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: 22 October 2018
Submission Date and Time: 12 November 2018 at 9:30am.
Return Date: 3 December 2018

This assignment is worth *30%* of the total marks available for this module. The penalty for late or non-submission is an award of zero marks.

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 your report, the java implementation and the two levels of the game that you have constructed.

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) may result in a mark of zero for the assessment or question part.

Assignment

Your task is to improve a computer player for the game Sokoban, using the provided java framework. Specifically, the aim is to implement a heuristic function which is inspired by the idea of so-called pattern databases. 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:

[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 report:

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

Feedback and suggestion for future learning

Feedback on your coursework will address the above criteria. Feedback and marks will be returned on 3 December 2018 via learning central. This will be complemented by general feedback in the lecture.

Getting Started

To get started with this coursework, first download `sokoban.zip` from Learning Central. In this archive, you will find a basic implementation of the Sokoban game. Sokoban is a puzzle game in which the player needs to move a number of boxes. An example of a Sokoban puzzle is shown in Figure 1(a). The boxes in this version of the game are represented as pink coloured rectangles, and the aim is to move these boxes to the positions which are indicated with a white dot. The possible actions available to the player are to move up, down, left or right (using the arrows on the keyboard). If the player is adjacent to a box, the player can push the box by moving in the direction of the box, provided that the cell next to the box (opposite from the player) is empty. The goal state for the puzzle from Figure 1(a) is shown in Figure 1(b).

The Java code in the `sokoban.zip` archive contains several classes. First, there are a number of classes related to the general operation of the game:

Sokoban Executing the main method from this class will launch the game.

GameDisplay takes care of the visualisation of the game.

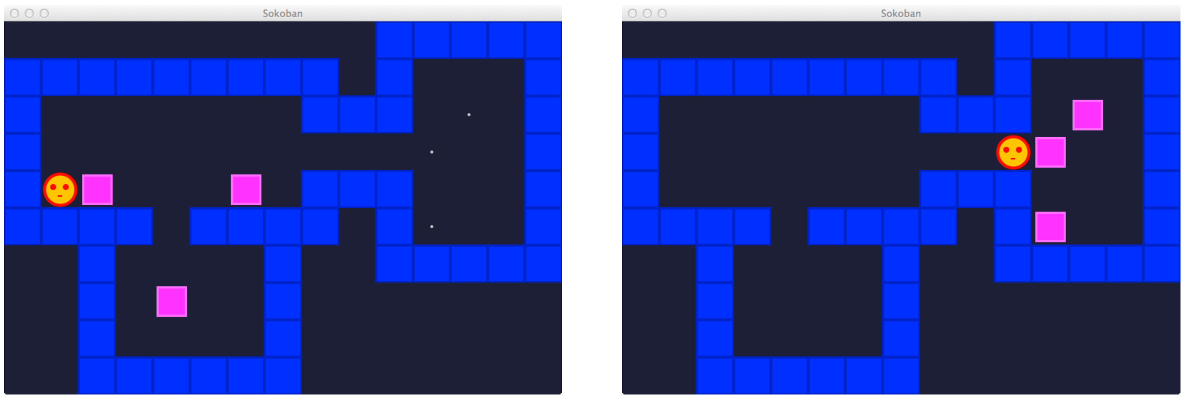
GameState represents a state of the game. Among others, it contains code for reading a level description from file, for checking which actions are legal in a given state, and for updating the game state after the player has taken a given action. This class also contains a method `setRelaxedState`, which you will be able to use for your implementation.

Position represents a position in the game (i.e. a row and a column from the given grid).

HumanPlayer implements the standard version of the game, where a human player needs to solve the puzzle (using the arrow keys on the keyboard).

SokobanAction encodes the types of actions that can be taken by a player of the game.

There are also some general methods related to the implementation of A*, which are entirely similar to the classes that you have already encountered in the lab class (**Action**, **AstarNode**,



(a) Initial state (b) Goal state

Figure 1: Screenshots of the Sokoban game used for the coursework.

State, **Node**). In addition, an implementation of an A* player for the game has also been provided:

SimpleSokobanAstarPlayer contains an implementation of the A* algorithm.

SimpleSokobanNode contains an implementation of the nodes that are used by the A* algorithm. This class also contains a method to compute the heuristic evaluation of a given node. In this implementation, this heuristic evaluation corresponds to the Manhattan distance.

Finally, there are three classes that directly relate to the coursework:

PatternHeuristicAstarPlayer contains an implementation of the A* algorithm, in which a so-called pattern database is constructed, which is used to implement an improvement of the Manhattan heuristic evaluation.

PatternHeuristicNode implements nodes that take advantage of the pattern database for computing the heuristic evaluation.

PatternDatabase contains a skeleton of the implementation of the pattern database. This is the class that you will need to implement.

The archive also contains a few sample levels. These levels are encoded as plain text files, so you can easily create your own (e.g. to test particular aspects of your computer player). An example of a level is:

```
wwwwwww
wg..g.w
w.pbg.w
w.bgb.w
w.bB..w
w.....w
wwwwwww
```

where `w` denotes a wall, `.` is an empty cell, `g` is a goal position, `p` is the position of the player (or `P` if the player is on a goal position), `b` is a block and `B` is a block which is on a goal position.

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

```
javac sokoban/*.java
```

To run the code on a Java Virtual Machine with 1GB of memory, you can use:

```
java -Xmx1g sokoban/Sokoban
```

In the implementation of the class `Sokoban`, you can change whether the game runs for a human player or for the computer player. For example, to run the basic A^* computer player, you can use:

```
SimpleSokobanAstarPlayer player = new SimpleSokobanAstarPlayer(state);
player.showSolution();
```

To use the human player, you should replace this by:

```
HumanPlayer player = new HumanPlayer(state);
```

[1]Replace / by \ on Windows.

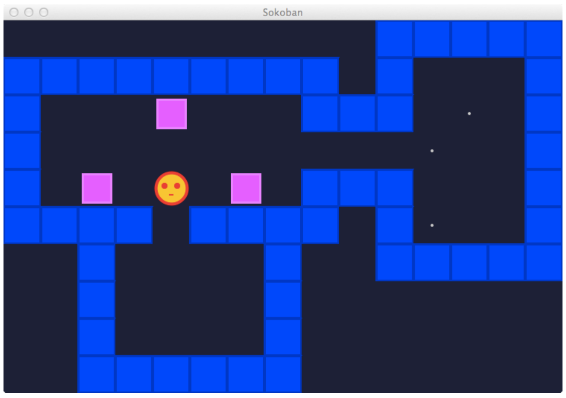


Figure 2: Example of a dead state: the level can no longer be solved from this state.

Assignment

Task 1: Implementing a pattern database heuristic (60 marks).
The Manhattan distance has two important limitations for this game. First, in some levels, the distance between a block and a goal position on the grid may not tell us much about the number of steps that are needed to actually move the block to that goal position. Second, an important issue in this game relates to the existence of so-called dead states. There are certain positions in the game which have the property that once a block occupies one of these positions, the game can no longer be solved. This is illustrated in Figure 2, where the problem is that one of the blocks is stuck against the top wall. The Manhattan heuristic fails to recognise such dead states, which means that a lot of computation time is wasted on exploring states that can no longer lead to any solution.

Your task is to implement a heuristic evaluation function which addresses both issues. This heuristic will rely on a pre-computed analysis of the level. Specifically, before solving the actual game, you should first determine how many steps are needed to solve a set of simplified (or "relaxed") problems. In these simplified problems, we keep the wall structure and goal positions of the given level, but we only consider a single block, and we vary the starting position of that block. Let us write *steps*(i, j) for the size of the optimal solution of the simplified problem where the block is initially at row i and column j, assuming that the initial position of the player is optimal (i.e. such that the resulting value of *steps*(i, j) is minimal).

The values *steps*(i, j) should be pre-computed for all positions and stored in an object of type `PatternDatabase`. Once these values are pre-computed, we can use them to efficiently compute an improved heuristic evaluation, when solving the actual game. Specifically, assume that we are in a state of the game where the positions of the blocks are given by $(i_1, j_1), ..., (i_n, j_n)$, then we can use *steps*$(i_1, j_1) + ... +$ *steps*(i_n, j_n) as our heuristic evaluation. Note that this heuristic is admissible and consistent, and can thus be used without giving up on the optimality guarantees of A^*.

Your task is to implement this strategy by completing the class `PatternDatabase`. Note that you will not need to make changes to any of the other classes. To generate the simplified problems, you can make use of the method `setRelaxedState` in the class `GameState`.

Task 2: Report (40 marks) You are asked to write a concise report discussing the following points. The total length of your report should not be longer than one page (5 marks will be deducted for each additional page).

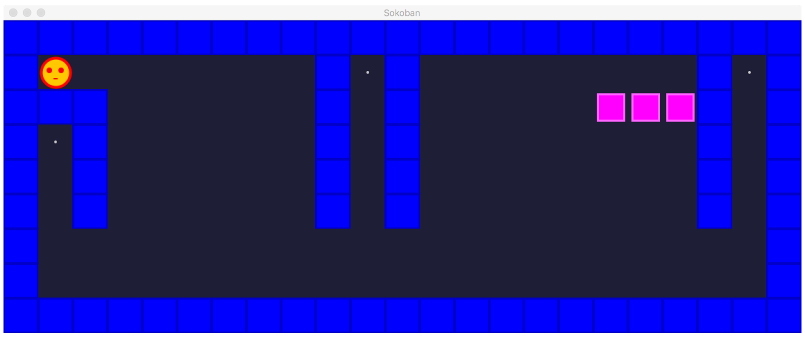


Figure 3: Example of level which is difficult to solve using A*.

1. Construct two levels for the game: one where the pattern database heuristic has a large impact (i.e. leads to a substantial decrease in execution time and/or number of expanded nodes) and one where its impact is negligible. You should include these examples as separate text files with your submission, respectively called `exampleLargeDifference.txt` and `exampleSmallDifference.txt`. In your report, you should provide a short explanation of why the heuristic is so effective in the first example, and why it is not effective in the second example. [20]

2. While the pattern database heuristic leads to a meaningful improvement in the performance of the computer player, there are some levels which can still not be solved with this heuristic, despite being easy to solve for humans. An example of such a level is shown in Figure 3. Describe why this level is difficult for the computer player to solve, regardless of which heuristic is used, and outline how you would adapt the computer player to make it solve such levels more efficiently. [20]

A discussion forum has been added on the learning central page for this module, which you can use for any questions you have about this coursework assignment.