A DEPTH-FIRST SEARCH APPROACH TO THE TERRACOTTA PUZZLE

1. Description of the Problem

TODO

2. Implementation with depth-first search

The idea of solving a puzzle with a depth-first algorithm goes back to Brandt et al. (2002), who solved a 3×3 puzzle. In theory, each piece of the puzzle can be put down in four different rotations, at any one of the nine different places. Brandt et al. (2002) already derived that even such a relatively small puzzle has a total of $4^8 \times 9! = 23,781,703,680$ possibilities of being arranged. Of course, with modern compute resources, one can try out the whole search space in hope of finding a solution, but this grossly inefficient. Brandt et al. (2002) recognized this and were able to find a solution after a mere 672 calls to the depth first algorithm instead. In what follows, we first describe depth-first search (DFS) algorithms and their use for solving puzzles, similar to what Brandt et al. (2002) did. We then present the implementation of the algorithm for our specific problem and point out the necessary adjustments to find a solution in our case.

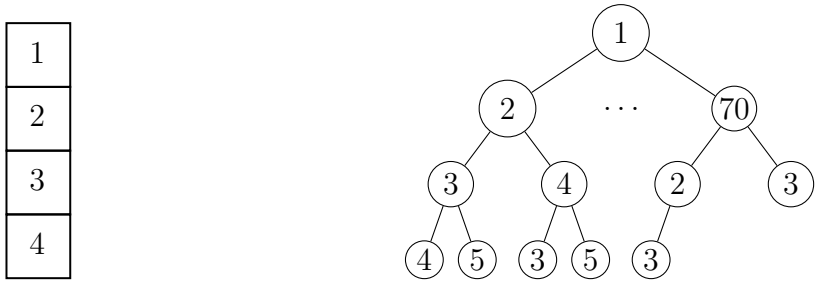
2.1. **Depth-First Search**

There are many references to DFS, and we refer the reader to Even (2011) for a more thorough treatment. Here we merely try to convey the intuition behind it. It is useful to think about the problem as a tree graph, where we start with a specific puzzle piece and then must choose the next piece.

For the simplest case, assume that we just want to arrange four pieces in a line. For the puzzle we consider here, there are 70 unique pieces[1]. Suppose, as represented in Figure 1, we start with the puzzle piece with id 1. For the second space, we then have the choice of every remaining piece (with indices 2-70), which results in 69 nodes for the second position in the puzzle and so on. As can be seen, the tree becomes extremely large quite quickly (given that just for the first position, there would be 70 separate trees to search.

Date: February 2023.

[1]In theory, we can also rotate all of them, which would simply add further nodes to the tree. We refrain from this here.



(A) Simple case for puzzle pieces (B) Tree representation of the search space

FIGURE 1. Starting point for search

The intuition of the depth first search is then rather simple. Instead of trying every possible arrangement, we let each branch grow until the final goal cannot be achieved (that is, until we do not find a piece that matches to our tree so far). As an example, suppose we follow the left branch in Figure 1b up to node 2. If we cannot find a fitting piece among the remaining 68 pieces, we know that *none* of the arrangements in the fourth layer can be a solution, and we can eliminate them from the search, that is, with 68 checks in the third layer, we can eliminate $68 \times 67 = 4,556$ possibilities. Now if the tree would be much deeper than four layers, this effect would compound and we can shrink the search space drastically.

Now all that remains to define is what to do if we cannot find a fitting puzzle piece. Here, Brandt et al. (2002) used the technique of *backtracking*. This is rather simple, whenever we are stuck, we just go back one level in the tree and choose another branch from there. We can then repeat the search we just performed until we have either found a solution or there are no more branches to select. In this case, we simple "backtrack" up one level. Following the example, if we cannot find a matching piece after we chose the sequence $1 - 2$, we backtrack to node 1, choose another piece (for example piece 3) and continue our search from there.

With this simple algorithm, Brandt et al. (2002) were able to find a solution after only 672 calls to the algorithm, a very manageable number of tries (even for a search by hand).

2.2. **Specifics of the Terracotta Puzzle**

Although the basic structure is similar to what was considered by Brandt et al. (2002), the Terracotta Puzzle we consider here is a lot larger. The issue with exponential growth in the solution-spce is that even small increases in the size of the

problem make the problem extremely difficult to solve. There are 70 unique pieces in our set, by the same reasoning as above, there will be $4^{69} \times 70! \approx 4.18 \times 10^{141}$ possible solutions, which is about one and a half times as many atoms as there are in the universe and about 1.76×10^{131} as many possible solutions than in the 3×3 case. If we are to find a solution, we will need an implementation of the depth-first search algorithm that saves as much time as possible for each compute step. The different steps are explained in the following paragraphs.

2.2.1. *Python Algorithms*

We opted to implement the algorithm in the Python programming language as a first try[2]. Python is not the fastest programming language, but it enables quick and easy prototyping. Given that the search space is large beyond imagination, faster computing will only get us that far in any case.

Algorithm 1 Recursive Depth-First Search (DFS)

Input: Set of available puzzle pieces, current index, fit function
Output: Vector of length 70, with puzzle piece indices at corresponding fields

```
if current index = 1 then
    index ← 1
    pieces ← All puzzle pieces
else if current index > 1 then
    pieces ← Available puzzle pieces
    index ← current index
end if
for piece_next in pieces do
    for rotation in {0, 90°, 180°, 270°} do
        if fits(piece_next) then
            pieces_next ← pieces_next − piece_next
            index_new ← index + 1
            DFS(pieces_next, index_new)
        end if
    end for
end for
```

We consider the case first where we arrange the puzzle pieces on a 7×10 rectangular grid, where we proceed by indices as depicted in Figure 2a. As a first try, we then implement a recursive form of the DFS algorithm as used in Brandt et al. (2002). The steps of the algorithm are summarised in Algorithm 1. Although the algorithm finds small solutions fairly quickly, it is not able to find a solution to our problem

[2]The algorithms can be found on github.com/phi-ra/terracotta-solver

1	2	3	4	5	6	7	8	9	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	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70

(A) Arrangement type I

1	8	15	22	29	36	43	50	57	64
2	9	16	23	30	37	44	51	58	65
3	10	17	24	31	38	45	52	59	66
4	11	18	25	32	39	46	53	60	67
5	12	19	26	33	40	47	54	61	68
6	13	20	27	34	41	48	55	62	69
7	14	21	28	35	42	49	56	63	70

(B) Arrangement type II

FIGURE 2. Different arrangements of the puzzle indices

(at least in reasonable time). After around 2 hours of compute time, the algorithm did not find a solution that goes beyond 50 pieces, a long shot from the desired 70 pieces.

To improve compute time, we can already reduce the search space drastically by observing that not all pieces are unique for every rotation. Given that there are four different directions that the arrows can point to, there should be $4 \times 4 \times 4 \times 4 = 256$ unique combinations. But we have already deduced that there are only 70 unique pieces up to a rotation. Note that this implies that not every piece has four different rotations without being equivalent to another piece. By realizing this we are only left with $256 \times 70! \approx 3.07 \times 10^{102}$ possible combinations, still huge but a lot smaller. The issue is that we then need to eliminate all rotations of the available set of pieces once we have used it. Thankfully, this can be implemented quite easily in Python with lookups in a *dictionary* a specific data structure of Python that is quite efficient.

Further, we can help our algorithm with the grid we chose. Since the DFS algorithm accelerates the search by cutting entire branches from the tree, initial arrangement in Figure 2a is not ideal. Since having to satisfy more constraints makes it more likely to fail (that is, to not find a fitting piece), we should have a layout that encourages early failure as much as possible. An arrangement in the form of 2b already does a better job, as puzzle piece 9 already needs to satisfy two constraints (fit above to puzzle piece 8 and to puzzle piece 2 on the left side). Of course, there are architectures that are even more suited for this, but it turns out that the arrangement in 2b is particularly easy to implement in Python.

Finally, we can take advantage of parallel processing. Since the DFS algorithm searches sequentially across the indices (that is, it starts with piece 1 in position one, then tries piece 2 on position 2 etc..), we can try different initial arrangements in parallel in the hope of finding one where the algorithm progresses faster. Most modern computer are able to process multiple tasks in parallel and we can (deterministically) shuffle the pieces for each task and run a few of them at the same time.

With all these improvements we now have an algorithm described in Algorithm 2. Now the algorithm proceeds quite quickly, finding a solution for up to 68 pieces after a few seconds and a solution for all 70 pieces after around two hours.

Algorithm 2 Accelerated Recursive Depth-First Search

Input: Set of available puzzle pieces, current index, fit function, rotation
Output: Vector of length 70, with puzzle piece indices at corresponding fields

```
if current index = 1 then
    index ← 1
    pieces ← All puzzle pieces
else if current index > 1 then
    pieces ← Available puzzle pieces
    index ← current index
end if
for piece_next in pieces do
    if fits(piece_next) then
        pieces_next ← pieces
        for rotation_piece in rotation[piece_next] do
            pieces_next ← pieces_next − rotation_piece ▷ Remove according to pre-calculated "rotation" dictionary
        end for
        index_new ← index + 1
        DFS(pieces_next, index_new)
    end if
end for
```

3. Solutions

References

Brandt, K., Burger, K., Downing, J. & Kilzer, S. (2002), 'Using backtracking to solve the scramble squares puzzle', *J. Comput. Sci. Coll* **17**(4).

Even, S. (2011), *Graph algorithms*, Cambridge University Press.