Wumpus

Specification

This practical involved creating the classic Hunt the Wumpus style game conforming to the basic specifications provided and then using our creativity we should implement further extensions. Our extensions/specifications are as follows:

- Graph implementation of the game board (not a torus)
- Treasure hunt race against another player over the network
- AI that can logically solve the game

Design

The design can be separated into two logically different parts: the representation of the cave system and the game play.

We chose a graph instead of the classic dodecahedron/matrix/torus representation of the board. This way we are able to create a more realistic cave system as well as a more complex one, as we can have an arbitrary number of caves connecting to a single cave. The way we do this is first we generate a simple graph without any cycles in it and then when that is finished we add in extra edges.
It is important to note that we want to create a graph that allows the player to win every single time. The problem here are the pits, which can not be stepped over and they can very easily block the path to the treasure and the exit. The way we deal with this issue and keep every single cave accessible is we take every cave that is connected to the one containing the pit and we connect those together, always allowing a path around the pit. This is a much quicker and computationally cheaper way of avoiding unplayability instead of searching the graph and using route finding algorithms and is something that stems from the more flexible structure of a general graph instead of a grid that is represented by a 2D array.
The other important design decision for the representation of the game is that instead of several objects, we represent anything that may be found in the caves as a list of “cave actions”. We represent these actions, namely finding the treasure, falling into a pit, being eaten by the Wumpus, being teleported by superbat and finding the exit, in an enumeration and then we store a certain number of these in a list within the cave objects. Based on these the game can then calculate whether we feel a breeze/stench/glittering next to them.
We chose to keep most of the original game rules/ effects such as the player is not forced to kill the Wumpus in order to win the game and we can not detect superbats in the adjacent caves.

The other big part of the design is the players, and how input is fetched from them. All three of them are extended from the abstract Player class which ensures that the transaction of data between the game and the player objects is in both ways uniform.
We have three kinds of players:

- KeyboardPlayer: Human player playing locally from a keyboard. Nothing very tricky here, we only implemented the necessary error handling, but nothing too remarkable
- AIPlayer: The way we approached the AI implementation is we assume every danger being present in every single cave and as we explore more and more of the cave system the more of these assumptions we can get rid of – basically the same way a human player would play.

We then improved this further using a breadth-first search algorithm.

- NetworkPlayer: This is a bit different from the previous two in that we handle its input differently and we only evaluate it once the game for both players has finished.

Testing

We tested the program with various erroneous inputs, as well as we tried feeding it CTRL+D. In both cases it handled these well:

```
Initialising game...
What is your name?
asd
Would you like to play over the network (y/n)?
a
Please enter a valid answer.
9
Please enter a valid answer.
n
Would you like to add another player? (y/n)
s
Please enter a valid answer.

Please enter a valid answer.
```

Screenshot 1: Erroneous input

```
You are in cave number 18!
There is a strong stench...
You can feel a light breeze...
You can move to the caves: 1, 2, 3, 5, 6, 9, 11, 12, 13, 16, 20
Would you like to shoot an arrow? (y/n)
no
Would you like to shoot an arrow? (y/n)
123
Would you like to shoot an arrow? (y/n)
Oh no! A critical error has occured during runtime: No line found
The system will now exit.
```

Screenshot 2: Erroneous input and supplying CTRL+D

We then tested the AI with the following map set up:

```
int NUMBER_OF_CAVES = 100;
int NUMBER_OF_BATS = 2;
int NUMBER_OF_PITS = 5;
int INITIAL_ARROWS = 3;
```

Screenshot 3: AI Set up #1

And the result was the following:

```
Games played: 10000
Games won: 5788
Wumpus: 508
Pit: 3738
```

Screenshot 4: First AI test

As we can see the AI won in the majority of the time, but it fell for pits way too many times. We improved it by implementing a breadth-first search algorithm to better the results. The results have improved dramatically:

```
You have reached the exit.
As you have collected the treasure you have exited the cave! Well Done!
Games played: 10000
Games won: 7052
Wumpus: 368
Pit: 2600
```

Screenshot 5: AI with Breadth-first algorithm

We then ran tests with a more complex cave set up:

```
int NUMBER_OF_CAVES = 200;
int NUMBER_OF_BATS = 1;
int NUMBER_OF_PITS = 5;
int INITIAL_ARROWS = 3;
```

Screenshot 6: Second AI set up

And the results after 10000 games were very impressive once again:

```
Games played: 10000
Games won: 7836
Wumpus: 292
Pit: 1879
```

Screenshot 7: AI results with second set up

Evaluation

We have completed the basic deliverable and have extended that with an AI, the option to play the game through the network as a race against another person or AI as well as we used a more complex data structure. All of these function properly, and they have built-in error handling, should anything go wrong, we can handle it gracefully, the user should not experience anything unusual. Evaluating the AI it seems that it has been implemented very well, but it is obviously still dependent on luck a lot. An explanation for this is that the AI sometimes has to guess a move, just as a normal player would. The chance of this happened is increased by the use of a graph.

Conclusion

In this practical the biggest challenge was finding the optimal representation for the data structure we wanted to use and how we could overcome the problems that arose from using such data structures (the array-adjacency matrix pairing in our case). This made us research more advanced graph walking algorithms (such as the breadth-first search algorithm, which we chose eventually) so that we could optimise our AI.

Acknowledgements

We used our code for the NetworkPlayer from our Backgammon practical, which is a modified version of the code that has been kindly provided to us for that practical by Prof. Saleem Bhatti.

References

“Graph Theory:DFS and BFS”; last accessed 21 April 2015;
http://www.cs.cornell.edu/~wdtseng/icpc/notes/graph_part1.pdf