Project Abalone

Rongyi Chen
Jason Jia
David Lu
Sam Tadey
Jeffery Wasty

Table of Contents

Table of Figures

Table of Tables

Abstract

Project Abalone is an attempt to create a game-playing agent for the board game Abalone. This document is the final version of our report encompassing parts 1, 2, and 3.

1 Problem Formulation

Much of this section will be familiar to readers of parts 1 and 2. The key difference in the agent came in the adoption of the decision-making heuristic which is discussed in a latter section.

1.1 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.

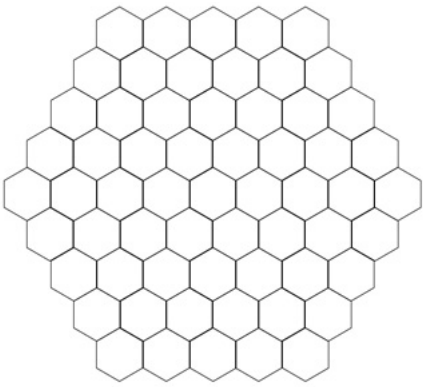

Figure 1 – The Abalone game board

1.1.1 Code representation of game board

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

Figure 2 - Code representation of the game board.

1.2 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.

1.3 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.

1.3.1 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.

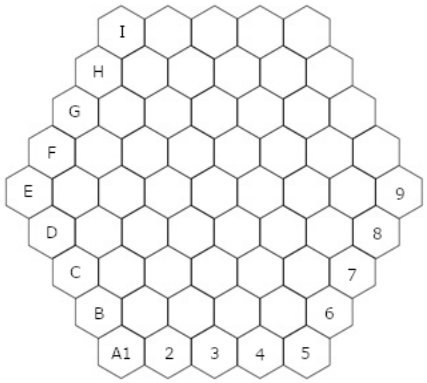

Figure 3 - Abalone board with labeled rows and diagonals

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.

1.3.2 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.

1.3.3 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.

1.3.4 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.

1.4 Initial state

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

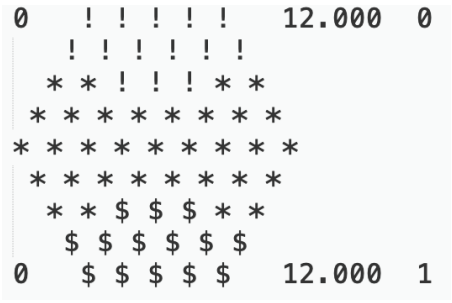

Figure 6 - The standard setup

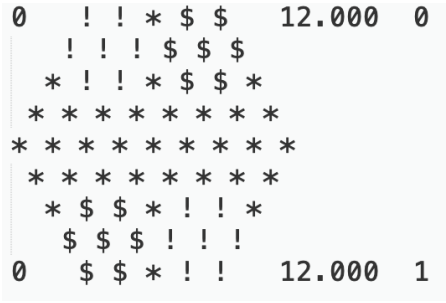

Figure 7 - The Belgian daisy setup

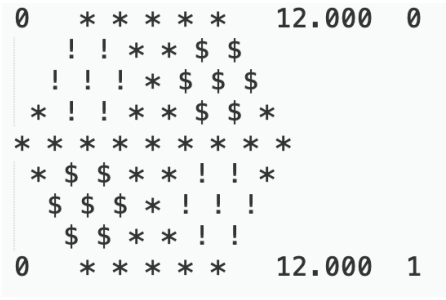

Figure 8 - The German daisy setup

1.5 Actions

1.5.1 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.

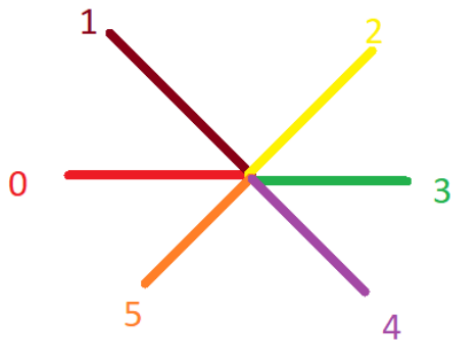

Figure 9 - The six directions of movement.

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 };
```

Figure 10 - An example row from the lookup table

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.

1.5.2 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
```

1.6 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

1.7 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).

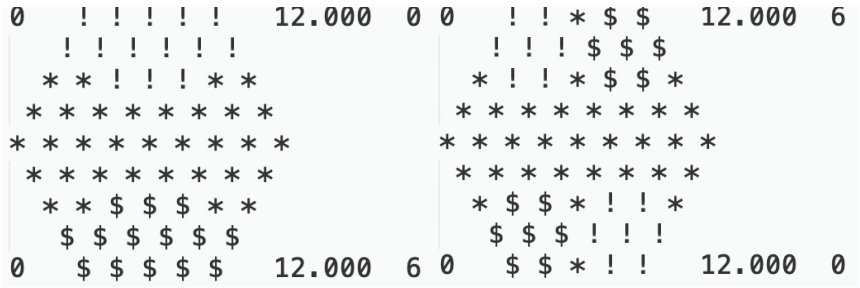

Figure 10 – The game ends when one of the following two states is reached. Note: the board can be of any configuration.

2 State space generation

By defining our actions, we know the legal game moves, and thus we know the resulting game states. The actions are not all intuitive, but are small in number, which comes in handy when checking validity of game moves. Game move representation on the other hand, while intuitive and suitable for our designed transition model, have many combinations, and so validation can be tough.

When states are generated, the state space is copied and all valid actions for that state are found by checking all possible actions. When all valid actions are found they are translated into game move representations, which are used to move from one state to the next. This flow can be represented as below:

All actions → *All valid actions* → *All valid moves* → *All valid next states*

We define two different rules for validating inline moves and board-side moves respectively.

2.1 Validation of inline moves

For validating inline move action, we define a set of valid inline moves. For this set, consider `O` to be one kind of piece, `@` to be another, `+` to be a valid move, and `!` to be a push off the board.

	For piece `O`	For piece `@`
One-piece move	`O+`	`@+`
Two-piece move	`OO+`, `OO@+`, `OO@!`	`@@+`, `@@O+`, `@@O!`
Three-piece move	`OOO+`, `OOO@+`, `OOO@!`, `OOO@@+`, `OOO@@!`	`@@@+`, `@@@O+`, `@@@O!`, `@@@OO+`, `@@@OO!`

Table 2 - The set of possible inline moves

Initially, we create a string called the "pattern". We then start from the marble closest to the trailing end and keep adding the next marble, in the defined direction of movement, to "pattern" until we either (a) hit an empty cell, (b) reach the end of the board, or (c) fail a check. The checks are done after each marble is added to see if "pattern" and checks to see if the "pattern" this far exists in the set of valid moves. If it does, it means the action is valid and we can continue, if not, then there is no need to continue with an invalid move, and the move fails.

2.2 Validation of broadside moves

For validation of broadside moves, we check, for each marble moving, if the cell it is moving to is empty. If all cells are empty, the action is **valid**. If one of the cells is either occupied by another marble or is out of board, the action is **invalid**.

For determination of the 'best' move, we rank each possible move through a heuristic. Each state resulting from a move can be given a score (based off the move), and we can begin to iterate through all the possible states, from any given states. The program keeps track of the best state and will return this state when either it has exhausted its search of possible states, or the time limit is reached. The move to reach that state is the move the program decides upon.

2.3 Game move representation

Game move representations are generated by checking each possible action of the game. If the action is deemed valid, a game move representation object is generated and is added to the list of available moves for the AI to take.

An example of a board-side move representation is:

```
[48=0, 41=1, 55=0, 49=1]
```

generated by the action {n=2, direction=2, index=48}

Each element in the array represent the next state of a cell on the game board. The left-hand side value is a number range from 0 to 60, which represents the index of the cell. The right-hand value is the state of the cell where 0 represents empty, 1 represents a white marble, and 2 represents a black marble.

Therefore, the example representation above translates to two white marbles will move from cells 48 and 55 to cells 41 and 49 respectively, using a board-side movement.

This same representation can be applied to inline moves. A representation of:

```
[1=0, 7=1, 14=1]
```

generated by the action {n=1, direction=4, index=1}

means that, two white marbles will move from cells 1 and 7 to cells 7 and 14.

3 Search strategy and performance enhancement

3.1 The winning heuristic function

The winning design as selected by section 6, is the heuristic by **Jason Jia**.

3.1.1 Heuristic Function Design

Two popular strategies that are used in Abalone game heuristic function are

- Evaluate marbles' closeness to the board center
- Evaluate marbles' adjacency

Using the above two strategies as a foundation, I have adopted below, strategies to maximize the AI performance while minimizing the side effect of AI searching power (maintain the level of depth of search and increase winning probabilities).

1. No trade off
    a. AI marbles are worth more points than the opponent's marble
2. Aggressive attitude
    a. Pushing opponent's marble away from center earns more points than AI marbles move to the center.
3. Defensive with a shape of six-star structure.
    a. Six-star structure means a marble are surrounded by six ally marbles.
    b. According my experience, this structure is really stable and hard for opponent to push, which always result in dividing opponent's army into two, making the army more vulnerable to attack.
4. Send a 'spy' to opponent army

a. A ‘spy’ means a marble that is surrounded by six enemy marbles

b. This will lower the adjacency for opponents’ marbles and the ‘spy’ will also be in a safe position.

3.1.2 Heuristic Function Constants/weights

POSITION_WEIGHT_MAP = {
1, 1, 1, 1, 1,
1, 2, 2, 2, 2, 1,
1, 2, 3, 3, 3, 2, 1,
1, 2, 3, 4, 4, 3, 2, 1,
1, 2, 3, 4, 5, 4, 3, 2, 1,
1, 2, 3, 4, 4, 3, 2, 1,
1, 2, 3, 3, 3, 2, 1,
1, 2, 2, 2, 2, 1,
1, 1, 1, 1, 1,
};

Figure 11 - Jason's position weight map

Constants:

- ai_marble_value = 110
- opponent_marble_value = 100
- aggressive_factor = 1.2
- six_star_structure_bonus = 5
- spy_structure_bonus = 5
- Neighbour_bonus = 1

Following the strategies listed in 4.1, below are the desired actions ordered by priority (with the weight assigned to the heuristic function):

1. Prevent AI marbles are pushed off the board

   ai_marble_value

2. Push Opponent marble off the board

opponent_marble_value

3. Push Opponent marble away from the center of board

   aggressive_factor * difference in Position_Weight_Map per marble

4. AI marbles try to be as close as possible to the center of board

   difference in Position_Weight_Map per marble

5. Marble forms six-star structure

   six_star_structure_bonus per marble

6. Spy in enemy

   spy_structure_bonus per marble

7. Marble has one ally neighbour

   neighbour_bonus per direction per marble

3.1.3 Heuristic Function Evaluation and Weight Selection

The heuristic function is evaluated using our frontend AI vs AI mode. Having two AI with different heuristic function or different weight against each other and selected the one with best performance. The results of this evaluation can be seen in section 6.

3.1.4 Results

Game	Black-player (heuristic owner)	White-player (heuristic owner)	Score	Winner
1	Jason	Jeff	6/5	Jason
2	Jason	David	6/4	Jason
3	Jason	Sam	6/5	Jason
4	Rongyi	Jason	3/3	Tie

Table 3 - Individual game results: Jason's excerpts

3.2 Performance enhancements

We attempted to increase the performance of the AI program by using classic techniques, node ordering, transposition table, quiescence searching, and multithreading. Looking at the results, the transposition table and quiescence search did not improve the performance of the AI, but rather slows down the searching. We believe this is caused by logic errors and bugs in code, but due to the limit of time, we decided to drop the two enhancements for now until further investigation is made.

3.2.1 Node ordering

Node ordering is the first enhancement we are using to increase the performance of Alpha-Beta pruning.

The ordering rule is simple: the actions that move more marbles will be searched first. This is because we believe that that actions moves more marbles will produce better result most of the time. More marbles being moved means you are moving groups of three more often than groups of two, and as well, you are disrupting more of the opponent's defensive structure.

In fact, the AI with node ordering did prune more nodes and searched fewer nodes comparing to the AI without node ordering. Since less time is taken for searching each level, the AI can search deeper within a given amount of time.

3.2.2 Multithreading

There is one other enhancement used in the AI program, which is multithreading the program to search Alpha-Beta Search tree in parallel. We use a fixed size thread pool with a size of 4 threads runs in parallel. As a result, we utilize the idle CPU cores to achieve better computational power which means better search result within a given amount of time. Although the 4 branches that are searched in parallel cannot prune each other, the final outcome of utilizing Multithreading is much better than single-thread AI.

3.3 Decision towards the heuristic function

The decision among the group was to award the title of best heuristic, and therefore use in the competition, to that heuristic which could outperform

the others. That is, we decided on a purely empirical test among the heuristics. The reason for this was

a) We felt it was the simplest and most fair, and

b) we each arrived at the same conclusion to use board weights (prioritizing the center), and clustering.

Thus, the differences would really come down to, who was able to find the best weightings between the two (and between knocking out pieces), and how fast did their code run. Several tests were run, by different group members, the results of which can be seen below. **Note that for each test, the standard board layout was used:**

Game	Black player (heuristic owner	White player (heuristic owner)	Score	Winner
1	Jason	Jeff	6/5	Jason
2	Jason	David	6/4	Jason
3	Jason	Sam	6/5	Jason
4	Jeff	Sam	6/4	Jeff
5	Jeff	David	3/1	Jeff
6	Sam	David	2/2	Tie
7	Sam	Rongyi	0/5	Rongyi
8	Rongyi	Jason	3/3	Tie
9	Rongyi	Jeff	4/4	Tie

Table 4 - Individual game results

Player (heuristic owner)	Percent of games won (%)
Rongyi	66.6
Jason	**87.5**
David	17.7
Sam	12.5
Jeff	62.5

Table 5 - Game summary. Percent of games won were 1 point for a win, 0.5 point for a tie, 0 for a loss, summed, and divided by 1 point for each game played.

Based on the results above, **Jason Jia**'s heuristic performed the best, and therefore his heuristic was the one we decided to go with for the remainder of the project.

4 Team member project contribution

- Problem formulation
    - All (group discussion)
- Game playing agent
    - Backend – All
    - Frontend – Jason
- State Space Generation

    - Rongyi and Jason
- Heuristics
    - All (Jason's was picked as best)
- Search strategy and performance enhancements
    - All (mainly Rongyi and Jason)
- Documentation
    - All (mainly Jeff)

5 References

[1] Campos, P., & Langlois, T. (2009). *Abalearn: Efficient Self-Play Learning of thegame Abalone.* Lisbon, Portugal: INESC-ID, Neural Networks and Signal Processing Group.

[2] Lemmens, N. (2005). *Constructing an Abalone Game-Playing Agent.* Maastricht, Netherlands: Maastricht University.

[3] Ozcan, E., & Hulagu, B. (2004). A Simple Intelligent Agent for Playing Abalone Game: ABLA. *Proc. of the 13th Turkish Symposium on Artificial Intelligence and Neural Networks* (pp. 281-290). Istanbul: Yeditepe University.

[4] Papadopoulos, A., Konstantino, T., Chrysopoulos, A. C., & Mitkas, P. A. (2016). Exploring Optimization Strategies in Board Game Abalone for Alpha-Beta Search. *2012 IEEE Conference on Computational Intelligence and Games (CIG)* (pp. 63-70). Granada, Spain: IEEE. Retrieved from https://issel.ee.auth.gr/wp-content/uploads/2016/04/Exploring-Optimization-Strategies-in-Board-Game-Abalone-for-Alpha-Beta-Search.pdf

6 Appendix I: Alternative heuristics

6.1 Rongyi's heuristic

A simple position weighted heuristic. Since the goal of abalone is to push opponent's marbles out of the board, the most efficient way to push opponent's marble and prevent from being pushed is to stay in the middle of the board. Therefore, each location of the board is assigned a weighted value, the closer it is to the middle, the higher the value is.

```
int[] POSITION_WEIGHT_MAP = new int[]{
1, 1, 1, 1, 1,
1, 2, 2, 2, 2, 1,
1, 2, 3, 3, 3, 2, 1,
1, 2, 3, 4, 4, 3, 2, 1,
1, 2, 3, 4, 5, 4, 3, 2, 1,
1, 2, 3, 4, 4, 3, 2, 1,
1, 2, 3, 3, 3, 2, 1,
1, 2, 2, 2, 2, 1,
1, 1, 1, 1, 1,
};
```

Figure 12 - Rongyi's position weight map

Each existing marble will also worth 50 points so that the AI will tend to push opponent's marble out of the board and keep its own marble from being knocked off.

I believe simple is the best. Although this heuristic function is very simple, it does achieve the goal of abalone - get together and push. opponents' marbles out. An advantage of this heuristic function is that it does very little calculation. Computation time matters since the heuristic function is evaluated at every single leaf of the search tree. This heuristic function takes little time to compute at runtime. In conclusion, I think this heuristic function will perform in reality competition.

Game	Black player (heuristic owner)	White player (heuristic owner)	Score	Winner
1	Sam	Rongyi	0/5	Rongyi
2	Rongyi	Jason	3/3	Tie
3	Rongyi	Jeff	4/4	Tie

Table 6 - Individual game results: Rongyi's excerpt

6.2 David's Heuristic

6.2.1 Design

My design took into consideration the piece's placement on the board, clustering, three in a row, and knocking an enemy piece out. All of these factors have weights that are attributed.

The closer a piece is to the middle of the board the higher its heuristic becomes. This is because fighting for control of the center is crucial in the early and mid-game. Having a developed cluster in the middle of the board stops the opponent from advancing. The edge spaces are weighted much less than the rest which encourages the AI to move out of there as soon as possible.

1, 1, 1, 1, 1,
1, 5, 5, 5, 5, 1,
1, 5, 8, 8, 8, 5, 1,
1, 5, 8, 10, 10, 8, 5, 1,
1, 5, 8, 10, 13, 10, 8, 5, 1,
1, 5, 8, 10, 10, 8, 5, 1,
1, 5, 8, 8, 8, 5, 1,
1, 5, 5, 5, 5, 1,
1, 1, 1, 1, 1

Figure 13 - David's position weight map

Clustering is defined to have six adjacent pieces next to one particular piece. It is important to have pieces next to you, so it is easier to build three in a row pieces for push or defend. Having a piece in all direction awards three points to the heuristic.

Three in a row is similar to clustering because it allows defense and push. My function checks if there are two additional pieces next to any given piece in a particular direction. These include left to right, left right diagonal, right left diagonal. Each three in a row is awarded one point.

Knocking off a piece brings me one step closer to winning the game and I valued it very highly. If knocking off a piece does not cause a huge detriment to my position, I will most likely take it. Each piece is assigned 50 points as a constant value, so if I can knock off an opponent piece, my heuristic will gain 50 points over the enemy.

6.2.2 Results

Game	Black-player (heuristic owner)	White-player (heuristic owner)	Score	Winner
1	Jason	David	6/4	Jason
2	Jeff	David	3/1	Jeff
3	Sam	David	2/2	Tie

Table 7 - Individual game results: David's excerpt

6.3 Sam's heuristic

6.3.1 Description

The heuristic that I created was made up of four main components. The components are piece advantage, ally adjacency, and two components related to piece position on the board.

The piece advantage component counts each piece on the board and returns a score relative to the player being analyzed. States that offer a piece advantage with this heuristic will score much higher than other states. The higher this heuristic is weighted the more aggressive the agent will become.

Ally adjacency in abalone is a critical component of the game. Having ally pieces adjacent to each other ensure safety while setting up aggressive plays. In my heuristic, pieces that are adjacent to other ally pieces score points for each of those allies.

Piece position is of the utmost importance in the game of Abalone. Your pieces are the safest in the center of the board, and scoring points requires that opponent pieces are on the outskirts of the board. The states score points for having ally pieces close to the center of the board, and opponent pieces toward the outsides.

The points awarded to pieces based on position are calculated through the following arrays. POSITION_WEIGHT_MAP is used for ally pieces, POSITION_PUSH_MAP for opponent pieces.

```
POSITION_WEIGHT_MAP =
      1, 2, 2, 2, 1,
    2, 4, 4, 4, 4, 2,
  2, 4, 7, 7, 7, 4, 2,
2, 4, 7, 10, 10, 7, 4, 2,
1, 4, 7, 10, 15, 10, 7, 4, 1,
2, 4, 7, 10, 10, 7, 4, 2,
  2, 4, 7, 7, 7, 4, 2,
    2, 4, 4, 4, 4, 2,
      1, 2, 2, 2, 1,
```

Figure 14 - Sam's position weight map

```
POSITION_PUSH_MAP =
   15, 10, 10, 10, 15,
   10, 7, 7, 7, 7, 10,
  10, 7, 3, 3, 3, 7, 10,
10, 7, 3, 2, 2, 2, 7, 10,
15, 7, 3, 2, 1, 2, 3, 7, 15,
10, 7, 3, 2, 2, 3, 7, 10,
  10, 7, 3, 3, 3, 7, 10,
   10, 7, 7, 7, 7, 10,
   15, 10, 10, 10, 15,
```

Figure 15 - Sam's position push map

6.3.2 Weights

The three sub-heuristics that come together to make the heuristic are weighted differently. Piece advantage is more heavily weighted than the other heuristics. States that protect ally pieces and capture enemy pieces are sought after. Weighted second are the position heuristics, as overall position on the board offers advantages. Lastly, clustering is weighted the least in this heuristic. I decided this could be weighted less because the POSITION_WEIGHT_MAP encourages clustering toward the center of the board already, so there is a little bit of overlap in those two sub-heuristics.

Constants:

1000 x piece advantage value

10 x sum of position push map of opposing colored marbles

100 + position on position weight map

50 + number of allies adjacent

I chose to weight these values more towards piece advantage and opponent position because I wanted my heuristic to be more aggressive. As testing went on, I found that my heuristic suffered in defensive strategy because of this.

6.3.3 Results

Game	Black player (heuristic owner)	White player (heuristic owner)	Score	Winner
1	Jason	Sam	6/5	Jason
2	Jeff	Sam	6/4	Jeff
3	Sam	David	2/2	Tie
4	Sam	Rongyi	0/5	Rongyi

Table 8 - Individual game results: Sam's excerpt

6.4 Jeff's heuristic

When playing the game of Abalone, I noticed two key points. The player whose pieces were in the middle had a lesser chance of having pieces bumped off (simply by being further from the edge), and pieces that were isolated were easy pickings for the other player putting in mind the idea of

cohesion and keeping your pieces together. For my heuristic, I expressed these two points.

Note: The position map and heuristic calculator may look similar among group members. This was a conscious decision to make comparison of heuristics between the members, easier. The backbone behind the calculator, where the heuristics are differentiated, is of course made by each member individually.

6.4.1 Being in the center

For the pieces to want to be in the center, a map was constructed, similar to that used for initial board setup. This map had different weightings for each square, with the squares at the center having a higher weighting compared to those near the edge, the map can be seen below:

```
private static final int[] POSITION_MAP = new int[] {
        1, 1, 1, 1, 1,
        1, 2, 2, 2, 2, 1,
        1, 2, 5, 5, 5, 2, 1,
        1, 2, 5, 8, 8, 5, 2, 1,
        1, 2, 5, 8, 10, 8, 5, 2, 1,
        1, 2, 5, 8, 8, 5, 2, 1,
        1, 2, 5, 5, 5, 2, 1,
        1, 2, 2, 2, 2, 1,
        1, 1, 1, 1, 1,
};
```

Figure 16 - Jeff's position weight map

For the calculation involving this map, the agent looks at each marble on the board and tries to maximize the overall 'heuristic' value of its own color.

6.4.2 Being cohesive

Cohesiveness can be looked at as a value for each piece, that describes the number of friendly pieces around it. For my heuristic and increase in these surrounding pieces leads to an increase in cohesion score, but not at a 1:1 rate. My experiences when playing were that, while having a duo of pieces, or a triplet, is much better than a single piece, having a 'blob' where pieces have four or more neighbors each, is much more desirable. That is the cohesion score should have a relationship to neighbor number that is not linear, but closer to quadratic. The formula I settled on to best describe this behavior is seen below:

```
adjacent = (int) (1.5 * adjacent) + 3;
```

The cohesion is added to the overall heuristic score in the same way centeredness was added, through the heuristic calculator.

6.4.3 Weightings

My initial inclination was that a shifting towards centering would product a better result, as in most board games I've played (chess, Go, shogi) control of the center is paramount. I found the most success with equal weightings of centering and clustering, and for myself that makes the most sense. You cannot win without one or the other, and when deciding what to focus on for a turn (if you're forced to pick), the decision comes down to the state of the board.

For the knocked-out pieces, I found that having an extremely high weight, so the program almost always attempts to knock out a piece, is the best. In my opinion, knocking out is the best decision to make, as it forces limits on the opponent's next move. Now they may not be able to consider a 2-for-1 sacrifice, if they already have 5 pieces removed.

6.4.4 Results

The heuristic was evaluated according to section 3, that is games were played between the different heuristics, and the one with the best record was chosen as winner. Below are the results of the heuristic matches: 2 wins, 1 draw, and 1 loss in 4 games.

Game	Black player (heuristic owner)	White player (heuristic owner)	Score	Winner
1	Jason	Jeff	6/5	Jason
2	Jeff	Sam	6/4	Jeff
3	Jeff	David	3/1	Jeff
4	Rongyi	Jeff	4/4	Tie

Table 9 - Individual game results: Jeff's excerpts

6.4.5 Conclusion

The reason I believe this approach works best is not because of the two-factor approach described above, though I do consider it the best approach. Each of our group members arrived at the same conclusion that these two factors were key. What separates this heuristic from the rest, is that the weighting given to both centeredness and cohesion can be modified, to find an optimal balance. Despite this, I was unable to find a balance so that my heuristic one the most games, coming in third instead.

The conclusion for this heuristic is that it is very good, and may be improved with further optimizations to weighting, but was not the best choice for this project.