1 Implementation

This assignment was programmed using C++11.

1.1 Part 1

Most of the functionality for part 1 is encapsulated in a class called Matrix. The class was created specifically for this assignment. The complete class diagram can be seen in Figure 1.

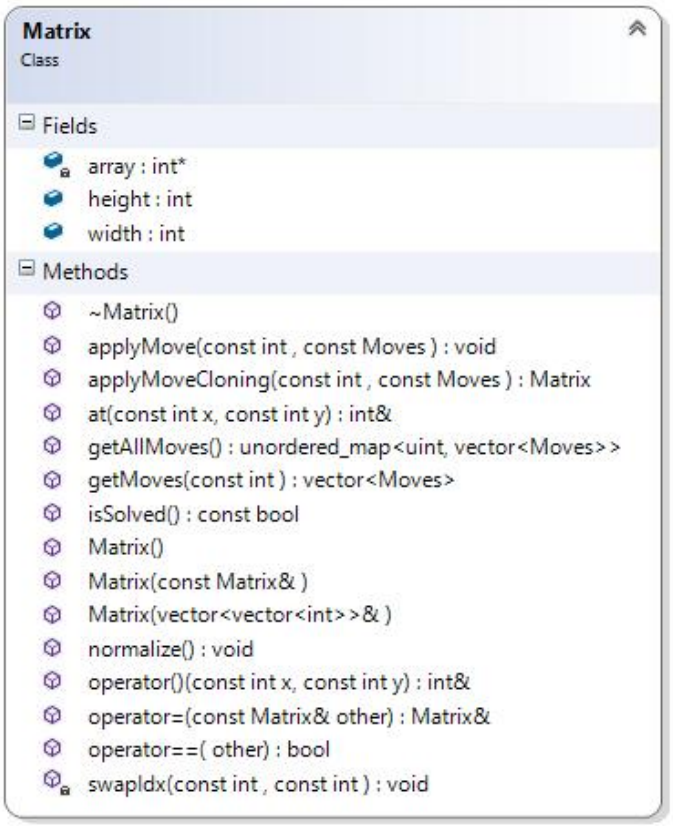

Figure 1: Class diagram for the Matrix class

1.1.1 State representation

To represent the state in the program, the main function first loads the .txt file containing the game state. Next, a Matrix object is created from the contents of the text file. The Matrix class provides functions for printing and cloning of a game state. For printing, the << operator is overloaded. For cloning, a copy constructor is provided as well as an overloaded = operator.

1.1.2 Puzzle Complete Check

The completeness of a puzzle is checked by iterating over all the rows and columns, searching for the value -1. If this value is present in the current game state, the puzzle is not solved, otherwise its solved.

1.1.3 Move Generation

The two functions “getMoves” and “getAllMoves” calculate and return all possible moves for a single piece and all pieces respectively. A move is considered possible for a given piece, if the value(s) of the cell(s) at the position the piece would be moving to is/are 0. Moves are encoded in an enum class shown in Figure 2. It is worthwhile to mention that a “Move” alone does not include a reference to a piece. These two values are managed separately throughout the program.

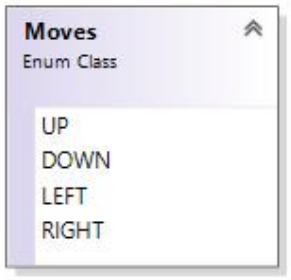

Figure 2: Move representation within the program

A Move can be applied to the current game state by calling the “applyMove” function. To create a new instance of the current game state that represents the applied move the function “applyMoveCloning” must be called.

1.1.4 State Comparison

Two game states are considered equal, if the contents of their matrices are the same. This is checked in the overloaded == operator.

1.1.5 Normalization

The normalization is implemented following the algorithm from the assignment sheet. The two functions “normalize” and “swapIdx” provide this functionality.

1.1.6 Random Walks

The random walk is implemented in a function named “randomWalk”. A shortened version of the function is shown in Codesnippet 1: The randomWalk function. It performs a possible, but random move on a random piece within a loop. The executed

move and the new game state are printed in every iteration.. It uses the previously created functions within the Matrix class, hence it can be kept short.

```
void randomWalk(Matrix& m, const uint n) {
    cout << m << endl;
    for (int i = 0; i < n; i++) {
        // 1. getAllMoves
        std::unordered_map<uint, std::vector<Moves>> allMoves =
m.getAllMoves();
        // 2. select one piece and move at random
        // .. LEFT OUT FOR READABILITY ..
        uint rand_piece = ..
        Moves rand_move = ..
        // 3. Execute move
        m.applyMove(rand_piece, rand_move);
        // 4. Normalize
        m.normalize();
        cout << "(" << rand_piece << "," << rand_move << ")"
            << endl << endl << m << endl;
        // 5. if goal, stop. else goto 1.
        if (m.isSolved()) return;
    }
}
```

Codesnippet 1: The randomWalk function. It performs a possible, but random move on a random piece within a loop. The executed move and the new game state are printed in every iteration.

1.2 Part 2

For the search algorithms Breadth First Search (BFS), Depth First Search (DFS) and Iterative Deepening Depth First Search (IDDFS) two new classes were built. Both class diagrams are shown in Figure 3.

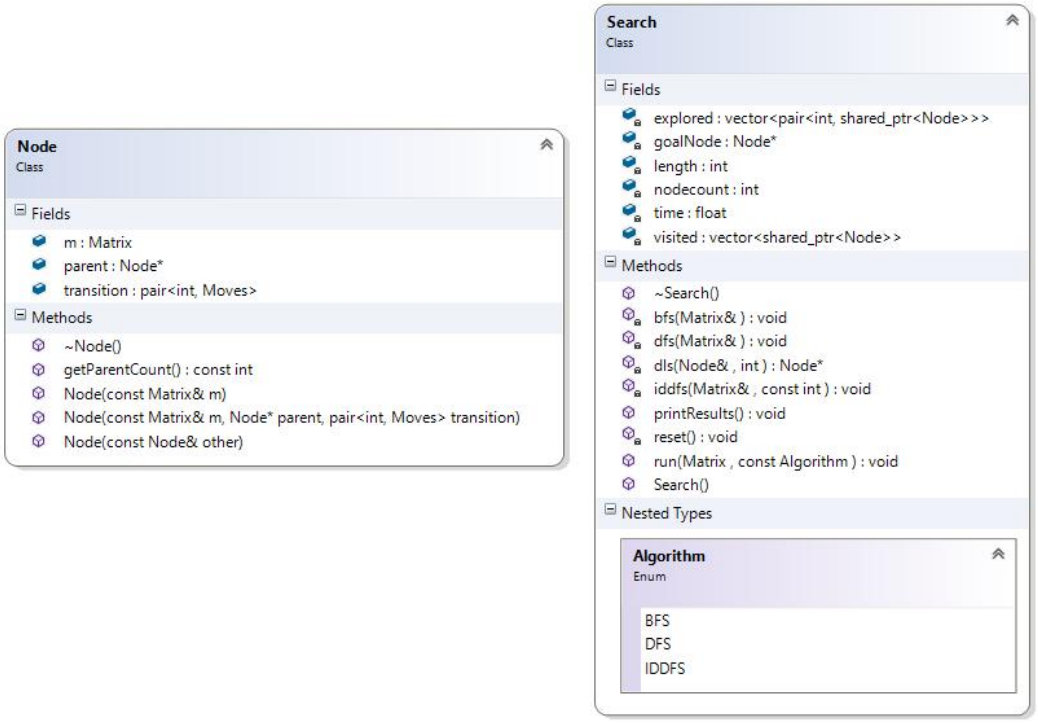

Figure 3: Class diagrams for the Node and Search classes. The algorithm type (BFS, DFS, IDDFS) is an enum that is nested in the Search class.

The class "Node" represents a node inside a search graph. It encapsulates a matrix, a pointer to its parent node and the transition that had to be executed to get from the parent node to this one. Additionally, the class provides copy constructors and a function to count the number of parents all the way back to the root node which does not have a parent node. Figure 4 visualizes the classes member variables.

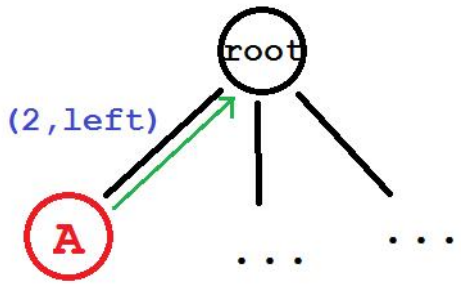

Figure 4: Visualization of the Node classes member variables. The current game state (red), the transition to get from the root node to node A (blue) and the pointer to the parent node (green)

The class "Search" encapsulated all three search algorithms as well as a "printResults" function, to output the steps to get from the root node to the goal node and meta data. This is possible due to every nodes pointer to its parent. When the goal is reached, the path to the root node can be traced back by following the first

parent pointer, then the next one …etc. The class measures the runtime and counts the number of explored nodes in the "run" function.

With this approach, the programs functionality can simply be extended to support additional search algorithms without having to change much of the underlying concept.

1.2.1 Breadth First search

The Breadth First Search algorithm is based on the FIFO (First In First Out) concept where a newly explored node is immediately added to a queue if it was not already visited. The pseudo code is shown in Codesnippet 2.

```
Function-bfs(Matrix)
    create queue q               // FIFO
    create list v                // visited
    create Node root from Matrix
    add root to q and v

    while q is not empty
       dequeue Node n from q
       if n is goal
          nodecount = size of v // Search class
          goalNode = n          // member variables
          return
       for every child of n
          if child is not in v
             add child to q and v
```

Codesnippet 2: Pseudo code for the BFS algorithm based on the C++ source code. The colored letters emphasize the most important elements.

1.2.2 Depth First search

The Depth First Search algorithm is based on the FILO (First In Last Out) concept, often referred to as a stack. A newly discovered node is immediately pushed to the stack if it was not already visited before. The pseudo code is shown in Codesnippet 3.

```
Function-dfs(Matrix)
    create stack s              // FILO
    create list v               // visited
    create Node root from Matrix
    add root to s and v

    while s is not empty
       pop Node n from s
       if n is goal
          nodecount = size of v // Search class
          goalNode = n          // member variables
          return
       if n is not in v
```

```
14            add n to v
15            for every child of n
16               push child to s
```

Codesnippet 3: Pseudo code for the DFS algorithm based on the C++ source code. The colored letters emphasize the most important elements.

1.2.3 Iterative Deepening Depth First Search

The Iterative Deepening Depth First Search algorithm is based on recursion and an increasing depth limit for the search tree. A list of already visited nodes is accumulated, until the search tree for the current depth limit is completed. When not solution is found, the limit is increased, the visited list is cleared and the search starts again from the root node. The pseudo code is shown in Codesnippet 4 and Codesnippet 5. The function "dls" recursively calls itself for all the children of a node if a child was not already explored and is not further away from the root node (deeper in the search tree) than the current one.

```
1  Function-iddfs(Matrix, limit)   // limit = depth limit
2      create list v               // visited (global variable)
3      create Node root from Matrix
4
5      for d = 0; d < limit; d++   // d = depth
6         add root to v
7         Node n = dls(root, d)
8         if n is not null
9            return n
10        clear v
```

Codesnippet 4: Pseudo code for the IDDFS algorithm based on the C++ source code. The colored letters emphasize the most important elements

```
1  Function dls(Node n, depth)
2      if depth = 0 and n  is goal
3         return n
4      if depth > 0
5         for every child of n
6            if n is already in v and its depth > depth
7               break;
8            else
9               nodecount++         // global variable
10              add n to v
11              return dls(child, depth - 1)
12     return null
```

Codesnippet 5: Pseudo code for the IDDFS algorithm based on the C++ source code. The colored letters emphasize the most important elements