Ozploding bozmbs – Bomberman in Oz

Alexandre Gobeaux and Gilles Peiffer

***Abstract*—In this paper, the methodology and design choices behind our implementation of the Bomberman game in Oz are given. Both the turn-by-turn part of the game controller as its simultaneous part are explained, and an attempt is made to clarify why the authors took certain decisions with regards to how the final product works. On top of that, an algorithm was implemented in order to control the players' moves, reacting to various messages it receives containing either requests or information about the game. This was done in at various levels of complexity, building increasingly efficient players. On top of the mandatory parts of the project, various supplementary features and embellishments are also included in the implementation.**

In order to validate our product, interoperability tests were also carried out with both the reference player and other teams' players. This paper presents a brief summary of the results of these tests.

I. Introduction

BOMBERMAN is a famous game.

II. Controller structure

A. Turn-by-turn controller

B. Simultaneous controller

III. Player structure

A. General structure

IN order to play the game intelligently, players have to store all relevant information about a game; in order to do this in a clean manner, the authors opted to use record structures, which can easily be modified with the `AdjoinList` function. These records change vary for more advanced players, as they need to store more data about the game in order to make use of their intelligent strategies, but the records always contain at least the following fields:

- `id`: ID of the player;
- `state`: state of the player;
- `lives`: number of remaining lives;
- `pos`: current position;
- `sppos`: assigned spawn position;
- `bombs`: number of bombs owned;
- `map`: map of the game;
- `score`: number of points.

Players have to respond to different messages they can get from the controller, in the form of a stream of instructions. In order to handle each of these messages, multiple so-called "handler functions" were created, one for each possible instruction. Most of the messages that a player can receive in its stream are relatively straightforward; in this paper, only the most important instruction, `doaction(ID Action)` is explained.

The `doaction(ID Action)` message asks the player for its ID and its next action (place a bomb at its current location or move to a neighbouring location). Multiple players were implemented, with the difference between them lying mainly in the way the "optimal" action is computed. These differences are explained in Section III-B.

B. Decision algorithms

1) Basic player: In our basic random player, `Player001Kardashian.ozf`, the decision algorithm looks like this:

- If the player has no bombs left, move to a random neighbour with uniform probability over the acceptable moves.
- If the player has bombs left, the player has a $10\,\%$ chance to drop a bomb, and a $90\,\%$ chance to move to a randomly chosen acceptable neighbour.

This strategy is not very efficient: the player does not avoid bombs, and does not hunt for points.

2) Advanced player: Our next player, `Player001Tao.ozf`, is slightly more advanced; its decision algorithm tries to avoid standing in dangerous areas, i.e. too close to bombs. This means that the advanced player has to keep track of where bombs are located. In order to do this,

Alexandre Gobeaux – 42191600
E-mail: alexandre.gobeaux@student.uclouvain.be
Gilles Peiffer – 24321600
E-mail: gilles.peiffer@student.uclouvain.be

one must simply add an argument to the `summary` record: `bomblist`, which stores a list of bombs that are currently on the map. When the player has to determine its next action, it uses the following algorithm:

- If the player has no bombs left, move to the least dangerous acceptable neighbour, where the danger rating of a tile is found by initialising it to zero, and then adding points if the tile is in the blast radius of a bomb (taking into account that walls and boxes stop fire from spreading, as well as the distance from the bomb).
- If the player has bombs left, the player has a $10\,\%$ chance to drop a bomb, and a $90\,\%$ chance to move according to the algorithm above.

This strategy is more successful than the basic player, and most of the time manages to avoid bombs. It does not however hunt for points or bonuses, and can make wrong choices when trying to avoid bombs (forcing itself into a dead end, for example).

3) Intelligent player: The intelligent player, `Player001Turing.ozf`, builds further on what the advanced player does, by adding the ability to intelligently search for points and bonuses, with the latter being preferred in most cases. The algorithm is the following:

- If the player has no bombs left, move to the least dangerous acceptable neighbour, where the danger rating of a tile is found by initialising it to zero, and then adding points if the tile is in the blast radius of a bomb (taking into account that walls and boxes stop fire from spreading, as well as the distance from the bomb). If multiple neighbours are equally safe, run a breadth-first search for points and one for bonuses. If possible, the player tries to go for a bonus first, given that the distance to the next bonus is not too big compared to the distance to the next point[1]. The next move on the path to
- If the player has bombs left, the player has a $10\,\%$ chance to drop a bomb, and a $90\,\%$ chance to move according to the algorithm above.

[1] Heuristically, we found a ratio of 2 to work fairly well.

IV. Interoperability

V. Extensions

VI. Conclusion

The conclusion goes here.

References

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.