Project Report

Group member

U077151L Lai Xiaoni
U077194E Qiao Li
U077181Y Zeng Qiang

Content

1. Problem statement

Game Objective:
Keep the Ozz alive as long as possible.

Game Operations:

1. Place the current tile at an empty x,y coordinate
2. Destroy an occupied tile at a filled x,y coordinate (only possible if ooze has not started to flow through the tile yet)
3. Do nothing(as this operation has no impact to the game strategy, we will avoid use this operation)

Game events:
Time Line 0----------6----------9----------12----------15----------18----------
start ozz ozz ozz ozz ozz
Each Time only one Operation is permitted.

Game status:
Status includes board with pieces and Queue.
They are represented as char[7][7] and char [5] respectively.

Game Score:

1. Each tile where the ooze flows through: +2
2. Bonus for each cross tile where the ooze flows through both ways: +5 (a total of (2x2)+5 = 9)
3. Each destroyed tile: -3

Problems:
Our agent needs to choose an operation with a best coordinates as parameter to maximize the score.

General strategy:
1. Choose appropriate operation;
2. Search for the best position for the operation;

To implement the strategy we need:
1. A way to decide whether it is time to take destroy operation;
2. An efficient search algorithm;
3. An utility function used by search algorithm to calculate the desirability of a certain game status;

2. Related Work

Recursive Best-First Search(RBFS)
A memory bounded search algorithm that always keeps track of second best node and the best node of its children. When no child that is better than second best node found, it recurs back to search the second best node.
The algorithm is derived from Best-First Search.
Its space complexity is O(bd) where b is the branching factor and d is the maximum search depth.
Its time complexity depends on the number of nodes generated thus depend on the description of the goal.
It's complete, however, not guaranteed to be optimal.

Utility Function
The use of Utility function by search algorithm is to provide logically proved measurement for a certain game status. The criteria can be the distance to the goal state, maximum scores etc.

3. Proposed Approach

Search Algorithm
3.1 The idea of RBFS

At any stage during the game, the agent needs to determine in which slot of the board the next given piece can be place. Therefore, it needs a search algorithm which can tell it the most suitable next slot. After some careful consideration, we decided to adopt the idea of Recursive Best First Search algorithm. It possesses some characteristics that can satisfy the demands of our agent:

Firstly, if the agent only determines the best slot for current given piece, it is

quite probable that the agent commit some shortsighted mistakes. For example, given a two-element queue of {'z','|'}, on top of the original 'e' piece, the shortsighted agent may simply place the current 'z' in the slot directly above 'e' because 'z' and 'e' can be connected vertically. But the next piece '|' would have no place to go. A more intelligent agent would be able to foresee such possibility and returns a slot that can maximize the score in the long run. The agent's such ability highly depends on its search algorithm. In this game, we have a queue with five elements each round. The search algorithm thus has to be able to explore five pieces ahead in order to give the best slot. Therefore, it is important to have a recursive algorithm which can expand a utility tree with five levels deep.

Secondly, having a recursive algorithm which can foresee the best steps within five rounds may be ideal, but the space and time complexities would be huge if we explore all possible combinations of five pieces on a 7*7 board. Thus suitable pruning is necessary. In the search algorithm we have implemented, only the best and second best steps (placing the pieces in these slots can result in highest or second highest utilities of the board) are kept in each level. The rest steps in that level are pruned away even though there may be very slight possibility that the steps turn up higher utilities in the next level. Then, the search algorithm explores to the deeper level and if it finds that future steps after the current second best step can turn up higher utilities, it will return the current second best step as the true "best" instead of the current best step.

The following is the pseudo code of the skeleton of our search algorithm. It returns an integer array [x][y][utility] where x-y is the coordinates of best slot.

```
int[] RBFS(currBoard, qCounter, queue) {
   newPiece<-queue[qCounter];
   list<-list of x-y coordinates in the searching range
   bestUtility = -∞;
   secondBestUtility = -∞;
   bestBoard, secondBestBoard;
   bestX, bestY, secondBestX, secondBestY;
   loop (each x-y in the list) {
        newBoard <- currBoard + newPiece in x-y slot;
        newUtility <- getUtility(newBoard);
        if (qCounter < MAX_LEVEL) {
             if (newUtility>bestUtility) {
                  replace secondBest*** with best***;
                  replace best*** with new***;
             } else if (newUtility>secondBestUtility) {
                  Replace secondBest*** with new***;
             }
        } else {
```

```
                If (newUtility > bestUtility) {
                    Replace best*** with new***;
                }
            }
        }
        If (qCounter <MAX_LEVEL) {
            Score1<-RBFS(bestBoard, qCounter+1,queue);     //recusion
            Score2<-RBFS(secondBestBoard,qCounter+1,queue);    //recursion
            If(score2>score1)return int[secondBestX][secondBestY][secondBestUtil];
            Else return int[bestX][bestY][bestUtil];
        } else {
            Return int[bestX][bestY][bestUtil];
        }
    }
```

3.2 Slots in the searching range

3.2.1 End slots, neighboring slots, all available slots

The list of x-y coordinates used for each level of searching is constructed in a way such that some slots are purposely placed in the front part of the list. These slots are considered to have higher priority to be searched than other slots, because from our gaming experience they are more probable to be the next best slots. The reason why we consider different priorities of slots to be searched is to control the placement of pieces in a certain region; this is particularly useful in the beginning phase of the game when the board is almost empty.

There are three different types of slots to which we pay special attention to:
1) End slots: slots that are immediately at the opening of all pipes;
2) Neighboring slots: slots that are immediately surrounding the end slots;
3) All available slots: all the empty slots in the current board except end slots and neighboring slots.

The end slots are appended to the list first, followed by neighboring slots, and then by all-available slots. In such way, when the board is almost empty and many slots gives the same utility, the agent will still tend to place pieces surrounding the main pipe, so as to maximize the score.

3.2.2 Special consideration when the board is full

When the board is full, all the above three types of slots would no longer exist as they are all defined as empty slots. In this case our agent is unable to advance any more; this is the bottleneck of our algorithm. To solve this problem, all slots in the board would be considered in the searching range

whenever the agent finds that the board is full.

3.2.3 Special consideration when the opening of main pipe is blocked

The agent would not place piece that could block the main pipe, as such placement must have very low utility value. However, it is possible that the agent place a piece in an empty region earlier and later the main pipe is built longer and longer until its opening is blocked by the previously placed piece. In this case, the slot at the opening of main pipe must be considered in the searching range; not only that, it must be put in a higher priority in the searching list.

Utility Function

The main purpose of out utility function is to indicate the desirability of a given state in this game. Given a 7*7 board with the placed tiles, the utility function will be able to return an integer.

To calculate the utility, we make use of two objects, namely *Utility* and *PipeRecord*. When a new *Utility* object is created with a given board, the board will be traced with a recursive method to find out the main pipe and the sub-pipes in the board. The pipes will be identified with another int[7][7] called *status*. Slots in the main pipe will be indicated as 1, empty slots as 0 and any other positive numbers indicate other pipes. All the slots in the same pipe will have the same number in *status*. During the tracing, the information of each pipe will be stored in different *PipeRecord*s in a list. The information includes the 1) length of the pipe, 2) the two ends of the pipe and 3) the distance of the pipe from the main pipe.
After creating the Utility object with a board, the search algorithm can obtain the utility value with the method *getUtility()*. The utility value is calculated as follows:

1. Every slot in the main pipe is given 200 points. A valid slot lengthens the main pipe is highly encouraged.
2. The slots in sub pipe nearest to the main pipe are given 40 points each. The next sub pipe is given 30 per slot, the third sub pipe is given 20 and so on. This is to encourage the sub pipes to be nearer to the main pipe's end and at the same time, discourage a high number of sub pipes in the game. The closeness of the sub pipe to the main pipe is determined by the shortest distance between any ends of the sub pipe to the end of the main pipe.
3. If the end of the main pipe is blocked by another pipe or by the wall, a penalty of 500 will be given. For example, an end of '|' has a '-'above it or the end is at board[3][0], this end is considered blocked.
4. If the ends of any other sub pipes are blocked, a penalty of 100 will be given.
5. 200 points will be added if a cross tile ('+') can be used for both directions.

The final utility value can be positive or negative. With the same number of non-empty slots in a board, a higher utility value is more desirable.

Task allocation:

Lai Xiaoni	• Search algorithm coding; • Relevant part in the report; • Editing of report;
Qiao Li	• Utility Function; • Relevant part in the report;
Zeng Qiang	• Skeleton code; • Rest of the report and its final formatting; • Other minor Admin tasks;

.

Terminology

Board: The 7X7 gaming Area consisting of 49 squares.

Slot: One of the squares on the Board that to be place a piece.

Piece: A tile that represents a unit pipe through which the Ozz can flow.

Unit pipe:

'+'	<cross piece>
'|'	<vertical pipe>
'-'	<horizontal pipe>
'q'	<tile with left and up outlets>
'e'	<pipe with up and right outlets>
'c'	<right and down outlets>
'z'	<down and left outlets>

Pipe: A longest continuous series of pieces on the board that the Ozz can flow through.

Main pipe: The pipe that linked to the starting piece i.e. piece (3, 3).

Sub pipe: The pipe that do not link to the starting piece.