Foundations of Artificial Intelligence

M. Helmert
S. Sievers
Spring Term 2023

University of Basel
Computer Science

Exercise Sheet 2

Due: March 12, 2023

Important: for submission, consult the rules at the end of the exercise. Non-adherence to the rules will lead to your submission not being corrected.

Exercise 2.1 (1+1 marks)

Characterize the following environments by describing if they are *static / dynamic*, *deterministic / non-deterministic / stochastic*, *fully / partially / not observable*, *discrete / continuous*, and *single-agent / multi-agent*. Explain your answers.

(a) Risk (see, e.g., `https://en.wikipedia.org/wiki/Risk_(game)`)

(b) moving a robotic arm

Exercise 2.2 (0.5 + 0.5 marks)

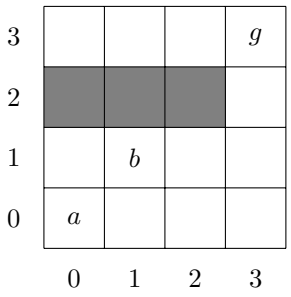
Determine if the following statements about state spaces $\mathcal{S} = \langle S, A, cost, T, s_0, S_\star \rangle$ are correct or not. Explain your answer.

(a) If all actions have equal costs, each solution for $\mathcal{S}$ is optimal.

(b) There is no solution for $\mathcal{S}$ if $T = \emptyset$.

Exercise 2.3 (2 marks)

In the *Sokoban* problem, there is an agent who can move in a grid of which a subset of cells is blocked by walls. In the general form of the problem (see, e.g., `https://en.wikipedia.org/wiki/Sokoban`), there is a set of boxes, located at their initial cells in the grid, that must be pushed to goal cells in the grid. The agent can freely move to any cell not occupied by a wall or a box. The agent can only *push* boxes: if they stand next to a box and the cell behind the box from the point of view of the agent is unoccupied, they can push the box to that free cell, which also moves the agent to the cell the box was previously. The objective is to get the boxes to the goals with as few pushes as possible. The number of moves of the agent do not matter.
Consider the following simplified example Sokoban problem instance with a grid of size 4×4, a single box starting at location b which must be pushed to the goal location g, and the agent starting at location a. Gray cells indicate walls while white cells can be entered.



Formalize the state space of the above Sokoban problem instance. Specify all parts of the state space, i.e., the set of states, the set of actions, the cost function, the set of transitions, the initial state and the set of goal states.

Exercise 2.4 (3.5+1.5 marks)

The task in this exercise is to write a software program. We expect you to implement your code on your own, without using existing code (such as examples you find online) except for what is provided by us. If you encounter technical problems or have difficulties understanding the task, please let us – the tutors or assistant – know *sufficiently ahead of the due date*.

On the website of the course, you find Java code (`state-spaces.zip`) that implements the black box interface for state spaces that was discussed in the lecture, as well as an interface for states and actions. The code includes an example implementation of the interface for the blocks world problem. You can test the implementation by invoking the `StateSpaceTest` class, which creates a set of random successor states starting from the initial state and afterwards performs a so-called random walk, i.e., it iteratively picks a successor at random and expands it.

To run the program, first compile it with

```
javac StateSpaceTest.java
```

from a shell (on Linux) and then run it with

```
java StateSpaceTest blocks blocks-problem.txt
```

The sole purpose of the provided blocks world implementation is to serve as an example that helps you for your own implementation.

Your task is to implement the provided interface for the *Sokoban* problem, where there is an agent who can move in a grid of which a subset of cells is blocked by walls. In the general form of the problem (see, e.g., `https://en.wikipedia.org/wiki/Sokoban`), there is a set of boxes, located at their initial cells in the grid, that must be pushed to goal cells in the grid. The agent can freely move to any cell not occupied by a wall or a box. The agent can only *push* boxes: if they stand next to a box and the cell behind the box from the point of view of the agent is unoccupied, they can push the box to that free cell, which also moves the agent to the cell the box was previously. The objective is to get the boxes to the goals with as few pushes as possible. The number of moves of the agent do not matter. Unlike in the general version, where any assignment of boxes to goal positions is a valid goal state, each box has a dedicated goal position in our simplified variant.

Everything should be implemented in a file called `SokobanStateSpace.java`. **In your handin, only submit your `SokobanStateSpace.java` as a solution to this exercise.**

(a) Implement the state space of the Sokoban problem (for variable grid size, number of boxes, positions of walls, initial position of the agent and initial and goal positions for each box).

(b) Implement the `buildFromCmdline` function for the Sokoban problem such that it parses an input file with the following format:

- three space separated numbers in the first line indicate width w and height h of the grid (8×7 in the sample instance in the file `sokoban-problem.txt`) and the number of boxes n (7 in the example).
- the following h lines with w zeroes or ones per line indicate if the respective grid cell can be entered by the agent (1) or if there is a wall (0).
- the next line gives the initial grid cell of the agent.
- the final n lines indicate the initial grid cell of each box, followed by their respective goal grid cell.

The file `sokoban-problem.txt` from the archive encodes an example instance. Make sure that your code recognizes invalid inputs such as invalid numbers in the grid description, invalid initial states where the agent or boxes are on blocked cells, or invalid goal specifictions.

You can test your implementation with the command

```
java StateSpaceTest sokoban sokoban-problem.txt
```

Notice that the command uses the new keyword `sokoban` (rather than `blocks` as in the example above). Also note that the problem is not expected to be solved by the provided random walk.

Submission rules:

- Exercise sheets must be submitted in groups of two students. Please submit a single copy of the exercises per group (only one member of the group does the submission).
- Create a single PDF file (ending .pdf) for all non-programming exercises. Use a file name that does not contain any spaces or special characters other than the underscore "_". If you want to submit handwritten solutions, include their scans in the single PDF. Make sure it is in a reasonable resolution so that it is readable, but ensure at the same time that the PDF size is not astronomically large. Put the names of all group members on top of the first page. Either use page numbers on all pages or put your names on each page. Make sure your PDF has size A4 (fits the page size if printed on A4).
- For programming exercises, only create those code textfiles required by the exercise. Put your names in a comment on top of each file. Make sure your code compiles and test it. Code that does not compile or which we cannot successfully execute will not be graded.
- For the submission: if the exercise sheet does not include programming exercises, simply upload the single PDF. If the exercise sheet includes programming exercises, upload a ZIP file (ending .zip, .tar.gz or .tgz; *not* .rar or anything else) containing the single PDF and the code textfile(s) and nothing else. Do not use directories within the ZIP, i.e., zip the files directly.
- Do not upload several versions to ADAM, i.e., if you need to resubmit, use the same file name again so that the previous submission is overwritten.