Cardiff School of Computer Science and Informatics

Coursework Assessment Pro-forma

Module Code: CM3112
Module Title: Artificial Intelligence
Lecturer: Steven Schockaert
Assessment Title: Coursework
Assessment Number: 1
Date Set: 2 November 2020
Submission Date and Time: 23 November 2020 at 9:30am
Return Date: 14 December 2020

This assignment is worth 40% of the total marks available for this module. If coursework is submitted late (and where there are no extenuating circumstances):

1. If the assessment is submitted no later than 24 hours after the deadline, the mark for the assessment will be capped at the minimum pass mark;
2. If the assessment is submitted more than 24 hours after the deadline, a mark of 0 will be given for the assessment.

Your submission must include the official Coursework Submission Cover sheet, which can be found here:

https://docs.cs.cf.ac.uk/downloads/coursework/Coversheet.pdf

Submission Instructions

You should submit a single zip file, containing the source code of your implementation of Task 1 and a PDF version of your answers to Task 2.

Description		Type	Name
Cover sheet	**Compulsory**	One PDF (.pdf) file	[student number].pdf
Solution	**Compulsory**	One zip file	[student number].zip

Any deviation from the submission instructions above (including the number and types of files submitted) will result in a reduction in marks for the assessment of 5%.

Staff reserve the right to invite students to a meeting to discuss coursework submissions

Assignment

This assignment consists of two tasks. In Task 1, you are required to implement a minimax computer player for the game Snake, using the java framework that is provided. In Task 2, you are required to answer a number of questions about an (unseen) pathfinding method called bidirectional search.

Detailed instructions can be found in the appended description.

Learning Outcomes Assessed

1. Choose and implement an appropriate search method for solving a given problem.
2. Define suitable state spaces and heuristics.
3. Understand the basic techniques for solving problems.

Criteria for assessment

Credit will be awarded against the following criteria.

For the implementation in Task 1:

- [Correctness] Does the code correctly implement the required algorithm?
- [Efficiency] Is the computational complexity of the implementation optimal?
- [Clarity] Is the code clearly structured and easy to understand?

For the questions in Task 2:

- [Correctness] Does the answer demonstrate a solid understanding of the problem?
- [Clarity] Is the answer well-structured and clearly presented?

Marks for Task 1 will be assigned based on the following benchmarks:

1st (70-100%)	The code is correct, efficient and clearly structured.
2.1 (60-69%)	The code is correct but may be too inefficient to perform well (when given a fixed time limit).
2.2 (50-59%)	The code is largely correct, but also contains some mistakes.
3rd (40-49)	A reasonable attempt towards a fully working implementation has been made, but the code is not finished.

Marks for Task 2 will follow the marking scheme that is provided in the attached detailed description.

Feedback and suggestion for future learning

Feedback on your coursework will address the above criteria. Feedback and marks will be returned at the latest on 14 December 2020 via learning central. General feedback on the coursework will be provided in the subsequent online lecture.

Task 1: Minimax (70 marks)

Getting started

In the `snake.zip` archive, you will find a basic implementation of a multiplayer version of the game Snake. In this game, each player has a snake, which moves across a grid (see the screenshot in Figure 1). The snakes move in turn, one grid cell at a time. The grid also contains a yellow disc. If a snake 'eats' this yellow disc, it will grow in size and a new disc will appear at a randomly chosen location. If a snake hits another snake or the outer boundary of the grid, it dies. The aim of the game is to have the longest surviving snake after a given number of steps.

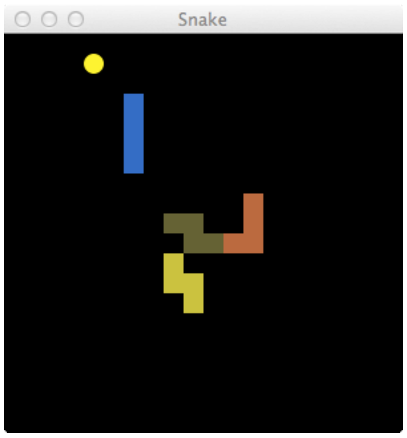

Figure 1: Screenshot of the Snake framework used for the coursework.

Game framework

The main classes of the java framework are:

Snake This class represents an instance of the game, and will repeatedly call the `doMove` method of the (remaining) players in turn, until the game is over. This class also contains the main method.

SnakePlayer An abstract class which contains some code that is common to all types of players. The method `doMove` is left abstract.

HumanPlayer An extension of `SnakePlayer` which allows a human to play the game, using the keyboard arrows.

RandomPlayer An extension of `SnakePlayer` which implements a very basic computer player, making random moves.

AStarPlayer An extension of `SnakePlayer` which uses the A^* algorithm to choose the move which gets the snake closest to the yellow disc.

Position Used for representing grid positions in `AStarPlayer`.

Node Used for representing nodes in `AStarPlayer`.

GameState This class represents a state of the game.

GameDisplay This class takes care of the visualisation of the game.

To compile the code, from outside the directory that contains the source files, you can use[1]:

```
javac snake/Snake.java
```

To run the code you can use:

```
java snake/Snake
```

In addition to the provided framework, you are allowed to reuse any code that was provided as part of this module. However, you are not allowed to use any other existing code or libraries. If you use any external sources for solving your coursework (e.g. pseudocode from a website), you should add a comment with a reference to these sources and a brief explanation of how they have been used. You are allowed to discuss high-level aspects of your solution with other students, but you should disclose the names of these students in a comment.

Assignment

The objective of this task is to implement a computer player for Snake based on the minimax algorithm. Particular challenges are the fact that the game involves chance elements (since we don't know in advance where the yellow discs will appear) and the fact that this game can be played with more than two players.

Basic framework. When implementing a multiplayer variant of the minimax algorithm, there are different assumptions that could be made about how the other players will behave (e.g. whether they will collaborate to defeat you). In your implementation, you should assume that each opponent will always play the move that is best for themselves (i.e. they will not collaborate). [40]

Iterative deepening. In your implementation, you should use iterative deepening to ensure that your player makes its move within the available time. [10]

Chance nodes. To deal with the uncertainty about where future yellow discs will appear, you will also need to consider chance nodes. Normally such a chance node would have one child for each possible position where a yellow disc might appear, which would make the search far too slow. To address this, you should instead use chance nodes with 5 children, corresponding to 5 randomly chosen samples of locations where the disc might appear. [10]

Evaluation. You are also expected to carry out an experimental evaluation of your minimax-player, by letting it compete in a four-player game against two instances of AStarPlayer and one instance of RandomPlayer. You should run this evaluation 20 times and report the number of times your player has won, in a comment block at the top of the source file of your minimax player. You should also specify how much time your computer player was given to make each move. Your submitted code should be set up such that running the main method of snake/Snake.java carries out this experiment. [10]

[1]Replace / by \ on Windows.

Task 2: Pathfinding (30 marks)

For this task, you are expected to answer some questions about bidirectional search.

The idea of bidirectional search is to simultaneously run two instances of graph search. The first instance is called the forward search. For this instance, the frontier is initialised with the initial state, as usual; we call this frontier the "forward frontier". For the second instance, called the backward search, the corresponding frontier is instead initialised with the goal states. This frontier is called the "backward frontier". Bidirectional search alternates between iterations of forward search and iterations of backward search. More precisely, the pseudocode of bidirectional search is as follows:

```
While a solution has not yet been found:
    if the forward frontier contains more elements than the backward frontier
        Expand a node from the backward frontier
    else
        Expand a node from the forward frontier
```

Expanding a node from the forward frontier is done in the normal way. Expanding a node from the backward frontier is done similarly, except that the actions are reversed, i.e. the pair (a, b) is considered as an action for the backward search if (b, a) is a valid action of the initial (i.e. forward) search problem.

To check whether a solution has been found, we now need to check whether the "forward path" and the "backward path" meet, rather than checking whether a goal state has been found. More precisely, in the basic version of bidirectional search the search finishes when the forward frontier and backward frontier have a non-empty intersection. This idea of "meeting in the middle" has several advantages, which can speed up the search process quite substantially in some applications.

1. Suppose the branching factor of the forward search and backward search are the same constant b. Suppose furthermore that there is a unique goal state, and let d be the length of the optimal path from the initial state to this unique goal state.
   (a) What is the largest number of nodes that may need to be stored in the frontier at the same time when using breadth-first search for this search problem? [4]
   (b) What is the largest number of nodes that may need to be stored in the frontier at the same time when using iterative deepening for this search problem? [4]
   (c) What is the largest number of nodes that may need to be stored in the two frontiers when using bidirectional search, if breadth-first search is used for both the forward and backward search? [4]
2. Explain why it is not possible to use iterative deepening for both the forward and the backward search. [8]
3. Suppose uniform cost search is used for both the forward search and backward search. When a solution is found (i.e. when the forward and backward frontiers have a non-empty intersection), is it guaranteed that this solution is optimal? Justify your answer by either giving a proof for why this is the case or by presenting a counter-example (i.e. a search problem where this first solution would not be optimal). [10]