Bachelor thesis
Computer Science

Radboud University

Optimisations and Generalisations of Retrograde Analysis

Author:
Julian Neeleman
s4156129

First supervisor/assessor:
Sjaak Smetsers
`s.smetsers@science.ru.nl`

Second assessor:
Georgios Gousios
`gousiosg@gmail.com`

July 6, 2015

Abstract

When presented with a puzzle, one tends to think from the starting point towards the goal. What steps should I take from where I am to get where I want to be? It takes more creative thinking to think in the opposite direction; What is the last step towards reaching my goal? And the step before that one? Most often the traditional forward approach works like a charm, especially when there is one starting point and one finish line. However, when the options to choose from are countless we are often at a loss when deciding what direction to take. It is in these situations that we should apply the alternative. These applications are not limited to real life, and when faced with a problem of this nature in computer science we formalise our thinking into retrograde analysis.

Contents

Chapter 1

Introduction

There are numerous projects that process huge amounts of data. Experiments at CERN for example, generate 30 petabytes of data every year[6]. All this information is processed in enormous, costly data centres. BOINC[3], a grid computing network set up by Berkeley University, hosts research projects that have run for decades on hundreds of cores. This research aids in a better understanding of global warming, finding cures for diseases and proving or disproving mathematical conjectures. For example, there is a project dedicated to finding world-record size Mersenne prime numbers[9] and one aiming to find pulsars using data collected by telescopes and satellites[8].

One can look at the large scale and add more processing cores to these grids, but these hardware solutions are expensive. Alternatively we can look at optimising the software used. Tiny improvements to small computations can result in massive time gain and money savings, because they are used billions or even trillions of times. Take for example Karatsuba's algorithm for multiplication[10], an arithmetic operation that is used with extremely high frequency in virtually all software.

In this paper we look into bitboards[2], an efficient and compact data structure used in chess engines and the representation of other board games. We will answer the question:

> What improvements can be made to the bitboard data structure and what can we use bitboards for?

We will show that bitboards can be used to efficiently check for symmetries in board states. We also introduce a new method for minimising the amount of processor operations to perform these checks faster than with current techniques.

Furthermore, we perform a case study on the puzzle game Tilt. The study will attempt to put the methods described into practise and give an indication of possible further applications.

We apply the techniques developed in our research to retrograde analysis, an algorithm that can be considered to be 'inverted breadth-first search'. This brings us to the more general question this thesis will answer:

> How can we improve the retrograde analysis algorithm?

Chapter 2

Preliminaries

Before digging into our research we explain some concepts that may be unfamiliar to the reader. We assume an understanding of the C++ language[5] and the breadth-first search algorithm[4].

2.1 Tilt Game

Tilt is a single-player sliding puzzle game developed by Thinkfun[11]. The game comes in a box containing the 5x5 game board, two green blocks, four blue blocks and six grey ones. The game board has a hole in the middle. A puzzle is created by distributing these blocks over the board. Usually, one sets up the board using one of the challenge cards supplied with the game. These cards give the player an indication of difficulty and guarantees that the configuration is solvable. We will be generating our own puzzles during the research.

Once the board is set up, we can make moves by tilting the board left, right, up or down so that all the blue and green blocks slide downhill. Note that the blocks can not stop before they hit either the edge of the board or another block! The grey blocks are stationary.

The objective of the game is to have all the green blocks fall through the hole in the middle of the board. Blue blocks are not allowed to fall into the hole, even after the last green block has been removed.

Clearly, not all configurations are solvable. Sometimes there is no way to remove a green block from the board, other times it might be impossible to prevent a blue block from falling through the hole. It is also possible to go from a solvable position to an unsolvable one.

To understand the order of magnitude of our search tree, we can sum over the binomial coefficients for different amounts of blocks of each colour:

$$\sum_{a=0}^{6}\sum_{b=0}^{4}\sum_{c=0}^{2}\binom{24}{a}\cdot\binom{24-a}{b}\cdot\binom{24-a-b}{c}\approx 10^{11}$$

Where the number 24 comes from the number of squares on the board. This equation does not take into account board symmetries and other search tree pruning factors, which we will show have a significant impact on the number of configurations we have to evaluate. It does, however, give us an upper bound to work with. To compare, Rush Hour has $3.6\cdot 10^{10}$ states[13] and the solved[12] game Connect Four has a state-space size of around 10^{13}. With the state of current computation power it is possible to determine the 'hardest' puzzle in Tilt. We consider a puzzle to be harder if its minimal solution requires more moves.

2.2 Bitboards

Traversing a large search tree comes with some logistical problems. For one, we want to keep the entire tree in memory for quick access. This can be achieved by compressing the information about a game state into one or more so-called bitboards.

A bitboard is essentially an array of bits. Being able to compress the state of a board game to such a structure does not only allow us to store huge amounts of states in memory, it also allows us to use low-level processor bit operations to rapidly manipulate a state. We will use this property extensively throughout this paper.

Chess engines are the best-known application of bitboards. Because a chess board is exactly 64 squares, maps of piece population on the board can be stored in a 64-bit integer perfectly. There are many uses for bitboards in chess and most modern engines use a combination of techniques. For example, if we want to efficiently store the positions of all black and white pawns on the board, we could initialise two bitboards at the start of the game like this:

```
 WHITE     BLACK
00000000 00000000
00000000 11111111
00000000 00000000
00000000 00000000
00000000 00000000
00000000 00000000
```

```
11111111 00000000
00000000 00000000
```

Now if we want to make sure no square is occupied by two pawns, we would normally compare the values in both boards for each square. Making clever use of our bitboard representation however, a simple bit-wise AND operation will suffice:

```
00000000     00000000
00000000     11111111
00000000     00000000
00000000 AND 00000000 = 0
00000000     00000000
00000000     00000000
11111111     00000000
00000000     00000000
```

The result is 0, which can be interpreted as 'no collision'. Any non-zero result would imply a collision. This operation only takes a 64-bit processor one clock-cycle, which saves a lot of time considering how many times a chess engine would like to check for collisions while traversing a search tree.

2.3 Retrograde Analysis

Breadth-first search (BFS) is one of the most commonly-used algorithms in computer science and artificial intelligence. Imagine a maze with one entrance and one exit. A simple BFS implementation would be able to find a path between these two points given that the maze is small enough. If we now add a second exit, the algorithm's performance should not be affected - a path to one exit would still suffice. Adding a second entrance however, and requiring the algorithm to find a path to the exit from both entry points, would mean having to run the BFS once for each entrance.

A better idea would be to find a path in the opposite direction; perform a BFS starting from the exit and work towards the entrances. Both should be found in one iteration.

To make the analogy for Tilt, one could consider the entry points to be any (solvable) puzzle and every final (solved) state serves as an exit. As mentioned earlier we want to find the puzzle that requires the most moves to solve. In other words: what is the maximum distance between any entrance point and exit? Considering that in this analogy there are - as shown earlier - roughly 100 billion entry points and - as will be explained later - about 100 million exits, the inverted approach is more efficient than performing BFS from all possible configurations.

When applying retrograde analysis to something with rules more complex than those of a maze we need to take into account that moves may not be symmetrical in nature. Take for example chess. If we move the queen piece to a new position on the board we could assume that moving it back to its original location reverses the move. However, if the queen captured an enemy piece in the process of moving we need to 'resurrect' this piece to completely undo the move. As a matter of fact we need to consider every rule of the game and determine its inverse effect, which can sometimes be quite difficult and indeed literally counter-intuitive.

For Tilt the same holds; not all moves are the same when reversed. If, for example, a green block falls through the hole by making a move, a move in the opposite direction does not return it to the board. A board has only one successor for any given move, but may have many (or no) predecessors. Consider a blue stone in the top-right corner of an otherwise empty Tilt board. Moving to the left will put the stone in the top-left corner. This is the configuration's only successor for moving left. However, if we look into the preceding state of this configuration given that the last move was to the right we find that there are several options. In fact, the blue stone may have been in any of the five squares of the top row.

Chapter 3

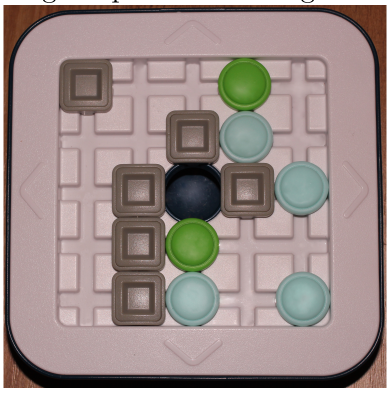
Research

We will now discuss performance improvements that can be made to using retrograde analysis and bitboard representations in software. After explaining the theory, these techniques will be demonstrated in a case study solving Tilt.

3.1 Grouped bit shifting

We explained in our preliminaries what bitboards are. They are a necessity for implementing our solution to Tilt. The rapid execution of bitboard permutations will prove to be essential for the feasibility of traversing the entire game tree.

There are several ways to perform bit permutations efficiently[1][14], but using bit shifts combined with masks covers most of these techniques. We use these permutations frequently in our implementation to perform moves and check for board symmetry. It is therefore useful to try to further improve existing bit permutation algorithms.



To demonstrate how we improve bit shifting, let us look at an example. The image above shows a configuration in Tilt that we would like to rotate 90 degrees clockwise. We can make this abstract by encoding each square in the grid as two bits, where 00 is an empty square, 01 represents a green block, 10 a blue one and 11 a solid grey block. Using this representation, our rotation looks like this:

```
11 00 00 01 00      00 00 00 00 11
00 00 11 10 00      11 11 11 00 00
00 11 00 11 10 -->  10 01 00 11 00
00 11 01 00 00      00 00 11 10 01
00 11 01 00 10      10 00 10 00 00
```

More generally, we are performing the permutation as shown below, where we have labelled each of the 50 bits. Note that because we require two bits to encode a square, all permutations take place in pairs.

```
[00 01] [02 03] [04 05] [06 07] [08 09]
[10 11] [12 13] [14 15] [16 17] [18 19]
[20 21] [22 23] [24 25] [26 27] [28 29]
[30 31] [32 33] [34 35] [36 37] [38 39]
[40 41] [42 43] [44 45] [46 47] [48 49]
                   |
                   v
[40 41] [30 31] [20 21] [10 11] [00 01]
[42 43] [32 33] [22 23] [12 13] [02 03]
[44 45] [34 35] [24 25] [14 15] [04 05]
[46 47] [36 37] [26 27] [16 17] [06 07]
[48 49] [38 39] [28 29] [18 19] [08 09]
```

We can deduce from these representations exactly how much each bit needs to be shifted in order to be in the right place. For example, bit 38 was moved 4 places to the right (+4). We can represent the pair-wise shifts like this:

```
+08 +16 +24 +32 +40
-04 +04 +12 +20 +28
-16 -08 +00 +08 +16
-28 -20 -12 -04 +04
-40 -32 -24 -16 -08
```

From this figure we see that there are 20 distinct shifts required to obtain the permutation. We could implement this naive approach in C++ for a given bitboard b. The masks are not written out in the implementation below, but they are hard-coded bitboards representing the section of the board that the bits are 'projected' on to. For example, mask18 is equal to the binary number 11, representing the bottom-right square of the board. This is the section we are projecting the top-right corner of the board onto by shifting the board 40 to the right. This is in line with the rotation we are trying to achieve.

```
b' =    ((b << 40) &  mask0) |
        ((b << 32) &  mask1) |
        ((b << 28) &  mask2) |
        ((b << 24) &  mask3) |
        ((b << 20) &  mask4) |
        ((b << 16) &  mask5) |
        ((b << 12) &  mask6) |
        ((b <<  8) &  mask7) |
        ((b <<  4) &  mask8) |
        ( b        &  mask9) |
        ((b >>  4) & mask10) |
        ((b >>  8) & mask11) |
        ((b >> 12) & mask12) |
        ((b >> 16) & mask13) |
        ((b >> 20) & mask14) |
        ((b >> 24) & mask15) |
        ((b >> 28) & mask16) |
        ((b >> 32) & mask17) |
        ((b >> 40) & mask18);
```

We need 56 bit operations to perform this permutation, which is slow when we are executing rotations repeatedly. However, we can significantly reduce the number of operations by combining groups of bits that move in the same direction. In our example we have groups of bits that need to be shifted by 4, 8 and 12 positions respectively. Instead of performing these shifts independently, we can group them together such that we shift the 4-group and the 12-group by 4, and the 8-group and the 12-group by 8. This way, we have shifted the 12-group by $4 + 8 = 12$, exactly as we wanted. Let us generalise this example.

After determining a set V containing the distinct shift-distances in our permutation, we would like to find a set C such that for each value v in V, we can find a subset C' of C for which the sum of the elements equals v. More importantly, we would like this set to contain as few elements as possible. In our example from earlier, we found that for the set $V = \{4, 8, 12\}$ there is $C = \{4, 8\}$ satisfying these constraints. We prove a few properties of C.

Theorem 3.1.1. *A minimal set C such that for every v in V, there exists a subset C' of C for which the sum of elements equals v, exists.*

Proof. Clearly, taking $C = V$ gives a set that satisfies the desired properties. For each value v in V, just take the singleton set of v in C. The sum of this set equals v. This gives us an upper bound for the size of a minimal set. $\square$

We will now also give a lower bound for the size of the minimal set.

Theorem 3.1.2. *For the minimal set C for V, the following bounds hold:* $\lceil log(\#V)\rceil \leq \#C \leq \#V$

Proof. Theorem 1 proves the upper bound, so we only need to prove the lower one. The number of possible sum values for subsets of C is smaller than or equal to the number of subsets of C, or the size of the power set of C, being $2^{\#C}$. Because of this, for C to qualify as a set for V, $2^{\#C} \geq \#V$ needs to hold, or $\lceil log(\#V)\rceil \leq \#C$ $\square$

The above bounds give promising best-case results for our technique. It does prove difficult however, to use anything better than brute force to find a minimal set. An implementation of a simple brute force approach can be found as an appendix to this paper. It is fast enough for permutations of up to 64 bits. Using this implementation, we find for $V = \{-40, -32, -28, -24, -20, -16, -12, -8, -4, 0, 4, 8, 16, 20, 24, 28, 32, 40\}$ a minimal set $C = \{-40, 4, 8, 16, 32\}$ (this set is not unique). We see that $2^{\#C} = \#V$, so we have encountered a best-case scenario. We need to recalculate the masks, and then change our calculation to the following:

```
b' =    ((b << 40) & mask19) |
        ((b >>  4) & mask20) |
        ((b >>  8) & mask21) |
        ((b >> 16) & mask22) |
        ((b >> 32) & mask23);
```

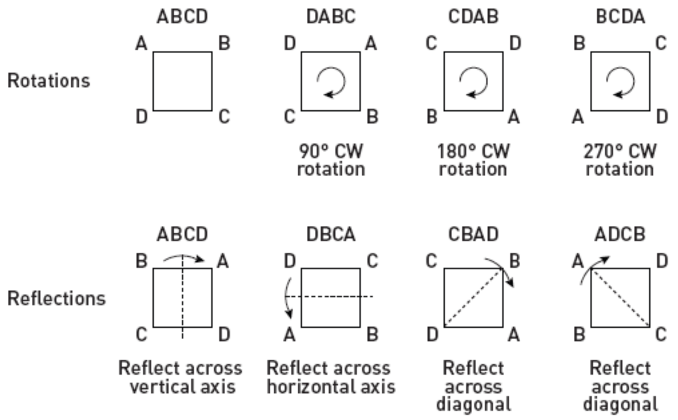
The masks are calculated to be the OR-combinations (or union) of the masks used in the naive implementation. So, mask19 is the union of the group of bits that is being shifting 40 spaces to the left:

```
mask19 = mask0 | mask1 | mask2 | mask3 | mask4 | mask5 |
         mask6 | mask7 | mask8 | mask9
```

These OR-operations are usually performed before compilation and hard-coded, not repeatedly calculated throughout execution. Using grouped bit shifting we now only need 14 bit operations instead of 56, reducing the time required to rotate a Tilt bitboard by 75%.

3.2 Bitboard symmetries

Now that we have discussed how to pre-calculate bit operation sequences for fast execution of bitboard permutations, we can show an application. Often times, bitboards are used to represent configurations that show symmetry. Most commonly when analysing board games we are dealing with a square board.



Shown above are the eight symmetries of a square board. For chess and checkers, only some of these symmetries are applicable, because of the checkered colouring of the squares and additional constraints on piece mobility. For Tilt however, full use can be made of the symmetries; it has a 5x5 board with a hole right in the centre. When performing retrograde analysis or other exhaustive techniques used for analysing the states of a game it is wise to check for every configuration whether it has already been further evaluated under symmetry.

Similarly to how we constructed an efficient algorithm for rotating a Tilt bitboard by 90 degrees clockwise in the previous section, we can find one for each of the symmetries of the square. We can then perform a statistical test to see how many states we can eliminate from our search.

```
bool keep(b_board b) {
    if ((mirr_h(b) < b) |
        (mirr_v(b) < b) |
        (mirr_d(b) < b) |
        (mirr_cd(b) < b) |
        (rot90(b) < b) |
        (rot180(b) < b) |
        (rot270(b) < b)) {
        return false;
    }
    return true;
}
```

The idea of the code above is to apply each of the seven permutations (excluding the identity) to them. If any of the resulting bitboards has a smaller numerical value, we eliminate the original from our search. This 'smaller than' criterion is arbitrary. We could implement this technique using any other strict total order. Running this code for a large number of randomly generated boards shows that almost 87.5% of Tilt configurations can be omitted from analysis, or seven out of eight. This is in line with the intuition that most boards are not identical under rotation or reflection. An example of an exception is the empty board, which is identical to itself under all symmetries.

3.3 Statistical verification of retrograde analysis

When analysing a huge search tree, we must verify the results. Besides the obvious unit testing and possible static analysis we can also reinforce the correctness of our results with statistical analysis. We want to apply to this in such a way that it does two things:

- Show inclusiveness. We want to make sure that our implementation is exhaustive. No solvable states should be omitted from the search (unless they are symmetrical to another)
- Show exclusiveness. States that are unsolvable should not be found in a retrograde analysis. We also want to verify the solution found for solvable states.

The reason we take a statistical approach is that checking every configuration is not feasible time-wise and defeats the purpose of using retrograde analysis. The obvious drawback to the statistical approach is that it might not cover all special cases. We now show a generalised approach to testing both properties.

3.3.1 Inclusiveness

Of the two, inclusiveness is the harder property to verify in retrograde analysis. The basic concept is to make a random list of solvable states, for example by generating states and using a breadth-first search to check if a solution exists. If additional symmetry constraints apply, we should take this into account as well. See the section about bitboard symmetries for this. We then add a check to our retrograde analysis algorithm to check states we are evaluating against our list.

This check however, can impact performance significantly if the randomly generated boards are stored in a simple vector or list. For our implementation of Tilt we are planning to do billions of look-ups. Checking against a hash table is recommended. Furthermore, states that are checked should be removed from the table. Because of the constant time-complexity of hash table look-ups the speed loss is negligible.

3.3.2 Exclusiveness

To statistically test for exclusiveness we randomly pick out states while evaluating and run a breadth-first search to confirm that the solution we found by working backwards from the solved state is in line with the result of the forward approach. The number of states that are picked out for verification can be scaled to balance between performance and rigour. For the verification of Tilt we will check one in every ten thousand states. This frequency is acceptable for this specific game because the BFS search trees do not often exceed a size of over a hundred thousand nodes. For games where the search tree is not as disconnected as it is for Tilt the sample size most likely needs to be smaller.

Another possibility is to generate boards that are unsolvable and do something similar as we did for checking inclusiveness, throwing an error when evaluating an unsolvable state.

3.4 Case study: Tilt

In the preliminaries we briefly introduced Tilt. We now use this game as a case study for the techniques explained earlier in this paper. Throughout the study will be parts of the C++ code used. For the full code and unit tests, see the appendix.

3.4.1 State representation

We have already covered how we can represent a Tilt configuration in a 50-bit bitboard. Additionally, we need to keep track of how many green blocks are in the puzzle, because we only allow for the maximum of two that are included in the standard game box. We keep a counter for this in our struct.

```
typedef uint64_t b_board; // board represented in 50 bits

struct b_board_counter {
    b_board b;
    int counter;
    int green; // keeps track of number of green blocks having been introduced to t

    // compare defined for use in memory checks
    bool operator==(const b_board_counter& rhs) {
        return b == rhs.b && green == rhs.green;
    }
};
```

We make the observation that when making a move to the left or right, the rows move independently. To clarify: what a specific row looks like when moving the entire board to the left is not affected by the composition of the rest of the board. Similarly, the columns move independent of each other when moving up or down. We can utilise this to calculate board states after moving per row or column. While the number of board configurations is around 100 billion, the number of row/column configurations is slightly less than $4^5 = 1024$ (we can omit rows that use more green or blue blocks than are in the game box). This allows us to pre-calculate a move table. For each of the rows, we calculate the outcome when we move it left or right. We then add it to the list of predecessor states of the resulting row. This way, we can easily extract all preceding rows from our database using only bitboard comparison. For example, say we have a row that from left to right consists of a green block, a blue block and three empty squares. The bit row representation of which is the following:

```
01 10 00 00 00
Gr Bl Em Em Em
```

This state can be reached by several states by moving the board left:

```
01 10 00 00 00
01 00 10 00 00
01 00 00 10 00
01 00 00 00 10
00 01 10 00 00
00 01 00 10 00
00 01 00 00 10
00 00 01 10 00
00 00 01 00 10
00 00 00 01 10
```

The list of preceding board states for any given board is equal to the Cartesian product of the lists for each of the rows. Of course, we have only been concerned with moves to the left for this example. Moves in other directions are implemented analogously. This way, the calculation of the predecessor states of a board can be reduced to only table look-ups and bit operations to extract and insert rows.

3.4.2 Generating final states

Vital to any implementation of retrograde analysis is the correct generation of all the final states. That is, all the configurations of the board that are in a solved state. For Tilt this comes down to finding all the configurations without green blocks. We also want to eliminate symmetrical duplicates.

```
bool gen(std::vector<b_board>& boards, std::vector<std::vector<b_board> >& memory ,
    if (valid(memory, b, symm_check)) {
        boards.push_back(b);
    }
    if (blocks.empty()) {
        return true;
    }
    int block = blocks[blocks.size() - 1];
    blocks.pop_back();
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            if (get(b, i, j) == 0 && (i != 2 || j != 2)) {
                b_board new_b = place(b, i, j, block);
                gen(boards, memory, blocks, new_b, symm_check);
            }
        }
    }
    return true;
}
```

This can be achieved by first recursively adding blocks to an empty game board, using the above algorithm, meanwhile checking for symmetries. We then recursively add blue blocks. Again, we check for symmetries. There are roughly 25.000 grey block configurations, and about 100 million final states altogether.

3.4.3 Parallelisation

When searching through the huge search tree of a game like Tilt, we would like to evaluate states in parallel. In order to do that we need to be certain that the states we evaluate do not have overlapping search trees, otherwise we would risk doing double calculations. In Tilt, this split into sub-problems is easily achieved.

We know from the rules of the game that grey blocks never move during the course of the game. A path between a state where the board contains four grey blocks and one where there are only two grey blocks can therefore not exist. Stronger even, the exact configuration of the grey stones is constant throughout the evaluation of a given final state. We can thus group final states by their grey block configuration, splitting the problem into 25.000 smaller ones. We then do not require any shared memory and 100% CPU use is attainable by having one instance of the code run on each of the available cores, giving each instance a different input file with independent sub-problems.

Chapter 4

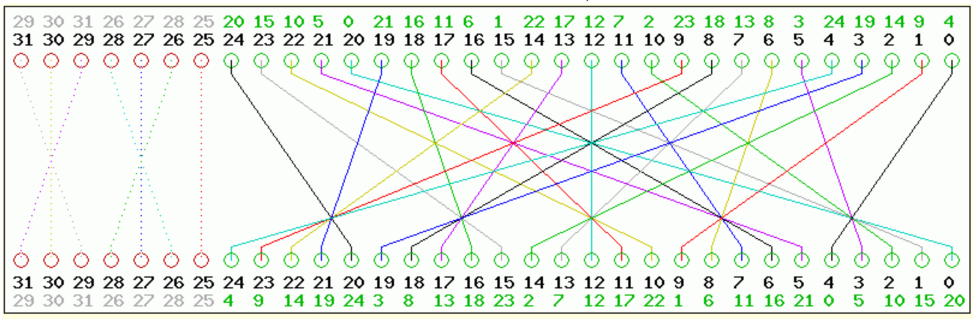
Related Work

Now let us compare the results of our research to those of the work we have built on.

4.1 Bit permutations

A masters thesis by Victor Allis in which he explains how he solved Connect-Four[12] pointed us into the direction of bitboards. In his thesis he describes how a board can be checked for a win-condition in just a few bit shifts. This result gave hope for elegant bitboard solutions for related problems.

There are many tools and tutorials available that determine efficient processor-operation sequences for bit permutations[2][7], but none of them seem to be aware of bit group shifting as described in our research. If we enter the rotation we used as an example earlier into the on-line tool found at `http://programming.sirrida.de/calcperm.php`, we obtain the following result (note that because of the 32-bit limit we have simplified the permutation by merging pairs of bits into one bit):



Best method found: Reordered Beneš network (about 30 cycles on superscalar processors):

```
   x = bit_permute_step(x, 0x010a0700, 4);  // Butterfly, stage 2
   x = bit_permute_step(x, 0x00630040, 8);  // Butterfly, stage 3
   x = bit_permute_step(x, 0x21213020, 2);  // Butterfly, stage 1
   x = bit_permute_step(x, 0x0000c618, 16);  // Butterfly, stage 4
   x = bit_permute_step(x, 0x00d6002e, 8);  // Butterfly, stage 3
   x = bit_permute_step(x, 0x01040d0f, 4);  // Butterfly, stage 2
```

The tool suggests a method using 30 cycles, where we managed to cut the required number of operations down to 14. While there are some cases where the tool provides a significantly better method, these make use of processor-specific commands such as byte-reversal. For general purposes our algorithm is a useful addition to the spectrum of bit permutation techniques.

4.2 Rush Hour

The paper by Sébastien Collette, Jean-François Raskin and Fredéric Servais on Rush Hour[13] served as the main inspiration for our research. Although they used symbolic computation, they mention in their results that retrograde analysis was a feasible approach to solving puzzles of this type. They saw drawbacks in the computational requirements for this methods. Optimisations to the algorithm were also required, as well as case-specific tweaks.

While time took care of the computational requirements (the computer we used was much faster than what they had access to in 2006), our research resulted in the necessary improvements to retrograde analysis.

Chapter 5

Conclusions

In our research we have managed to combine existing algorithms and techniques. In some instances we have even managed to add to the theory. The case study also gave us a 'hardest puzzle' for Tilt, a satisfying result.

5.1 Theory

The largest contribution to theory this paper makes is the improvement to bitboard permutations. Even though an efficient algorithm was not found, a brute-force method worked fine for boards up to 64 bits. Grouped bit shifting relies entirely on shifts-by-n operations. Today, virtually every processor is equipped with a such an operator. Software using this technique can therefore be executed on any modern computer.

5.2 Case Study

A result emerged after about twenty minutes of running the Tilt code on all eight cores of a home computer equipped with an Intel i7-4790K 4.00GHz x64-processor. We averaged around 10 million state-evaluations per second, checking a total of about 12 billion configurations. Looking back at our estimate for the number of configurations, we see that we were able to prune almost 90% of the search tree.

All techniques described in this paper were incorporated in our implementation and the code has almost complete unit-test coverage.

The hardest puzzle found was the one used earlier as an example for bitboard rotation:

5.3 Further research

There are several related topics worth looking into. These come to mind:

- Finding a more efficient method to compute shift sequences for grouped bit shifting.
- Combining grouped bit shifting with existing techniques[1] might result in even faster permutations on specific platforms.
- Applying the described theory to more puzzles. Rush hour would be an interesting choice, because similar research[13] using different methods has been done in this game.

Bibliography

[1] Bit permutations. http://programming.sirrida.de/bit_perm.html.

[2] Bitboard. https://en.wikipedia.org/wiki/Bitboard.

[3] Boinc. https://boinc.berkeley.edu/.

[4] Breadth-first search. https://en.wikipedia.org/wiki/Breadth-first_search.

[5] C++. http://www.cplusplus.com/.

[6] Cern. http://home.web.cern.ch/about/computing.

[7] Chess programming. https://chessprogramming.wikispaces.com/Bitboards.

[8] Einstein@home. http://einstein.phys.uwm.edu/.

[9] Great internet mersenne prime search. http://www.mersenne.org/.

[10] Karatsuba's algorithm. https://en.wikipedia.org/wiki/Karatsuba_algorithm.

[11] Thinkfun. http://www.thinkfun.com/.

[12] Victor Allis. A knowledge-based approach of connect-four, 1988. http://www.informatik.uni-trier.de/~fernau/DSL0607/Masterthesis-Viergewinnt.pdf.

[13] Sébastien Collette, Jean-François Raskin, and Fredéric Servais. On the symbolic computation of the hardest configurations of the rush hour game, 2006. http://www.ulb.ac.be/di/algo/secollet/papers/crs06.pdf.

[14] Pradyumna Kannan. Magic move-bitboard generation in computer chess, 2007. http://www.pradu.us/old/Nov27_2008/Buzz/research/magic/Bitboards.pdf.

Appendix A

Appendix

A.1 Retrograde Analysis

```
#include "stdafx.h"
#include <stdint.h>
#include <assert.h>
#include <iostream>
#include <vector>
#include <queue>
#include <fstream>
#include <string>
#include <algorithm>
typedef uint64_t b_board; // board represented in 50 bits
typedef unsigned b_row; // there are four regular rows in the board, and one center one with a hole
const int MAX_GREEN = 2, MEM_SIZE = 1000000;
/* row struct with number of green blocks */
struct b_row_counter {
    b_row r;
    int counter;
    // compare defined for use in reverse look-up
    bool operator==(const b_row& rhs) {
        return r == rhs;
    }
};
/* board struct that keeps track of moves required to solve */
struct b_board_counter {
    b_board b;
    int counter;
    int green; // keeps track of number of green blocks having been introduced to the board
    // compare defined for use in memory checks
    bool operator==(const b_board_counter& rhs) {
        return b == rhs.b && green == rhs.green;
    }
};
std::vector<b_row> left_table[0x400]; // after init_tables(), retrograde states are found in a vector
std::vector<b_row> right_table[0x400]; // same as left_table, but for right-moves
std::vector<b_row_counter> left_c_table[0x400]; // same as left_table, but for center row
std::vector<b_row_counter> right_c_table[0x400]; // same as left_c_table, but for right-moves
/*
 * draws the puzzle and outputs moves required to complete
 */
void draw_board(b_board_counter bc) {
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            int v = (bc.b >> (2 * i + 10 * j)) & 0x3;
            std::cout << v << " ";
        }
        std::cout << std::endl;
    }
    std::cout << bc.counter << std::endl << std::endl; // output number of moves to complete
}
/*
 * perform matrix transposition on the bitboard
 */
b_board transpose(b_board b) {
    assert(b < 0x4000000000000); // boards should never exceed 50 bits
    return ((b << 32) & 0x30000000000) |
    ((b << 24) & 0xC00C0000000) |
        ((b << 16) & 0x300300300000) |
        ((b << 8) & 0xC00C00C00C00) |
        (b & 0x3003003003003) |
        ((b >> 8) & 0xC00C00C00C) |
        ((b >> 16) & 0x30030030) |
        ((b >> 24) & 0xC00C0) |
        ((b >> 32) & 0x300);
}
/*
 * mirrors board vertically
 */
b_board mirr_v(b_board b) {
    return ((b << 8) & 0xF03C0C03C0F0) |
        ((b >> 4) & 0xC0300C0300C0) |
        ((b >> 8) & 0x300C0300C03);
}
/*
 * reverses bit row
 */
b_row reverse(b_row r) {
    assert(r < 0x400);
    return ((r >> 8) & 0x3) |
        ((r >> 4) & 0xC) |
        (r & 0x30) |
        ((r << 4) & 0xC0) |
        ((r << 8) & 0x300);
}
/*
 * converts a bit row to an array
 */
unsigned * b_row_to_array(b_row r) {
    unsigned *a = new unsigned[5];
    a[0] = (r >> 8) & 3;
    a[1] = (r >> 6) & 3;
    a[2] = (r >> 4) & 3;
    a[3] = (r >> 2) & 3;
    a[4] =  r & 3;
    return a;
}
/*
 * converts an array into a bit row
 */
b_row array_to_b_row(unsigned * a) {
    return (a[0] << 8) |
        (a[1] << 6) |
        (a[2] << 4) |
        (a[3] << 2) |
        a[4];
}
/*
 * convert bit row to array, execute a move to the left and convert back to bit row
 * this function is only used to initialize move tables
 */
b_row move_b_row(b_row r) {
    assert(r < 0x400); // rows should never exceed 10 bits
    unsigned * a = b_row_to_array(r);
    bool change = true;
    while (change) {
        change = false;
        for (int i = 0; i < 4; i++) {
            if (a[i] == 0 && (a[i + 1] == 1 || a[i + 1] == 2)) {
                a[i] = a[i + 1];
                a[i + 1] = 0;
                change = true;
            }
        }
    }
    r = array_to_b_row(a);
    delete a;
    return r;
}
/*
 * convert bit center row to array, execute a move to the left and convert back to bit center row
 * this function is only used to initialize move tables
 */
b_row move_bc_row(b_row r) {
    assert(r < 0x400); // center rows should never exceed 10 bits
    unsigned * a = b_row_to_array(r);
    assert(a[2] == 0); // center rows should have holes
    if (a[3] == 2 || (a[4] == 2 && a[3] != 3)) {
        return 0x30; // return 0000110000 to signal an illegal board position
    }
    if (a[0] == 0 && a[1] != 3) {
        a[0] = a[1];
        a[1] = 0;
    }
    if (a[3] != 3) {
        a[3] = 0;
        if (a[4] != 3) {
            a[4] = 0;
        }
    }
    r = array_to_b_row(a);
    delete a;
    return r;
}
/*
 * converts a set of five rows to a board
 */
b_board rows_to_board(b_row * rows) {
    return ((b_board)rows[0] << 40) |
        ((b_board)rows[1] << 30) |
        ((b_board)rows[2] << 20) |
        ((b_board)rows[3] << 10) |
        (b_board)rows[4];
}
/*
 * converts a board to a set of five rows
 */
b_row * board_to_rows(b_board b) {
    assert(b < 0x4000000000000); // boards should never exceed 50 bits
    b_row * rows = new b_row[5];
    rows[0] = (b >> 40) & 0x3FF;
    rows[1] = (b >> 30) & 0x3FF;
    rows[2] = (b >> 20) & 0x3FF;
    rows[3] = (b >> 10) & 0x3FF;
    rows[4] = b & 0x3FF;
    return rows;
}
/*
 * count number of green blocks in a row
 */
int count_green(b_row r) {
    int result = 0;
    unsigned * a = b_row_to_array(r);
    for (int i = 0; i < 5; i++) {
        if (a[i] == 1) {
            result++;
        }
    }
    delete a;
    return result;
}
/*
 * initialize a left-move table where each index represents the list of row states that can be reached
 * by making a left-move. similarly we initialize a right-move table. up- and down-move tables are not
 * generated; we do a transpose -> move -> transpose to achieve the same result. special tables are
 * initialized for the center row
 */
void table_init() {
    for (b_row r = 0; r < 0x400; r++) {
        if ((r & 0x30) == 0x0) {
            // row is also a center row
            b_row_counter r_counter = { r, count_green(r) };
            b_row r_left_c = move_bc_row(r); // perform left-move
            b_row r_right_c = reverse(move_bc_row(reverse(r))); // reverse the row, perform a left-move and reverse back
            left_c_table[r_left_c].push_back(r_counter);
            right_c_table[r_right_c].push_back(r_counter);
        }
        b_row r_left = move_b_row(r); // perform left-move
        b_row r_right = reverse(move_b_row(reverse(r))); // reverse the row, perform a left-move and reverse back
        left_table[r_left].push_back(r);
        right_table[r_right].push_back(r);
    }
}
/*
* moves board left
*/
b_board move(b_board b) {
    b_row * rows = board_to_rows(b);
    b_row new_rows[5];
    for (int i = 0; i < 5; i++) {
        if (i == 2) { // center row
            for (int j = 0; j < 0x400; j++) {
                if (std::find(left_c_table[j].begin(), left_c_table[j].end(), rows[i]) != left_c_table[j].end()) {
                    new_rows[i] = j;
                }
            }
        }
        else { // regular row
            for (int j = 0; j < 0x400; j++) {
                if (std::find(left_table[j].begin(), left_table[j].end(), rows[i]) != left_table[j].end()) {
                    new_rows[i] = j;
                }
            }
        }
    }
    delete rows;
    if (new_rows[2] == 0x30) {
        return b;
    }
    return rows_to_board(new_rows);
}
/*
* moves board in given direction
*/
b_board move(b_board b, int dir) {
    switch (dir) {
    case 0: // left
        return move(b);
    case 1: // right
        return mirr_v(move(mirr_v(b)));
    case 2: // up
        return transpose(move(transpose(b)));
    default:
        break;
    }
    return transpose(mirr_v(move(mirr_v(transpose(b))))); // down
}
/*
* checks whether any green blocks remain on the board
*/
bool solved(b_board b) {
    b_row * rows = board_to_rows(b);
    for (int i = 0; i < 5; i++) {
        if (count_green(rows[i]) > 0) {
            delete rows;
            return false;
        }
    }
    delete rows;
    return true;
}
/*
* calculates all successors (reachable states) for a given state
*/
std::vector<b_board_counter> successors(b_board_counter bc) {
    std::vector<b_board_counter> result;
    b_board_counter new_bc;
    new_bc.counter = bc.counter + 1;
    for (int i = 0; i < 4; i++) {
        new_bc.b = move(bc.b, i);
        result.push_back(new_bc);
    }
    return result;
}
/*
* solves given board using simple breadth-first search
*/
int solve(b_board b) {
    std::vector<b_board_counter> q; // queue
    std::vector<std::vector<b_board> > memory(MEM_SIZE); // hash table
    b_board_counter bc;
    bc.b = b;
    bc.counter = 0;
    q.push_back(bc);
    int i = 0; // memory index
    while (i < q.size()) {
        do {
            if (i == q.size()) {
                return 0; // no solution was found
            }
            bc = q[i];
            i++;
        } while (std::find(memory[bc.b % MEM_SIZE].begin(),
            memory[bc.b % MEM_SIZE].end(),
            bc.b) != memory[bc.b % MEM_SIZE].end());
            memory[bc.b % MEM_SIZE].push_back(bc.b);
        if (solved(bc.b)) {
            return bc.counter; // solution found, return number of moves
        }
        std::vector<b_board_counter> succ = successors(bc); // calculate reachable states
        q.insert(q.end(), succ.begin(), succ.end()); // add to end of queue
    }
    return 0;
}
/*
 * calculates predecessors through Cartesian product of rows, per direction
 */
std::vector<b_board_counter> cartesian(std::vector<b_row> pred_rows[4], std::vector<b_row_counter> pred_c_rows, b_board_counter orig, int c_row_green)
    for (int i = 0; i < 4; i++) {
        if (pred_rows[i].empty()) {
            return {}; // if one of the vectors is empty, then the cartesian product is empty
        }
    }
    if (pred_c_rows.empty()) {
        return {};
    }
    std::vector<b_board_counter> results;
    for (int i0 = 0; i0 < pred_rows[0].size(); i0++) {
        for (int i1 = 0; i1 < pred_rows[1].size(); i1++) {
            for (int i2 = 0; i2 < pred_c_rows.size(); i2++) {
                for (int i3 = 0; i3 < pred_rows[2].size(); i3++) {
                    for (int i4 = 0; i4 < pred_rows[3].size(); i4++) {
                        b_row rows[5] = { pred_rows[0][i0],
                            pred_rows[1][i1],
                            pred_c_rows[i2].r,
                            pred_rows[2][i3],
                            pred_rows[3][i4] };
                        b_board_counter bc;
                        bc.b = rows_to_board(rows);
                        bc.counter = orig.counter + 1;
                        bc.green = orig.green + pred_c_rows[i2].counter - c_row_green;
                        if (!(bc == orig) && bc.green <= MAX_GREEN) {
                            results.push_back(bc);
                        }
                    }
                }
            }
        }
    }
    return results;
}
/*
 * calculates all predecessing legal boards to the input board. predecessors are in fact elements of the
 * Cartesian product set of rows/columns, with case distinction between left- and right-moves
 */
std::vector<b_board_counter> predecessors(b_board_counter bc) {
    std::vector<b_board_counter> results;
    b_board_counter tbc = bc;
    tbc.b = transpose(bc.b); // transposed board
    b_row * rows = board_to_rows(bc.b), * columns = board_to_rows(tbc.b); // extract rows/columns from board
    // left
    std::vector<b_row> left_rows[4] = { left_table[rows[0]],
        left_table[rows[1]],
        left_table[rows[3]],
        left_table[rows[4]] };
    std::vector<b_board_counter> left_cart = cartesian(left_rows, left_c_table[rows[2]], bc, count_green(rows[2]));
    results.insert(results.end(), left_cart.begin(), left_cart.end());
    // right
    std::vector<b_row> right_rows[4] = { right_table[rows[0]],
        right_table[rows[1]],
        right_table[rows[3]],
        right_table[rows[4]] };
    std::vector<b_board_counter> right_cart = cartesian(right_rows, right_c_table[rows[2]], bc, count_green(rows[2]));
    results.insert(results.end(), right_cart.begin(), right_cart.end());
    // up
    std::vector<b_row> up_rows[5] = { left_table[columns[0]],
        left_table[columns[1]],
        left_table[columns[3]],
        left_table[columns[4]] };
    std::vector<b_board_counter> up_cart = cartesian(up_rows, left_c_table[columns[2]], tbc, count_green(columns[2]));
    // transpose boards again to return to initial orientation
    for (int i = 0; i < up_cart.size(); i++) {
        up_cart[i].b = transpose(up_cart[i].b);
    }
    results.insert(results.end(), up_cart.begin(), up_cart.end());;
    // down
    std::vector<b_row> down_rows[5] = { right_table[columns[0]],
        right_table[columns[1]],
        right_table[columns[3]],
        right_table[columns[4]] };
    std::vector<b_board_counter> down_cart = cartesian(down_rows, right_c_table[columns[2]], tbc, count_green(columns[2]));
    // transpose boards again to return to initial orientation
    for (int i = 0; i < down_cart.size(); i++) {
        down_cart[i].b = transpose(down_cart[i].b);
    }
    results.insert(results.end(), down_cart.begin(), down_cart.end());
    // clear pointer memory and return results
    delete rows;
    delete columns;
    return results;
}
/*
 * retrograde analysis algorithm. the last board analysed is returned as result.
 * the breath-first search nature of the algorithm guarantees that this is the hardest
 * configuration for the given subproblem
 */
b_board_counter retrograde(std::vector<b_board> roots) {
    assert(!roots.empty()); // a root must be provided
    std::vector<b_board_counter> q; // queue
    std::vector<std::vector<b_board> > memory(MEM_SIZE); // hash table
    for (int i = 0; i < roots.size(); i++) {
        b_board_counter bc;
        bc.b = roots[i];
        bc.counter = 0;
        assert(solved(bc.b)); // roots are required to be final states, thus solved
        bc.green = 0;
        q.push_back(bc);
    }
    b_board_counter last = q[0];
    int i = 0; // memory index
    while (i < q.size()) {
        //std::cout << memory[0].size() << std::endl;
        b_board_counter bc;
        do {
            if (i == q.size()) {
                return last; // all reachable states have been found
            }
            bc = q[i];
            i++;
        } while (std::find(memory[bc.b % MEM_SIZE].begin(),
            memory[bc.b % MEM_SIZE].end(),
            bc.b) != memory[bc.b % MEM_SIZE].end());
        memory[bc.b % MEM_SIZE].push_back(bc.b);
        last = bc;
        std::vector<b_board_counter> pred = predecessors(bc); // calculate predecessors
        q.insert(q.end(), pred.begin(), pred.end()); // add to end of queue
    }
    return last; // all reachable states have been found
}
int _tmain(int argc, _TCHAR* argv[]) {
    table_init(); // initiate move tables
    b_board_counter hardest = { 0, 0, 0 }; // keep track of hardest puzzle found
    int n = 0; // number of subproblems solved
    std::string t;
    std::cin >> t;
    std::ifstream input(t + "-blue_output.txt");
    std::cin.ignore();
    int N; // size of root
    // read in subproblems
    while (input >> N) {
        n++;
        std::vector<b_board> roots;
        for (int i = 0; i < N; i++) {
            b_board b;
            input >> b;
            roots.push_back(b);
        }
        b_board_counter bc = retrograde(roots); // enter retrograde analysis
        if (bc.counter != solve(bc.b)) {
            std::cout << "MISMATCH" << std::endl;
        }
        if (bc.counter >= hardest.counter) {
            hardest = bc;
            draw_board(hardest);
        }
        if (n % 100 == 0) {
            std::cout << n << std::endl;
        }
    }
    draw_board(hardest);
    std::cout << "Done." << std::endl;
    std::cin.ignore();
    return 0;
}
```