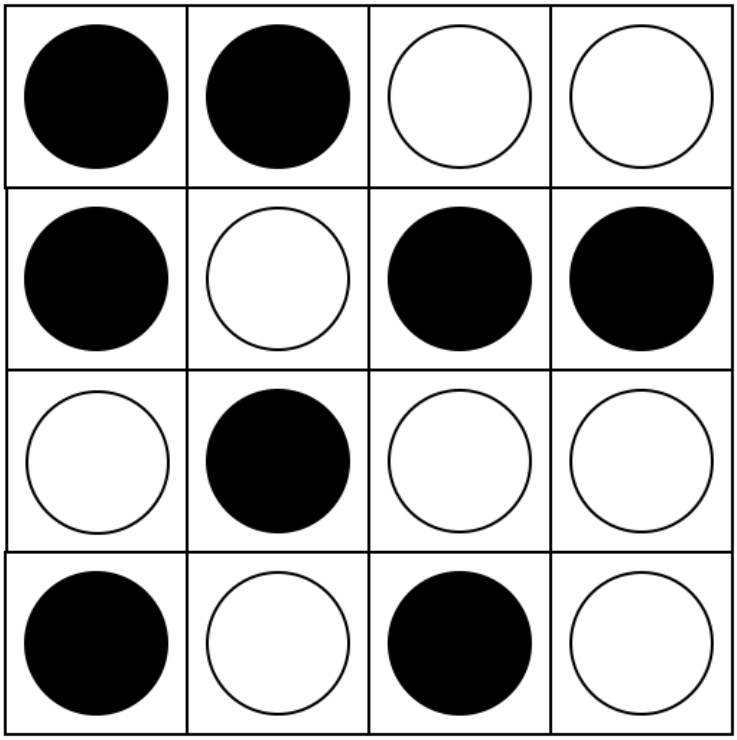
Graphs and Puzzles

Inas Bosch, Yanis Bosch

December 24, 2020



1 Introduction

In this paper, we will try to analyse puzzles consisting of a square grid or matrix of coloured points.

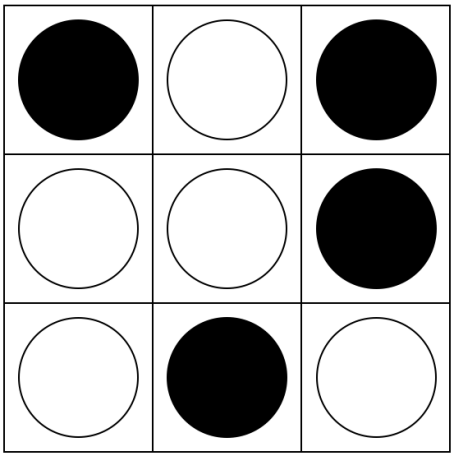


Figure 1: *Example of a 3x3 puzzle with black and white dots*

The goal is to get from one configuration of the grid to another one, a solution, whilst only being able to modify the puzzle in a specific way. We can rotate 2×2 square of dots inside the larger square, either clockwise or counter-clockwise. We note that the puzzles can vary in size

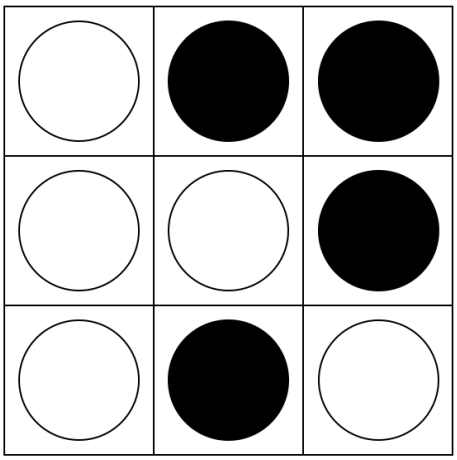


Figure 2: *Puzzle obtained from the puzzle from figure 1 by rotating the top left square clockwise*

and have a varying number of colours. Our goal is to either find an algorithm to go from one configuration of the puzzle to another one, whilst using as few movements as possible, or try to compute all possible configurations in advance to find the ideal set of movements. We will see that we started off with the latter method.

It is also worth mentioning that we will not talk about 2×2 puzzles as we can cleary see that it is often impossible to go between 2 configurations of such a puzzle.

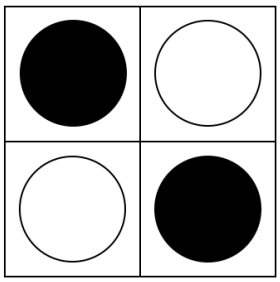


Figure 3: *Example of a* 2×2 *puzzle*

Remark 1. Throughout this paper we will often talk about *solving* a puzzle. We will then mean by this that we try to go from one configuration of a puzzle to a another given configuration, a solution.

Remark 2. We will sometimes also represent puzzles as a matrix with numbers instead of a grid with dots, where different numbers simply represent differently coloured dots. Hence another way of representing the puzzle from figure 1 would be:

$$\begin{matrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{matrix}$$

where 0 represents a white dot and 1 and black dot.

2 A first approach

2.1 Main idea

The first, and simplest way to solve given puzzles, and to get an idea which kinds of puzzles could be solved, if any, was to simply compute a tree containing all configurations of puzzles that one could obtain starting with our solution and then rotating the 2×2 squares.

To do so, we created a class *tree*, containing a *root* and a set of *nodes*. Each *node* had both a parent *node*, and a set of children *nodes*. We denoted the *parent* of the root *node* by a *None* object.

In order to compute the tree, we went through all the leaf nodes of our tree on the lowest level one by one. For each one we went through all possible movements, applied them to the puzzle, and checked if the newly obtained puzzle was already in the tree or not. If not, we added it as a child of the leaf node we modified.

We repeated this process until no new puzzles could be obtained.

2.2 Results

Initially we wanted to figure out if all types of puzzles could be solved. We started with 3×3 puzzles with 3 colours and 4×4 puzzles with 2 colours. We chose some random configuration of each type of puzzle and built our tree from there. In the end, we simply printed the amount of puzzles in the tree, and compared this to the total number of possible configurations, and saw that in both cases, it matched. Hence it is possible to find a set of movements to switch between any two configuration of these types of puzzles.

We can also note that the tree we build will in fact give us the shortest way to go from any configuration to the configuration of the root. Indeed, when building the i th level of the tree, we will go through all puzzles that can be reached in i steps from the root, and only add them if they are not already in a higher level, i.e. that we have not yet found a shorter way to reach them.

2.3 Disadvantages

It is important though to see that this method gives us only the shortest path between the root and a random configuration, and not between any two random configurations.

We also wanted to go further and check if 3×3 puzzles with 9 colours could be solved. But whereas we had 1680 configurations for puzzles with 3 colours, this new type of puzzle had 362880 possible configurations. This meant that despite leaving our program to run for 2 days we were unable to get a result.

We can note here that figuring out how to solve a 3×3 with 9 colours was of particular importance. It would allow us to solve any larger puzzle, with as many colours as it has dots. Indeed we would be able to exchange any two dots of the puzzle by combining exchanges inside 3×3 squares, thusly allowing us to solve any larger puzzle.

3 Applying *Dijktsra's algorithm* to our problem

3.1 Main idea

We now wanted to find a method that allowed us to find the shortest path between any two random configurations of a given type of puzzle. To do so we turned to *Dijsktra's algorithm*, more noteably used to find the shortest path between two cities given a set of roads and a set of cities. More generally put, it is an algorithm that, given a graph and a source node, constructs a minimum spanning tree, i.e. a set of edges that connects all vertices, and has the least possible combined *weight* of edges.

Hence we first computed a graph, i.e. a set of edges and a set of vertices, containing all possible configurations of a given puzzle. Whereas usually the weight of an edge would be of particular significance, for example the length of a road, we will here have all edges have an equal weight of 1, as each edge will represent one movement on our puzzle.

With our graph completed, we can now find the shortest possible path between two random configurations by creating a minimum spanning tree with one of the configurations as a source node, finding the other one in the minimum spanning tree, and following the edges, or movements, in said tree until we reach the source node.

It is also worth mentioning that this method allows us to easily go between any two random configurations as we do not need to calculate each possible configuration again each time we have a different source or root node.

3.2 Results

In practice, except for the fact that it allowed us to go between any two random configurations without computing all the configurations again, it yielded few useful results, as when we only needed to compute one path between two configurations, i.e. we didn't reuse the tree, the first method wound up being faster.

However we later realised that it could be useful, as we managed to adapt it to solve puzzles with 2 colours of size larger than 3.

4 First step towards an algorithmic approach

From now on, we'll use coordinates to refer to each single square within the puzzle. The origin $(0, 0)$ is the first square in the first line. If we move from one square to the one on its right, we increase the first coordinate by 1. If we go to a lower square, we increase the second coordinate by 1 as well.
We will describe every rotation by a tuple of the form $(X, Y, direction)$. The coordinates (X, Y) are those of the first square in the first line of the 2×2 square that we rotate. *direction* is a boolean - if it's true (T), we turn the square clockwise, if it is false (F), we turn it anti-clockwise.
We now consider the following 3×3 puzzle with 9 colours represented by numbers from 1 to 9.

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

We tried to exchange the 1 and 2 in the puzzle without changing the rest of it and found the following sequence of rotations:

$$\begin{array}{ccc} \boxed{\begin{array}{cc} 1 & 2 \\ 4 & 5 \end{array}} & 3 \\ 6 \\ 7 \quad 8 \quad 9 \end{array} \xrightarrow{(0,0,F)} \begin{array}{ccc} 2 & 5 & 3 \\ \boxed{\begin{array}{cc} 1 & 4 \end{array}} & & 6 \\ \boxed{\begin{array}{cc} 7 & 8 \end{array}} & & 9 \end{array} \xrightarrow{(0,1,F)} \begin{array}{ccc} 2 & 5 & 3 \\ & \boxed{\begin{array}{cc} 4 & 8 \end{array}} & 6 \\ & \boxed{\begin{array}{cc} 1 & 7 \end{array}} & 9 \end{array} \xrightarrow{(0,1,F)} \begin{array}{ccc} 2 & \boxed{\begin{array}{cc} 5 & 3 \end{array}} \\ 8 & \boxed{\begin{array}{cc} 7 & 6 \end{array}} \\ 4 \quad 1 \quad 9 \end{array}$$

$$\xrightarrow{(1,0,F)} \begin{array}{ccc} 2 & 3 & 6 \\ 8 & \boxed{\begin{array}{cc} 5 & 7 \end{array}} \\ 4 & \boxed{\begin{array}{cc} 1 & 9 \end{array}} \end{array} \xrightarrow{(1,1,T)} \begin{array}{ccc} 2 & \boxed{\begin{array}{cc} 3 & 6 \end{array}} \\ 8 & \boxed{\begin{array}{cc} 1 & 5 \end{array}} \\ 4 \quad 9 \quad 7 \end{array} \xrightarrow{(1,0,T)} \begin{array}{ccc} 2 & 1 & 3 \\ 8 & \boxed{\begin{array}{cc} 5 & 6 \end{array}} \\ 4 & \boxed{\begin{array}{cc} 9 & 7 \end{array}} \end{array}$$

$$\xrightarrow{(1,1,T)} \begin{array}{ccc} 2 & 1 & 3 \\ \boxed{\begin{array}{cc} 8 & 9 \end{array}} & & 5 \\ \boxed{\begin{array}{cc} 4 & 7 \end{array}} & & 6 \end{array} \xrightarrow{(0,1,T)} \begin{array}{ccc} 2 & 1 & 3 \\ 4 & \boxed{\begin{array}{cc} 8 & 5 \end{array}} \\ 7 & \boxed{\begin{array}{cc} 9 & 6 \end{array}} \end{array} \xrightarrow{(1,1,F)} \begin{array}{ccc} 2 & 1 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

Since we can permute 1 and 2, we can now also exchange any 2 directly adjacent dots by performing a set of rotations to bring them to positions $(0, 0)$ and $(1, 0)$, using the movements that allowed us to exchange 1 and 2 before, and then performing the first set of rotations, each in the other direction, and in reverse order.

Now suppose we want to exchange 1 and 3. To do so we can first exchange the elements in positions $(0, 0)$ and $(1, 0)$, then the ones in positions $(1, 0)$ and $(2, 0)$, and lastly exchanging the elements in positions $(0, 0)$ and $(1, 0)$ again.

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \xrightarrow{1 \leftrightarrow 2} \begin{array}{ccc} 2 & 1 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \xrightarrow{1 \leftrightarrow 3} \begin{array}{ccc} 2 & 3 & 1 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \xrightarrow{2 \leftrightarrow 3} \begin{array}{ccc} 3 & 2 & 1 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

Using a similar method we can now exchange any two dots on the same row or column inside a 3×3 puzzle.

This then allows us to solve any 3×3 puzzle with 9 colours, which in turn, as mentioned in 2.3, allows us to solve any larger puzzle.

5 A simple algorithm

5.1 Main idea

As mentioned before, calculating all possible configurations in advance was a highly inefficient method, only allowing us to solve small puzzles. To get around this issue, we tried to find an

algorithm, which, given the current state of the puzzle, chooses a next movement, and does so until the puzzle has been solved.

We know that such an algorithm can exist, as we discovered that any puzzle larger than 3×3 can be solved. In order to implement our algorithm, we used the sets of movements found in section 4 to allow us to switch dots.

Firstly we went through the whole puzzle to search the dots that should be at the first level, and one by one, put them in the first level, regardless of the order inside said level. We repeated this process for each level until each dot was at the right height.

We then iterated through each level, switching the dots around to put them into the right place, to end up with the configuration of a given solution.

5.2 Results

This approach proved far better in terms of computational speed, being able to solve 20×20 puzzles in a short time.

5.3 Drawbacks

Whereas this approach allowed us to solve far larger puzzles, it was very inefficient in terms of the number of moves used. Trying to go from the following configuration:

$$\begin{array}{cccc} 400 & 399 & \dots & 381 \\ 380 & 379 & \dots & 361 \\ \vdots & \vdots & \ddots & \vdots \\ 19 & 18 & \dots & 1 \end{array}$$

to this solution:

$$\begin{array}{cccc} 1 & 2 & \dots & 20 \\ 21 & 22 & \dots & 40 \\ \vdots & \vdots & \ddots & \vdots \\ 381 & 382 & \dots & 400 \end{array}$$

yielded a solution consisting of a total of 133592 steps, which, as we will see further on, is very far from being optimal.

6 Optimising our algorithm

6.1 Main idea

The first thing one could notice, was that when moving a dot upwards to the level it should be at, it does not matter wether the levels below it are modified or not, meaning we can simply perform one rotation per level to bring the dot up to one level under which it should be, and then put it in the right level without modifying that level. To do so as efficiently as possible, we figured out how to move up a dot in a 3×3 without changing the first row, doing this in far fewer movements than it took us before, where we left the rest of the puzzle untouched.

Let us suppose we want to move the first dot of the second row to the first row. The method from Section 4 would have yielded the following result:

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \xrightarrow{1 \leftrightarrow 2} \begin{array}{ccc} 4 & 2 & 3 \\ 1 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

using a total of 11 movements, whereas with our new method which used the set of movements:

$$(0, 1F) \to (0, 0, F) \to (0, 1, T) \to (0, 0, T)$$

we obtained the following result in 4 movements:

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{matrix} \xrightarrow{4 \to 1} \begin{matrix} 4 & 2 & 3 \\ 1 & 8 & 6 \\ 7 & 5 & 9 \end{matrix}$$

We figured out many similar ways of switching dots whilst only preserving the first row or second row. We were also able to find more effective ways of switching two dots while preserving the rest of the 3×3, finding a solution for each specific needed case. All of these sets of movements are listed in the following section.

6.2 Results

All the optimisations mentioned before allowed us to solve the example in the Section 5.3 in 13208 steps instead of 133592.

6.3 Drawbacks

Whilst this algorithm allowed us to solve our 20×20 in approximatively 10 times less steps, we will see that there is still room for improvement.

Another issue with the program was that it would not work if there were multiple dots with the same colour. If we suppose that we have the following puzzle:

$$\begin{matrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{matrix},$$

and the following solution:

$$\begin{matrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{matrix},$$

Then the program would first try to put all elements that should be in the first row in the first row. To do so it would first go through the values of our first puzzle, starting at the top left and then going through row after row from left to right.
It would first check the number 1 and notice that the solution contains a number 1 in the first row, hence it would not move it. It would do the same for next two values in the puzzle, which are also 1. It would thus incorrectly think that the first row contains all the elements it should. It would now try to find three 1's in the last two rows, but would only find one, thus resulting in an infinite loop.

This issue can be avoided by assigning a different number to each dot, even if they have the same colours. However this will result in a solution with more steps than necessary.

7 Useful permutations

Listed here are the sets of movements to switch dots in a 3×3 puzzle, some of which preserve the first line, the first two lines or the whole puzzle. We will represent the 3×3 puzzle as follows:

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

- **Preserve the first line**:
  - $1 \leftrightarrow 2$: $(0,0,F),(1,0,F),(0,1,T),(1,0,T)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 2 & 1 & 3 \\ 7 & 4 & 6 \\ 8 & 5 & 9 \end{array}$$

  - $2 \leftrightarrow 3$: $(1,0,T),(0,0,T),(1,1,F),(0,0,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 3 & 2 \\ 4 & 6 & 9 \\ 7 & 5 & 8 \end{array}$$

  - $1 \leftrightarrow 3$: $(0,0,F),(0,1,T),(1,0,T),(1,0,T)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 3 & 2 & 1 \\ 4 & 6 & 5 \\ 7 & 8 & 9 \end{array}$$

  - $7 \to 1$: $(0,0,F),(0,1,T),(0,0,T)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 7 & 2 & 3 \\ 1 & 5 & 6 \\ 8 & 4 & 9 \end{array}$$

  - $4 \to 1$: $(0,1,F),(0,0,F),(0,1,T),(0,0,T)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 4 & 2 & 3 \\ 1 & 8 & 6 \\ 7 & 5 & 9 \end{array}$$

  - $8 \to 2$: $(0,0,T),(0,1,F),(0,0,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 8 & 3 \\ 4 & 2 & 6 \\ 5 & 7 & 9 \end{array}$$

  - $5 \to 2$: $(0,1,T),(0,0,T),(0,1,F),(0,0,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 5 & 3 \\ 7 & 2 & 6 \\ 4 & 8 & 9 \end{array}$$

– $9 \to 3$: $(1,0,T),(1,1,F),(1,0,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 9 \\ 4 & 5 & 3 \\ 7 & 6 & 8 \end{array}$$

– $6 \to 3$: $(1,1,T),(1,0,T),(1,1,F),(1,0,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 6 \\ 4 & 8 & 3 \\ 7 & 5 & 9 \end{array}$$

- **Preserve the two first lines**:

– $4 \leftrightarrow 6$: $(0,1,T),(1,1,F),(1,1,F),(0,1,T),(0,1,T),(1,1,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 3 \\ 6 & 5 & 4 \\ 9 & 8 & 7 \end{array}$$

- **Preserve all the lines**:

– $4 \leftrightarrow 7$: $(0,0,F),(1,1,F),(0,0,T),(0,1,T),(1,1,T)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 3 \\ 7 & 5 & 6 \\ 4 & 8 & 9 \end{array}$$

– $6 \leftrightarrow 9$: $(1,0,T),(0,1,T),(1,0,F),(1,1,F),(0,1,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 9 \\ 7 & 8 & 6 \end{array}$$

– $5 \leftrightarrow 8$: $(0,1,F),(0,1,F),(0,0,F),(1,1,F),(0,0,T),(0,1,T),(1,1,T),(0,1,F),(0,1,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 8 & 6 \\ 7 & 5 & 9 \end{array}$$

– $7 \leftrightarrow 8$: $(0,1,T),(1,1,T),(0,0,F),(1,1,F),(0,0,T)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 8 & 7 & 9 \end{array}$$

– $8 \leftrightarrow 9$: $(1,1,F),(0,1,F),(1,0,T),(0,1,T),(1,0,F))$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 9 & 8 \end{array}$$

– $4 \leftrightarrow 5 : (0,1,F),(0,0,F),(1,1,F),(0,0,T),(0,1,T),(1,1,T),(0,1,T)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 3 \\ 5 & 4 & 6 \\ 7 & 8 & 9 \end{array}$$

– $5 \leftrightarrow 6 : (1,1,T),(1,0,T),(0,1,T),(1,0,F),(1,1,F),(0,1,F),(1,1,F)$

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array} \longrightarrow \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 6 & 5 \\ 7 & 8 & 9 \end{array}$$

8 Second Algorithm

8.1 Main idea

In this second algorithm, we tried to write an iterative function which first solved the first row and first column of a puzzle of size $n \times n$ ($n \geq 4$). We do this by first finding the first element in the first row and by moving it to $(0,1)$. Then we search for the second element and move it to $(0,2)$, making sure to not move the first element. We repeat these steps until we reach the last element in the first row. After finding it, we move it to $(n-1,1)$. The following series of permutations applied to the rightmost and topmost 3×3 square in our puzzle allow us to move it to $(n-1,0)$ without disturbing the first row:

$$\begin{array}{ccc} 1 & 2 & \circ \\ \bullet & \bullet & 3 \\ \bullet & \bullet & \bullet \end{array} \xrightarrow{(0,0,F)} \begin{array}{ccc} 2 & \bullet & \circ \\ 1 & \bullet & 3 \\ \bullet & \bullet & \bullet \end{array} \xrightarrow{(1,0,F)} \begin{array}{ccc} 2 & \circ & 3 \\ 1 & \bullet & \bullet \\ \bullet & \bullet & \bullet \end{array} \xrightarrow{(0,0,T)} \begin{array}{ccc} 1 & 2 & 3 \\ \bullet & \circ & \bullet \\ \bullet & \bullet & \bullet \end{array}$$

This gives us a puzzle like this:

$$\begin{array}{ccccc} 1 & 2 & 3 & \cdots & n \\ \bullet & \bullet & \bullet & \cdots & \bullet \\ \bullet & \bullet & \bullet & \cdots & \bullet \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \bullet & \bullet & \bullet & \cdots & \bullet \end{array}$$

Afterwards we repeat the same steps for the first column until we reach the last element. We move it to $(1,n-1)$ and then apply the following rotations to move it to its place without disturbing the first row and the rest of the first column:

$$\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & \bullet & \bullet & \bullet \\ 9 & \bullet & \bullet & \bullet \\ \circ & 13 & \bullet & \bullet \end{array} \xrightarrow{(0,1,T)} \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 9 & 5 & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet \\ \circ & 13 & \bullet & \bullet \end{array} \xrightarrow{(0,2,T)} \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 9 & 5 & \bullet & \bullet \\ \circ & \bullet & \bullet & \bullet \\ 13 & \bullet & \bullet & \bullet \end{array} \xrightarrow{(0,1,F)} \begin{array}{cccc} 1 & 2 & 3 & 4 \\ 5 & \bullet & \bullet & \bullet \\ 9 & \circ & \bullet & \bullet \\ 13 & \bullet & \bullet & \bullet \end{array}$$

We then get the following configuration:

$$\begin{array}{ccccc} 1 & 2 & 3 & \cdots & n \\ 2 & \bullet & \bullet & \cdots & \bullet \\ 3 & \bullet & \bullet & \cdots & \bullet \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n & \bullet & \bullet & \cdots & \bullet \end{array}$$

We then solve the second row and second column in the puzzle by working in the $(n-1) \times (n-1)$ yet unsolved square in the puzzle. We repeat this, until the unsolved square is a 3×3 puzzle.

In order to solve the remaining 3×3 puzzle, we solve the first row as we did before. Then we locate the element that should be placed at $(0, 2)$ and move it to $(0, 1)$. We do the same for the element that should be placed at $(0, 1)$ and move it to $(1, 1)$. We rotate the leftmost and topmost 2×2 square:

$$\begin{array}{ccc} 1 & 2 & 3 \\ 7 & 4 & \bullet \\ \bullet & \bullet & \bullet \end{array} \xrightarrow{(0,1,F)} \begin{array}{ccc} 1 & 2 & 3 \\ 4 & \bullet & \bullet \\ 7 & \bullet & \bullet \end{array}$$

Now all that remains to do is to solve the 2×2 square at the bottom left.
We can rotate this square once or twice in either directions to get one of the following cases:

$$\begin{array}{ccc} 1 & 2 & 3 \\ 4 & \mathbf{6} & \mathbf{5} \\ 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & \mathbf{9} \\ 7 & 8 & \mathbf{6} \end{array} \quad \begin{array}{ccc} 1 & 2 & 3 \\ 4 & \mathbf{8} & 6 \\ 7 & \mathbf{5} & 9 \\ 1 & 2 & 3 \\ 4 & 5 & \mathbf{8} \\ 7 & \mathbf{6} & 9 \end{array} \quad \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & \mathbf{9} & \mathbf{8} \\ 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{array}$$

In the last case we don't need to do anything, our square is solved. In all other cases, we can exchange our elements using the permutations found in section 7.

8.2 Results

This new algorithm allowed us to solve puzzles of size all the way up to 30×30. If we apply it to the same puzzle and solution as in 5.3, we needed only 7126 rotations to solve our puzzle.

We also used it to solve puzzles with only two colours by using Djikstra's algorithm to solve our 3×3 puzzles instead of the aforementioned method.

8.3 Drawbacks

Whilst this algorithm seems to give a well optimised set of movements, it does not necessarily give us the shortest way of solving our puzzle.

9 Examples

We take as an example the following puzzle:

$$\begin{array}{cccc} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{array}$$

This will be our solution. We desire to solve the following puzzle:

1	1	0	0
1	1	0	1
1	0	0	0
0	1	0	1

Then we have:

Algorithm used	Number of rotations
First algorithm	530
Second algorithm	20
Second algorithm applied to Djikstra's	17

10 Comparisons in 3×3 puzzles with two colours

We now generated all possible configurations for 3×3 puzzles with the two colours white (0) and black (1), where white was represented 5 times and black 4. Then for every possible combination of a solution and a puzzle amongst all possible combinations, we computed the maximum number of rotations needed. We also stored the second largest number of needed rotations.

For our first algorithm, the two largest numbers of needed rotations are 240 and 264. The respective solutions and puzzles are:

Solutions	Puzzles	Number of rotations	Number of rotations (Djikstra)
1 1 1 0 0 0 0 0 1	0 1 0 0 1 1 0 0 1	264	3
1 1 0 1 1 0 0 0 0	1 0 1 0 0 1 0 1 0	240	4
1 0 1 1 1 0 0 0 0	1 0 1 0 0 1 0 1 0	240	4
0 1 1 1 0 1 0 0 0	0 0 1 1 1 0 0 1 0	240	3
1 1 1 1 0 0 0 0 0	0 0 0 1 1 0 1 1 0	240	6

For the second algorithm, we get:

Solutions	Puzzles	Number of rotations	Number of rotations (Djikstra)
0 0 1 0 1 1 1 0 0	1 1 0 1 0 0 0 0 1	21	4
0 0 1 0 0 0 1 1 1	1 1 0 1 0 0 0 0 1	20	4
1 0 0 1 0 0 0 1 1	0 0 1 0 0 1 1 1 0	20	4
0 1 1 0 1 1 0 0 0	1 1 0 1 1 0 0 0 0	20	4

We notice that the worst case in the first algorithm require so many more rotations than our second one does. The second one still needs four times more rotations than what would be possible, but is still reasonably fast.

11 Code

11.1 First approach

CODE ICI

11.2 Djikstra's algorithm

CODE ICI

11.3 First algorithm

CODE ICI

11.4 Second algorithm

We kept the same class *puzzle*, as well as the methods $\_\_str\_\_$, *rotate* and $\_\_eq\_\_$.
We added the following methods: