Creating a solver for Kwirk (Puzzle Boy)

Ellen Wittingen
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
e.m.wittingen@student.utwente.nl

ABSTRACT

Kwirk, or Puzzle Boy in Japan, is a video game containing a planning problem where characters need to be moved to the exit through a room with obstacles such as blocks, turnstiles, and holes in the floor. At the time of writing, only one Kwirk solver exists which makes use of external memory, which might be slow and not very memory-efficient. Therefore, the aim of this study is to investigate whether an algorithmic solver can be created which does not require external memory. Further objectives are to find out what features make a Kwirk solver more efficient and find the shortest solutions. For this investigation multiple solvers were explored. 16 out of 30 puzzles from the game-mode *Going up?* could be solved. Most solvers ran out of memory on the remaining puzzles. The shortest solutions out of all solvers were found by a simple A* solver and a multi-layered A* solver that only generates subgoals for holes. None of the solvers stood out with regards to solving a puzzle most efficiently.

Keywords

Kwirk, Puzzle Boy, solver, planning problem, puzzle, subgoals, multi-layered

1. INTRODUCTION

Planning problems are problems for which a sequence of tasks needs to be determined to accomplish a certain goal [11]. The video game Kwirk, or Puzzle Boy in Japan, is such a problem. It is a puzzle game from 1989 by Atlus Co., Ltd. for the Game Boy. Each puzzle represents a maze containing one or more characters, obstacles, and an exit, where the goal is to navigate all characters to the exit. Only one character can be controlled at a time, but the player can switch between them. The obstacles consist of walls, blocks, turnstiles, and holes in the floor. Walls cannot be interacted with and nothing can penetrate them. Blocks vary in shape and can be pushed by characters if the space behind the block is empty or a hole. When a block is pushed into a hole, that section of the hole is turned into normal floor tiles and the block disappears. Turnstiles have one to four arms attached to a central pivot. When one of the arms is pushed the turnstile turns 90 degrees in the direction it is being pushed in. However, this is not possible when an obstacle or character is blocking an arm from turning. [15, 16]

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
35th Twente Student Conference on IT July 2nd, 2021, Enschede, The Netherlands.
Copyright 2021, University of Twente, Faculty of Electrical Engineering, Mathematics and Computer Science.

At the time of writing, no published research could be found on solving Kwirk puzzles. There does exist a Kwirk solver by Vladimir Panteleev which uses external memory to manage its large state-space [13]. This technique might not be very memory-efficient, as the use of external memory is required. It might be slow as well, if it explores such a large state-space.

This raises the following questions: Can Kwirk puzzles be solved algorithmically without using external memory? If this is the case, what is the most efficient method to solve Kwirk puzzles? And what kind of solvers can find the shortest solutions?

To answer these questions, this paper discusses the following: First, the modelling of the puzzles and the inner workings of various solvers are described. After that, experiments on the solvers are discussed and the results are analysed. Lastly, some possible improvements upon the solvers are described.

2. REQUIREMENTS

A valid solver should conform to the following requirements: If a puzzle is solvable, the solver should return a sequence of tasks (e.g. pushing a block or the arm of a turnstile) that results in all characters reaching the exit. The returned solution should not violate any of the rules of Kwirk. Furthermore, for each character-obstacle interaction, the character should be able to reach the obstacle from its current position without any other interactions with obstacles. Lastly, if a puzzle is not solvable, the solver should inform the user of that.

3. EXISTING SOLUTIONS

At the time of writing, no published research could be found on solving Kwirk puzzles. However, there does exist a solver by Vladimir Panteleev [13]. This solver uses a technique called External Memory Delayed Duplicate Detection, which temporarily stores part of the state-space in external memory. When new states are added, it is not immediately checked whether the state is a duplicate. Instead, an entire set of new states is checked for duplicates at a later time, as that is more efficient when the external memory is slow [4].

Requiring external memory to manage the large state-space might be a sign that the solver in [13] is not very memory-efficient. Solving the puzzles might be slow as well, if such a large state-space is explored. However, due to time constraints it was not possible to confirm these inferences. In any case, there might be a more memory-efficient or faster method to solve Kwirk puzzles, or at least a method that does not require external memory.

4. ARCHITECTURE

A game related to Kwirk is Sokoban, where the goal is to push several blocks to marked positions in a maze [10]. In both games blocks can be pushed around so some ideas from Sokoban solvers have been incorporated into the Kwirk solver. However, there are also differences between Sokoban and Kwirk. The most notable difference is that in Sokoban blocks must be moved to goal positions instead of characters. Another difference is that Sokoban puzzles always have one character, whereas Kwirk puzzles can contain more than one. Furthermore, Kwirk puzzles additionally contain turnstiles and holes. Finally, in Sokoban all blocks take up exactly one tile whereas in Kwirk the sizes vary. All these differences mean that solving Kwirk is still quite different from Sokoban. Because Kwirk puzzles involve more kinds of obstacles and possibly multiple characters, it can arguably be more complicated to solve them.

In this section the model used to represent Kwirk puzzles and two kinds of solvers are discussed. The entire architecture was implemented in Java 11.

4.1 Model

A model of the Kwirk puzzles is required to solve them, which will be described in next few sub-sections.

4.1.1 State representation

Each state represents the current configuration of the maze. In a state the positions each obstacle occupies and what positions each character can reach are stored. A position is considered reachable if it is not occupied by any obstacle and the character can walk to it without interacting with any obstacles along the way.

Two states are considered different if the occupied positions of at least one of the obstacles differs, or if one of the sets of reachable positions is different. These sets of reachable positions are not linked to the characters itself when comparing states, so if two characters swapped places and nothing else changed, it is considered the same state. This is possible because every character has the exact same abilities and goal. This state equality should reduce the number of states generated.

4.1.2 State-transitions

When generating state-transitions, the Sokoban solver by Hernandez-Belmonte et al. [6] only looks at which directions each box can be pushed in. Then, for each box-direction pair, a path from the robot to the position from which to push the box is calculated. This avoids generating states in which only the robot is moved, but no boxes.

The generation of state-transitions for the solvers in this study works in a similar fashion. The solvers only consider in which directions each block and turnstile arm can be pushed. An obstacle-direction pair is only considered valid if the following holds: First of all, the new positions that the pushed obstacle will occupy should not blocked by some other obstacle. Secondly, at least one character should be able to reach a position from which they can push the obstacle. Lastly, the position the character will occupy after pushing the obstacle should not be occupied already.

One major difference with the Sokoban solver, is that the solvers in this study do not calculate paths from the character to the obstacle. Instead, the set of reachable positions stored in the state is used to check whether an obstacle is reachable. This technique most likely reduces the number of calculations for each obstacle-direction pair. This means that most tasks in the solution only specify from what position which character should push which obstacle in what direction. The solution can also contain tasks to move a character to a different position (see Section 4.1.3), but those do not specify a path either.

4.1.3 Multi-character puzzles

Some puzzles contain multiple characters, which can cooperate with each other or block moves from others.

Cooperation.

Sometimes the cooperation of multiple characters is required to solve a puzzle. Figure 1 shows an example of such a puzzle. The block at position (2,2) must be moved north at some point to fill the hole at (1,0). This is only possible by pushing the block to (1,2) so it can then be pushed north from (1,3). However, after the block has been pushed to (1,2), position (1,3) is no longer reachable. This means that a character needs to move to (1,3) before the block is pushed.

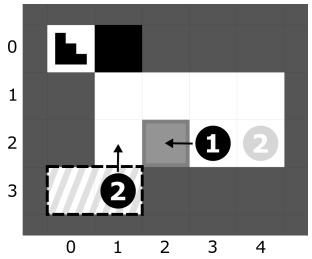


Figure 1: A two-character combo move. The light grey circle is the original position of character 2 and the black circle marked 2 is its new position.

For these situations so-called combo-pushes were implemented. When a block can be pushed, such as pushing the block at (2,2) west in Figure 1, it is checked whether the positions perpendicular to the push direction will still be reachable after the block is pushed (positions (1,1) and (1,3)). The position in push direction itself (position (0,2)) is ignored, as a character there will push the block back to its original position. If such a position is unreachable after the block's move, it is checked whether the block could be pushed if a character were there. If this is the case, a new state-transition is generated where first a character is moved to that position, then the original push is performed, and then the block is pushed by the moved character. This means that the block moved two positions in a single state-transition. Similar checking is done on the new state-transition to determine whether a triple combo-push is possible. Quadruple combo-pushes are not calculated, as that would push the block around in a circle back to its original position.

Characters blocking moves.

When an obstacle is pushed, it could be a case that a character is blocking the obstacle from moving. Therefore, each character that is blocking the push is moved to a position they can reach before the obstacle is pushed.

There is a slight complication with this, as the chosen position can influence what the character can reach in the future. For example, when looking at Figure 2, if a character is moved to area `1`, then after the block has been moved west, it cannot reach the 1x1 block near the southwest corner. Whereas if the character is moved to area `3`, it could reach that block after the push. Therefore, for every combination of characters and areas a new state-transition is created.

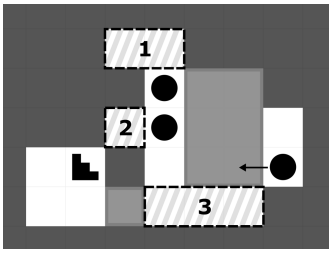


Figure 2: Multiple areas (marked with numbers and circled with black) are created when the block is moved west.

In the puzzle in Figure 2, there are three separate areas characters can move to before the block is pushed west. The possible character-area combinations are: two characters in area 1, two characters in area 3, one character in area 1 and one in area 2, one character in area 1 and one in area 3, and one character in area 2 and one in area 3.

4.2 Simple solvers

Some simple solvers were explored first. These solvers repeatedly explore state-transitions to new states until the goal state is found, which is when all characters can reach the exit. Solvers with the following search algorithms have been investigated: Breadth First Search (BFS), Depth First Search (DFS), A* and Fringe. These specific algorithms were chosen because they are diverse in terms of the order in which states are opened, which could help to find out what kind of search algorithm is the best.

4.2.1 BFS and DFS

BFS opens first all the siblings of a node prior to opening its children. DFS does the opposite, and opens all the node's children before opening any of its siblings.

4.2.2 A search*

A* is a best-first algorithm which always opens the node with the lowest cost [5]. To accomplished this, the algorithm keeps the nodes in its frontier sorted by cost. This cost is calculated by the cost function in Section 4.2.4.

4.2.3 Fringe search

Fringe search [1] is a variation upon Iterative Deepening A* (IDA*), which only opens nodes with a cost below a certain threshold. When there are no more nodes with a low enough cost, the threshold is raised and the search is restarted with a completely new search tree [9].

Fringe search differentiates itself from IDA* by not rebuilding the search tree when the search is restarted. Björnsson, Enzenberger, Holte and Schaeffer argue that this is an improvement upon IDA* because it causes fewer states to be opened. The downside is that it still has to revisit nodes and keep them in the frontier as long as they are above the current threshold. However, the frontier does not have to be fully sorted (which is a requirement for A*), which saves time [1].

Fringe keeps track of two lists, the `now` and the `later` list. The nodes in the `now` list are inspected during the current iteration. If the cost of the node exceeds the current threshold, the node is added to the end of the `later` list. Otherwise, the node's children are added to the front of the `now` list. When the `now` list is empty, the threshold is set to the minimum cost of the nodes in the `later` list and the `later` list becomes the new `now` list.

The simple Kwirk solvers conform to the pseudo-code in [1], and, as suggested in the paper, a single doubly-linked list was used to represent the `now` and `later` list. The nodes are opened in the same order as IDA*.

4.2.4 Cost function

The simple A* and Fringe solvers both use a cost function to calculate the cost of each node. This cost function, $f(n)$, is calculated as follows: $f(n) = g(n) + h(n)$, where n represents the current node, and $g(n)$ the cost of the path from the start-node to the current node. $h(n)$ is a heuristic function which returns an estimated cost of the cheapest path from the current node to the goal. The exact definitions of $g(n)$ and $h(n)$ are problem-dependent, so the following were chosen for Kwirk: $g(n)$ is simply the number of predecessors of the current node. This makes sure that if there are two solutions with the same $h(n)$, the shortest solution is chosen. $h(n)$ is defined as follows: For the character whose obstacle-interaction caused the transition to the current state, the set of reachable positions are compared with those of an x number of the state's predecessors. The more positions that are only reachable in the current state and not in any of the predecessors, the lower the result of $h(n)$ will be. This choice of $h(n)$ encourages characters to explore areas that they have not visited in a while. For the solvers $x = 10$ was chosen, as after some prior testing on small puzzles, it seemed the best choice.

4.3 Multi-layer solvers

A Sokoban solver by Botea, Müller and Schaeffer [2] consists of two layers: a global and a local solving layer. On the global layer, a Sokoban puzzle is split up into rooms and tunnels and each state-transition on this layer represents moving a box from one room or tunnel to another. The local solving layer then tries to find a path of state-transitions that achieves a single state-transition on the global layer. This idea seemed promising, so two Kwirk solvers using this technique were explored. In the next sub-sections, the details of these solvers are explained.

4.3.1 Global layer

The global layer in the Kwirk solvers somewhat differs from the Sokoban solver in [2]. Instead of rooms and tunnels, a Kwirk puzzle is split up into rooms and connectors, where connectors are obstacles separating any two rooms. A room is a set of positions which is closed off from other areas because it is surrounded by a combination of walls and obstacles. For example, in Figure 3, four rooms can be distinguished. When an obstacle in the surrounding of a room is also part of the surrounding of another room, it is considered a connector, as removing the obstacle would connect the two rooms. In the puzzle in Figure 3, there are four connectors, the other obstacles are part of the rooms.

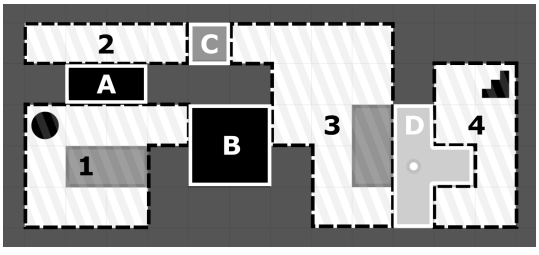


Figure 3: Interpreted rooms and connectors by the global layer. The rooms are numbered and circled with black and the connectors are lettered and circled with white.

The global layer calculates a sequence of sub-goals, using BFS, which move a character from its start-room to the room that holds the exit. When there are multiple characters, the room of an arbitrary character is chosen as the start-room. A sub-goal represents getting rid of a connector in such a way that a character can walk from one room

to the next. Each sub-goal also holds goal-positions, which are positions in the next room. So when any of the goal-positions are reachable, the sub-goal is considered solved.

When looking at Figure 3, there are two possible sequences: going through hole `B` and turnstile `D`, or going through hole `A`, block `C` and turnstile `D`.

A different kind of sub-goal is generated for each kind of connector, so one sub-goal for blocks, one for turnstiles and one for holes. After generating these, sub-goals involving turnstiles and holes are first, recursively, split into multiple sub-goals before the sequence is returned.

As can be seen in Figure 4, when the global layer has determined the sub-goal sequence to solve, each sub-goal is solved individually by the local layer. The local layer uses the goal-state of the last sub-goal in the sequence as its start-state. When a sub-goal cannot be solved, the global search layer tries to find a new sub-goal sequence and solve it. This keeps going till a sub-goal sequence is found of which all sub-goals can be solved or when there are no more new sub-goal sequences. If all sub-goals in the sequence were solved it means that a solution has been found. If not, all discovered sequences that have not been fully explored yet are attempted again. This is required to ensure completeness because some sub-goals can be solved multiple ways. This means that the local solver can generate multiple goal-states for each sub-goal, which could each be used as a start-state for the next sub-goal. The solver finally stops when all possible ways to solve each sub-goal sequence have been exhausted.

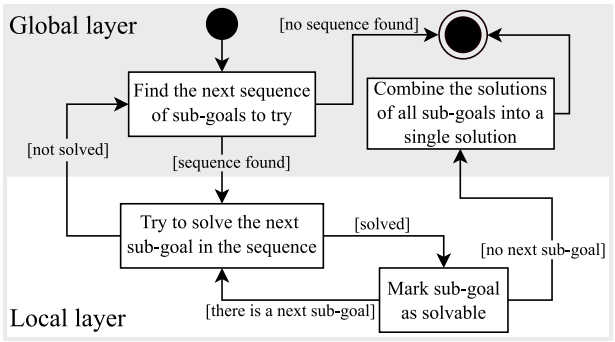


Figure 4: The general flow through the global and local layer of the multi-layer solvers when solving a puzzle.

Turnstile sub-goals.

For a sub-goal for a turnstile, the turnstile needs to be rotated to let characters reach the next room. However, the arms of the turnstile could be blocked from turning by other obstacles. Therefore, a sub-goal is generated for every obstacle that is blocking the turnstile from turning in both directions. No sub-goals are generated for obstacles that only block the turnstile from turning in one direction, unless the turnstile can only turn in that direction because walls are blocking the rotation in the other direction. To give an example, in Figure 3 the block west of turnstile `D` needs to be moved out of the way before it can be rotated, so a sub-goal is generated for it.

Hole sub-goals.

To solve a sub-goal for a hole, the hole needs be filled with blocks in a certain order. This sub-problem seems similar to a Sokoban puzzle, and is arguably the most complicated part of the multi-layered solvers.

First, a sequence of blocks is calculated that can fill the hole in such a way that a character can walk from one room to the next. This is done by putting blocks in the hole such that the new block touches the previous block, or in case there is no previous block the edge of the hole. This means that in the local solving layer the block can be pushed over the previous blocks to the correct position. To give an example, in the puzzle in Figure 3 there are multiple possibilities to fill hole `B`. For instance, the block in room `1` could be used, or a combination of block `C` and the block in room `3`.

For each chosen block a sequence of sub-goals is generated. Each sequence contains a set of sub-goals which navigate a character from the room bordering the hole to the room containing the block, and one sub-goal with the goal to move the block to the hole.

Only one sub-goal is generated to move the block to the hole to avoid too much complexity. It can make solving the sub-goal more difficult for the local layer. However, this disadvantage is somewhat countered by the fact that the same path as from the hole to the block might be taken in reverse. In that case, some connections between the rooms might already be open or gone, as characters already interacted with those when traversing to the block.

When searching for a sub-goal sequence from the hole to a block, the solver could come across another hole. Since a block can only fill one hole, any of the blocks that are reserved for the current hole are not considered when calculating sub-goals for those holes.

4.3.2 Local layer

The local layer is quite simple as it mostly uses the same implementation as the simple solvers. However, there are a few additions such as only opening a limited number of states, deadlock detection and move ordering. To guide the local layer in solving a sub-goal, cost functions are used. Therefore, only the A* and the Fringe solvers in Section 4.2 could be used as a base, as BFS and DFS do not make use of cost functions.

Cost functions.

The local layer solves three kinds of sub-goals: moving a block out the way, rotating a turnstile out of the way, and moving a block towards a hole (which replaces the hole-sub-goals). For each of these sub-goals a different $h(n)$ is used for the cost function.

For moving a block out of the way, states which are generated by moving the target block are favoured most. Moving any obstacle further away from the goal-positions is also favoured, as that clears space around the target block.

When the sub-goal is to rotate a turnstile out of the way, turning the target turnstile is favoured most. If another obstacle is pushed, obstacles closer to the target room are favoured, as that might aid in freeing space around the target turnstile in order to rotate it.

For moving a block towards a hole, moving the target block towards the goal is highly favoured. For any other obstacle interaction, the closer to the hole, the better, as that could mean an obstacle in the front of the hole is being moved out of the way.

Opening a limited number of states.

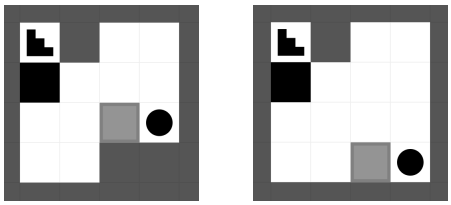
The first addition to the local layer is that it stops trying to solve a sub-goal after opening x states. x depends on the Manhattan distance between the goal-position of the current sub-goal and the position of the character used to transition to end-state of the last sub-goal. The further

the distance, the higher x will be. The position of the character was chosen because the transition to the end-state is always an interaction with the target obstacle of the last sub-goal. This means that the position of the character is somewhere near that obstacle.

When filling a hole, blocks often need to be moved through multiple rooms to reach it, as mentioned in Section 4.3.1. This means more states need to be visited than normal. Therefore, when the sub-goal is to move a block to a hole, x is multiplied by 3.

Deadlock detection.

Some basic deadlock detection is performed for sub-goals that move a block towards a hole. If the hole is north of the block, the block needs to be pushed northwards at some point. This is only possible if there is no wall south of the block or if the block cannot be moved west or east towards such a position. Figure 5a shows a situation which is not considered a deadlock, as the block can be moved west to create an empty space south of the block. Figure 5b, however, is considered a deadlock state so no children will be generated for it.



(a) No deadlock. (b) Deadlock.

Figure 5: A puzzle without and with a deadlock.

The Sokoban solvers in [2, 7, 8] also make use of deadlock detection. A pre-computed deadlock table is used which holds all combinations of walls and blocks in a 5x4 area that are a known deadlock. In contrast, the Kwirk solvers calculate whether there is a deadlock when the state is opened, as only one kind of deadlock is detected.

Move ordering.

When generating children for a state, the children are generated in a certain order. Because the frontier is mostly unordered in Fringe search, the first child generated is also the first child that is opened (if its cost is below the current threshold). Therefore, the order of the children can have influence on how fast a solution is found with Fringe search. So, the choice was made to implement a simple move ordering. The first children are always generated by state-transitions using the same obstacle as the obstacle defined in the sub-goal it is solving. After those, all other children are generated (without any specific ordering).

5. EXPERIMENTATION

Some experiments were performed to see what puzzles can be solved. The solvers were tested on the 30 puzzles from the game-mode *Going up?*. Each solver was given a maximum of 15 minutes solving time for every puzzle. The experiments were run on an Intel® Core™ i5-8250U CPU (at 1.60GHz). For each puzzle a new instance of Java was started which was given 2GB RAM (at 2400MHz) initially and was allowed to use up to 4GB of RAM.

First, the simple and the multi-layer solvers from Section 4 were tested. After that, some variations upon these were tested as well. There are many more variations that could be have tested than the ones mentioned in the next sub-sections, but due to time constraints this was not possible.

During prior testing, most solvers could solve 13 puzzles within a minute. This means that 17 puzzles could potentially not be solved within the time limit. Since six solvers plus variations had to be tested, testing could take up a lot of time if the time limit is too high. With the choice of 15 minutes, testing a single solver could take $17*15/60 = 4.25$ hours at maximum, which seemed reasonable.

Each puzzle-solver combination was only tested once due to time constraints. However, the solvers were implemented in such a way that states are always opened in the same order, so there is no randomness involved. This means that the results of all the solved puzzles are accurate. For unsolved puzzles there can be some inaccuracy as it depends on when the solver ran out of time or memory.

5.1 Heuristics for the simple solvers

As discussed in Section 4.2.4, the simple A* and Fringe solvers use a cost function that favours states which let characters explore positions that have not been explored in the last x moves. These heuristics were tested for $x = 10$, $x = 20$, and $x = 30$ to see what value for x is best.

A different cost function was also tested, which calculates the average Manhattan distance between the characters and the exit. The lower this average exit-distance, the more the state is favoured. The Manhattan distance was chosen instead of the actual minimal path distance, as it is less complicated to compute.

5.2 Only sub-goals for holes

After prior testing, it seemed that the simple solvers outperformed the multi-layer solvers. Therefore, a variation on the multi-layer solvers was tested which only generates sub-goals for holes and not for other obstacles. Sub-goals for holes were chosen as some guidance for filling holes is probably useful. This means less sub-goals will be generated, so the local layer will have less guidance, which could bring its behaviour closer to that of the simple solvers. This variation is based on the A* multi-layer solver.

5.3 No deadlock detection

To see the influence of the deadlock detection in Section 4.3.2 on the multi-layer solvers, a variation on the multi-layer A* solver without deadlock detection was tested.

5.4 Fewer new sequences and start-states

In Section 4.3.1 it was discussed that when a sequence of sub-goals cannot be solved, the solver tries to find a new sequence to solve. However, it might take multiple visits to solve a sequence, so a variation was tested where each sequence is tried 10 times before a new sequence is tried.

It can also take several tries before a solution for a sub-goal is found. Therefore, another variation was tested which tries to solve a sub-goal with the same start-state 10 times before trying a new start-state.

The last variation in this category is simply a combination of the two aforementioned variations. All these variations were based on the multi-layer A* solver.

6. RESULTS

In Table 2, 3, and 4, results that are better than the base solver are coloured light grey and results that are worse are coloured dark grey. $\Delta\mathbf{s}$ is the number of states opened by the variation divided by the number of states opened by the base solver. Only puzzles where the number of moves or sequences are different, or where $\Delta\mathbf{s}$ is greater than 1.1 or less than 0.9 are shown.

Table 1: Results of every base solver for every puzzle that could be solved by at least one solver.
m is the number of moves in the solution, **s** number of opened states, and **seq** the number of different sub-goal sequences explored. In the **solved** column, *MEM* means the solver ran out of memory and *SLOW* means the puzzle was not solved within 15 minutes. The best **m** and **s** for each puzzle are coloured light grey, and the worst are coloured dark grey.

	BFS			DFS			A*			Fringe			Multi-layer A*				Multi-layer Fringe			
puzzle	**solved**	**m**	**s**	**solved**	**m**	**s**	**solved**	**m**	**s**	**solved**	**m**	**s**	**solved**	**m**	**s**	**seq**	**solved**	**m**	**s**	**seq**
easy-1	✓	4	24	✓	15	40	✓	4	8	✓	4	5	✓	4	8	1	✓	4	18	1
easy-2	✓	4	39	✓	5	33	✓	4	18	✓	4	16	✓	4	16	1	✓	4	26	1
easy-3	✓	4	15	✓	6	11	✓	4	7	✓	6	10	✓	4	8	1	✓	4	17	1
easy-4	✓	5	22	✓	5	7	✓	5	19	✓	5	20	✓	5	20	1	✓	5	8	1
easy-5	✓	8	374	✓	34	262	✓	8	217	✓	8	174	✓	8	119	1	✓	8	124	1
easy-6	✓	9	46	✓	86	275	✓	9	26	✓	9	33	✓	9	28	1	✓	9	10	1
easy-7	✓	8	597	✓	51	96	✓	8	284	✓	8	175	✓	10	587	2	✓	10	437	2
easy-8	✓	9	374	✓	26	210252	✓	9	144	✓	9	158	✓	9	2931	5	✓	9	89	1
easy-9	✓	10	9633	✓	22	2607	✓	10	513	✓	10	458	✓	10	68	1	✓	11	384	3
easy-10	✓	7	263	✓	44	113	✓	7	70	✓	7	55	✓	9	73	1	✓	14	581	7
average-1	✓	10	7333	✓	165	4303	✓	10	3743	✓	11	2892	✓	10	14140	8	✓	11	17013	8
average-4	✓	109	3218	✓	181	1016	✓	109	2573	✓	109	2299	✓	109	1724	1	✓	109	1106	1
average-5	✓	30	346583	✓	2599	193692	✓	30	360502	SLOW	-	84684	✓	38	308747	1	✓	163	514944	1
hard-1	✓	25	15203	✓	59	11315	✓	25	14100	✓	25	7106	✓	25	7685	1	✓	27	4859	1
hard-3	✓	34	135064	✓	58	136681	✓	34	122213	✓	36	65592	✓	34	212599	4	✓	66	775553	4
hard-5	MEM	-	212170	✓	699	17379	✓	12	24644	SLOW	-	13196	MEM	-	85845	1000	MEM	-	106780	908

6.1 Base solvers

The results of the base solvers are shown in Table 1. The simple DFS and A* solvers solved the most puzzles, but all solutions of A* are shorter. BFS finds almost all solutions, but opened more states on average than A*, Fringe and multi-layer A*. Fringe and multi-layer A* are on par overall, though Fringe can solve one fewer puzzle. Multi-layer Fringe has worse results compared to the other solvers, but it does find shorter solutions than DFS.

14 puzzles from the game-mode *Going up?* could not be solved by any solver. BFS, DFS, A* and multi-layer A* run out of memory on these puzzles before the 15-minute mark, whereas Fringe exceeds the time limit instead. With multi-layer Fringe it differs per puzzle whether it runs out of memory or time. Fringe only opens a fraction of the states of the other solvers on these 14 puzzles. On these puzzles together Fringe opened 387.967 states, whereas A* opened 1.643.491 states and BFS and DFS even more.

6.2 Heuristics for the simple solvers

Changing x from 10 to 20 or 30 in the cost function that favours exploring new positions had almost no effects on the results of the simple A* and Fringe solvers.

Table 2: Results of using the average exit-distance as the cost function.

	Simple A*				Simple Fringe			
puzzle	**solved**	**m**	**s**	**Δs**	**solved**	**m**	**s**	**Δs**
easy-1	✓	4	10	1.250	✓	7	12	2.400
easy-2	✓	4	67	3.722	✓	4	29	1.813
easy-3	✓	4	7	1.000	✓	4	8	0.800
easy-4	✓	5	15	0.789	✓	5	14	0.700
easy-6	✓	9	21	0.808	✓	9	26	0.788
easy-7	✓	8	402	1.415	✓	8	242	1.383
easy-8	✓	9	91	0.632	✓	11	68	0.430
easy-9	✓	10	2684	5.232	✓	10	1737	3.793
easy-10	✓	7	42	0.600	✓	8	55	1.000
average-1	✓	10	2598	0.694	✓	11	1649	0.570
average-4	✓	109	2677	1.040	✓	111	2479	1.078
average-5	✓	30	257066	0.713	SLOW	0	78699	0.929
hard-3	✓	34	118515	0.970	✓	34	62981	0.960
hard-5	MEM	0	180630	7.330	SLOW	0	21277	1.612

As can be seen in Table 2, using the exit-distance as the cost function results in fluctuations in the number of opened states. However, this evens out when considering all puzzles together. It performs worse solution-wise, as hard-5 can no longer be solved by simple A* and the simple Fringe solver finds more longer than shorter solutions.

6.3 Only sub-goals for holes

Only generating sub-goals for holes results in opening fewer states on average, as can be seen in Table 3. Every solution now has the same number of moves as the simple A* solver. Another observation is that all of the puzzles with changed results contain more obstacles than holes. Furthermore, less sequences were explored.

Table 3: Results of only generating sub-goals for holes.

puzzle	**solved**	**m**	**s**	**Δs**	**seq**
easy-2	✓	4	18	1.125	1
easy-3	✓	4	7	0.875	1
easy-7	✓	8	282	0.480	1
easy-8	✓	9	141	0.048	1
easy-9	✓	10	508	7.471	1
easy-10	✓	7	70	0.959	1
average-1	✓	10	3674	0.260	1
average-5	✓	30	351744	1.139	1
hard-3	✓	34	173635	0.817	2
hard-5	✓	12	24371	0.284	1

6.4 No deadlock detection

The variation without deadlock detection only affected three puzzles (easy-5, hard-1, and hard-3), where the number of opened states nearly doubled.

6.5 Fewer new sequences and start-states

As can be seen in Table 4a, exploring fewer new sequences can solve one puzzle extra for which only one sequence was explored instead of 1000. It does open more states for one puzzle.

Table 4b shows the results of exploring fewer new start-states. The number of states opened is different for four puzzles, but those results even out when considering them together. One puzzle has fewer moves in the solution and one puzzle more, so that evens out as well.

The solver that combines both variations can solve one puzzle more, as can be seen in Table 4c. Overall, slightly fewer sequences were explored compared to both variations. This combination solved two puzzles with more moves and one puzzle with less.

Table 4: Results of variations regarding the frequency with which new sequences or start-states are tried.

(a) Fewer new sequences

puzzle	solved	m	s	Δs	seq
average-1	✓	10	18539	1.311	8
hard-5	✓	13	144	0.002	1

(b) Fewer new start-states

puzzle	solved	m	s	Δs	seq
easy-7	✓	8	486	0.828	2
easy-8	✓	9	24500	8.359	5
average-1	✓	10	10304	0.729	8
average-5	✓	43	384795	1.246	1
hard-5	MEM	-	79179	0.922	801

(c) Combination of both variations

puzzle	solved	m	s	Δs	seq
easy-7	✓	8	263	0.448	1
easy-8	✓	9	26117	8.911	5
average-1	✓	14	3116	0.220	5
average-5	✓	43	384795	1.246	1
hard-5	✓	13	72	0.001	1

7. DISCUSSION

Some remarks can be made about the results and about which solvers are the best for what situations.

7.1 Not enough memory

First of all, most solvers run out of memory on more difficult puzzles. None of the variations seemed to solve this problem, so it is probably caused by the fact that the state-space simply takes up too much space. For the 30 puzzles, the minimum size of a state was approximately 6kB and the maximum 31kB. This size heavily depends on the size of the room and the number of characters and obstacles. The fact that the Fringe solver does not run out of memory but opened a lot less states before the 15-minute mark, is probably caused by the problem mentioned in Section 4.2.3, where all nodes below the current threshold are checked every single time the threshold is raised. So, the Fringe solver probably spends a portion of its time on that, instead of on exploring states.

7.2 Base solvers

The fact that hard-5 could only be solved by two base solvers could be explained by the fact that it contains a lot of obstacles. This means a lot of states are possible, and BFS and Fringe just do not find the "correct" states soon enough. Also, this makes a lot of different sub-goal sequences are possible, which is why the multi-layer solvers could not solve the puzzle either, as most sequences are not solved on the first try. And with many sequences it takes quite long before the same sequence is tried again.

7.3 Heuristics for simple solvers

No direct explanation could be found why changing x from 10 to 20 or 30 had almost no effects, so more research is required to find out.

Variations with the exit-distance cost function probably find longer solutions for some puzzles because characters sometimes need to move further away from the exit temporarily, to move obstacles out of the way or to move towards blocks that will be pushed into a hole. Those actions are less favoured by the exit-distance cost function than by the cost function which favours exploring new areas.

7.4 Only sub-goals for holes

That the variation which only generates sub-goals for holes performs the same as the simple A* solver with regards to the solution size can be explained by the fact that less sub-goals are generated. This means that the local layer does more heavy lifting than usual, and the local layer, as mentioned in Section 4.3.2, is an adapted version of the simple A* solver. Why this variation only affected puzzles that mostly contain more obstacles than holes, can be explained by the fact that for these puzzles the number of possible sequences and sub-goals in a sequence are significantly reduced.

7.5 No deadlock detection

It makes sense that not using deadlock detection increased the number of states visited, as that means that deadlock states can generate children, which are also deadlocked, and thus more useless states are generated. However, it is peculiar that this only affected three puzzles. A possible explanation could be that for most puzzles a solution is found before any deadlock state is opened, but this cannot be said with any certainty.

7.6 Fewer new sequences and start-states

Opening fewer sequences.

The variation that sticks with the same sub-goal sequence for longer did not have a large effect on the results. This makes perfect sense for puzzles that already only found one sequence in the base solver. For the other puzzles this result could possibly be explained by the following: The first sub-goal sequence that is tried might need to open significantly more states to reach the goal-state than later sequences. The default multi-layer solver goes through multiple sequences and might find the solution in one of the later sequences, but not that many states are opened per sequence. Whereas in this variation the solution is found with the first sequence, but more states were explored for that one sequence. Therefore, the overall number of opened states could be comparable.

Exploring fewer start-states.

The same reasoning can be applied to the variation that explores the same start-states for longer. A lot of states might have to be opened to get from the first start-state to a solution for the sub-goal, whilst a later start-state might need to open less states to find the goal-state.

Combination of both variations.

The result of the combination of both variations showed a mix of the results of both variations, which makes sense.

7.7 Best solvers

Finding the shortest solution.

For finding the shortest solutions either the simple A* solver or the multi-layer A* solver that only generates sub-goals for holes could be used. They both found the shortest solutions out of all solvers, though there is a chance that even shorter solutions exist that no solver could find. The simple A* solver is much less complex than its multi-layered counterpart, so the simple A* solver is more advantageous. However, this claim is based on the results of 16 puzzles, so more testing is required to be more certain.

Finding a solution efficiently.

The number of states opened can be used as a measure of efficiency. For opening as few states as possible, no solver can be suggested with confidence. Each solver seems to optimise for different puzzles and there seems to be little to no pattern to it. However, it can be said with some certainty that multi-layered solvers should use deadlock detection, as it reduced the number of states visited for at least a few puzzles.

8. FUTURE WORK

The solvers in this paper could solve 16 out the 30 puzzles from the *Going up?* game-mode of Kwirk. This means that further improvements are required to solve the remaining 14 puzzles. This section highlights a few of those.

8.1 Memory used

As seen in Section 6.1, the solvers tend to run out of memory on larger puzzles. There are several potential solutions to this problem.

8.1.1 Efficient state-space representation

First of all, the state-space could be encoded in such a way that it takes up less space in memory. Using binary decision diagrams (BDDs) [3] to represent the state-space is one such technique.

8.1.2 Space efficient search algorithms

Another solution is to create a solver using a search algorithm that does not store the state-space or only a portion of it, such as the IDA* algorithm [9].

8.1.3 Pruning

Lastly, the number of states in the state-space could be reduced by pruning states that are not considered useful.

More deadlock detection.

Some minor pruning is already incorporated in the multi-layer solvers in the form of deadlock detection (see Section 4.3.2). However, due to time constraints, only one form of deadlock detection was incorporated. This means that more deadlock detection is possible. For example, when a 1x1 block is pushed into the corner between two arms of a turnstile, that turnstile will never be able to turn, as that 1x1 block cannot be pushed away. So, this would be a deadlock state for a sub-goal which involves that turnstile.

Relevance cuts.

Junghanns and Schaeffer [8] consider only exploring state-transitions that are "relevant", though a few exceptions are permitted. A relevant state-transition is a transition which is influenced by the last m transitions, where the definition of "influence" is domain dependent. This means that the search stays more local. It could be a useful addition to the local layer of the multi-layer solvers, which currently only use a cost function to encourage this behaviour.

8.2 Multi-character sub-goals

As mentioned in Section 4.3.1, the multi-layer solvers choose an arbitrary character to base the sub-goal sequence on. This has the drawback that the character closest to the exit could be chosen, which would mean that no sub-goals are generated that aid characters further away. However, sub-goals could be generated in such a way that the start-positions of all characters are taken into consideration. This might might make solving multi-character puzzles more efficient.

8.3 More testing

It is hard to conclude from the results what solvers optimise for what kind of puzzles, as the game-mode *Going up?* contains many different puzzles. The current solvers could be tested more extensively to find out what features are good for what kind of Kwirk puzzles. For example, puzzles only containing blocks could be tested with increasingly more blocks to see what solvers perform best on puzzles that only contain blocks. Also, more variations upon the solvers could be tested, such as different cost functions for the sub-goals.

8.4 FESS

A completely different approach to solving Kwirk puzzles could be taken as well. Shoham and Schaeffer created a solver for Sokoban called FESS [14] and claim that it is the first solver to solve all the Sokoban instances of a set of puzzles used in XSokoban [12]. This solver uses a so-called feature space to decide what state in the state-space to evaluate next. This technique could potentially be applied to solving Kwirk puzzles too.

9. CONCLUSION

In this study, multiple solvers for Kwirk were explored. The most a single solver could solve was 16 out of 30 puzzles of the game-mode *Going up?*. Most solvers ran out of memory when trying to solve the remaining puzzles. It could be concluded from the results in Table 1 and 3 that the A* solver and the multi-layer A* solver which only generates sub-goals for holes found the shortest solutions out of all solvers. Each solver seemed to optimise the number of opened states for different puzzles with little to no pattern as to which kind of puzzles. This means that there is no conclusive answer to what solver is the most efficient with regards to opening as few states as possible. However, deadlock detection, though only on three puzzles, did increase the efficiency.

Improvements could be made to solve more puzzles. To use less memory, a more efficient state representation, a space-efficient search algorithm, or pruning could be used. The generation of sub-goal sequences for multi-character puzzles could be improved. More experiments could be performed to find out what features are best for what kinds of puzzles. A solving technique called FESS could potentially be used as well.

10. REFERENCES

[1] Y. Björnsson, M. Enzenberger, R. C. Holte, and J. Schaeffer. Fringe Search: Beating A* at Pathfinding on Game Maps. *CIG*, 5:125–132, 2005.

[2] A. Botea, M. Müller, and J. Schaeffer. Using Abstraction for Planning in Sokoban. In J. Schaeffer, M. Müller, and Y. Björnsson, editors, *Computers and Games*, pages 360–375, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[3] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986.

[4] S. Edelkamp and D. Sulewski. External Memory Breadth-First Search with Delayed Duplicate Detection on the GPU. In R. van der Meyden and J.-G. Smaus, editors, *Model Checking and Artificial Intelligence*, pages 12–31, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum

Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[6] U. H. Hernandez-Belmonte, V. Ayala-Ramirez, and R. E. Sanchez-Yanez. Pushing boxes with a mobile robot in Sokoban-like scenarios. In *CONIELECOMP 2011, 21st International Conference on Electrical Communications and Computers*, pages 147–151, 2011.

[7] A. Junghanns and J. Schaeffer. Sokoban: Enhancing general single-agent search methods using domain knowledge. *Artificial Intelligence*, 129(1):219–251, 2001.

[8] A. Junghanns and J. Schaeffer. Sokoban: Improving the search with relevance cuts. *Theoretical Computer Science*, 252(1-2):151–175, 2001.

[9] R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[10] R. R. Leme, A. G. Pereira, M. Ritt, and L. S. Buriol. Solving Sokoban Optimally with Domain-Dependent Move Pruning. In *2015 Brazilian Conference on Intelligent Systems (BRACIS)*, pages 264–269, 2015.

[11] D. McDermott. Automated Planning. In *Encyclopedia of Computer Science*, page 117–119. John Wiley and Sons Ltd., GBR, 2003.

[12] A. Myers. XSokoban. `https://www.cs.cornell.edu/andru/xsokoban.html`, July 1997.

[13] V. Panteleev (CyberShadow). *DDD-Kwirk*. GitHub, Feb. 2010. `https://github.com/CyberShadow/DDD-Kwirk`.

[14] Y. Shoham and J. Schaeffer. The FESS Algorithm: A Feature Based Approach to Single-Agent Search. In *2020 IEEE Conference on Games (CoG)*, pages 96–103, 2020.

[15] A. Vark. *Puzzle Boy / Kwirk*. Hardcore Gaming 101, Aug. 2016. `http://www.hardcoregaming101.net/puzzle-boy-kwirk/`.

[16] Video Games. Tomate auf Trab: Kwirk. pages 68–68, 1991. German magazine. Issue 1. `https://www.kultboy.com/index.php?site=t&id=2965`.