Matúš Rusňák ID : 116286

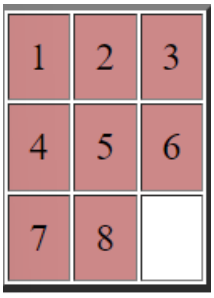
Prehľadávanie stavového priestoru - Assignemnt 1

1. Task definition

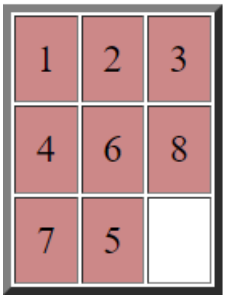
In this assignment I was given problem D. Which was to compare results of 2 different heuristics with the use of greedy algorithm. The algorithm is used to try to solve given puzzle board to look like another board. The program will either print out that it is not possible or give you the correct order of operators to use to solve the puzzle.

For example if picture A can be moved around to look like picture B.

Picture A)



Picture B)



The only way you can change the board is to move the neighboring tiles to an empty slot. In this scenario it is possible by using move operators in this order: [RIGHT, DOWN, LEFT, UP]. By using this order of operators, the board on picture A will look like the board on picture B. Therefore, it is solvable.

Operators = [UP, DOWN, LEFT, RIGHT]

2. Algorithm used

To solve this problem a simple algorithm is used:

1. Create first node with starting board and store in yet not processed nodes.
2. If no unprocessed nodes exist, end with failure, meaning no possible solution exists
3. Fetch best next node from still unprocessed nodes and set it to current node (best = node with lowest heuristic value, how I calculate heuristic is explained down in point 3)
4. If this node has heuristic value 0 end with success and print out solution
5. From possible operators (where can you move next) create children nodes and store current node to processed nodes.
6. Store children nodes to yet unprocessed nodes.
7. Go back to step 2

After this algorithm successfully runs, we will know if given boards are possible to solve or not. And if yes how to solve them

Matúš Rusňák ID : 116286

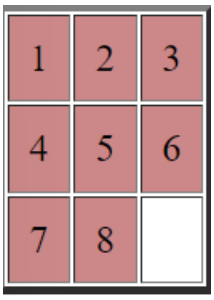
3. Heuristic calculation

As I mentioned before the point of my assignment is to compare results of 2 different heuristics. So, let's see how we calculate them.

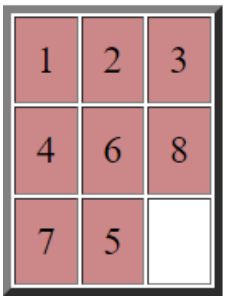
3.1 Heuristic 1:

This is calculated by sum number of tiles that are different from wanted board. Example:

Board A:



Wanted board:

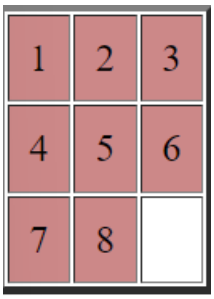


In this case tiles with values 5,6,8 are not in the correct state meaning heuristic value for board A =3.

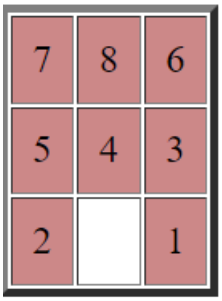
3.2 Heuristic 2

This is calculated offset of X and Y coordinates from wanted board.

Board A:



Wanted board:



Now value for board A will look like this: 4 + 3 + 1 + 1 + 1 + 1 + 2 + 2 = 15

This value is stored in node and based on it we choose which node will be processed next.

4. Code

4.1 Node

```python
class Node:
    """
    Body of node which stores necessary information.
    matrix = current state of board
    parent = pointer to its parent node from which matrix was created
    heuristic_value = calculated heuristic from end point of matrix
    """

    def __init__(self, matrix: np.array, parent, last_operator: str, heuristic_value: int) -> None:
        self.matrix = np.array(matrix)
        self.parent = parent
        self.last_operator = last_operator
        self.heuristic_value = heuristic_value
```

This node stores current state of board, it's parent node which is used at the end to determine the correct order of operators to solve given matrix and heuristic value of current board state. This is all the information that I needed to store in order to solve this problem. These nodes get stored in dictionaries by creating a hash from their board. They get stored in dictionary processed nodes if they have been already processed or in dictionary unprocessed nodes if they yet still wait to be processed.

4.2 Hashmaps class

```python
class Hashmaps:
    """
    class that stored processed and still unprocessed_nodes in dictionaries
    key to dictionary is a hash that is uniquely generated from every matrix
    """

    def __init__(self):

        self.processed_nodes = {}
        self.unprocessed_nodes = {}

    def _create_hash_from_matrix(self, matrix: np.array) -> str:
        """
        function that creates unique hash for given matrix with the help of sha256
        this hash is used to store nodes into dictionaries
        """
        hash = (sha256(matrix).hexdigest())
        return hash
```

This class stores 2 dictionaries. One for already processed nodes and one for still unprocessed nodes.

In the picture you can also see how I create hash. The function takes one argument matrix from which is created unique hash that is used to store nodes in dictionaries. Class also has other methods like add_processed, check_processed, add_unprocessed and others that are self-explanatory.

4.2.1 Find best next node

```
def find_best_next_node(self) -> Node:
    """
    method that find the best next node by sorting unprocessed_nodes by heuristic value
    then return that node with lowest heuristic value to be processed
    :return: Node with lowest heuristic value
    """
    lowest_heuristic_node = min(self.unprocessed_nodes.values(), key=attrgetter('heuristic_value'))
    hash = self._create_hash_from_matrix(lowest_heuristic_node.matrix)
    self.unprocessed_nodes.pop(hash)
    return lowest_heuristic_node
```

This is the function that I use to find node with the lowest heuristic value from unprocessed node. This is a crucial part of the algorithm because it narrows down which operator was best to use and removes that node from unprocessed. Meaning we will know that node was already processed.

4.3 Helper class

```
class Helper:
    """
    class that help with heuristic calculations, string manipulations,
    creating children from possible operations on given matrix and creating nodes
    """

    def __init__(self, end_matrix: np.array, hashmaps: Hashmaps, heurisitc_config: int) -> None:
        self.hashmaps = hashmaps
        self.end = end_matrix
        if heurisitc_config == 1:
            self.calc_heuristic = self.heuristic_1
        else:
            self.calc_heuristic = self.heurisitc_2
```

This class contains many methods. Heuristic calculations, creation of nodes, finding possible operators and overall processing of a node and others. The main class Puzzle uses this class to solve this puzzle

4.4 Puzzle class

```
class Puzzle:
    """
    class which takes given matrix and wanted matrix.
    solves given problem whether it is possible or not and gives and output to the console:
    1. if its solvable it prints the correct order of needed operations
    2. if its not solvable it informs that wanted matrix its not possible to achieve
    """

    def __init__(self, start: list, end: list, heuristic_config: int) -> None:
        self.start = np.array(start)
        self.end = np.array(end)
        self.hashmaps = Hashmaps()
        self.helper = Helper(self.end, self.hashmaps, heuristic_config)
        self.solve()
```

This is the main class that start the program. At takes as an argument starting state board, wanted state board and heuristic_config to set which heuristic calculation it should use. The correct examples of arguments:

start/end = [[1, 2, 3], [4, 5, 6], [7, 8, ‘m’]] – where ‘m’ represents empty slot in board

heuristic_config = 1 or 2 | where 1 = heuristic_1 calculation, and 2 = heuristic_2 calculation

One thing to note is that ‘start’ and ‘end’ have to have the same m*n size, but ‘m’ and ‘n’ can be whatever value you want

4.4.1 Solve method

```
def _solve(self, current_node: Node) -> bool:
    """
    main method processes node and find best next node to processes until heuristic_value = 0 or no more unprocessed
    nodes exists meaning possible solution does not exist
    """
    found_solution = True
    while current_node.heuristic_value != 0:
        self.helper.process_node(current_node)
        current_node = self.hashmaps.find_best_next_node()

        if len(self.hashmaps.unprocessed_nodes) == 0:
            found_solution = False
            break
    if found_solution == True:
        solution = self._create_solution(current_node)
        print(solution)
        return True
    else:
        print("Solution for {} array to move into {} array does not exist!".format(self.start, self.end))
        return False
```

This method solves the problem. The while loop inside repeats itself until it finds a node with heuristic value 0 meaning solution does exists or when dictionary with unprocessed nodes is empty meaning a solution does not exist. After that it find the correct order of operators and prints it out to console

5. Testing

In these tests I mainly focus on how much time it took to run, how much memory was used, how many nodes where processed and a comparison of time between the 2 heuristic calculations. These results have been generated on my PC and way wary based on which machine you run them. Results are printed into the console by running program test.py. Here are my results:

Test table

Test name	Heuristic type	Total time(s)	Number of nodes	Peak memory (KB)
1_1	1	0.043994	188	103.1563
1_2	2	0.042003	119	83.53125
2_1	1	0.06	191	114.4688
2_2	2	0.089598	191	159.5938
3_1	1	381.2928	181439	13877.16
3_2	2	313.0597	181439	13872.41
4_1	1	0.666605	2084	1347.219
4_2	2	0.630565	1124	785.8125
5_1	1	0.683589	1770	1394.906
5_2	2	2.209301	2692	2459.625
6_1	1	22.08319	28120	19753.13
6_2	2	0.701382	992	762.6875

5.1 Test table legend

Test name – name of the function in test.py

Heuristic type – type of heuristic used to calculate heuristic value, as mentioned above (type 1 or 2)

Total time – total time to run algorithm

Number of nodes - number of processed nodes until a conclusion is reached

Peak memory – peak memory of what algorithm needed to allocate

6. Conclusion

Results show heuristic type 2 is faster in most cases. It also mostly processes less nodes than heuristic type 1. That may be caused by the value calculation itself. By calculating heuristic 1 we only get number of tiles not in the correct place. But by calculating heuristic 2 we also get how far away the tile from its correct place is. Therefore, the next selected node with lowest heuristic value is most likely closer to solution than just lowest value node from heuristic type 1. Not to be confused by test 3 results but they were deliberately made to not have a solution. We can see a large number of processed nodes until we run out of them. Meaning the solution exists. This test however ran a long time. Therefore, I am also including a test_demo.py that skips this test

7.Dependencies

In order to run this program, you will need:

1. Python 3.10.7
2. NumPy 1.23.3

8. Sources

“Picture A, Picture B, Wanted board “- http://www2.fiit.stuba.sk/~kapustik/z2d.html, 2022