Softwarepraktikum SS 2019

Assignment 2

Group - 6

Roman Vuskov	376809	roman.vuskov@rwth-aachen.de
Songran Shi	357791	songran.shi@rwth-aachen.de
Marlene Damm	379945	marlene.damm@rwth-aachen.de
Marvin Gazibarić	378154	marvin.gazibaric@rwth-aachen.de

Task 1

1.1 Network Connection

The network connection between our client and the central game server is handled using Java.Net sockets. When provided with a host name and a port, our program directly connects to the server at program startup.

The `ServerConnection` class represents an established connection to the server. Internally it wraps around a `java.net.Socket` instance that is created in the constructor and closed upon calling `close()`.

Apart from that, the Api of `ServerConnection` consists of two functions:

- `awaitMessage()` blocks the execution, until a message is fully received and then returns that message.
- `sendMessage(Message msg)` sends the given message to the server.

Messages are wrapped into instances of the `Message` class. It holds the message type, and the data in binary representation. This is a more robust implementation in comparison to handling everything externally and just providing the whole message as binary data.

The message type is represented by the `Message.Type` enum that contains a variant for every existing message type. It also handles the conversion between the binary values and the enum variants. This provides a very convenient way of abstraction over the individual binary values, so that we don't have to remember them or look them up.

We also don't want to care about handling 16 and 32 bit values ourselves, as this requires error-prone bit shifting operations. Luckily, Java provides `DataInputStream` and `DataOutputStream`, which wrap around other input and output streams and provide an abstraction layer over the immediate binary data. That way, we can directly read and write `int` values, i.e. the 32-bit message size value from and into the socket.

Also there is the `ByteBuffer` class from the `java.nio` package, that provides similar functionality, but instead wraps around `byte` arrays. We are using this to decode the message content.

1.2 Message Handling

Our Reversi client is now able to parse and handle every type of message. On move request messages (type four), the `requestMove()` method of the new singleton `AI` class is called, which returns the next move we are going to make. Currently, the sum of all heuristic functions is used to determine the best move to make out of all

legal moves. This approach has inherent problems and open questions like "how much mobility points is one stability point worth?". It is only used in this early stage of development and will be replaced by a more suitable solution.

Task 2

The asymptotic complexity analysis in the following sections should be taken with a grain of salt since the game board consists of at most 2500 tiles, making constants not entirely negligible. Actual performance testing must still be conducted to acquire more accurate complexity measures.

2.1 Legal Moves

We implemented a more efficient method for determining the set of legal moves from any given game state during the Building Phase. Before this, search for legal moves is realized by sweeping through all tiles on the map and inspecting move validity using our isLegal-method from Assignment 1 Task 3. The old method is especially inefficient in the beginning stage of the game when there are few stones on the board.
Instead, the new method getLegalMoveTiles exploits a particular property of Reversi rules: any legal move during the Building Phase requires at least one opponent stone to be captured in between two of the player's stones. We can thus use the player's stones already placed on the board as end points of such capture moves and search radially outward. This operates very similar to isLegal (which starts out from the new tile instead) but inspects far fewer tiles.
For the bombing phase any tile that is not a hole can be targeted. Our method simply checks each tile for this property and returns all non-hole tiles.

2.2 Mobility, Bonus and Uncertainty Phase

These three heuristics were trivial to implement and require negligible computational resources.

2.2.1 Mobility

As defined in the previous report, mobility is simply a measure of the number of legal moves available to the player in a given game state. Since we have already implemented getLegalMoveTiles, we can simply evaluate the size of the set of legal moves.

2.2.2 Bonus

Both override bonus and bomb bonus heuristic can be quickly calculated using global state parameters. Currently, the valuation of a bomb vs. an override stone only depends on stateless variables, i.e. it stays the same over the entire game. It thus causes an AI to always choose the same bonus request for any captured bonus tile which might be sub-optimal. Further analysis of state-dependency is therefore required for future versions of this heuristic.

2.2.3 Uncertainty Phase

For this heuristics we inspect each tile of the map for any choice/ inversion tiles left. This implementation could be improved by tracking the number of choice/inversion tiles as a game state attribute. However, we decided to postpone this change due to other issues with this heuristic: In test runs it turned out that on some maps, certain tiles are unreachable or rarely captured over the entire game in practice. Since our prescription prohibits the use of other heuristics until all choice and inversion tiles are removed, some alterations must be implemented for a proper game AI.

2.3 Clustering

2.3.1 Implementation

The pseudo-code (1) below demonstrates our implementation of the clustering heuristic, which follows the prescription in the previous report exactly:

Algorithm 1 Clustering Heuristic

Require: GameState, PlayerNr

```
function CLUSTERING()
    for each Player do
        if Player is my Player then
            rivalry[Player] ← −1
        else
            rivalry[Player] ← (m_j · (2r+1)^2) / ((p−1) · (|N_j − N_1| + 1))

    for each Player's stone do
        for each stone in bomb diameter do
            BombedStoneCount[stoneOwner]++
        for each Player do
            clusteringSum += BombedStoneCount[Player] · rivalry[Player]

    return clusteringSum /((2 · (2 · bombRadius + 1) − 1)^2
```

2.3.2 Complexity Analysis

The computation of rivalry factors is trivial in time complexity. For the clustering factor of each stone, all tiles within one bomb diameter must be considered. For the total clustering factor, the clustering factor of each stone must be considered. The total time complexity of this heuristics is therefore in $\mathcal{O}(n \cdot r^2)$ where n is the number of the player's stones on the board and r is the bomb radius (excluding edge cases where bomb area grows exponentially with bomb radius).

2.4 Stability

2.4.1 Implementation

We implemented the stability heuristics according to prescription in the previous report, with the exception of step 4, i.e. stability resulting from completely filled rows/columns is not included. Two possible implementations of step 4 have been considered:

- Tracking approach
  - Determine all rows, columns and diagonals from map geometry before the start of the game.
  - Track fill level of each as global game state attribute during the game.

  - For the stability value of a certain stone inside the stability heuristic method, check the fill level of the row/column/diagonal the stone belongs to.

- Search approach
  - For each of the player's stone, search radially outward for empty tiles.
  - If no empty tile is found before arriving at a hole or the original tile (path could be a circle), the row/column/diagonal must have been completely full and the stone is stable.

The tracking approach is rather difficult to implement due to custom transitions making weird geometries possible. The search approach on the other hand is computationally expensive ($\mathcal{O}(n \cdot d \cdot l)$), where n is the number of the player's stones on the board, d is the number of directions and l is the average length of straight paths). For this reason we postponed the implementation of this particular step. The rest of the rating function still provides a reasonable heuristic:

Algorithm 2 Stability Heuristic

```
Require: GameState, PlayerNr
  function STABILITY()
    for each Player's stone do
      for each direction do
        if Player's stone's neighbour in direction is hole then
          Stable[direction] ← Player's stone

    while Stable[] contains unpropagated stones do
      for each direction do
        for each stone in Stable[direction] do
          if stone's neighbour's owner in direction == Player then
            Stable[direction of arrival] ← stone's neighbour
          stone_propagated ← TRUE

    return sum of sizes of Stable[]
```

2.4.2 Complexity Analysis

A stone can propagate each stability direction for at most one step, i.e. each stone will be evaluated at most twice: once during initial stability assignment (step 1 in the previous report) and once during stability propagation (step 2 - 3). Therefore, this heuristic belongs in the complexity class $\mathcal{O}(n \cdot d)$ where n is the number of

the player's stones on the board and d is the number of stability directions to be examined.