Bachelor thesis
Computing Science

Radboud University

Analysis of Confined Blocks World

Author:
Julius Landsman
s1010919

First supervisor/assessor:
prof. dr., H. Zantema (Hans)
h.zantema@cs.ru.nl

Second assessor:
prof. dr., J.H. Geuvers
(Herman)
h.geuvers@cs.ru.nl

January 10, 2021

Abstract

Blocks World is a logical planning game. While it has been widely used in the field of artificial intelligence, it has also received some backlash due to the artificial nature of the game. In this thesis, we analyse a slightly more realistic variation of Blocks World, in an attempt to make Blocks World results more useful in the real world. Most notably, this variation puts a limit on the scale of Blocks World. We show how to verify whether problems of this variation are solvable, and provide an algorithm that solves these problems. We also prove that finding an optimal solution for this variation is NP-hard, putting this problem in the same complexity class as most other versions of Blocks World.

Contents

Chapter 1

Introduction

One of the most well-known planning domains in artificial intelligence is Blocks World, where the goal is to build one or more vertical stacks of blocks. Blocks World is an instance of a 'toy problem': a problem that does not have all the complexity of a real-world problem [13]. The simplicity of toy problems make them useful to compare the performance of different algorithms and heuristics, as well as being crucial in fields such as game design. However, toy problems such as Blocks World can also have a negative connotation [16]. One might think toy problems are too artificial to be useful in practice. Conversely, one might try to use the solution of a toy problem in the real world, without considering the limitations of this solution.

While there are several known versions of Blocks World, the bulk of the current research has been on the most basic version, Elementary Blocks World (EBW). In this paper we will present a different variation, Confined Blocks World (CBW), which is meant to be a slightly more realistic variation of Blocks World. The main difference between these versions is that, unlike EBW, CBW puts a limit on both the number of stacks of blocks and the height of these stacks. As the real world also has these limits, knowledge about CBW can help bridge the gap between Blocks World and real-world problems (for example: sorting boxes in a warehouse, which clearly has a limited capacity). We will provide an algorithm that can solve CBW problems, and we will show that finding an optimal solution for CBW problems is NP-hard (similar to EBW).

We will start by explaining how exactly Blocks World works, and what is currently known about it (chapter 2). We will then introduce and define Confined Blocks World, and present our main findings about this variation, such as our algorithm to solve problems and the hardness of finding an optimal solution (chapter 3). We will discuss existing literature related to this subject (chapter 4), and conclude our research (chapter 5).

Chapter 2

Background

2.1 What is Blocks World?

2.1.1 Elementary Blocks World

The most well-known and basic version of Blocks World is Elementary Blocks World (EBW) [11]. EBW is about cube-shaped blocks, which can be stacked on top of each other by placing a block precisely on top of another block. A vertical stack of one or more blocks is also called a 'tower' of blocks.
An EBW 'state' consists of a finite number of blocks, which are stacked into towers on top of a table.
A state can only be changed by moving a block from the top of a tower, either onto another tower or to the table (and thus creating a new tower). Only a single block can be moved at a time.
An EBW problem consists of two states: an initial state and a goal state. The main objective is to move the blocks, one at a time, from the initial state to the goal state. Ideally this is done in the least amount of moves possible. A simple EBW problem and its solution can be seen in figure 2.1.

Other properties of EBW are:

The table may hold an unbounded number of towers. This means all blocks can be on the table simultaneously.

Towers may consist of an unbounded number of blocks. This means all blocks can be in the same tower.

Every block is unique; a state cannot have more than 1 of the same block.

Every block has the same size. This means every block is either on top of a single other block or on the table.

The positioning of the towers on the table is irrelevant.

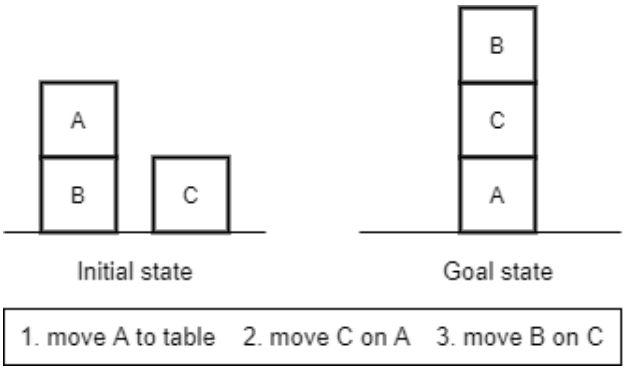

Figure 2.1: An EBW problem and a possible solution

2.1.2 Other versions of Blocks World

While EBW is the most widely used and well-known version of Blocks World, there are also several other versions that have been researched throughout the years. Some notable examples of these are:

The original Blocks World used by Winograd [18], which contained blocks of different sizes, colors and shapes, such as pyramids.

Varying Block-Size Blocks World (VBW) [11], an adaptation of EBW. As the name suggests blocks can have different sizes, and as a result multiple blocks can sit on top of a single other block.

Limited Table-Size Blocks World (LBW) [11], an adaptation of EBW where the table is only large enough to hold a certain number of towers.

Varying Block-Size, Limited Table-Size Blocks World (VLBW) [11], which combines the elements of both VBW and LBW.

The version of Blocks World used by Chenoweth [6]. This version is also similar to EBW, but here blocks are not necessarily unique. This means blocks can appear multiple times in a single problem.

2.2 Solving Blocks World

In this section we will broadly describe what is currently known about solving Blocks World problems. There are two main things to focus on: finding any solution, and finding an optimal solution. A solution is optimal if the number of blocks getting moved during the solution is the lowest possible; this means it is impossible to solve the problem in fewer moves than that of an optimal solution. Note that a single block getting moved multiple times counts as multiple moves. The amount of blocks in a given Blocks World problem generally gets indicated with n, so we will do the same in this section.

2.2.1 Finding any solution

Finding any solution to EBW (or any other variation where the table size is unbounded) is trivial: since the table is large enough to hold every block simultaneously, any problem can simply be solved by moving every block to the table first, and then building the goal state from the ground up. We can conclude the following:

> For any Blocks World version where the table is large enough to hold all blocks simultaneously, a solution of $2n$ moves can easily be found.

Finding a solution when the table size is limited is a bit more tricky. For LBW one can kind of simulate the table by first moving all blocks to two temporary towers. Whenever a block is needed for the goal state, move all blocks above that block to the other temporary tower, and then move the now-free block to its position in the goal state. This method can solve LBW problems in $O(n \log n)$ moves. More details can be found in [11].
Finding any solution for VLBW is a lot harder, but how hard exactly this is has not been explored.

2.2.2 Finding an optimal solution

Unlike finding any solution, finding an optimal solution of a Blocks World problem is very hard. This can be shown by analyzing the decision problem: 'Given a Blocks World problem and a positive integer L, is there a solution that solves the problem in L moves or less?'.
For EBW, this decision problem has been proven to be NP-complete [11]. Thus, finding an optimal solution for EBW problems is NP-hard. A similar result has been found for VBW and LBW. For VLBW however, the decision problem was just found to be NP-hard. For some VLBW problems the shortest solution has exponential length, so there is no NP-time algorithm that can find an optimal solution for all VLBW problems [11].
Chenoweth proved NP-completeness for the decision problem of his version, so finding an optimal solution is also NP-hard for Blocks World problems with duplicate blocks [6].

While finding an optimal solution for EBW is NP-hard, polynomial time algorithms have been found that come close to finding an optimal solution. [11] describes an algorithm that runs in $O(n^3)$ time, which solves CBW problems in no more than twice the length of the optimal plan. [16] improves the implementation of this algorithm to make it run in $O(n)$ time. It also introduces two other algorithms that run in $O(n)$ time, of which the most e□cient one finds a solution that is on average only 1.05 times as long as the optimal solution.

Chapter 3

Confined Blocks World

In this chapter we will analyse Blocks World with a limit on both the number of towers and the height of the towers. We call this variation of blocks world Confined Blocks World (CBW). We denote the maximum height of the towers as h, and the maximum mumber of towers as m.
Since there is a limit on the size of the table, finding any solution for a CBW problem is not trivial. On top of that, the limit on the tower height means the trick used for finding an LBW solution cannot be used here either. We will provide an algorithm that can solve CBW problems (if a solution is actually possible), and show how hard finding an optimal solution for CBW problems is.
We will denote a CBW state as a 'configuration'. Such a configuration can be seen as a grid with h rows and m columns, where every block has to be within that grid. An example can be seen in figure 3.1, where $h = m = 3$.

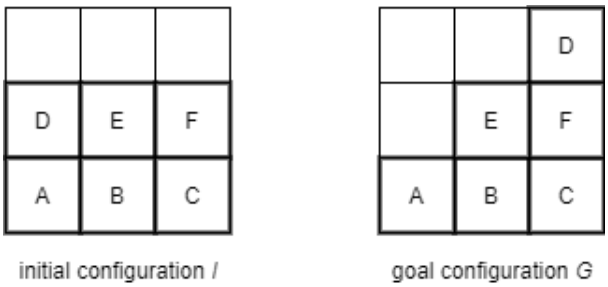

Figure 3.1: A simple CBW problem

3.1 Definitions

3.1.1 Configuration

As stated above, a CBW configuration is visualised as a grid with h rows and m columns. We define row 1 as the bottom row, row h as the top row, etc. Similarly we define column 1 as the left-most column, column m as the right-most column, etc. Position (x, y) indicates the slot at column x and row y.
A CBW configuration is built from a set of blocks B. 0 indicates an empty slot, so $0 \notin B$.

Definition 3.1 (Configuration)
We formally define a configuration C as a total function:

$$C : \{1, ..., m\} \times \{1, ..., h\} \to B \cup \{0\}, \text{where}$$
$$\forall (i, j) \in \{1, ..., m\} \times \{1, ..., h\} : (C(i, j) = 0 \wedge j < h) \Rightarrow C(i, j + 1) = 0$$

Note that every block in a CBW configuration is unique, meaning the part of function C with codomain B is injective.
We can then identify a block in 2 ways: by its position in the configuration it's in, or by its name. A block with name 'NAME' will be represented by [NAME]. For example, in figure 3.1 we have $I(3, 2)$ = [F], and $I(3, 3) = 0$.

Bear in mind that a side effect of this notation is that columns have a specific position in a configuration, which is not the case for towers in an EBW state.

3.1.2 Auxiliary functions

Auxiliary 3.1 (top)
The function $\texttt{top}(x, C)$ indicates the height of the highest block in column x in configuration C. In more formal terms:

$$\texttt{top}(x, C) = \max(\{j \in \{1, ..., h\} \mid C(x, j) \in B\})$$

This means $\forall i \in \{1, ..., m\} : (\texttt{top}(i, C) = h) \vee (C(i, \texttt{top}(i, C) + 1) = 0)$.
In figure 3.1 we get $\texttt{top}(3, I) = 2$, and $\texttt{top}(3, G) = 3$.

Auxiliary 3.2 (clear)
A block is said to be clear if it is free to be moved. In other words, a block is clear if there are no blocks on top of it. To reflect this we define the function $\texttt{clear}(b, C)$, which returns `True` if block b is clear in configuration C. For example, in figure 3.1 we get $\texttt{clear}([A], I) = \texttt{False}$, and $\texttt{clear}([D], I) = \texttt{True}$.

Auxiliary 3.3 (pos)
Lastly, we define the function pos(b, C), which returns the position of a block b in configuration C. So pos(b, C) = (x, y) when $C(x, y) = b$. For example, in figure 3.1 we have Pos([D], I) = (1, 2) and Pos([D], G) = (3, 3).

3.1.3 Moving blocks

Like every Blocks World variant, the only possible action for CBW is moving a block. A CBW configuration can therefore only be changed by moving a single block from one column to another. We call this a 'basic move', which we define as follows:

Definition 3.2 (move)
We define a basic move with the function move(x_S, x_D, C), which moves the highest block in column x_S to column x_D in configuration C. The move function is only possible if $C(x_S, 1)$ *2* B ^ $C(x_D, h)$ = 0. This function results in a new configuration C' where:
$C'(x_S, \text{top}(x_S, C)) = 0$,
$C'(x_D, \text{top}(x_D, C) + 1) = C(x_S, \text{top}(x_S, C))$, and
$C'(i, j) = C(i, j)$ for every other slot.
The problem in figure 3.1 can be solved in a single basic move. We get G = move(1, 3, I).

3.2 Solving CBW

A CBW problem can be indicated as (I, G), where I is the initial CBW configuration, and G is the goal configuration. The aim of a CBW problem is to transform I into G with a sequence of basic moves. Here, the maximum height h and table size m should be the same for I and G. For a problem to be solvable, clearly the set of blocks B should also be the same for I and G. We will therefore only look at CBW problems where this is indeed the case. The number of blocks in a problem will be indicated by n, so n = *jBj*.
One last assumption we make is that for all CBW problems $m > 2$ holds (more info on this can be found in section 3.2.5).

3.2.1 Solvability

However even with the rules described above, not every CBW problem is solvable. Consider the example in figure 3.2. Here it is impossible to ever move block [A] to a di□erent position, since the only way to make block [A] clear is by filling up the other two columns completely. This means the problem cannot be solved. As we can see, the number of blocks n is simply too high for the given dimensions m and h to solve to problem.

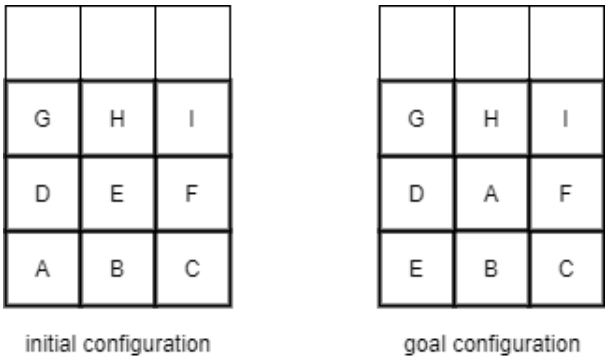

Figure 3.2: An unsolvable CBW problem

Lemma 1: If $n \quad h(m \quad 1)$ holds for a CBW configuration C, any block in C is able to be moved to a di□erent position eventually.

Proof: Let C be a CBW configuration for which $n \quad h(m \quad 1)$ holds, and $C(x, y)$ be the block we want to move. Let r be the number of blocks in column x, so r = top(x, C). Since $n \quad h(m \quad 1)$, and the total number of slots in any CBW configuration is $m \quad h$, we get that the number of empty slots in C is $m \quad h \quad n \quad m \quad h \quad h(m \quad 1) = h$. There are at least h empty slots in C, and there are $h \quad r$ empty slots in column x. This means there are at least $h \quad (h \quad r) = r$ empty slots in other columns. Since column x has r blocks, we can move every block in column x to a di□erent column, including block $C(x, y)$.

As we can see, if a CBW configuration has at least h empty slots, we can guarantee that every block can be moved. If we look at the problem in figure 3.2 again we find that $m = 3$, $h = 4$ and $n = 9$. We get that there are $m \quad h \quad n = 3$ empty slots in the configuration. And indeed $3 < h$, so some blocks cannot be moved to a di□erent position (specifically, the blocks in the bottom row). For this problem it holds that $n = h(m \quad 1) + 1$. From this and lemma 1 we can conclude that that $n = h(m \quad 1)$ is indeed the highest n can be to guarantee that every block can be moved.

If there are h empty slots in a configuration, we know that every block can at least be moved. But the question still remains if all CBW problems with h or more empty slots can be solved. To answer this we need to try finding an algorithm that solves all CBW problems for which $n \quad h(m \quad 1)$. We tried two approaches for this: building the goal configuration up column by column, and building the goal configuration up row by row.
The first approach works if all columns in the goal configuration have the same number of blocks, but it can lead to trouble if there are columns with h

blocks. Consider the problem in figure 3.3. The leftmost column is finished, so moving a block away from there would seem to be counterproductive. However, switching blocks B and D for the middle column is impossible without moving block E at some point.

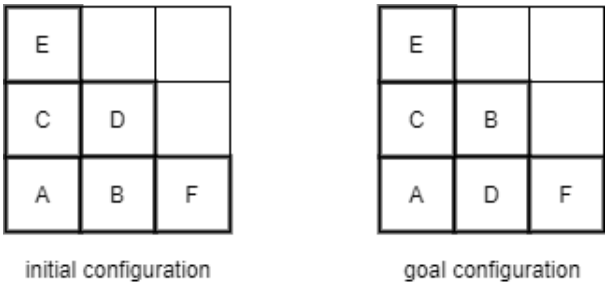

Figure 3.3: A tricky CBW problem

The second approach on the other hand does always work. We came up with an algorithm that uses this approach to solve any CBW problem for which $n \quad h(m \quad 1)$. This algorithm does this in no more than $3hn + 6n$ basic moves. For simplicity, we will denote this function as $F(h, n)$ for the rest of this chapter:

$$F(h, n) = 3hn + 6n$$

We can now prove the following theorem:

Theorem: For any CBW problem (I, G) where $n \quad h(m \quad 1)$, the initial configuration I can be transformed to the goal configuration G in no more than $F(h, n)$ basic moves.

To prove the above theorem, we define the algorithm for `solve`(I, G), which solves any CBW problem (I, G) where $n \quad h(m \quad 1)$. We will first define the functions `HMove` and `VMove`, which will be centerpieces of this algorithm. Both these functions use a series of basic moves to move a block from an initial position (x_S, y_S) to another position (x_D, y_D). `HMove` is used if $x_S \neq x_D$, and `VMove` is used if $x_S = x_D$. These functions could be combined into a single function, but for clarity we seperate them here.
For both of these functions $n \quad h(m \quad 1)$ is a requirement, so we assume this holds for the rest of this section.

3.2.2 HMove

HMove(x_S, y_S, x_D, y_D, C) moves block $C(x_S, y_S)$ to position (x_D, y_D), where $x_S \neq x_D$. This function can be used whenever the following properties hold:
- $C(x_S, y_S) \in B$
- $1 \leq x_D \leq m$
- $1 \leq y_D \leq h$
- $y_D \leq \text{top}(x_D, C) \Rightarrow y_D = \text{top}(x_D, C) + 1$

Our implementation of HMove with explanation can be found in appendix A.1.

Using $C' = \text{HMove}(x_S, y_S, x_D, y_D, C)$ transforms C into a configuration C' with the following properties:
For column x_D:
- $\forall j \in \{1, ..., y_D - 1\} : C'(x_D, j) = C(x_D, j)$ (no changes)
- $C'(x_D, y_D) = C(x_S, y_S)$

If $y_D \leq \text{top}(x_D, C)$:
- $\forall j \in \{y_D + 1, ..., \text{top}(x_D, C)\} : C'(x_D, j) = C(x_D, j - 1)$
- $\forall j \in \{\text{top}(x_D, C) + 1, ..., h\} : C'(x_D, j) = 0$

If $y_D = \text{top}(x_D, C) + 1$:
- $\forall j \in \{y_D + 1, ..., h\} : C'(x_D, j) = 0$

For column x_S:
- For $\forall j \in \{1, ..., y_S - 1\} : C'(x_S, j) = C(x_S, j)$ (no changes)

If $y_D \leq \text{top}(x_D, C)$:
- $C'(x_S, y_S) = C(x_D, \text{top}(x_D, C))$
- $\forall j \in \{y_S + 1, ..., h\} : C'(x_S, j) = C(x_S, j)$ (no changes)

If $y_D = \text{top}(x_D, C) + 1$:
- $\forall j \in \{y_S, ..., h - 1\} : C'(x_S, j) = C(x_S, j + 1)$
- $C'(x_S, h) = 0$

For every other column:
- No changes

An example where $y_D \leq \text{top}(x_D, C)$ and one where $y_D = \text{top}(x_D, C) + 1$ can be seen in figures 3.4 and 3.5 respectively.

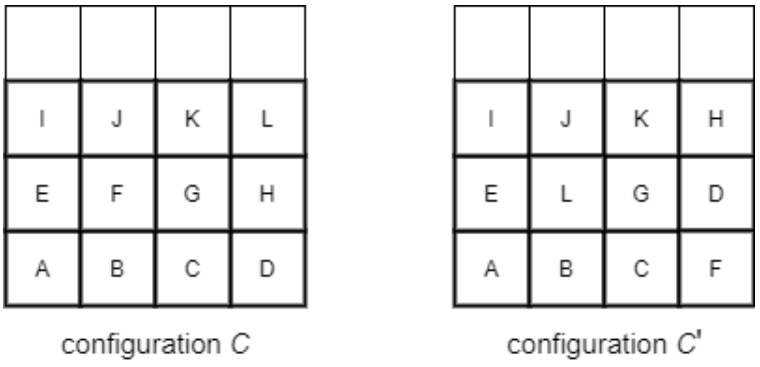

Figure 3.4: C^0 = HMove$(2, 2, 4, 1, C)$

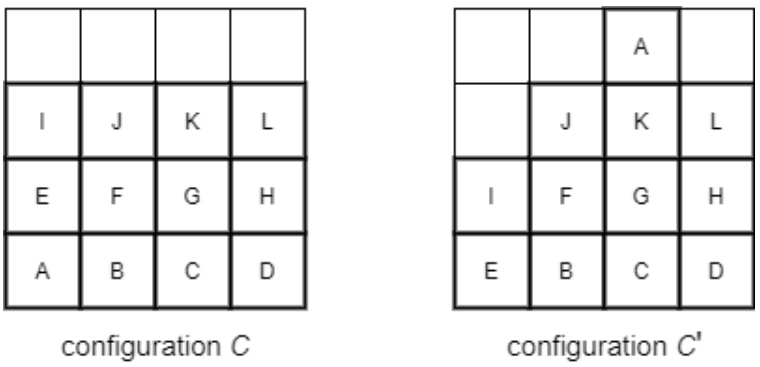

Figure 3.5: C^0 = HMove$(1, 1, 3, 4, C)$

In particular the properties in the following lemmas will be important to us. The proof of these lemmas can be found in appendix A.2.

Lemma 2: Executing HMove(x_S, y_S, x_D, y_D, C) takes at most 2(top(x_S, C) y_S) + 2(top(x_D, C) y_D) + 8 basic moves.

Lemma 3: When executing C^0 = HMove(x_S, y_S, x_D, y_D, C), the following properties hold for configuration C^0:
- $C^0(x_D, y_D) = C(x_S, y_S)$
- $\forall j \in \{1, .., y_D - 1\} : C^0(x_D, j) = C(x_D, j)$
- $\forall j \in \{1, .., y_S - 1\} : C^0(x_S, j) = C(x_S, j)$
- $\forall i \in \{1, .., m\} : (i \neq x_S \wedge i \neq x_D) \Rightarrow (\forall j \in \{1, .., h\} : C^0(i, j) = C(i, j))$

In other words, HMove(x_S, y_S, x_D, y_D, C) ensures not only that block $C(x_S, y_S)$ gets moved to its destination (x_D, y_D), but also that no blocks below y_S in column x_S get changed, no blocks below y_D in column x_D get changed, and no blocks in any other column get changed.

3.2.3 VMove

VMove(x, y_S, y_D, C) moves block $C(x, y_S)$ to position (x, y_D). This function can be used whenever the following properties hold:
- 1 x m
- y_S top(x, C)
- y_D top(x, C)

Our implementation of VMove with explanation can be found in appendix B.1.

Using C' = VMove(x, y_S, y_D, C) transforms C into a configuration C' with the following properties:
For column x:
If $y_S = y_D$:
- No changes

If $y_S > y_D$:
- $8j \in \{1, ..., y_D - 1\} : C'(x, j) = C(x, j)$ (no changes)
- $C'(x, y_D) = C(x, y_S)$
- $8j \in \{y_D + 1, ..., y_S\} : C'(x, j) = C'(x, j - 1)$
- $8j \in \{y_S + 1, ..., h\} : C'(x, j) = C(x, j)$ (no changes)

If $y_S < y_D$:
- $8j \in \{1, ..., y_S - 1\} : C'(x, j) = C(x, j)$ (no changes)
- $8j \in \{y_S, ..., y_D - 1\} : C'(x, j) = C'(x, j + 1)$
- $C'(x, y_D) = C(x, y_S)$
- $8j \in \{y_D + 1, ..., h\} : C'(x, j) = C(x, j)$ (no changes)

For every other column:
- No changes

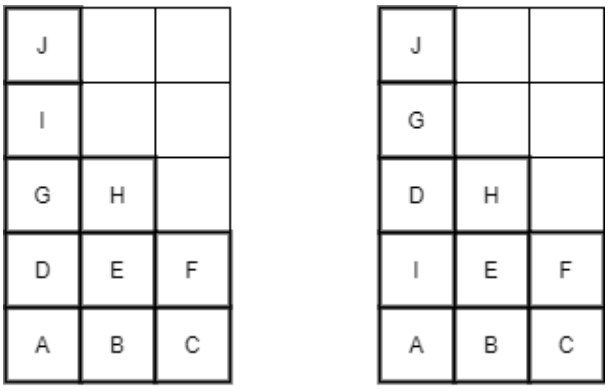

Figure 3.6: C' = VMove(1, 4, 2, C)

An example can be seen in figure 3.6, where $y_S > y_D$.

Again, in particular the properties in the following lemmas will be important to us. The proof of these lemmas can be found in appendix B.2.

Lemma 4: Executing VMove(x, y_S, y_D, C) where y_S y_D takes at most 2(top(x, C) y_D) + 6 basic moves.

Lemma 5: When executing C^0 = VMove(x, y_S, y_D, C) where y_S y_D, the following properties hold for configuration C^0:
- $C^0(x, y_D) = C(x, y_S)$
- $\forall j \in \{1, ..., y_D$ $1\} : C^0(x, j) = C(x, j)$
- $\forall i \in \{1, ..., m\} : i \neq x \Rightarrow (\forall j \in \{1, ..., h\} : C^0(i, j) = C(i, j))$

In other words VMove(x, y_S, y_D, C) moves block $C(x, y_S)$ to position (x, y_D), and if y_S y_D then in the resulting configuration no columns besides column x are changed, and in column x no blocks below y_D are changed.

3.2.4 Finding the solution

With these functions, we can now prove the main theorem:

Theorem: For any CBW problem (I, G) where n $h(m$ 1), the initial configuration I can be transformed to the goal configuration G in no more than $F(h, n)$ basic moves.

Proof:
We can apply the following algorithm to transform a CBW configuration I into G in no more than $F(h, n)$ basic moves. For these configurations it holds that n $h(m$ 1). C indicates the current configuration in the algorithm.

Algorithm solve(I, G):

```
1 C = I
2 for (j_D = 1, j_D  h, j_D + +):
3    for (i_D = 1, i_D  m, i_D + +):
4       if G(i_D, j_D) ∈ B:
5          (i_S, j_S) = pos(G(i_D, j_D), C)
6          if (i_S = i_D):
7             C = VMove(i_S, j_S, j_D, C)
8          else:
9             C = HMove(i_S, j_S, i_D, j_D, C)
10 return C
```

This algorithm builds up G from bottom to top; first every block that is in row 1 in G gets moved to its goal position in C, then row 2, etc.

Lemma 3 ensures that every time HMove(i_S, j_S, i_D, j_D, C) is used, no columns besides columns i_S and i_D change. On top of that, no blocks below position (i_S, j_S) and no blocks below position (i_D, j_D) are changed. Because the algorithm builds up the goal configuration from the bottom up, positions (i_S, j_S) and (i_D, j_D) cannot be below blocks that were moved to their goal position earlier during the algorithm. This means that lemma 3 essentially ensures that every block that is moved to its goal position during line 7 or 9 of the algorithm, won't get moved again by a HMove afterwards.

Similarly, since the goal configuration is built from the bottom up, we know every time VMove(i_S, j_S, j_D, C) is used in line 9 it holds that $j_S \quad j_D$. This means we can use lemma 5, which ensures that every time VMove(i_S, j_S, j_D, C) is used no columns besides column i_S change, and no blocks below position (i_S, j_D) are changed. Again position (i_S, j_D) cannot be be below blocks that were moved to their goal position earlier during the algorithm, since j_D goes up linearly during the algorithm. So lemma 5 ensures that all blocks that were moved to their goal position earlier cannot have their position changed again during an VMove either.

As a result of these 2 lemmas, we know that every block that gets moved to its goal position in line 7 or 9, will stay in that position forever. This means we can move all blocks to their correct position from the ground up without worrying about displacing them again afterwards. Once this is done for every position, we result in configuration G.

Maximum Basic Moves

There are 2 lines in the algorithm where basic moves get made: line 7 and line 9. One of these lines is executed every time $G(i_D, j_D)$ *2* B, which happens exactly n times.

During line 7, at most 2(top(i_S, C) $\quad j_D$) + 6 basic moves get made (lemma 4).

During line 9, at most 2(top(i_S, C) $\quad j_S$) + 2(top(i_D, C) $\quad j_D$) + 8 basic moves get made (lemma 2).

This means that in the worst case scenario, line 9 gets called n times, and line 7 zero times. In this scenario we end up with a total of

n (2(top(i_S, C) $\quad j_S$) + 2(top(i_D, C) $\quad j_D$) + 8) maximum basic moves.

To convert this term into a function of n, m and/or h we need to find the average values of top(i_S, C) $\quad j_S$ and top(i_D, C) $\quad j_D$ throughout the algorithm.

For top(i_S, C) $\quad j_S$ this is very hard to compute, as these values will both be completely dependent on the input CBW configurations. Therefore we will use the theoretical worst-case scenario: top(i_S, C) = h, and j_S = 1, so top(i_S, C) $\quad j_S \quad h \quad$ 1.

For $\texttt{top}(i_D, C) \quad j_D$ we can slightly improve this: Since j_D goes up linearly from 1 to h throughout the algorithm (line 2), we know that the average of $\texttt{top}(i_D, C) \quad j_D$ cannot be more than $0.5h$. This results in:
n (2($\texttt{top}(i_S, C) \quad j_S$) + 2($\texttt{top}(i_D, C) \quad j_D$) + 8)
n (2($h \quad 1$) + 2($0.5h$) + 8).

This means the total number of basic moves of the algorithm cannot be more than:

n (2($\texttt{top}(i_S, C) \quad j_S$) + 2($\texttt{top}(i_D, C) \quad j_D$) + 8)
n (2($h \quad 1$) + 2($0.5h$) + 8) =
n (3h + 6) =
3hn + 6n = $F(h, n)$ 2 $O(hn)$

3.2.5 Other observations

Optimality

One thing to note is that the number of basic moves that get made in the above algorithm is by no means the most optimal. As we will prove later finding an optimal solution is NP-hard, but even in the setting of polynomial algorithms several changes can be made to decrease the number of basic moves that get made. One example would be to constantly check for every clear block if it can be moved constructively (meaning it can be moved to its goal position, and every block below that goal position is also in its goal position) in a single move. If so, make the constructive move and then implement a way to ensure that block doesn't gets moved again.
However, implementing additions like this would clutter up both the algorithm and its respective proofs. Since one of the main aims of our algorithm is to be clear, we chose not to include these changes.

CBW problems with 1 or 2 columns

As stated before, we only considered CBW problems for which $m > 2$ holds. This is because CBW problems with only 1 or 2 columns are either trivial or impossible to solve. For m = 1 this is easy to see: since there is only one column, blocks simply cannot get moved. Therefore a problem (I, G) is only solvable if $I = G$.
If m = 2, blocks can only get moved back and forth between the two columns. Without a third column is it impossible to swap the positions of two blocks. This means the order of blocks in which they appear in the two columns can never be changed. Consider the example in figure 3.7. Block [A] will always be below block [C] in the left column, and always be above [C] in the right column. Without a third column, it is not possible to change this. As

a result, a problem (I, G) can only be solved if the order of the blocks in which they appear in I is the same as in G.

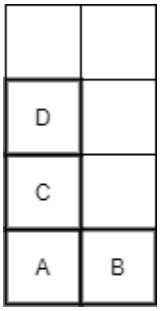

Figure 3.7: A CBW configuration where m = 2

Block Uniqueness

Since EBW was the original inspiration for this problem, we defined a CBW configuration to be one where no block can appear twice in a single configuration. The algorithm above assumes this and wouldn't work if a configuration had duplicate blocks.
However, we can still very easily transform a CBW problem where blocks appear multiple times into a problem that can be solved by the algorithm. We can do this by changing both the initial and goal configuration such that every block appears to be unique: For every block that appears multiple times in the configurations, add a di□erent identifier to each of those blocks (for example an integer). Say we have 6 yellow blocks, we now identify these blocks by yellow-1,...,yellow-6. After this is done every block is unique again, and the algorithm can solve the problem (although some very suboptimal moves might get made).

3.3 Finding an optimal solution

As stated in chapter 2, an optimal solution for Blocks World is a solution that makes the lowest number of basic moves possible. Finding an optimal solution for EBW problems (as well as most other versions of Blocks World) is known to be NP-hard. In this section we will show that this is also the case for CBW. To prove this, we follow the standard procedure of converting the optimization problem into a yes/no decision problem. We then prove NP-completeness for this decision problem.

3.3.1 NP-completeness of CBW

To prove NP-completeness, we need to convert the problem of finding an optimal solution for CBW into a yes/no decision problem. We then reduce a known NP-complete problem to an instance of a CBW problem. For this purpose we will use the FEEDBACK ARC SET problem. This problem is defined as follows:

> Given a directed graph (V, E) and an integer k, is there a set of edges F such that $jFj \quad k$ and the graph $(V, E \quad F)$ is acyclic?

This problem is known to be NP-complete [9].
We define *CBW SOLUTION* to be the following decision problem:

> Given a CBW problem (I, G) and an integer $L > 0$, can (I, G) be solved in L basic moves or less?

Our proof is based on the initial NP-completeness proof for EBW by Gupta and Nau [11], and as a result shares a lot of similarities.

Lemma 6: *CBW SOLUTION* is in NP.

Proof: Suppose S is a certificate for the problem, so S is a sequence of L or less basic moves. We can then simply perform every basic move in S, starting with configuration I. If the resulting configuration is equal to G, S is a correct solution. This can clearly be decided in polynomial time, so the problem is in NP.

Definition 3.3 (reduction)

We show that an arbitrary instance of FEEDBACK ARC SET can be transformed into an instance of a CBW problem in a polynomial number of steps. Let (V, E, k) be an instance of FEEDBACK ARC SET, where (V, E) is a directed graph, and k is the maximum number of edges that can be removed. We may assume without loss of generality that V is the set

of integers $\{1, 2, .., p\}$ for $p = |V|$. E can be seen as the set of edges $\{(v_1, w_1), (v_2, w_2), ..., (v_q, w_q)\}$ for $q = |E|$.
We then define the following instance (I, G, L) of *CBW SOLUTION*:

1. $L = 2q + k$
2. For both I and G:
   $m = p + 2q$
   $h = 2p$
3. I is defined as follows:

   For the leftmost p columns
   For every vertex $v \in V$, Let $I_v \subseteq E$ be the set of incoming edges to v, and $O_v \subseteq E$ be the set of outgoing edges from v. We can then say that $I_v = \{(i_1, v), (i_2, v), ..., (i_{|I_v|}, v)\}$, and $O_v = \{(v, o_1), (v, o_2), ..., (v, o_{|O_v|})\}$.

   We then have for every $v \in V$:
   $\forall j \in \{1, ..., |I_v|\} : I(v, j) = [v, \text{In}, i_j]$
   $\forall j \in \{1, ..., |O_v|\} : I(v, |I_v| + j) = [v, \text{Out}, o_j]$
   $\forall j \in \{|I_v| + |O_v| + 1, ..., 2p\} : I(v, j) = 0$

   An example can be seen in figure 3.8. Note that the order of the edges in I_v and O_v is irrelevant. The important thing is that all 'In' blocks are below all 'Out' blocks in I.

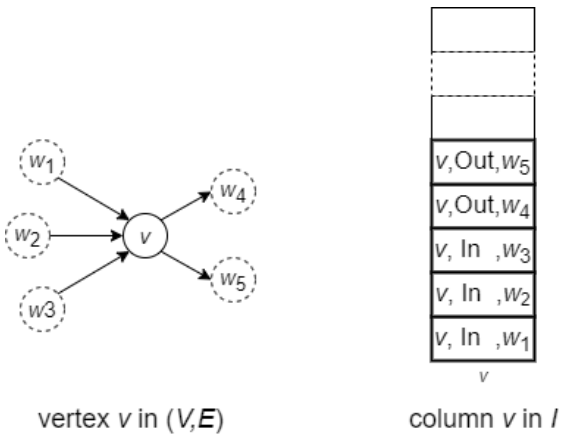

Figure 3.8: A vertex and its corresponding column in I

   For the rightmost $2q$ columns
   $\forall (i, j) \in \{p + 1, ..., p + 2q\} \times \{1, ..., 2p\} : I(i, j) = 0$.
   In other words, the rightmost $2q$ columns in I are empty.

4. G is defined as follows:

   For the leftmost p columns:
   $\forall(i,j) \in \{1,...,p\} \times \{1,...,2p\} : G(i,j) = 0$.
   So the leftmost p columns in G are empty.

   For the middle q columns:
   As stated before, E is the set of edges $\{(v_1,w_1),(v_2,w_2),...,(v_q,w_q)\}$.
   Then for all $r \in \{1,...,q\}$ we know $(v_r,w_r) \in E$, and we have the following:
   $G(p+r,1) = [w_r, \text{In}, v_r]$
   $G(p+r,2) = [v_r, \text{Out}, w_r]$
   $\forall j \in \{3,...,2p\} : G(p+r,j) = 0$.

   An example can be seen in figure 3.9.

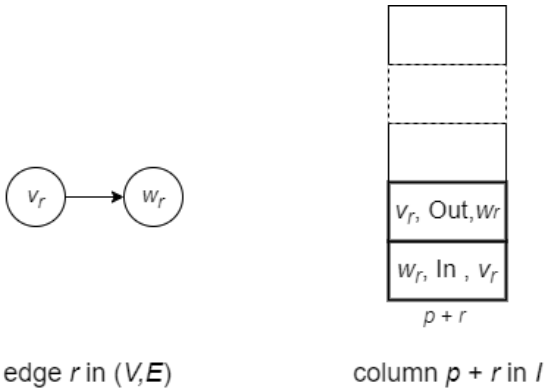

Figure 3.9: An edge and its corresponding column in G

   For the rightmost q columns:
   $\forall(i,j) \in \{p+q+1,...,p+2q\} \times \{1,...,2p\} : G(i,j) = 0$.
   So the rightmost q columns in G are empty.

An example of a reduction can be seen in figure 3.10. As we can see there are 2 blocks for every edge in E: One for the incoming part of the edge, and one for the outgoing part. This means the total number of blocks (n) for problem (I,G) is exactly $2q$.

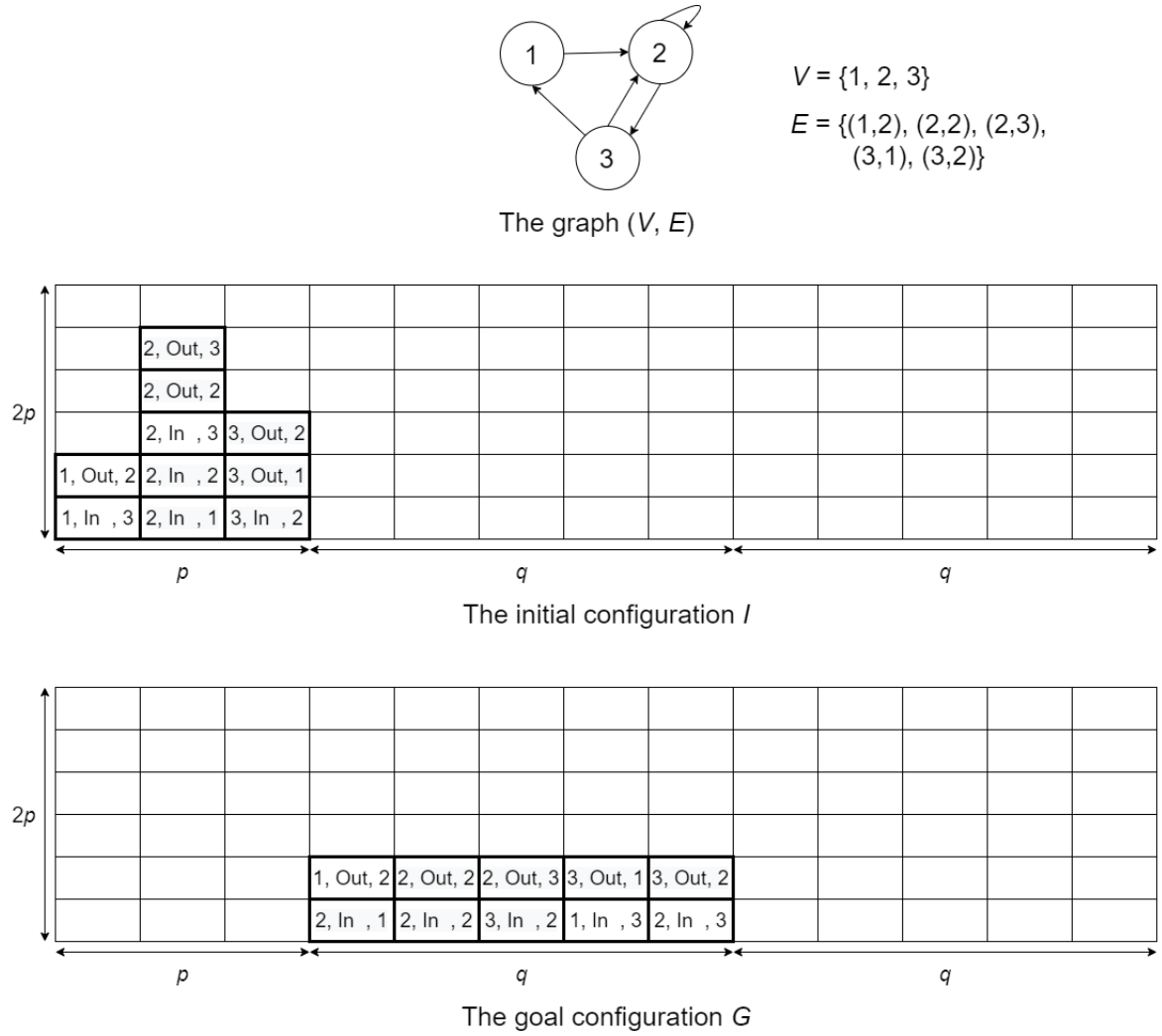

Figure 3.10: A graph (V, E) and the derived CBW problem (I, G)

Lemma 6: If (V, E) has no cycles, the optimal solution for (I, G) contains exactly $2q$ basic moves.

Proof: Assume (V, E) is an acyclic graph, and (I, G) is the corresponding CBW problem.
Since (V, E) is acyclic, we know it has a topological ordering $v_1, v_2, ..., v_p$ such that there can be no edge (v_i, v_j) *2* E for $i \quad j$. This means vertex v_p only has possible incoming edges, and no outgoing edges. From this it follows that column v_p in I only has blocks of the format $[v_p, \text{In}, w]$ for some w *2* V, and no blocks of the format $[v_p, \text{Out}, w]$. This means the 'In' blocks in this column aren't blocked by the 'Out' blocks from moving, and can simply all

be moved to their goal position. For example, let the top block of column v_p be $[v_p, \text{In}, w]$ for some $w \in V$. Let (w, v_p) be the rth edge in E. We can then perform $\texttt{move}(v_p, p + r, I)$ to move the block to its goal position in a single basic move. We can then repeat this for every 'In' block under this. Once this is done, every block in column v_p has been moved to its goal position in a single basic move.
We can then continue with vertex $v_{p\ 1}$. From the topological ordering we know that the only possible outgoing edge for this vertex is edge $(v_{p\ 1}, v_p)$, and hence the only possible 'Out' block in column $v_{p\ 1}$ is $[v_{p\ 1}, \text{Out}, v_p]$. But if this block exists then from the previous part we know that block $[v_p, \text{In}, v_{p\ 1}]$ has already been moved to its goal position. The goal position of $[v_{p\ 1}, \text{Out}, v_p]$ is on top of this block, so we can simply move it there in a single basic move. This unblocks the 'In' blocks of column $v_{p\ 1}$, which can again also be moved to their goal position in a single basic move. Once this is done, the blocks in the important section of column $v_{p\ 1}$ have all been moved to their goal position in a single basic move.
We can repeat this process for vertices $v_{p\ 2}, v_{p\ 3}, ..., v_1$. Eventually every block has been moved to its goal position in a single basic move. Configuration I contains 2 blocks for every edge in E (one for the incoming part and one for the outgoing part). Since these $2q$ blocks all get moved to their goal position in a single basic move, it takes $2q$ basic moves to solve this problem optimally. The optimal solution also can't be less than $2q$ basic moves, since none of the $2q$ blocks can be in their goal position in I.

Lemma 7: For every cycle in (V, E), one of the 'Out' blocks corresponding to the edges in this cycle has to be moved twice to solve (I, G).

Proof: Assume (V, E) contains a cycle $(v_1, v_2, ..., v_p, v_1)$.
This means $(v_1, v_2), (v_2, v_3), ..., (v_p, v_1) \in E$.
In the goal configuration G we then get $[v_1, \text{Out}, v_2]$ on top of $[v_2, \text{In}, v_1]$, $[v_2, \text{Out}, v_3]$ on top of $[v_3, \text{In}, v_2]$, ..., and $[v_p, \text{Out}, v_1]$ on top of $[v_1, \text{In}, v_p]$.
But in configuration I we have $[v_1, \text{Out}, v_2]$ above $[v_1, \text{In}, v_p]$, $[v_2, \text{Out}, v_3]$ above $[v_2, \text{In}, v_1]$, ..., and $[v_p, \text{Out}, v_1]$ above $[v_p, \text{In}, v_{p\ 1}]$.
This means every 'In' block that corresponds to an edge in the cycle is locked from getting moved to its final location in a single basic move. We can only resolve this by moving a single 'Out' block to a temporary location, which is what the right-most q columns are for. We can for example do this with block $[v_2, \text{Out}, v_3]$. Say (v_2, v_3) is the rth edge in E. We can move $[v_2, \text{Out}, v_3]$ to its temporary location in column $p + q + r$ with $C = \texttt{move}(v_2, p + q + r, C)$. This unlocks block $[v_2, \text{In}, v_1]$. Once $[v_2, \text{In}, v_1]$ is clear and can be moved to its final location, we can move block $[v_1, \text{Out}, v_2]$ on top of it, which in turn unlocks $[v_1, \text{In}, v_p]$, etc. Eventually block $[v_3, \text{In}, v_2]$ will be able to get moved to its final location with $C = \texttt{move}(v_3, p + r, C)$, at which point we

can finally move $[v_2, \text{Out}, v_3]$ on top of it with C = move($p + q + r, p + r, C$). As we can see, moving a single 'Out' block to its temporary location resolves the lock created by the cycle that its corresponding edge is part of.

Lemma 8: (I, G) has a solution of L basic moves or less i (V, E) has a feedback arc set of size k or less.

Proof: (!): Suppose (I, G) has a solution of L basic moves or less. Let P be the length of an optimal solution for (I, G), so $P \quad L$. We know from lemma 6 and 7 that the optimal solution moves all $2q$ blocks either once or twice. Hence, $P \quad 2q$. Let T be the set of blocks that are moved twice in the optimal solution of (I, G) (once to their temporary spot and once to their final location). Since $2q$ blocks need to be moved to their final position, we get that:

$$jTj = P \quad 2q \quad L \quad 2q = k$$

From the proof of lemma 7 we know that every block in T is associated with an edge that is part of a cycle, and is of the format $[x, \text{Out}, y]$ for some edge (x, y) *2* E that is in a cycle. Furthermore, from lemma 7 it follows that for every cycle at least one of the 'Out' blocks associated with the edges in the cycle needs to be moved twice. Thus, T contains blocks $[x_1, \text{Out}, y_1]$,...,$[x_r, \text{Out}, y_r]$ such that every cycle in (V, E) contains one of the edges $(x_1, y_1), ..., (x_r, y_r)$.
So (V, E) has a feedback arc set of size $r = jTj \quad k$.

( ): Suppose (V, E) has a feedback arc set $F = (x_1, y_1), ..., (x_r, y_r)$ such that $r \quad k$. From the proof of lemma 6 we know that (I, G) has $2q$ blocks that are not in their final position, and if (V, E) has no cycles all of these $2q$ blocks only need to get moved once. From the proof of lemma 7 we know that a cycle in (V, E) produces a lock in (I, G), and this lock can only be resolved by moving block $[v, \text{Out}, w]$ to a temporary position for any edge (v, w) which is in the cycle. If F is a feedback arc set of (V, E), it means that every cycle in (V, E) has an edge that is in F. This means that to solve (I, G) we can move all blocks $[x_1, \text{Out}, y_1]$,...,$[x_r, \text{Out}, y_r]$ to their temporary position whenever they become clear. Since the associated edges are part of all the cycles in (V, E), all the locks in (I, G) will get resolved by moving these blocks to their temporary positions. This means all other blocks in (I, G) can get moved to their final position in a single basic move. The result is a solution for (I, G) of $(2q \quad r) + 2r = 2q + r \quad 2q + k = L$ basic moves.

Theorem: *CBW SOLUTION* is NP-complete.

Proof: Lemma 8 shows that an arbitrary instance of FEEDBACK ARC

SET can be reduced to an instance of a CBW problem. This reduction runs in polynomial time, so this means *CBW SOLUTION* is NP-hard. Lemma 5 shows *CBW SOLUTION* is in NP. Thus we can conclude that *CBW SOLUTION* is NP-complete.

Corollary: Finding an optimal solution for CBW problems is NP-hard, but no worse.

Proof: If an optimal solution for CBW can be found, then for any L one can immediately tell whether a solution of L basic moves or less exists. So from the theorem, finding an optimal solution is NP-hard.

To prove finding an optimal solution is no worse than NP-hard, suppose we have an oracle which tells whether the answer is yes or no for an instance (I, G, L) of *CBW SOLUTION*. Then given any CBW problem (I, G), we can find the length L of the optimal solution for (I, G) by repeatedly guessing a value for L and asking the oracle for (I, G, L). Once we know L, we can figure out the first basic move of the optimal solution by repeatedly guessing a first move move(g_S, g_D, I) and asking the oracle to solve $(I^0, G, L \quad 1)$, where I^0 = move(g_S, g_D, I). Once we know the first basic move, we can figure out the rest of solution in the same way. This process makes at most polynomially many calls to the oracle.

3.3.2 CBW with pre-established restrictions

In the previous section we reduced a FEEDBACK ARC SET instance to a CBW problem. For this problem m = p + 2q, and h = 2p, and n = 2q.

But if we had specific pre-established restrictions on the CBW problem, this reduction would likely not work. Say we wanted to prove NP-completeness for all CBW problems where n = $h(m \quad 1)$. In section 3.2 we showed that these problems are always solvable. However, since this equation does not hold for the CBW problem resulting from the reduction, our current NP-completeness proof does not work. To solve this, we can adapt the current reduction as follows:

Below the current configurations of both I and G, we add a bunch of 'dummy rows'. These are rows that are completely filled with blocks, and are entirely equal in I and G. More formally, if row y is a dummy row, then:
$\forall i \in \{1, ..., m\} : I(i, y) \in B \wedge I(i, y) = G(i, y)$.
The names of the blocks in these rows are irrelevant, as long as they are equal in I and G.

In the case of n = $h(m \quad 1)$, we can achieve this property by adding 2p^2 + 4pq $\quad$ 2q $\quad$ 2p dummy rows below the original configurations. This results in m staying the same (p+2q), but h increasing from 2p to 2p^2 +4pq $\quad$ 2q. Furthermore, the bottom 2p^2 + 4pq $\quad$ 2q $\quad$ 2p rows are now completely filled

with blocks. If we add the original $2q$ blocks to this, the new total number of blocks is:

$$
\begin{aligned}
& n = \\
& (2p^2 + 4pq \quad 2p \quad 2q)(p + 2q) + 2q = \\
& (2p^2 + 4pq \quad 2q)(p + 2q) \quad (2p^2 + 4pq) + 2q = \\
& (2p^2 + 4pq \quad 2q)(p + 2q) \quad (2p^2 + 4pq \quad 2q) = \\
& (2p^2 + 4pq \quad 2q)(p + 2q \quad 1) = \\
& h(m \quad 1)
\end{aligned}
$$

So indeed, the property $n = h(m \quad 1)$ holds. An example of this can be seen in figure 3.11. Here the names of the dummy blocks are simply their coordinates.

This technique works because none of the blocks in the dummy rows have to be moved, since they are all already in their goal position. Since the original problem can be solved by moving every block either once or twice, the only reason to move a block from its goal position would be if moving that block is necessary to solve the problem (figure 3.3 depicts a situation like this). But in this case we know the original problem was solvable, which means the new problem is also solvable without moving any of the dummy blocks, so moving any dummy block would be suboptimal. This means the new CBW problem we just produced is essentially the same one as the original problem, but with the additional property that $n = h(m \quad 1)$.

Now let's say we not only have the requirement $n = h(m \quad 1)$, but also $m = h$. We can again achieve with a similar concept: on top of the previous adaptation, we now also add dummy columns. Since $h = 2p^2 + 4pq \quad 2q$ and $m = p + 2q$, the number of dummy columns we need to add is:
$(2p^2 + 4pq \quad 2q) \quad (p + 2q) = 2p^2 + 4pq \quad 4q \quad p$.
Since neither h nor the number of empty slots change, the property $n = h(m \quad 1)$ also still holds after adding these columns. An example of this can also be seen in figure 3.11.

As we can see NP-completeness can also be proven for CBW problems with pre-established restrictions, by adding an number of dummy rows and/or columns appropriate to the restriction.

Do keep in mind that this technique only works for restrictions where n, m and h are written in terms of each other, so e.g. a restriction of the form $m = C$ where C is a constant cannot be satisfied.

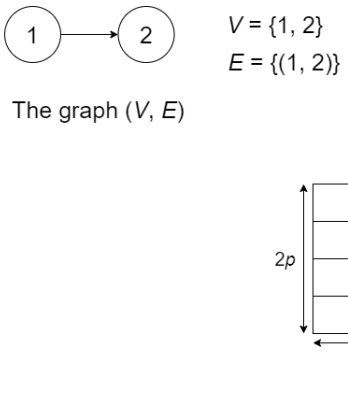

Figure 3.11: A simple reduction and two adaptations

Chapter 4

Related Work

Over the last few decades, Blocks World has been one of the most popular planning domains in artificial intelligence. Consequently, it has been researched several times before. Finding an optimal solution was first proven tp be NP-hard by S. V. Chenoweth [6]. However, in his version of Blocks World, blocks could appear multiple times in a given problem. The version of Blocks World with unique blocks, EBW, went on to become more popular, and was proven to be NP-hard in [10] and [11]. One of the authors of [11], J. Slaney, later revisits this paper with S. Thiébaux in [16], providing multiple improved algorithms to solve EBW problems. It also presents several other results about EBW, such as a method to generate random problem instances. Before this, Slaney and Thiébaux published several other papers on EBW, such as [14] and [15].

Multiple other planning systems for solving Blocks World have been described since, such as in [4], [12], [17] and [2]. Later, a deep learning-approach for solving Blocks World problems was presented in [1]. The results of this approach among others were then used in [5], which contains an analysis of Blocks World state space by evaluating the size of the search graph associated with Blocks World problems. [7] includes a very mathematical approach to Blocks World, as this paper formalizes Blocks World by giving a complete axiomatization of the domain.

Our research analyses a specific variation of Blocks World, where both the number of towers and height of the towers are limited. To the best of our knowledge, there is no public literature about this exact version of Blocks World. Nevertheless, general planning methods such as the ones the described in [3] and [8] can still provide to be useful in planning for this variation of Blocks World.

Chapter 5

Conclusions

In this thesis we have analysed Confined Blocks World, which is a slightly more realistic approach of Blocks World. We have found that solving a CBW problem is a lot more difficult than most other variations of Blocks World. While finding a solution for e.g. an EBW problem is always possible (assuming the blocks in the initial state and the goal state are the same) and trivial, this is not the case for CBW. Instead, there is only guaranteed to be a solution for CBW problems for which $n \quad h(m \quad 1)$ holds. In other words, the number of empty slots in a CBW problem needs to be at least as high as the limit on the height of the towers in that problem. We provided an algorithm that will solve any CBW problem for which this is indeed the case. It does this in $O(hn)$ basic moves.

We have also found that finding an optimal solution for a CBW problem is NP-hard, and the related decision problem is NP-complete. The proof for this can be adapted to fit CBW problems with pre-established restrictions on the dimensions and/or the amount of blocks. This puts CBW in the same complexity class as EBW, and most other variations of Blocks World.

Future research could focus on algorithms that can solve CBW problems more efficiently. As stated in section 3.2.5, the number of basic moves made in our algorithm can still be improved, as this was not one of our major goals. A more efficient algorithm can be made by implementing more heuristics, as well as improving the implementations of `CMove` and `HMove`.

Bibliography

[1] Edward Ayunts and Aleksandr Panov. Task Planning in "Block World" with Deep Reinforcement Learning. pages 3–9, 01 2018.

[2] A. Babiarz, T. Grzejszczak, A. Łęgowski, M. Niezabitowski, and J. Orwat. Planning and heuristics of assistant manipulator. In *2016 12th World Congress on Intelligent Control and Automation (WCICA)*, pages 2824–2828, 2016.

[3] Fahiem Bacchus and Froduald Kabanza. Using temporal logic to control search in a forward chaining planner. In *Proceedings of the 3rd European Workshop on Planning*, pages 141–153, 1995.

[4] A. Bieszczad and B. Pagurek. Neurosolver solves blocks world problems. In *Proceedings of International Conference on Neural Networks (ICNN'96)*, volume 2, pages 1215–1220 vol.2, 1996.

[5] A. Bădică, C. Bădică, I. Buligiu, L. I. Ciora, and F. Petcuşin. Quantifying Blocks World State Space. In *2020 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*, pages 1–7, 2020.

[6] Stephen V. Chenoweth. On the NP-Hardness of Blocks World. In *AAAI*, 1991.

[7] Stephen Cook and Yongmei Liu. A Complete Axiomatization for Blocks World. *Journal of Logic and Computation*, 13, 12 2001.

[8] Michael D Ernst, Todd D Millstein, and Daniel S Weld. Automatic SAT-compilation of planning problems. In *IJCAI*, volume 97, pages 1169–1176, 1997.

[9] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA, 1990.

[10] N. Gupta and D. Nau. Complexity Results for Blocks-World Planning. In *AAAI*, 1991.

[11] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2):223 – 254, 1992.

[12] B. Prasad. A Planning System for Blocks-World Domain. In *Proceedings ACS/IEEE International Conference on Computer Systems and Applications*, page 0059, Los Alamitos, CA, USA, jun 2001. IEEE Computer Society.

[13] S.J. Russell, S.J. Russell, P. Norvig, and E. Davis. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

[14] John Slaney and Sylvie Thiébaux. Blocks World Tamed–Ten thousand blocks in under a second. 1995.

[15] John Slaney and Sylvie Thiébaux. Linear time near-optimal planning in the blocks world. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1208–1214, 1996.

[16] John Slaney and Sylvie Thiébaux. Blocks World revisited. *Artificial Intelligence*, 125(1):119 – 153, 2001.

[17] M. Toussaint, N. Plath, T. Lang, and N. Jetchev. Integrated motor control, planning, grasping and high-level reasoning in a blocks world using probabilistic inference. In *2010 IEEE International Conference on Robotics and Automation*, pages 385–391, 2010.

[18] Terry Winograd. Understanding natural language. *Cognitive Psychology*, 3(1):1 – 191, 1972.

Appendix A

HMove

A.1 Algorithm

In this section we will show our implementation of HMove. For clarity we divide our algorithm in 5 steps. To help visualise this process, we will show the result after every step for the HMove of figure 3.4 (C' = Hmove$(2, 2, 4, 1, C)$), which can be seen in figure A.1. The values of the variables that are known after every step can also be seen.
Note that we assume $n \quad h(m \quad 1)$ ^ $m >$ 2 holds, since this is required for HMove to work.

Definition: Say block $C(x_S, y_S)$ *2* B needs to be moved to position (x_D, y_D), where x_S ~~6=~~ x_D. We define HMove(x_S, y_S, x_D, y_D, C) in 5 steps:

Step 1

```
1 List L1 = []
2 T1 = C
3 while (top(x_S, T1) > y_S):
4    p = any column for which p ~= x_S ^ top(p, T1) < h
5    L1.append(p)
6    T1 = move(x_S, p, T1)
7 T2 = T1
```

The purpose of this step is to make block $C(x_S, y_S)$ clear. It does so by moving every block above it to a di□erent column, so every block above position (x_S, y_S) gets moved away. This results in a temporary configuration T_2, for which clear$(C(x_S, y_S), T_2)$ = True.
From the proof of lemma 1 we know every block in a column can get moved out of that column if $n \quad h(m \quad 1)$. Thus, we know every block above position (x_S, y_S) can get moved away, so executing these lines is always possible. List $L1$ keeps track of the order of the columns to which the blocks of column x_S were moved. This will be used later.

Step 2

```
8  List L2 = []
9  PosChange = False
10 while (top(x_D, T_2)    y_D):
11    if (9q 2 f1, .., mg : i 6= x_S ^ i 6= x_D ^ top(i, T_2) < h):
12       L2.append(q)
13       T_2 = move(x_D, q, T_2) < h
14    else:
15       r = any column for which r 6= x_S ^ r 6= x_D
16       PosChange = True
17       L2.append( 1)
18       T_2 = move(r, x_D, T_2)
19       T_2 = move(x_S, r, T_2)
20       T_2 = move(x_D, x_S, T_2)
21       while (top(x_D, T_2)    y_D):
22          L2.append(x_S)
23          T_2 = move(x_D, x_S, T_2)
24 T_3 = T_2
```

The purpose of step 2 is to free up column x_D so that block $C(x_S, y_S)$ can get moved to its goal position there. To do so it moves every block that is at position (x_D, y_D) or above to a di□erent column.

We move these blocks to any column q for which $q \neq x_S \wedge q \neq x_D \wedge \texttt{top}(q, T_2) < h$, so any column with empty space that isn't x_S or x_D. However, since we now need to keep 2 columns open, we cannot ensure that such a column q always exists.

If we get the situation that such a column does not exist, the h or more empty slots of the configuration are all in columns x_S and x_D. In this case we enter the 'else' clause in line 14, and we need to use column x_S to move out the remaining blocks of column x_D. However, we also need to make sure that block $C(x_S, y_S)$ stays clear.

To accomplish this we use any column r for which $r \neq x_S \wedge r \neq x_D$. Since we know all the empty slots are in column x_S or x_D, currently $\texttt{top}(r, T_2) = h$. We then perform the three basic moves in lines 18-20. This swaps block $C(x_S, y_S)$ and the highest block of column r. Since block $C(x_S, y_S)$ is now at height h, it is clear, and can thus be moved later.

Finally, we can perform $T_2 = \texttt{move}(x_D, x_S, T_2)$ until $\texttt{top}(x_D, T_2) = y_D \quad 1$. Since every other column is full and there are at least h empty slots, this is always possible. Again, list $L2$ keeps track of the order of the columns to which the blocks of column x_D were moved. The boolean variable PosChange keeps track of whether block $C(x_S, y_S)$ got moved to column r or stayed in column x_S. Line 17 enables us to know later when exactly the else clause

of line 14 got entered, which enables us to move the blocks back later in the right way.
We end up in a configuration T_3 for which:
clear($C(x_S, y_S), T_3$) = True ^ $T_3(x_D, y_D)$ = 0 ^ (y_D = 1 _ $T_3(x_D, y_D$ 1) 2 B).

Step 3:

```
25 if (PosChange):
26    T4 = move(r, xD, T3)
27 else:
28    T4 = move(xS, xD, T3)
```

This step moves block $C(x_S, y_S)$ to the destination position (x_D, y_D).
If the else clause in line 14 got entered, block $C(x_S, y_S)$ is the highest block in column r in T_3, and if not it is still the highest block in column x_S. As a result, we move block $C(x_S, y_S)$ from one of those columns to column x_D, depending on the value of PosChange. Because of the properties of T_3, the result is a configuration T_4 for which $T_4(x_D, y_D) = C(x_S, y_S)$.

Step 4:

```
29 for (a = size(L2)   1; a > 0; a      ):
30    if (L2[a] =   1):
31       T4 = move(xS, r, T4)
32    else:
33       T4 = move(L2[a], xD, T4)
34 if (yD    top(xD, C)):
35    if (L2[0] =   1):
36       T4 = move(xS, r, T4)
37       T5 = move(xD, xS, T4)
38    else:
39       T5 = move(L2[0], xS, T4)
40 else:
41    if (L2[0] =   1):
42       T5 = move(xS, r, T4)
43    else:
44       T5 = move(L2[0], xD, T4)
```

This step moves the original blocks of column x_D in C back to column x_D.
These are the blocks that got moved during step 2. Since list $L2$ kept track of the order of the columns to which these blocks were moved, we can move these blocks back in the reverse of this order. This means the blocks in column x_D in the resulting configuration T_5 are in the same order as they are in C.

If we come across the value ' 1' in $L2$, we know this is the point at which block $C(x_S, y_S)$ and the original top block of column r were swapped. The block that was at position (r, h) before line 18 (let's call this block b) is currently at the top of column x_S. We know this because block b got moved to column x_S in line 20, and because of list $L2$ every block that got moved to column x_S since has been moved back to column x_D at this point. This means block b is currently clear in column x_S, and it can be moved back to column r (of which we know that the top block is at height h 1 at this point), where it originally came from.
Since an extra block has been moved to column x_D (this block of course being $C(x_S, y_S)$), the number of blocks of column x_D would increase by 1 in the final configuration. However, if top(x_D, C) = h the new number of blocks in column x_D would be h + 1, which is impossible. Because of this we move the last block, that is block $C(x_D, \text{top}(x_D, C))$, to column x_S instead of x_D. This way top(x_S, C) = top(x_S, C') and top(x_D, C) = top(x_D, C'). This is done in line 37 or 39, depending on whether -1 was the last item in $L2$. This move is always possible since block $C(x_S, y_S)$ has been moved away from column i permanently, which means top$(x_S, T_4) <$ top(x_S, C) h.
If y_D = top(x_D, C) + 1 however, we need every block in column x_D. In this case we can't move block $C(x_D, \text{top}(x_D, C))$ to column x_S, so line 34 ensures the move doesn't happen (which is fine since in this case we know top(x_D, C) + 1 = y_D h).

Step 5:

```
45 for (a = size(L1)  1; a  0; a    ):
46    T5 = move(L1[a], xS, T5)
47 C' = T5
48 return C'
```

This step moves all original blocks of column x_S in C back to column x_S in the current configuration. We do this the same way as in step 4. Since column x_S lost 1 block $C(x_S, y_S)$ and gained at most 1 block $(x_D, \text{top}(x_D, C))$, moving these blocks back is always possible. After this step we result in the final configuration C', with the properties outlined in section 3.3.2.

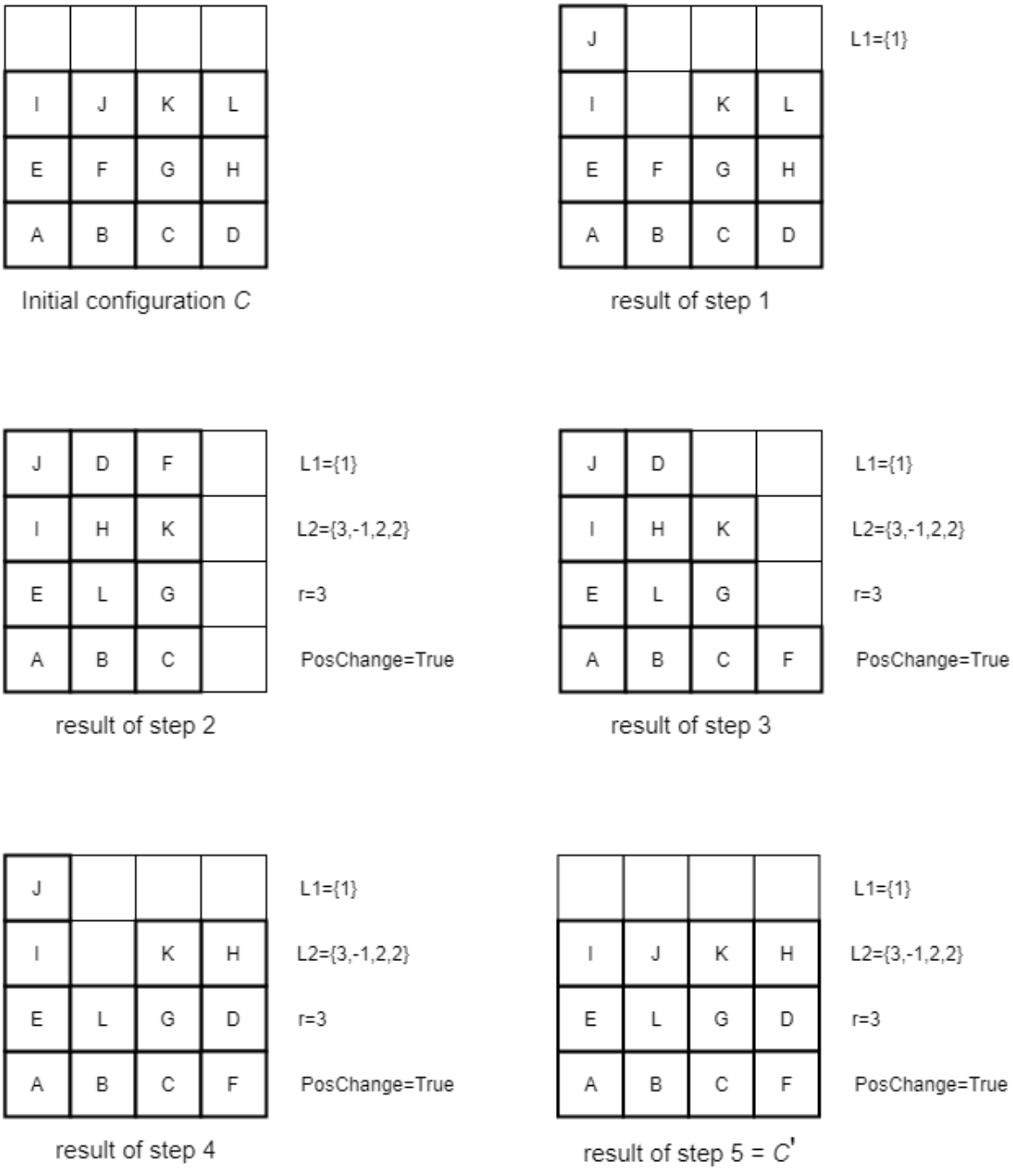

Figure A.1: C^0 = HMove$(2, 2, 4, 1, C)$ in 5 steps

A.2 Proofs

Lemma 1: Executing HMove(x_S, y_S, x_D, y_D, C) takes at most 2 (top(x_S, C) y_S) + 2 (top(x_D, C) y_D) + 8 basic moves.

Proof:
Step 1 only executes move in line 6. Line 6 is executed top(x_S, C) y_S times, so step 1 makes top(x_S, C) y_S basic moves.
For step 2 there are 2 cases. If the else-clause in line 14 never got entered, only the move in line 13 gets executed. This line gets executed top(x_D, C) y_D + 1 times (+1 because position (x_D, y_D) itself needs to be empty).
In case the else-clause in line 14 does get entered, the move in lines 13, 18, 19, 20 and 23 gets executed. Lines 18, 19 and 20 all get executed once. Lines 13 and 23 combined get executed the same number of times as line 14 in the first case, so top(x_D, C) y_D + 1 times. So in total top(x_D, C) y_D + 4 basic moves are made during step 2 in the second case.
Step 3 executes move once.
Step 4 executes one move for every item in $L2$. size($L2$) got increased by 1 every time line 12 or 22 got executed, plus possibly one time in line 17. So size($L2$) = top(x_D, C) y_D + 1 or size($L2$) = top(x_D, C) y_D + 2. If the if-clause in line 35 gets entered, move gets executed twice for a single item in $L2$ one time. This means step 4 executes move at most top(x_D, C) y_D + 3 times.
Similarly, step 5 executes move size($L1$) times. The size of $L1$ increases by 1 every time line 5 gets called. So in total step 5 executes move top(x_S, C) y_S times.

Overall, the move command gets executed at most top(x_S, C) y_S + top(x_D, C) y_D + 4 + 1 + top(x_D, C) y_D + 3 + top(x_S, C) y_S =
2 (top(x_S, C) y_S) + 2 (top(x_D, C) y_D) + 8 times.

Lemma 2: When executing C' = HMove(x_S, y_S, x_D, y_D, C), the following properties hold for configuration C':
- $C'(x_D, y_D) = C(x_S, y_S)$
- $\forall j \in \{1, .., y_D - 1\} : C'(x_D, j) = C(x_D, j)$
- $\forall j \in \{1, .., y_S - 1\} : C'(x_S, j) = C(x_S, j)$
- $\forall i \in \{1, .., m\} : (i \neq x_S \wedge i \neq x_D) \Rightarrow (\forall j \in \{1, .., h\} : C'(i, j) = C(i, j))$

Proof:
$\underline{C'(x_D, y_D) = C(x_S, y_S)}$
The property $T_4(x_D, y_D) = C(x_S, y_S)$ gets achieved in line 26 or 28. After this, there are 2 possible lines where blocks get moved away from column x_D. The first one is in line 37. However, since -1 cannot be the only element of $L2$, we know at least one other block got moved to column x_D (in line

33) before this line gets executed. This means block $C(x_S, y_S)$ is never the block that gets moved away during line 37.
The second possibility is during line 46, if $L1[a] = x_D$. But this line only moves blocks back that initially got moved away during line 6. We know block $C(x_S, y_S)$ never gets moved in line 6, which means it doesn't get moved in line 46 either.
It follows that after the property gets achieved in line 26 or 28, block $C(x_S, y_S)$ never gets moved again. This means the property $C'(x_D, y_D) = C(x_S, y_S)$ holds for the final configuration C'.

$\forall j \in \{1, .., y_D - 1\} : C'(x_D, j) = C(x_D, j)$
Blocks get moved out of column x_D during step 2 of the process. However, this only happens as long as $\text{top}(x_D, T_2) < y_D$. This means that no blocks at row $y_S - 1$ or below get moved out of the column, so their positions do not change.
We showed above that block $C(x_S, y_S)$ does not get moved again after step 3, at which point it is in (x_D, y_D). This means the blocks below it do not get moved either, and thus the blocks below y_D in column x_D never get moved during the algorithm. This means the property holds for C'.

$\forall j \in \{1, .., y_S - 1\} : C'(x_S, j) = C(x_S, j)$
Blocks get moved out of column x_S during step 1 of the process. This happens as long as $\text{top}(x_S, T_1) > y_S$, so no blocks at row y_S or below get moved out of the column during this step. The block at position (x_S, y_S) might get moved out in line 19, but no blocks get moved out of column x_S after this. This means the blocks in column x_S at row $y_S - 1$ or below don't get moved at all, so their positions never change. Blocks can also get moved out of column x_S during step 4, but these are only the blocks that got moved to the column first in line 20 or 23. This means the property holds for the final configuration C'.

$\forall i \in \{1, .., m\} : (i \neq x_S \wedge i \neq x_D) \implies (\forall j \in \{1, .., h\} : C'(i, j) = C(i, j))$
For this property we need to show that no columns besides x_S and x_D have changed. Lists $L1$ and $L2$ ensure that every block that gets moved to one of such columns during step 1 or 2, gets moved back from that column during step 5 or 4 respectively. The only other time in the algorithm that a different column gets changed is in lines 18-20. The top block of column r gets swapped with block $C(x_S, y_S)$. However, the initial top block of column r gets moved back to column r in line 31, 36 or 42. If it got there initially during step 1 or 2, it will still get moved back to its original column in step 4 or 5, because list $L2$ keeps track of when exactly the block got swapped. This means in the final configuration C' no columns besides column x_S and x_D have changed, so the property holds.

Appendix B

VMove

B.1 Algorithm

In this section we will show our implementation of VMove. Since this process is similar to HMove, our explanation of the algorithm will be shorter. Note that again we assume $n \quad h(m \quad 1) \wedge m > 2$ holds, since this is required for VMove to work. We define VMove as follows:

Definition: Say block $C(x, y_S)$ *2* B needs to be moved to position (x, y_D). There are 3 cases for VMove(x, y_S, y_D, C) (for one of which we will omit the algorithm):

case 1

```
1 if (y_S = y_D):
2    return C
```

This case is very straightforward. If $(x, y_S) = (x, y_D)$, we don't need to move any blocks.

case 2

```
3 if (y_S > y_D):
4    T = C
5    List L = []
6    while (top(x, T)    y_S):
7       p = any column for which p 6= x ^ top(p, C) < h
8       if (top(x, T) 6= y_S):
9          L.append(p)
10      T = move(x, p, T)
11   cb = pos(C(x, y_S), T)[0]
```

```
12  while (top(x, T)    y_D):
13    if (9i 2 f1, .., mg : i 6= x ^ i 6= cb ^ top(i, T) < h):
14      q = any column for which q 6= x ^ q 6= cb ^ top(x, T) < h
15      L.append(q)
16      T = move(x, q, T)
17    else:
18      r = any column for which r 6= x ^ r 6= cb
19      PosChange = True
20      L.append(-1)
21      T = move(r, x, T)
22      T = move(cb, r, T)
23      T = move(x, cb, T)
24      while (top(x, T)    y_D):
25        L.append(cb)
26        T = move(x, cb, T)
27  if (PosChange):
28    T = move(r, x, T)
29  else:
30    T = move(cb, x, T)
31  for (a = size(L)    1; a    0; a      ):
32    if (L2[a] =    1):
33      T = move(cb, r, T)
34    else:
35      T = move(L[a], x, T)
36  return T
```

The algorithm for this case is very similar to the algorithm of HMove. We first move out every block that is at position (x, y_S) or higher (lines 6-10), which is possible because there are at least h free spots in the configuration. The result is that block $C(x, y_S)$ is currently the highest block in a column cb where $cb \neq x$. We then move out every block that is at position (x, y_D) or higher to columns that aren't x or cb. Once that isn't possible anymore, we swap block $C(x, y_S)$ with the top block of column r, similar to step 2 of HMove. We then move the rest of the blocks to column cb (lines 12-26). This is again possible because there are h free spots in the configuration, and every column besides x and cb has no empty slots at this point. Now clear$(C(x, y_S), T)$ = True, and $T(x, y_D) = 0$. This means we can move block $C(x, y_S)$ to position (x, y_D) (lines 27-30). Finally, we move every block back into column i, similar to step 4 of HMove. We know that top(x, C) = top(x, C') (since no extra blocks get moved to any column), so we don't have to worry about moving the last block to another column.
The final result is a configuration C' where $C'(x, y_D) = C(x, y_S)$, and $\forall j \in \{y_D + 1, ..., y_S\} : C'(x, j) = C(x, y_S - 1)$.

case 3: $y_D > y_S$
This case is also possible, but the algorithm to do this is more complex than the previous one. Moreover, this case will never be called in our final algorithm for Solve(I, G). As a result we choose not to further elaborate upon this case.

B.2 Proofs

Lemma 3: Executing VMove(x, y_S, y_D, C) where y_S y_D takes at most 2 (top(x, C) y_D) + 6 basic moves.

Proof:
If y_S = y_D, then no basic moves are made and C gets returned. In this case VMove takes 0 basic moves.
If $y_S > y_D$, move gets executed multiple times. The first time is in line 9. This line gets called top(x, C) y_S + 1 times. We then have 2 cases again, similar to HMove. In the first case (where the else-clause in line 17 does not get called), the move in line 16 gets called y_S y_D times. Then either the move in line 28 or in line 30 gets called once. Line 33 doesn't get called, and line 35 gets called size(L) = top(x, C) y_D times.
In the second case, a few moves get added: The moves in lines 20, 21, 22 and 33 all get called once. So in total, the second case adds 4 more basic moves.

Overall, if $y_S < y_D$ the move command gets executed (top(x, C) y_S + 1) + (y_S y_D) + 1 + ($top(x, C)$ y_D)[+4] =
2 (top(x, C) y_D) + 2 times, or
2 (top(x, C) y_D) + 6 times.

Lemma 4: When executing C' = VMove(x, y_S, y_D, C) where y_S y_D, the following properties hold for configuration C':
- $C'(x, y_D) = C(x, y_S)$
- $\forall j \in \{1, ..., y_D$ $1\} : C'(x, j) = C(x, j)$
- $\forall i \in \{1, ..., m\} : i \neq x \implies (\forall j \in \{1, ..., h\} : C'(i, j) = C(i, j))$

Proof:
$C'(x, y_D) = C(x, y_S)$
The property $T(x, y_D) = C(x, y_S)$ gets achieved in line 28 or line 30. After this, no blocks get moved away from column x in the algorithm, which means block $C(x, y_S)$ will not get moved again after line 30. Thus, the property $C'(x, y_D) = C(x, y_S)$ holds for the final configuration C'.

$\forall j \in \{1, ..., y_D - 1\} : C'(x, j) = C(x, j)$

Blocks get moved out of column x in lines 10, 16 and 29. However, all of these lines only get executed while $\text{top}(x, T) < y_D$. This means that no blocks at row $y_D - 1$ or below get moved, so $C'(x, j) = C(x, j)$ for all $j < y_D - 1$.

$\forall i \in \{1, ..., m\} : i \neq x \implies (\forall j \in \{1, ..., h\} : C'(i, j) = C(i, j))$

List L ensures that every block that gets moved out of column x, eventually gets moved back in line 34. The only other time in the algorithm that a different column gets changed is in lines 21-23. The top block of column r gets swapped with block $C(x, y_S)$. However, block $C(x, y_S)$ gets moved to its goal position in column x, and the original top block of column r gets moved back to column r in line 33. If it got there initially during line 10, 16 or 25, it will still get moved back to its original column in line 34 because of list L. This means in the final configuration C' no columns besides column x will have changed, so the property holds.