Last updated on Monday, March 16, 2020

Sokoban Assignment

Intelligent Search – Motion Planning in a Warehouse

Key information

- Submission due at the end of **Week 08** (Sunday 26 April, 23.59pm)
- Submit your work via Blackboard
- Recommended group size: three people per submission.
  Smaller groups are allowed (1 or 2 people OK, but completion of the same tasks is required).

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 models a robot moving boxes in a warehouse and as such, it can be treated as an automated planning problem. Sokoban is an interesting challenge for the field of artificial intelligence largely due to its difficulty. Sokoban has been proven NP-hard. Sokoban is difficult not because of its branching factor, but because of the huge depth of the solutions. Many actions (box pushes) are needed to reach the goal state! However, given that the only available actions are moving the worker up, down, left or right, the branching factor is small (only 4!).

Unless specified otherwise, all the boxes are indistinguishable, there is no difference between pushing one box or another to a given target. **The worker can only push a single box at a time and is unable to pull any box**.

The aim of this assignment is to design and implement a planning agent for Sokoban

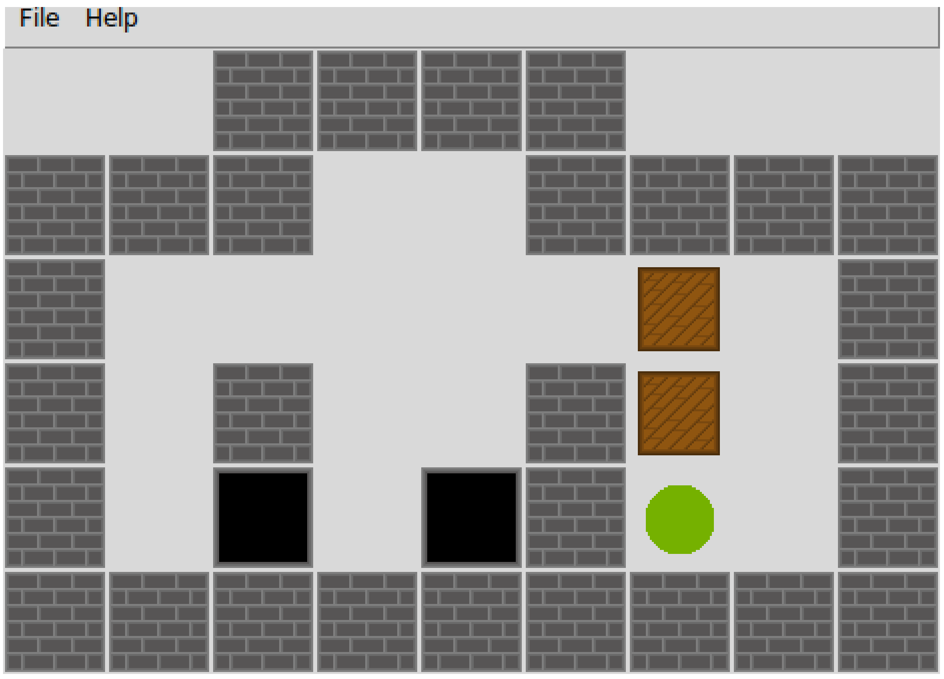

Illustration 1: Initial state of a warehouse. The green disk represents the agent/robot/player, the brown squares represent the boxes/crates. The black cells denote the target positions for the boxes.

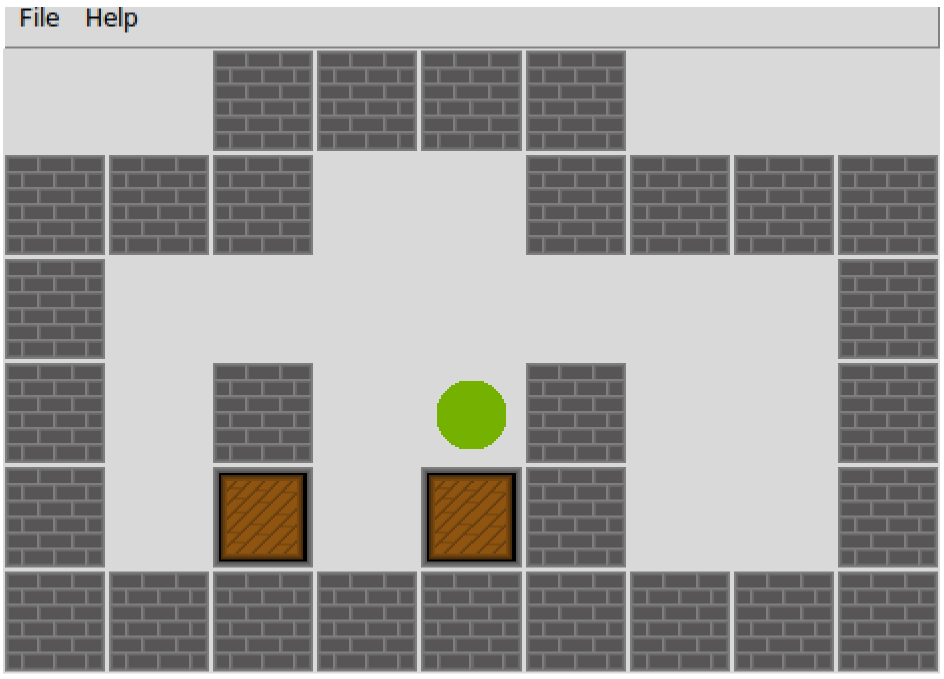

Illustration 2: Goal state reached: all the boxes have been pushed to a target position.

Approach

As already mentioned, Sokoban has a large search space with few goals, located deep in the search tree. However, admissible heuristics can be easily obtained. These properties suggest to approach the problem with an informed search. Suitable generic algorithms include A* and its variations.

After playing for a few games, you will realize that a bad move may leave the player in a doomed state from which it is impossible to recover. For example, a box pushed into a corner cannot be moved out. If that corner is not a goal, then the problem becomes unsolvable. We will call these cells that should be avoided **taboo cells**. During a search, we can ignore the actions that move a box on a taboo cell.

Another useful tool to reduce the search tree is to use ***macro moves.*** In the context of Sokoban, an ***elementary action*** is a one-step move of the worker. A ***macro action*** is the decision of a manager to have one specific box pushed to an adjacent cell. The macro action triggers itself an auxiliary problem; *can the worker go to the cell next to the specified box*. Note that the macro action can be translated into a sequence of elementary moves for the worker.

You will consider three scenarios

Scenario 1 - Elementary Actions

- In the first scenario, all actions have the same cost. The actions are elementary in the sense that an action moves the worker to an adjacent cell. Practically, you have to complete the function *solve_sokoban_elem* in the file *mySokobanSover.py*.

Scenario 2 – Macro Actions

- In the second scenario, all actions still have the same cost. The actions are macro in the sense that they focus on the motion of the boxes (not the many steps the worker has to do to reach a box). Practically, you have to complete the function *solve_sokoban_macro.*

Scenario 3 – Weighted Boxes

- In this third scenario, we assign a pushing cost to each box, whereas for the functions *solve_sokoban_elem* and *solve_sokoban_macro*, we were simply counting the number of actions executed (either elementary or macro). The actions in the third scenario are elementary in the sense that an action moves the worker to an adjacent cell. Practically, you have to complete the function *solve_weighted_sokoban_elem*.

In order to help you create an effective solver, you are asked to implement a few auxiliary functions (see the python file provided, *mySokobanSover.py*, for further details).

Puzzle representation in text files

To help you design and test your solver, you are provided with a 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 Figure 1 is code in a text file as

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

Files provided

- **search.py** contains a number of search algorithms and related classes.
- **sokoban.py** contains a class ***Warehouse*** that allows you to represent warehouses and to load puzzle instances from text files.
- **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 play!
- **mySokobanSolver.py** code skeleton for your solution. You should complete all the functions located in this file.
- **sanity_check.py** script to perform very basic tests on your solution. The marker will use a different script with different warehouses. You should develop your own tests to validate your code
- A number of puzzles in the folder 'warehouses'

Your tasks

Your solution **has to comply to** the same search framework as the one used in the practicals. That is, you have to use the classes and functions provided in the file *search.py*.

All your code should be located in a single file called *mySokobanSolver.py*. **This is the only Python file that you should submit**. In this file, you will find partially completed functions and their specifications. You can add auxiliary classes and functions to the file *mySokobanSolver.py*. When your submission is tested, it will be run in a directory containing the files *search.py* and *sokoban.py* and your file *mySokobanSolver.py*. If you break this interface, your code will fail the tests!

Deliverables

You should submit via Blackboard only two files

1. A **report** in **pdf** format **strictly limited to 4 pages in total** (be concise!)
    - One section to explain clearly your state representations, your heuristics, and any other important features needed to understand your solver.
    - Once section to describe the performance and limitations of your solvers.
    - Use tables and figures!
2. Your **Python file** *mySokobanSolver.py*

Marking Guide

- **Report**:   5 marks
    - Structure (sections, page numbers), grammar, no typos.
    - Clarity of explanations.
    - Figures and tables (use for explanations and to report performance).
- **Code quality**:   5 marks
    - Readability, meaningful variable names.
    - Proper use of Python constructs like dictionaries and list comprehension.
    - Header comments in classes and functions.
    - Function parameter documentation.
    - In-line comments.

- **Functions of mySokobanSolver.py :** 20 marks

  The markers will run python scripts to test your function.
    - **my_team():** 1 mark
    - **taboo_cells()**: 3 marks
    - **check_action_seq()**: 3 marks
    - **solve_sokoban_elem()**: 4 marks
    - **can_go_there()**: 3 marks
    - **solve_sokoban_macro()**: 3 marks
    - **solve_weighted_sokoban_elem()**: 3 marks

Marking criteria

- **Report**: 5 marks
  - Structure (sections, page numbers), grammar, no typos.
  - Clarity of explanations.
  - Figures and tables (use for explanations and to report performance).

Levels of Achievement

5 Marks	**4 Marks**	**3 Marks**	**2 Marks**	**1 Mark**
+Report written at the highest professional standard with respect to spelling, grammar, formatting, structure, and language terminology.	+Report is very-well written and understandable throughout, with only a few insignificant presentation errors. +Testing methodology and experiments are clearly presented.	+The report is generally well-written and understandable but with a few small presentation errors that make one of two points unclear. +Clear figures and tables. +Clear explanation of the heuristics used	Large parts of the report are poorly-written, making many parts difficult to understand. +Use of sections with proper section titles.	The entire report is poorly-written and/or incomplete. +**The report is in pdf format.**

To get "i Marks", the report needs to satisfy all the positive items of the columns "j Marks" for all j≤i. For example, if your report is not in pdf format, you will not be awarded more than 1 mark.

- **Code quality**: 5 marks
    - Readability, meaningful variable names.
    - Proper use of Python constructs like tuples, dictionaries and list comprehension.
    - Header comments in classes and functions.
    - Function parameter documentation.
    - In-line comments.

Levels of Achievement

5 Marks	4 Marks	3 Marks	2 Marks	1 Mark
+Code is generic and well structured. For example, auxiliary functions help increase the clarity of the code.	+Proper use of data-structures. +No unnecessary loops. +Useful in-line comments. +Header comments are clear. The new functions can be unambiguously implemented by simply looking at their header comments.	+No magic numbers (that is, all numerical constants have been assigned to variables with meaningful names). +Each function parameter documented (including type and shape of parameters)	+Header comments for all new classes and functions. +Appropriate use of auxiliary functions.	Code is partially functional but gives headaches to the markers.

To get "i Marks", the report needs to satisfy all the positive items of the columns "j Marks" for all j≤i.

Final Remarks

- Do not underestimate the workload. Start early. You are strongly encouraged to ask questions during the practical sessions.
- Email questions to f.maire@qut.edu.au
- Enjoy the assignment!