2A :: Intelligence artificielle

Lab 1: State Space Search with Sokoban

ENSICAEN - Computer Science Department

CS 2I1AE1: Artificial Intelligence

Lab 1: State-Space Search Problem with Sokoban

Duration: 1 session

In this lab, your Sokoban agent will find paths through a maze world pushing boxes around in a warehouse to get them to storage locations. You will implement *uninformed* and *informed search algorithms*.

Table of Contents

Introduction: Sokoban Transport Puzzle

Sokoban (meaning "*warehouse keeper*" in Japanese) is a type of transport puzzle. The game is played on a board of squares, where each square is a floor or a wall. Some floor squares contain boxes, and some floor squares are marked as storage locations.

The player is confined to the board, and may move horizontally or vertically onto empty squares (never through walls or boxes). The player can also move into a box, which pushes it into the square beyond. Boxes may not be pushed into other boxes or walls, and they cannot be pulled. The puzzle is solved when all boxes are at storage locations.

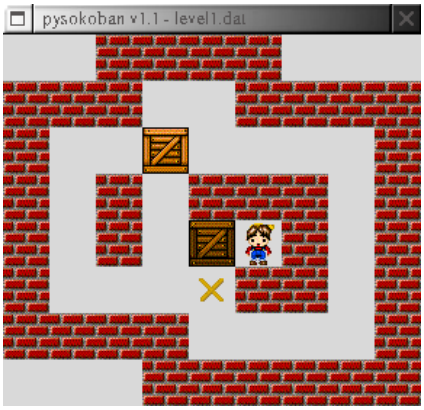

Figure 1: Example of a Sokoban puzzle.

General Instructions

1/ Download the project lab1.zip in your file system.

This archive contains all the code and supporting files for the lab. To complete the lab you only have to consider the module agents.py (you do not have to read the other files). The different agents are implemented as subclasses of the class `Agent` (cf. Design Pattern "Strategy").

The program uses TkInter which is the de-facto standard python GUI toolkit. If you work at home, you may need to install it.
Under Ubuntu do:

```
apt-get install python3-tk
```

2/ First test that the Sokoban game is working perfectly by running the following command, which allows you to play with the arrows keys:

```
python sokoban.py --grid puzzle1.txt
```

3/ Then, we provide the implementation of the Depth First Search algorithm in the `DFS` class of the module agents.py. You should observe successful behavior in all the seven puzzles. Below is given the performance of DFS for the first two puzzles:

```
./sokoban.py --agent DFS --grid puzzle1.txt
   (Number of explored nodes: 102; Number of moves: 15)

./sokoban.py -a DFS -g puzzle2.txt
   (Number of explored nodes: 2,852; Number of moves: 144)
```

Note that all the commands that appear in the lab are stored in the file `README.md`.

Also, note that the option '`--help`' displays the list of possible arguments:

```
./sokoban.py --help
```

Exercise 1: Breadth-first Search Algorithm (BFS)

Implement the Breadth-First Search algorithm in the method `search()` of the class `BFS` in the module agents.py. Base your work on the code of DFS class. If you understand the difference between DFS and BFS, you will need to change only a few lines of code!

The method prototype is:

```
def search( self, initial_state ):
```

where `initial_state` stores the initial configuration of the board.

Your function should return the list of successive directions Sokoban must follow to complete the puzzle. Directions belongs to 'left', 'right,' 'up', 'down' and are returned by the method `get_successor_states()`.

To help you program the algorithm, the class `SokobanState` provides some useful methods. Let `state` be an instance of the class `SokobanState`:

class SokobanState	
`a_state.is_goal_state()`	Returns true if the state is a valid goal state.
`a_state.get_successor_states()`	Returns all states reachable from the state as a list of triplets (state, direction, cost).

Some useful documentation on Python `list` and `set` classes:

`a_list = [1, 2]`	Creates a list with two elements.
`a_list.append(x) / a_list.insert(0, x)`	Inserts the element *x* at the end of the list / at the beginning of the list.
`a_set = set([x])`	Creates a new set with the element *x*.
`a_set.add(x)`	Adds the element *x* in the set.

Hint: Don't forget to detect already visited states in a closed list to speed up the search.

The performances for the first 4 puzzles should be:

```
./sokoban.py -a BFS -g puzzle1.txt
   (Number of explored nodes: 2,162; Number of moves: 13)

./sokoban.py -a BFS -g puzzle2.txt
   (Number of explored nodes: 6,422; Number of moves: 68)

./sokoban.py -a BFS -g puzzle3.txt
   (Number of explored nodes: 29,597; Number of moves: 72)

./sokoban.py -a BFS -g puzzle4.txt
   (Number of explored nodes: 10,022; Number of moves: 37)
```

Don't try to solve puzzle8 or puzzle9 with this algorithm

Exercise 2: Greedy Best-First Search Algorithm (GBFS)

Implement the Greedy Best-First Search algorithm in the `GBFS` class. Base your work on the UCS implementation which is provided in the same file. You may also use methods of the class `SokobanState`.

class SokobanState	
`a_state.is_goal_state()`	Returns true if the state is a valid goal state.
`a_state.get_successor_states()`	Returns all states reachable from the state as a list of triplets (state, direction, cost)
`a_state.heuristic()`	Returns the heuristic value for the state. The heuristic function is the sum of the Manhattan distance of each box to the nearest storage location.

Some useful documentation on `PriorityQueue`:

`a_queue = PriorityQueue()`	Creates an empty priority queue.
`a_queue.isEmpty()`	Tests if the queue is empty.
`x=a_queue.pop()`	Extracts the first element of the queue (the one with the highest priority).
`a_queue.push(x, cost)`	Adds the element *x* in the list with the priority *cost*.

Your code should find a solution for the first six puzzles with the following performances:

```
./sokoban.py -a GBFS -g puzzle1.txt
   (Number of explored nodes: 152; Number of moves: 15)

./sokoban.py -a GBFS -g puzzle2.txt
   (Number of explored nodes: 1,732; Number of moves: 78)

./sokoban.py -a GBFS -g puzzle3.txt
   (Number of explored nodes: 16,472; Number of moves: 150)

./sokoban.py -a GBFS -g puzzle4.txt
   (Number of explored nodes: 5,422; Number of moves: 53)

./sokoban.py -a GBFS -g puzzle5.txt
   (Number of explored nodes: 22,277; Number of moves: 87)

./sokoban.py -a GBFS -g puzzle6.txt
   (Number of explored nodes: 497; Number of moves: 31)
```

Exercise 3: A* Search Algorithm (ASS)

Implement A* search in the method `search` of the class `AstarS`. Base your work on the implementation of `GBFS`.

```
./sokoban.py -a ASS -g puzzle1.txt
   (Number of moves explored nodes: 792; Number of moves: 13)

./sokoban.py -a ASS -g puzzle2.txt
   (Number of moves explored nodes: 6122; Number of moves: 68)

./sokoban.py -a ASS -g puzzle3.txt
   (Number of moves explored nodes: 28,907; Number of moves: 72)

./sokoban.py -a ASS -g puzzle4.txt
   (Number of moves explored nodes: 8,672; Number of moves: 37)

./sokoban.py -a ASS -g puzzle5.txt
   (Number of moves explored nodes: 36,357 Number of moves: 73)

./sokoban.py -a ASS -g puzzle6.txt
   (Number of moves explored nodes: 902; Number of moves: 31)
```

Exercise 4: Iterative Deepening Search

Implement the Iterative Depth-First Search algorithms (IDS). If needed, assume that the solution is no longer than 500 moves.

Exercise 5: IDA*

Implement the IDA* algorithm. If needed, assume that the solution is no longer than 500 moves.

Because our heuristic is far from the real value, the IDA algorithm is not very efficient. So, do not use this algorithm with the puzzles #8, #9 and #10 (see exercise 6):

```
./sokoban.py -a IDASS -g puzzle1.txt
   (Number of explored nodes: 2,783; Number of moves: 13)

./sokoban.py -a IDASS -g puzzle2.txt
   (Number of explored nodes: 194,588; Number of moves moves: 68)

./sokoban.py -a IDASS -g puzzle3.txt
   (Number of explored nodes: 772,562; Number of moves moves: 72)

./sokoban.py -a IDASS -g puzzle4.txt
   (Number of explored nodes: 88,935; Number of moves moves: 37)

./sokoban.py -a IDASS -g puzzle5.txt
   (Number of explored nodes: 998,238; Number of moves moves: 73)

./sokoban.py -a IDASS -g puzzle6.txt
   (Number of explored nodes: 14,959; Number of moves moves: 31)
```

Exercise 6: Enhancement

To reduce drastically the size of the search tree, one idea is to consider only box moves and not Sokoban's moves. At each turn, the possible new states are those with boxes Sokoban can move.

Write a new version of the Sokoban game with the previous idea. You should change the module `sokobanframe.py` and rewrite the various search algorithms with the new states.

Try to solve puzzle8 and puzzle9 now.

Last modified: Thursday, 14 November 2024, 10:58 AM

◄ Project: Rational Agent in Wumpus World

Jump to...

Lab3_add ►