1 Implementation

This assignment was built on top of the first assignment. The following chapters detail the additional implementations for the second assignment.

1.1 Additions

To support the informed A* search algorithm a new class "CostNode" was added as well as different heuristic functions and the A* algorithm itself.

The class CostNode is derived from the already existing class Node. It contains a member variable "cost" which represents the total cost of the node. To be able to prioritize some nodes over others according to their cost, a struct "LessThanByTotalCost" is added. The struct holds only one function that compares the cost of two nodes. This function will be used to keep a list of nodes sorted (priority queue) for the A* implementation described later.

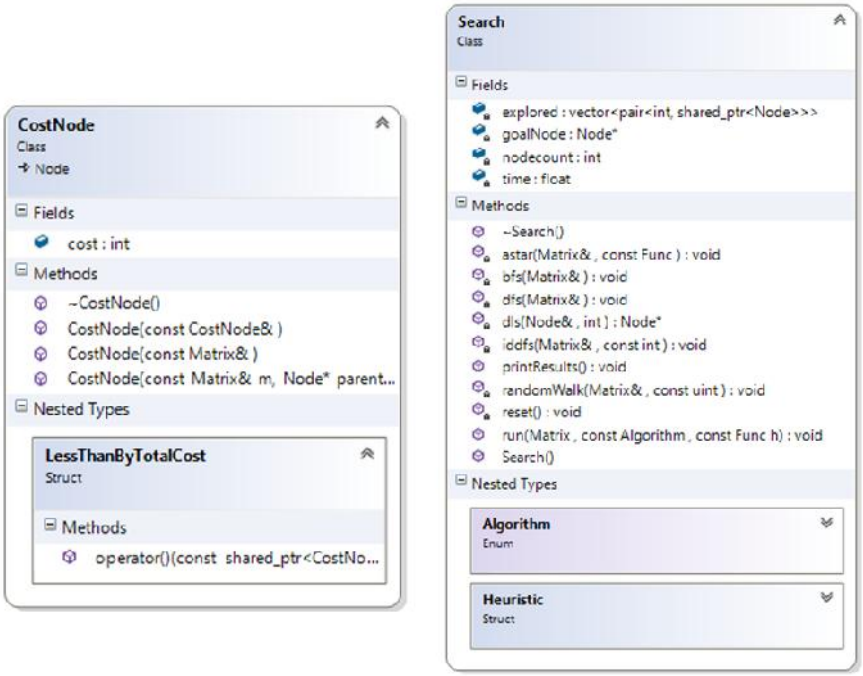

Figure 1: Class diagrams for the classes CostNode and Search

The Search class was extended to support the new A* search algorithm as well as two heuristic functions. The implementation for the algorithm is added to a new method. The heuristic functions are added to a struct nested in the Search class. This way, they can be selected from outside and passed to the A* method. The heuristic functions used here are described in Chapter 1.3.

1.2 A*

The A* search algorithm was implemented in such a way to speed up the tree search (as supposed to uninformed search algorithms e.g. breadth first) while maintaining an optimal solution. Its pseudo code is similar to the one of the breadth first search (see Codesnippet 1). The main difference is the usage of a priority queue instead of a strict FIFO queue.

```
Function-astar(Matrix)
    create priority queue q      // FIFO sorted by priority
    create list v                // visited
    create Node root from Matrix
    root.cost = heuristic        // f = g + h (g = 0 for root!)
    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
             child.cost = g + heuristic   // f = g + h
             add child to q and v
```

Codesnippet 1: Pseudo code for the A algorithm based on the C++ source code. The colored letters emphasize the most important elements. The differences from the breadth first pseudo code are highlighted in purple*

The priority queue is a sorted list, that sorts its elements by a custom criterion. Here, the criterion is the cost of a node which is the sum of the step cost and a heuristic function ($f = g + h$). When dequeuing an element from the priority queue, it is going to be the element with the lowest cost. This element is estimated to be closest to the goal and should be expanded further.

The performance of the A* algorithm largely depends on the heuristic function. It should be admissible (i.e. never overestimate the cost of a node) to push the search in the direction of the goal. Furthermore, every implementation should calculate the heuristic for any given node only once. There were two heuristic functions implemented for this assignment. Both are detailed in the following chapter.

1.3 Heuristic functions

1.3.1 Manhatten distance

The Manhatten distance describes the minimum number of steps necessary to reach a point in a grid layout assuming no diagonal movement (see Figure 2).

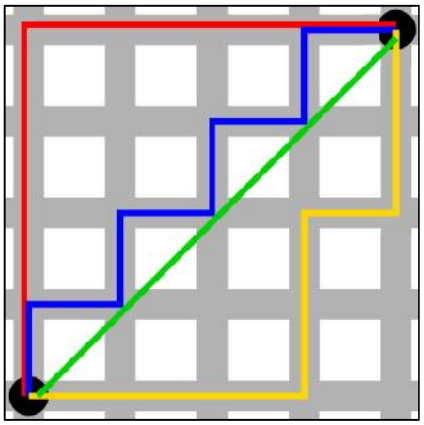

Figure 2: Visualisation of the Manhatten distance between two points. The Euclidean length is green. Assuming no diagonal movement, the step cost for the red, blue and yellow path is the same. Figure taken and modified from [1]

Here, the Manhatten distance is used to describes the distance between the master brick and the goal. This heuristic is admissible, because the closer the master brick is moved towards the goal, the smaller the Manhatten distance becomes. It never becomes smaller when moving the master brick further away from the goal.

To calculate the Manhatten distance for different master brick and goal sizes correctly, the center is taken as the reference. Consider the three configurations in Figure 3.

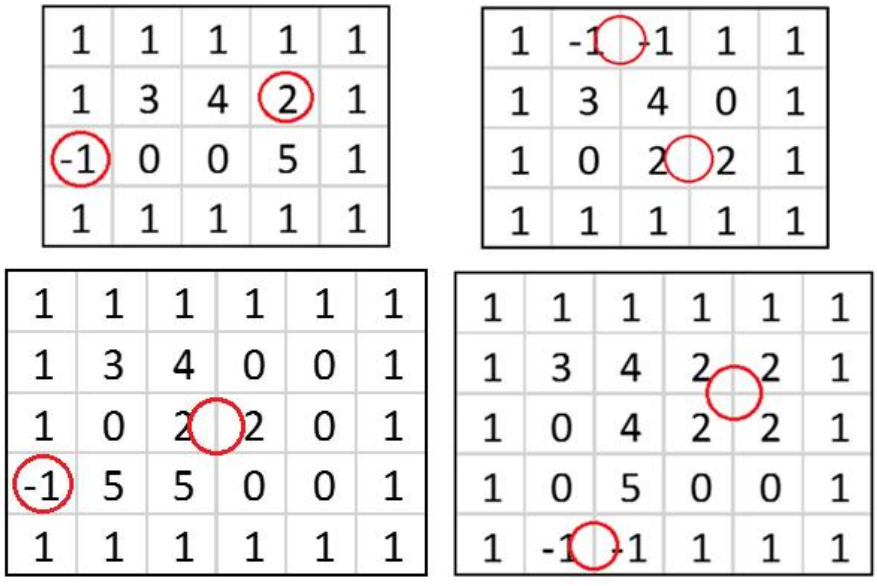

Figure 3: Three configurations to illustrate the master bricks and goals center. Manhatten distances from top left to bottom right: 4, 3, 3, 4

For a master brick or goal size of 1x1, the center is the absolute position of the cell in the grid (top left). For the sizes 2x1, 1x2 and 2x2 the center is taken (top right,

bottom left, bottom right). Table 1 details the configurations from Figure 3 with the master and goal positions. The Manhatten distance is calculated by the following formula:

$$distance = abs(masterX - goalX) + abs(masterY - goalY)$$

In some cases, a decimal number results from this calculation. Since there are no puzzles with a master brick size bigger than 2x2 here, the decimal place can simply be cut off.

Configuration Figure 3	**Master**	**Goal**	**abs(Master – Goal)**	**distance**	**Manhatten distance**
Top left	3|1	0|2	3|1	4	4
Top right	2.5|2	1.5|0	1|2	3	3
Bottom left	2.5|2	0|3	2.5|1	3.5	3
Bottom right	3.5|1.5	1.5|4	2|2.5	4.5	4

Table 1: Metadata from the configurations in Figure 3. The calculated Manhatten distance is in the most right column

***NOTE**: The approach described above for calculating the Manhatten distance by taking the center works correctly for the sizes 1x1, 1x2, 2x1 and 2x2.*

1.3.2 Blocking heuristic

The blocking heuristic was designed to, in contrary to the Manhatten distance, to also evaluate the positions of other bricks on the board. The problem with the Manhatten distance is, that it completely disregards the board position and outputs a different value only when the master brick is moved. However, in many board configurations multiple pieces other than the master brick must be moved before the master brick can be moved closer to the goal.

Therefore, the blocking heuristic counts the number of "blocking cells" between the master brick and the goal. A cell is considered blocking, if its value is >2 and it is positioned between the master brick and the goal. Regardless of the number of occurrences of the same value on the board it is always counted as a standalone value. Because the board layout is a grid and the master brick could reach the goal on different paths (see Figure 2), every cell in this area must be evaluated. Figure 4 shows the configurations from Figure 3 where the potentially blocking cells are highlighted in grey.

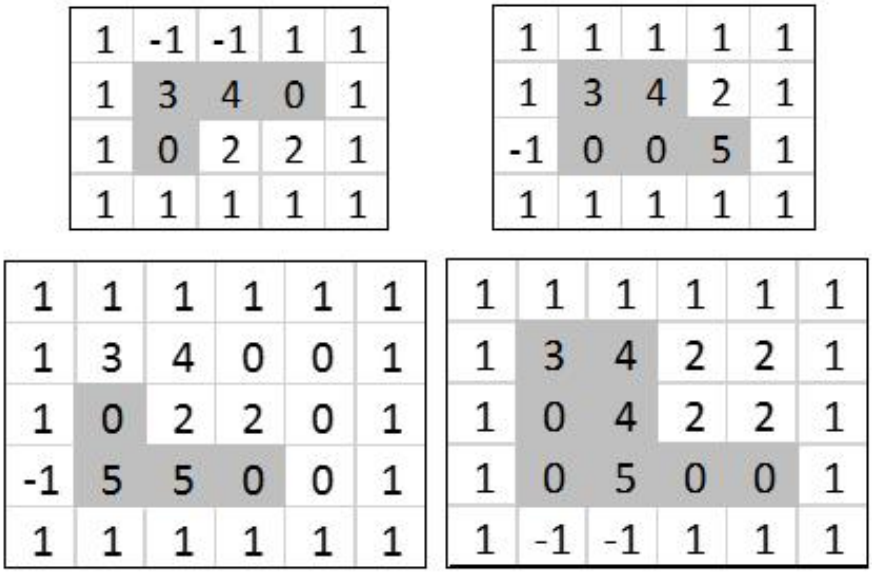

Figure 4: Board configurations from Figure 3. Potentially blocking cells are highlighted in grey. These cells must be evaluated. Actual blocking cells from top left to bottom right: 2, 3, 2, 4

Blocking cells are cells highlighted in grey that are not 0. While in three out of the four configurations in Figure 4 the whole board must be evaluated, the area becomes smaller when the master brick moves closer to the goal. For the bottom left configuration there are only four cells to be evaluated. The number of blocking cells is two.

The C++ implementation defines the area to be evaluated by the position and dimension of the master brick and the goal. Then the number of cells that contain a value >2 is counted.

Proof of admissibility

This heuristic is guaranteed to be admissible, because moving the master brick further away from the goal does not push the search closer to the goal (i.e. never overestimates the cost of a node). This is because moving the master brick further away from the goal will, in the worst case, result in a number of blocking cells that is equal to the previous number of blocking cells. This satisfies the condition:

$$h(n) \leq h^*(n) \quad \forall\, n$$

n is a node

h is the heuristic

h(n) is the cost to reach the goal from n

h*(n) is the actual cost to reach a goal from n

As an example, consider the bottom left configuration from Figure 4. The current number of blocking cells is two. Assume one would move the master brick to the right. This only introduces zeros to the area that must be evaluated. Therefore, the number of blocking blocks would still be two. If a move to the right would introduce not only zeros but for example one value >2, the resulting number of blocking cells would be three and thus, not be prioritized over the lower value of two when not moving to the right. The heuristic would be considered not admissible, if for the bottom left configuration, moving to the right would result in a lower value than two. This is never the case and therefore the heuristic is admissible.

2 References

[1] Wikipedia Taxicab geometry [Online]. Available: https://en.wikipedia.org/wiki/Taxicab_geometry. [accessed 05/18/17]

[2] Wikipedia Admissible heuristic [Online]. Available: https://en.wikipedia.org/wiki/Admissible_heuristic. [accessed 05/19/17]