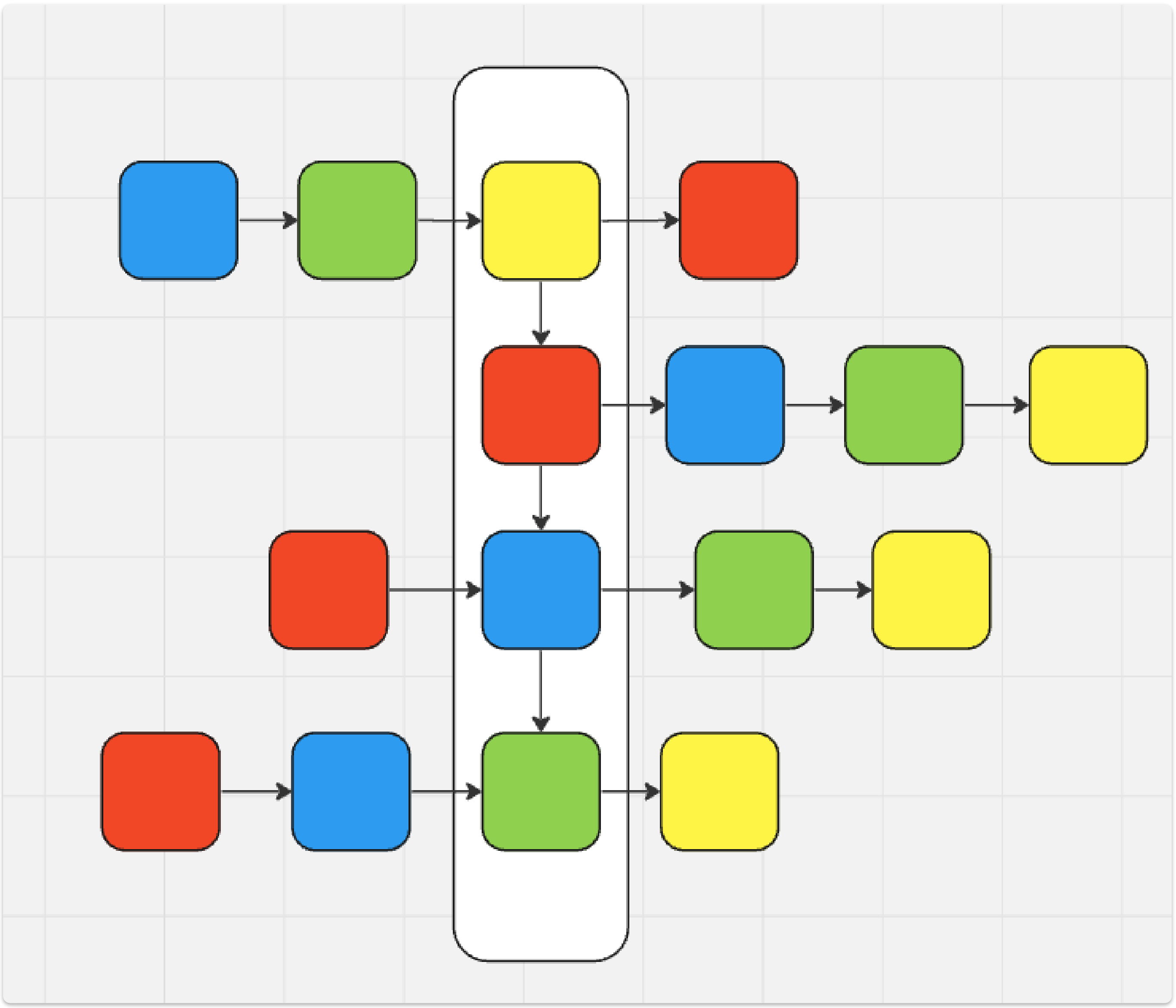
Øystein Seel

The task involves solving a cube tower puzzle with a handful og different algorithms. This report contains a description of algorithms, implementation, problem instances, the heuristic chosen, and analysis.

Above is an illustration of the initial tower with the cubes within in the white rectangle facing forward. The arrows illustrate the single possible direction of rotation. This illustrates a tree-like structure and all the algorithms implemented are well suited for the traversal of such a structure.



Algorithms:

Iterative Deepening Depth First Search (IDDFS), Breadth-First Search (BFS), Depth-First Search (DFS), and A* Search share the common goal of finding a solution path from an initial tower configuration to a goal state where all cubes are aligned by color. They all explore the space of possible configurations, generating and evaluating successor states to discover a sequence of moves that solves the puzzle. Each algorithm applies a different strategy to manage the exploration order and efficiency, aiming to optimize search performance regarding time, memory, or both. Here is a brief outline of all the algorithms:

BFS:

Searches the breadth of each level of the tree and uses a python `deque` for handling all the different states of the tower. The algorithm has an iterative approach with the use of a for and a while-loop.

DFS:

Searches the depth of the tower and stores all the visited states in a `set()`. It has a recursive approach.

A Star:

Creates a list of states to explore and prioritising them with the help of a cost function. This function calculates the cost from a start node to the target.

As long as the heuristic is properly chosen, the algorithm will provide the shortest path.

Iddfs:

Searches all the generated spaces within the limit of a given depth. If the depth is exceeds the limit, the algorithm terminates. This algorithm uses the space efficiency of DFS and the completeness of BFS.

Implementation:

Some time was spent initially getting a proof of concept with the provided precode. When the code worked, algorithms classes and other methods were separated into modules.

When all the algorithms had their own module, the analysis and creation of a main module started. There were some problems finding a suitable tool for memory usage, and the one selected performed poorly.

Problem instances:

Instead of running five specific configurations the main module used a method for randomly generating the initial configuration:

```python
def generate_random_colors():
    colors = ['red', 'blue', 'green', 'yellow']
    random_colors = random.choices(colors, k=4)
    return random_colors
```

The program was tested multiple times and the results behaved similarly regardless of the generated configuration. More on this in the analysis.

Heuristic:

Both Manhattan and Euclidean distance were tested for running the A* algorithm. They both performed worse then a more simple approach:

```python
def calculate_heuristic(configuration):
    return sum(1 for color in configuration if color != configuration[0])
```

One could say that this approach is more directly aligned with the programs objective and simplicity.

Analysis:

Of all the algorithms implemented, DFS was the worst performing. As show by the screenshot bellow, DFS has a significantly higher amount cube rotations.

Besides the number of rotations, DFS also had a higher execution time then the others. The same happened with the use of memory. Although the use of the `psutil` package did give some confusing results, where the usage for A star was negative (shown bellow) when calculated like this:

```python
memory_init = psutil.virtual_memory().used
# run algortihm....
memory_end = psutil.virtual_memory().used - memory_init
```

Plots

1. A*
   Balances efficiency and effectiveness by by prioritizing paths that appear closer to the goal. The three heuristics tested, mostly outperformed the following algorithms. Sometimes the IDDFS has slightly better performance.
2. IDDFS
   Combines the benefits of BFS and DFS and with a low max_depth it performed almost as good as A*

3. BFS
   Had some memory-intensive runs worst of all or second to last
4. DFS
   Always performed the most number rotations and generally ended up last in terms of execution time.

Performance in solving the puzzle depended on the puzzle's complexity and the algorithm's specific implementation details, such as heuristic accuracy for A* or depth limits for IDDFS. Without any similar details for BFS and DFS they rarely if never performed best.

Challenges:

Getting the generate moves method to work was rather problematic. Handling the different rotation methods resulted in multiple chained for-loops. At least this performance penalty was applied similarly to all the algorithms. Determining how many moves to generate on each method call was also an issue. Create more moves would take longer to run, but might involve a actual state that solved the puzzle.

Given the limited problem space it was difficult to distinguish the algorithms from each other. There wasn't enough time to implement the bonus challenge or expand the cubetower structure. Maybe an expansion of the problempsace might have improved the DFS performance.

⊗ 1 Error in region