COMP 3981 – Project Abalone

Written Report – Part 1

Rongyi Chen
Jason Jia
David Lu
Sam Tadey
Jeffery Wasty

Table of Contents

Abstract

Project Abalone is an attempt to create a game-playing agent for the board game Abalone. This document details the accomplishments of the first checkpoint, which include game board and move representation, problem formulation, and team member contributions.

1 Game board representation

The board is made of 61 squares and looks like as below. The squares must be labeled from columns 1 – 9 (which stretch from the top left to the bottom right) and rows A through I. How the board is represented in the code, however, is up to each team.

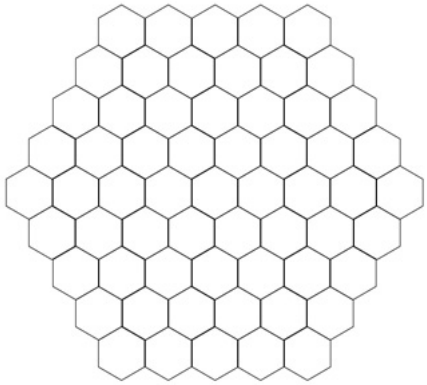

Figure 1 – The Abalone game board

1.1 Code representation of game board

For our program the game board is represented as an array, as seen below.

```
static char[] STANDARD_INITIAL_STATE = new char[]{
        '0', '0', '0', '0', '0',
        '0', '0', '0', '0', '0', '0',
        '+', '+', '0', '0', '0', '+', '+',
        '+', '+', '+', '+', '+', '+', '+', '+',
        '+', '+', '+', '+', '+', '+', '+', '+', '+',
        '+', '+', '+', '+', '+', '+', '+', '+',
        '+', '+', '@', '@', '@', '+', '+',
        '@', '@', '@', '@', '@', '@',
        '@', '@', '@', '@', '@',
};
```

Figure 2 - Code representation of the game board.

2 Game move representation

Game moves will be discussed in a further section on actions, but to summarize our game moves can be represented as value lookup, which takes in two inputs, the current square and the direction of movement. The value returned on lookup is the new square.

This is for a single piece. For a game move of multiple pieces, it is divided into several of these smaller actions.

3 Problem formulation

3.1 State representation

At any time, the Abalone game exists in a state with contains the board, its squares, the pieces (both those remaining, and those knocked out), the timer, and the move counter.

3.1.1 Board and squares

The Abalone board, as discussed above, is a board of 61 squares shaped as a pentagon, with each side being of length 5. The board is labeled as seen below.

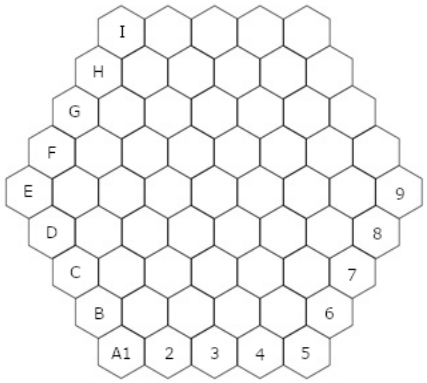

Figure 3 - Abalone board with labeled rows and diagonals

The board and its squares could thus be visualized as a 2-dimensional matrix, as seen below.

```
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * *
* * * * * * * *
* * * * * * *
* * * * * *
```

Figure 4 - Visualization of the Abalone board

This will be our board representation as we talk about the rest of the problem formulation. Note that the actual representation is the array seen in Figure 2.

3.1.2 Pieces

Each player controls 14 pieces, each a different color (traditionally black and white). We can include the pieces in our representation as below.

Figure 5 - Visualization of the Abalone board with its pieces

In this visualization, $ corresponds to the 1st player, and ! to the 2nd player.

In the code, the pieces are also indicated by different symbols.

3.1.3 Knocked out pieces

Since the goal of Abalone is to knock your opponent's pieces off the board. We will need to represent the knocked-out pieces in a way. This will take the form of a number in the bottom right, and in the top left, to indicate the number of knocked out pieces for player one and player two respectively.

3.1.4 Time and moves

The state also includes the current move the player is on (if there is a move limit) and the time left over from the last round (if there is a time limit). These will be present in the top right.

3.2 Initial state

Abalone allows for several initial setups. The three we will consider are the standard layout, the Belgian daisy, and the German daisy.

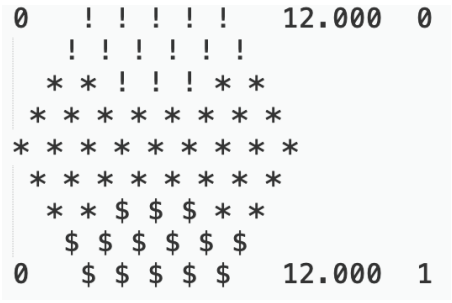

Figure 6 - The standard setup

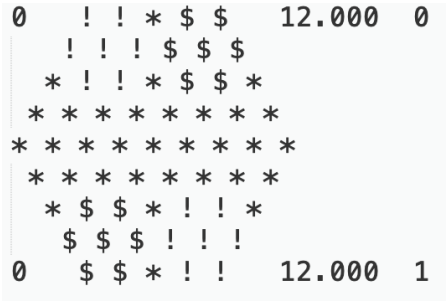

Figure 7 - The Belgian daisy setup

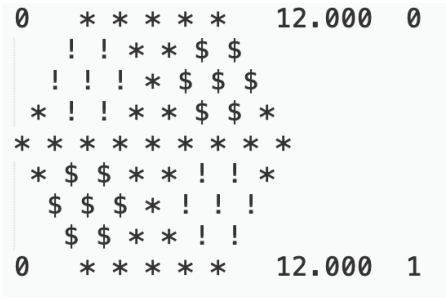

Figure 8 - The German daisy setup

3.3 Actions

3.3.1 Action definition and legal actions

In Abalone piece movement can be divided into three parts. First the piece must move to a square that exists, second the piece must move to a square that is legal, and third, the collection of pieces moving must be allowed, and each must follow the first two rules.

To check for existence, we can define a set of directions on the game board as seen below.

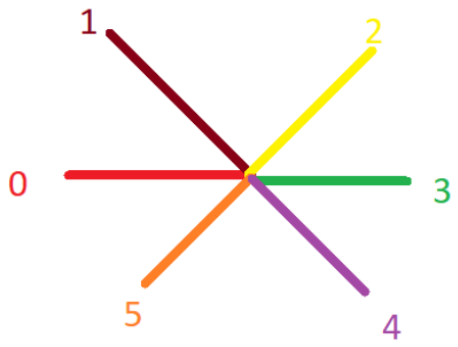

Figure 9 - The six directions of movement.

And we can code a value for each square, in each direction, in a look-up table. This will allow one to easily find if there is a square to move to, given a starting square and direction, and what the new square will be. Consider the example below:

```
mapTable[0] = new byte[] { -1, -1, -1, 1, 6, 5 };
```

This element of the table is for the 0^{th} square, and says that for directions of movement 0, 1, 2, the piece goes out of bounds, for direction 3, the new square is 1, for direction 4 the new square is 6, and for direction 5 the new square is 5.

To check for legality, we'll need to keep track of pieces moving, and pushed pieces, and make sure that the number moving outnumbers those that are

moved. This assumes there is a piece on the square we're moving to, if its empty the move is legal.

Finally, we look at the combinations of pieces that can move. Pieces can move together if they can be connected in a straight line, up to three pieces. This set can move in any direction.

3.3.2 Code representation of actions

The action is composed of 3 parameters,

1. **Movement Type**, an integer
2. **Movement Direction**, an integer
3. **The position of the marble that will be moved**, an integer

The **Movement Type** defines whether the action is an in-line move or is a board-side move.

a) If the value equals to **1**, it is an **in-line move**
b) If the value equals to **2** or **3**, it is a **board-side move** and the value represents the number of allied marbles to move together

The **Movement Direction** is an integer value range from 0 to 5 that represents one of the 6 directions. It defines the direction the marble(s) will move to

The **position of the marble** will be used along with **Movement Type** and **Movement Direction** to compute the indices of all the marbles that will be moved in this action.

Since only one position of the marble(s) is needed in the action, and the rest of the marbles' position is computed at runtime. The number of action combinations is fairly small.
It can be calculated by multiplying the range of each parameter.

```
Number of combinations = 3 * 6 * 61 = 1098
```

3.4 Transition model

The transition model is the actions and the resulting state. At first glance, there are many possible states, and so it would seem the transition model is very large. This is not the case, as it can be abstracted down to a few choice examples.

The actions are:

- Movement of a piece(s) with no knockoff of enemy pieces.
- Movement of a piece(s) with knockoff of enemy pieces.
- Movement of a piece, but it is the last turn.

Therefore, the transition model would look as follows:

Prior state	Action	Resulting State
	Movement of a piece(s) with no knockoff of enemy pieces.	
	Movement of a piece(s) with knockoff of enemy pieces.	
	Movement of a piece, but it is the last turn.	

Table 1 – The transition model for our project

4 Goal test

The game ends if one of the following conditions are met.

1. A player has had 6 of their pieces knocked off the board.
2. A player has exceeded their allotted time for a turn.
3. A stalemate condition is reached. This can be the maximum number of turns being reached, or maybe a certain number of moves happening without forward progress (defined beforehand).

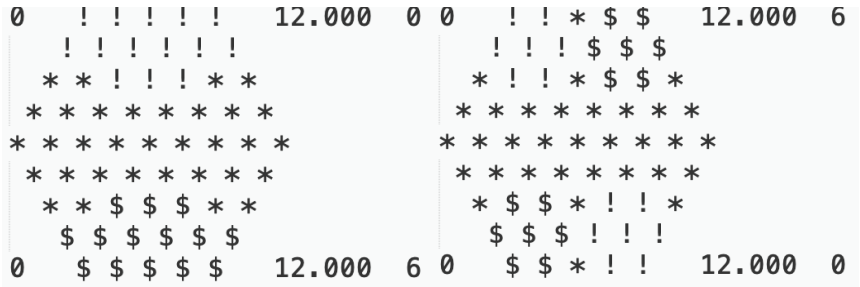

Figure 10 – The game ends when one of the following two states is reached. Note: the board can be of any configuration.

5 Team member contribution

Each team member contributed both to this document as well as the components handed in along with this document (ground rules & GUI). More specifically the contributions are as follows:

- Ground rules contract
    - All (Contract was borrowed from another course and adapted to our use case)
- Deciding on game board and game move representation
    - All (Talked about during first two team meetings)
- Problem formulation

    - All (Worked on this independently, then all came together for a meeting to discuss our ideas)
- GUI Input
    - Sam, Jeff – Timing and buttons found along top.
    - David, Jason, Rongyi – Working with game board to get moving pieces.
- GUI output
    - Sam, Jeff – Time history
    - David, Rongyi, Jason – Update board
    - Rongyi – Game score, move history, next move.
- Configuration
    - Sam, Rongyi, David – Initial layout, color selection, human vs computer.
    - Sam, Jeff – turn limit, move limit
- Documentation
    - Jeff, Rongyi