Softwarepraktikum SS 2016

Assignment 1

Group - 3

Eric Wagner	344137	eric.michel.wagner@rwth-aachen.de
Stefan Rehbold	344768	Stefan.Rehbold@rwth-aachen.de
Sonja Skiba	331588	sonja.skiba@rwth-aachen.de

Task 1

The three maps were added under \assignment_final\maps in the first assignment report directory.

Task 2

```
1  typedef struct Cell
2  {
3    uint8_t direction[MAX_PLAYER_COUNT];
4    uint16_t neighbor[MAX_PLAYER_COUNT];
5  } Cell;
6
7
8  class Map
9  {
10   private:
11     uint16_t overrideStones[MAX_PLAYER_COUNT];
12     uint16_t numberOfBombes[MAX_PLAYER_COUNT];
13     uint8_t playerMap[MAX_PLAYER_COUNT + 1];
14       // 0xFF means the player is not playing. playerMap[0] is also a
             player not playing.
15
16     void addNeighbour(uint32_t x1, uint32_t y1, uint8_t dir1,
         uint32_t x2, uint32_t y2, uint8_t dir2);
17
18   public:
19     char* board;
20
21     Map(std::string path);
22     Map(Map& toCopyMap);
23     virtual ~Map();
24     Cell getCell(uint8_t x, uint8_t y);
25     void draw();
26     bool isMoveValid(uint8_t x, uint8_t y, uint8_t player, uint8_t
         information);
27     int32_t evaluateForPlayingPhase(uint8_t player);
28     int32_t evaluateForBombingPhase(uint8_t player);
29
30 };
```

In the first stage of the creation of the data structure we parse the map to an two dimensional array and count the number of occupied cells. We need the number of cells to create array of the according size and because we are already reading the map in, we save it in a more handy format.
In the next stages of loading the map we split the information in a static part, which will remain unchanged during a game and a dynamic part. To anticipate a lot of map copying of the map to evaluate different possible moves, we did try to reduce the size of the dynamic part as much as possible.

The data structure we came up with consists of 3 parts, which are best explained with the help of an example. We latter we take the following four by four board:

-	-	-	-
-	1	2	0
-	b	-	0
0	0	0	c

We begin by copying the states off every cell in an one dimensional **board** array. This array is saves all the information that can change during a game. It has the size of number of cells from the board that not holes, that way copying of a lot of boards to execute different moves will be quick.

1	2	0	b	0	0	0	0	c

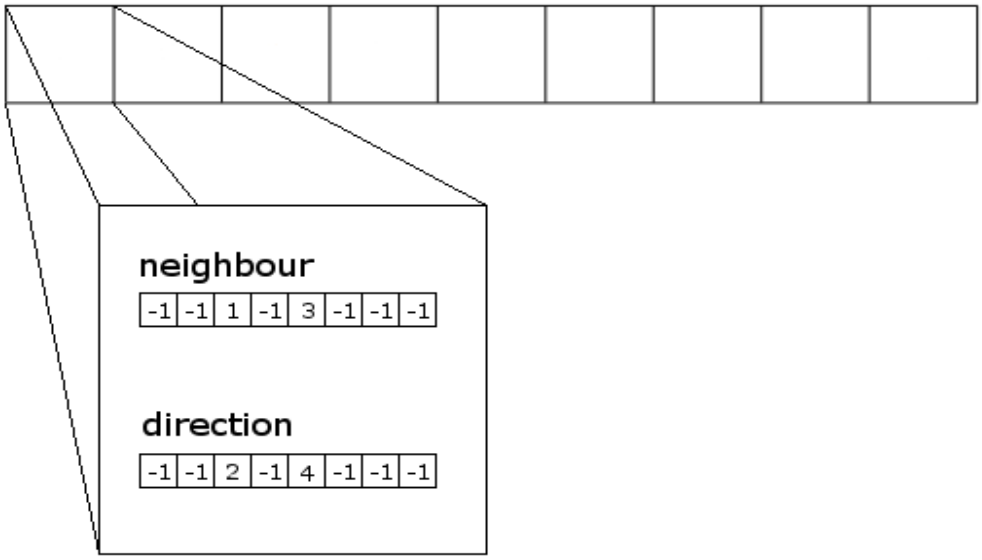
The second thing we do is to create our **offsetMap**. That is again two by two array off the same size than the initial board. In that array we save for every cell, how many cells come before it, while ignoring the walls. On the positions of walls, we simply write a -1.

-1	-1	-1	-1
-1	0	1	2
-1	3	-1	4
5	6	7	8

Finally, we need to save how all the cells are linked together. To archive this we created a struct **Cell**, which consists of two arrays, **neighbour** and **direction**, each with an entry for every possible direction.

neighbour[direction] saves the index of the neighbour in that that direction. Here again, we write an -1 if in a specific direction the cell borders no other cell.

As we can change the direction of a line throw artificially added transitions, **direction**[direction] saves the direction in which the map traversing algorithm has to continue looking.



Task 3+5

The algorithm we decided on works the following way:

- Check if the cell that should get placed on exists.
  - Case No: Return that the move is not valid.
- Check if the player has a override stone if the move needs it.
  - Case No: Return that the move is not valid.
- for each(direction of the cell)
  - while(next cell not an wall, the starting, an empty and a player cell)
    - Get the new cell.
    - Get the new direction.
  - if(next cell is from the player)
    - Reverse the direction.
    - while(next cell not the starting cell)
      - Mark that the cell needs to get recoloured.
      - Get the new cell.
      - Get the new direction.
- if(at least one cell was marked)
  - Recolour all marked cells.
  - switch(starting cell state)
    - case player or expansion: Take away one override stone of the player.
    - case inversion: Do the inversion.
    - case choice: Switch the stones of the player with stones of the chosen other player.
    - case bonus: Give the player a bomb or an override stone, depending on his choice.
    - case empty: Nothing special to do here.
  - Recolour starting cell.
  - Draw map.

  - Return that the move was valid.

- if(no cell was marked and starting stone is a expansion stone)
  - Recolour starting cell.
  - Take away one override stone of the player.
  - Draw map.
  - Return that the move was valid.

- Return that the move is invalid.

Task 4

The following aspects should be considered when rating the result of any given move.
build phase rating function:

- number of own tiles, weighted by stability *(sorted from best to worst)*:
  - stable tiles: tiles that can not be captured by the opponent at all
  - semi-stable tiles: tiles that can be captured by the opponent, but not immediately in the next move/hard to capture in general)
  - unstable tiles: tiles that can be captured by the opponent in the next move
  - high risk tiles: tiles that can be captured by the opponent in the next move and give them stable tiles
- number of bombs captured in that move * effectiveness of bombs *(=number of tiles a bomb hits)*
- number of own possible moves (maximize as far as possible)
- number of each opponents' possible moves (minimize as far as possible)
- minimized tile loss of player=((own playerindex) - (overall capturable inversion stones))
- number of available override stones (more valuable the closer it is to the end of the game/the less the opponents have left)

bomb phase rating function:

- number of own tiles
- difference to the other players' number of tiles