | 8073 | 0.5 | 15.806 | – | 217762 | 0.7 | – |
| X18 (11) | – | 86822 | 0.3 | – | – | 142594 | 0.5 | – |
| X19 (15) | – | 1986260 | 6.6 | – | – | 1367107 | 4.6 | – |
| X20 (18) | – | 1043626 | 3.5 | – | – | 1792506 | 6.0 | – |
| X21 (13) | – | 2015739 | 6.7 | – | – | 1734084 | 5.8 | – |
| X22 (27) | – | 1040950 | 3.5 | – | – | 1273951 | 4.2 | – |
| X23 (18) | – | 181666 | 0.6 | – | – | 192127 | 0.6 | – |
| X24 (22) | – | 193095 | 0.6 | – | – | 280814 | 0.9 | – |
| X25 (19) | – | 151960 | 0.5 | – | – | 98666 | 0.3 | – |
| X26 (13) | – | 318782 | 1.1 | – | – | 359926 | 1.2 | – |
| X27 (20) | – | 306165 | 1.0 | – | – | 184517 | 0.6 | – |
| X28 (20) | – | 2143441 | 7.1 | – | – | 1384314 | 4.6 | – |
| X29 (16) | – | 713988 | 2.4 | – | – | 891901 | 3.0 | – |
| X30 (18) | – | 328379 | 1.1 | – | – | 529960 | 1.8 | – |
| X31 (20) | – | 1835309 | 6.1 | – | – | 2070293 | 6.9 | – |
| X32 (15) | – | 139313 | 0.5 | – | – | 161368 | 0.5 | – |
| X33 (15) | – | 2414175 | 8.0 | – | – | 2533338 | 8.4 | – |
| X34 (14) | – | 152206 | 0.5 | – | – | 158227 | 0.5 | – |
| X35 (17) | – | 1124631 | 3.7 | – | – | 1545186 | 5.2 | – |
| X36 (21) | – | 147653 | 0.5 | – | – | 261154 | 0.9 | – |
| X37 (20) | – | 1139744 | 3.8 | – | – | 1153789 | 3.8 | – |
| X38 (8) | – | 459433 | 1.5 | – | – | 344254 | 1.1 | – |
| X39 (25) | – | 2733235 | 9.1 | – | – | 1063199 | 3.5 | – |
| X40 (16) | – | 100584 | 0.3 | – | – | 188692 | 0.6 | – |
| X41 (15) | – | 1881674 | 6.3 | – | – | 2606860 | 8.7 | – |
| X42 (24) | – | 1004244 | 3.3 | – | – | 957301 | 3.2 | – |
| X43 (9) | 148 | 3478 | 0.8 | 4.340 | – | 473279 | 1.6 | – |
| X44 (9) | – | 2610985 | 8.7 | – | – | 2180320 | 7.3 | – |
| X45 (17) | – | 275603 | 0.9 | – | – | 273820 | 0.9 | – |
| X46 (14) | – | 1918194 | 6.4 | – | – | 1321142 | 4.4 | – |
| X47 (16) | – | 2170530 | 7.2 | – | – | 2001872 | 6.7 | – |
| X48 (34) | – | 853768 | 2.8 | – | – | 1366267 | 4.6 | – |
| X49 (12) | – | 2735698 | 9.1 | – | – | 2823506 | 9.4 | – |
| X50 (16) | – | 1656835 | 5.5 | – | – | 1529224 | 5.1 | – |
| X51 (14) | – | 2389962 | 8.0 | – | – | 1945015 | 6.5 | – |
| X52 (18) | – | 186449 | 0.6 | – | – | 151754 | 0.5 | – |
| X53 (15) | 210 | 2 | 0.0 | 0.061 | 210 | 2 | 0.0 | 0.061 |
| X54 (16) | 267 | 61 | 0.2 | 0.401 | – | 188847 | 0.6 | – |
| X55 (12) | – | 623972 | 2.1 | – | – | 1194927 | 4.0 | – |
| X56 (16) | 237 | 36477 | 0.2 | 176.581 | – | 146741 | 0.5 | – |
| X57 (16) | – | 1318644 | 4.4 | – | – | 1961935 | 6.5 | – |
| X58 (15) | – | 1071200 | 3.6 | – | – | 1353109 | 4.5 | – |
| X59 (16) | 316 | 2629 | 0.2 | 14.248 | – | 90505 | 0.3 | – |
| X60 (13) | – | 924951 | 3.1 | – | – | 1277411 | 4.3 | – |

**Table 14:** Results of experiment 4 *(part 1)*

| | BFS+VL | | | | IDDFS+VL+PP+IMO | | | |
|---|---|---|---|---|---|---|---|---|
| **Puzzle** | **Path** | **Nodes** | **Speed** | **Time** | **Path** | **Nodes** | **Speed** | **Time** |
| *(stones)* | *moves* | | *1000/s* | *s* | *moves* | | *1000/s* | *s* |
| X61 (20) | – | 1019117 | 3.4 | – | – | 1743255 | 5.8 | – |
| X62 (16) | – | 68039 | 0.2 | – | – | 118133 | 0.4 | – |
| X63 (17) | – | 798570 | 2.7 | – | – | 725545 | 2.4 | – |
| X64 (16) | 411 | 9001 | 0.2 | 48.024 | – | 218724 | 0.7 | – |
| X65 (15) | – | 202961 | 0.7 | – | – | 166683 | 0.6 | – |
| X66 (18) | – | 1149154 | 3.8 | – | – | 795184 | 2.7 | – |
| X67 (20) | – | 2222865 | 7.4 | – | – | 1397815 | 4.7 | – |
| X68 (15) | – | 109113 | 0.4 | – | – | 204017 | 0.7 | – |
| X69 (18) | – | 1260995 | 4.2 | – | – | 913531 | 3.0 | – |
| X70 (18) | 349 | 10519 | 0.5 | 20.977 | – | 232413 | 0.8 | – |
| X71 (18) | – | 1571387 | 5.2 | – | – | 1860561 | 6.2 | – |
| X72 (16) | – | 53591 | 0.2 | – | – | 68433 | 0.2 | – |
| X73 (14) | – | 1995395 | 6.7 | – | – | 2219136 | 7.4 | – |
| X74 (16) | – | 2694280 | 9.0 | – | – | 1706684 | 5.7 | – |
| X75 (17) | – | 496840 | 1.7 | – | – | 172763 | 0.6 | – |
| X76 (17) | – | 138139 | 0.5 | – | – | 116434 | 0.4 | – |
| X77 (14) | – | 124413 | 0.4 | – | – | 437872 | 1.5 | – |
| X78 (8) | 142 | 2 | 0.1 | 0.017 | 142 | 2 | 0.1 | 0.017 |
| X79 (12) | 176 | 949 | 0.3 | 2.875 | – | 282913 | 0.9 | – |
| X80 (12) | 233 | 1399 | 1.0 | 1.369 | – | 112270 | 0.4 | – |
| X81 (12) | 191 | 1057 | 1.0 | 1.072 | – | 328673 | 1.1 | – |
| X82 (12) | 173 | 15 | 0.7 | 0.022 | – | 142682 | 0.5 | – |
| X83 (10) | ✓194 | 568 | 0.7 | 0.817 | – | 293527 | 1.0 | – |
| X84 (12) | 161 | 17852 | 0.1 | 132.219 | – | 81050 | 0.3 | – |
| X85 (15) | – | 544450 | 1.8 | – | – | 838530 | 2.8 | – |
| X86 (10) | – | 2992175 | 9.9 | – | – | 2230896 | 7.4 | – |
| X87 (12) | – | 1567963 | 5.2 | – | – | 1387214 | 4.6 | – |
| X88 (23) | – | 949355 | 3.2 | – | – | 1327185 | 4.4 | – |
| X89 (21) | – | 1189487 | 4.0 | – | – | 1322462 | 4.4 | – |
| X90 (25) | – | 42000 | 0.1 | – | – | 166924 | 0.6 | – |

**Table 15:** Results of experiment 4 *(part 2)*