Last updated on Tuesday, March 22, 2016

2016 CAB320 - Assignment One

Intelligent Search

Key information

- Code and report due at the end of **Week 09**
- Submission deadline: **Sun 08th of May, 11.57pm**
- Submit your work via Blackboard
- Group size: 1 or 2 people per submission

Overview

Sokoban is a computer puzzle game in which the player pushes boxes around a maze in order to place them in designated locations. It was originally published in 1982 for the Commodore 64 and IBM-PC and has since been implemented in numerous computer platforms and video game consoles.

The screen-shot below shows the GUI provided for the assignment. While Sokoban is just a game, it can represent a robot moving boxes in an actual warehouse and as such, it can be treated as an automated planning problem. Sokoban is an interesting problem for the field of artificial intelligence largely due to its difficulty. It has been proven NP-hard. Sokoban is difficult due to its branching factor of 4 (up, down, left, right) and the huge depth of the solutions (many pushes needed!). Additionally, a move may leave the puzzle in a state in which it is impossible to solve it, creating a state of deadlock. For example, a box in a corner cannot be moved out. If that corner is not a goal, then the problem becomes unsolvable.

As the boxes are indistinguishable, there's no difference between pushing one box or the other to a given target. **The player can only push a single box at a time and is unable to pull any box**.

The aim of *Assignment One* is to design and implement a solver for Sokoban.

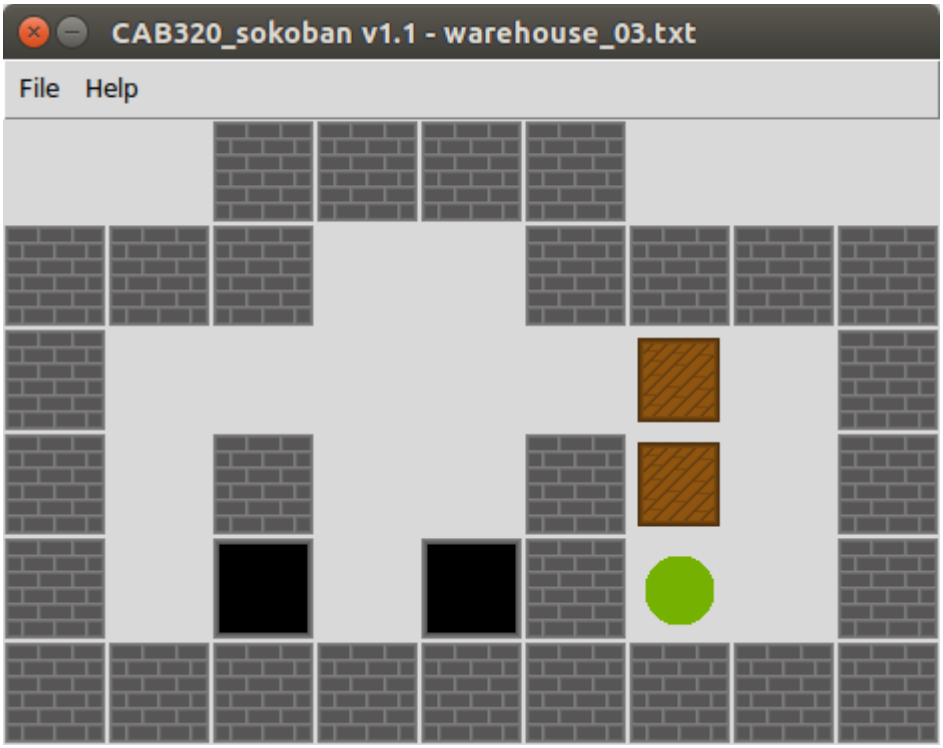

Illustration 1: Initial state of a warehouse. The green disk represents the agent/robot/player, the brown squares represent the boxes/crates. The black cells denote the target positions for the boxes.

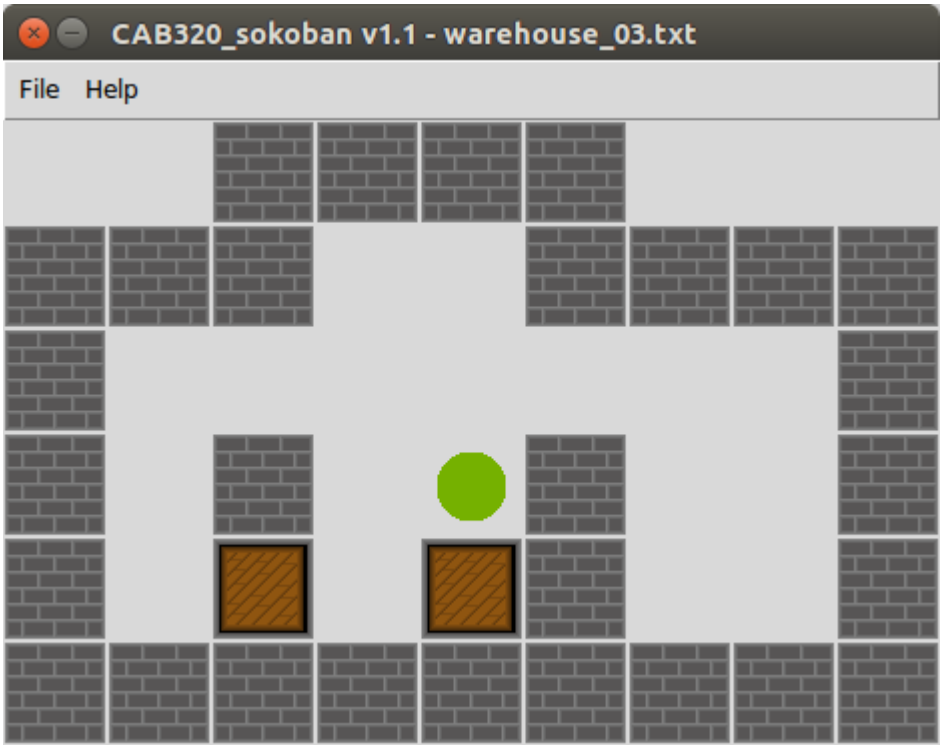

Illustration 2: Goal state reached: all the boxes have been pushed to a target position.

Puzzle representation in text files

To help you design your solver, you are provided with a large number of puzzles.

The puzzles and their initial state are coded as follows,

- **space**, a free square
- '#', a wall square
- '$', a box
- '.', a target square
- '@', the player
- '!', the player on a target square
- '*', a box on a target square

For example, the puzzle state of the first figure is code in a text file as

```
        #   #   #   #
#   #   #               #   #   #   #
#                       $           #
#           #           #   $       #
#           .       .   #   @       #
#   #   #   #   #   #   #   #   #   #
```

Suitable search algorithms

As already observed, Sokoban has a large search space with few goals located deeply in the tree. However, lower-bound heuristics can be created. These properties suggest to approach the problem with an informed search that finds sparsely distributed goals in a huge search space. A suitable generic algorithm is ***Iterative Deepening A*.*** Iterative Deepening A* (IDA*) combines iterative-deepening with the heuristic tree search A*.

Another useful trick to explore large search space is to use ***macro moves.*** In the context of Sokoban, a macro move would treat some sequences of elementary moves as a single move. An example of macro move is moving the agent from one cell to a remote cell next to a box.

Code provided

- **cab320_search.py** contains a number of search algorithms and search related classes.
- **cab320_sokoban.py** contains a class ***Warehouse*** that allows you to read a puzzle from a text file and display its content on the python console.
- **cab320_sokoban_gui.py** a GUI implementation of Sokoban that allows you to play and explore puzzles. This GUI program does not solve puzzles, it simply allows you to familiarize yourself with the game and play!
- A large number of puzzles

Your tasks

Your solution **has to comply** with the framework used in the practicals. That is, you have to use the classes and functions defined in the file **cab320_search.py**.

All your assessable code should be located in the file called **mySokobanSolver.py**

- **Task T1** Implement a **class SokobanPuzzle(Problem)** that can be used in conjunction with the search algorithms defined in ***cab320_search.py***
- **Task T2** Implement the function **checkActions()**
- **Task T3** Implement the function **tabooCells()** . The role of this function is to identify all cells that should be avoided. For example, a non-target cell in corner of a wall can become a trap for a box. The puzzle become unsolvable once a box is pushed in such a corner. Can you think of other cells that should be avoided?
- **Task T4** Create heuristics suitable for Sokoban.
  Hint: the heuristics for the sliding puzzle are relevant!
- **Task T5** Implement the function **solveSokoban_elementary()**
- **Task T6** Implement the function **solveSokoban_macro()**

Deliverables

You should submit via Blackboard a zip file containing

1. A report in pdf format **limited to 4 pages** of text (be concise!)
    - explaining concisely your heuristics, taboo cells, and important features of your solver
    - describing the performance and limitations of your solver
2. Your Python files **mySokobanSolver.py** and your own version of **cab320_search.py**

Draft Marking Scheme

- Report: 20 marks
- Code quality (readability, simplicity, structure, genericness, in line documentation): 10 marks
- T1 : 5 marks , T2 : 5 marks , T3 : 5 marks , T4 : 10 marks , T5 : 15 marks, T6 : 20 marks
- Note that if you can solve warehouse_5.txt , warehouse_9.txt and warehouse_11.txt under one second, then you are doing very well! Other puzzles can take significantly more time. For example, warehouse_59.txt requires about 20 minutes of number crunching on my laptop.

Final Remarks

- Do not underestimate the workload. Start early! You are strongly encouraged to ask questions during the practical sessions to check that you are on the right track.
- Email questions to f.maire@qut.edu.au

Have fun while learning!