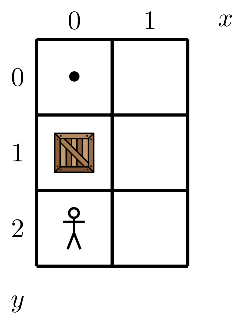
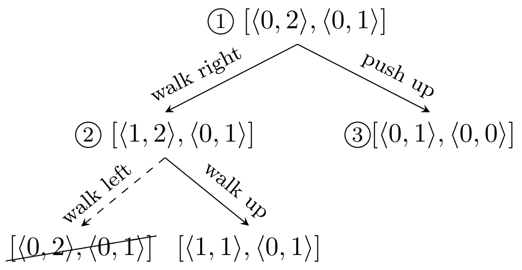
Sokoban

In the next exercise, we will consider the puzzle game *Sokoban*[1]. In this single-player game, we need to move a set of crates (▨) to certain goal squares (marked by •). To accomplish this, the character we control (웃) may move around by *walking* to the left, right, up or down one square at a time if it is vacant and inside the play area. We cannot move diagonally. Additionally, we may *push* a crate one square to the left, right, up or down if we stand directly next to it on the side opposite of the push direction and there is a vacant space in the push direction next to the crate. We cannot pull a crate or drag it sideways. When pushing a crate, we also move one step into the push direction ourselves. The level is completed once every crate is on a goal square. Which crate ends up at which goal square is irrelevant.



(a) Example Sokoban level.

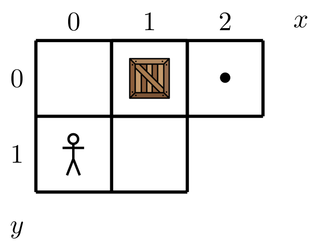


(b) The corresponding breadth-first search tree.

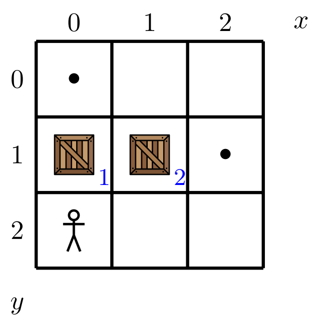
Figure 1: Sokoban example.

Example 1. *Figure 1a depicts an example level. Each square inside the play area of the level has an (x, y) coordinate. We start the level at square $\langle 0, 2\rangle$ with one crate at location $\langle 0, 1\rangle$. The crate has to be moved to the goal square $\langle 0, 0\rangle$. To avoid spending too much time drawing the search nodes of the search trees, you may (but do not have to) represent a game state of this level as a tuple of coordinates $[\langle x_0, y_0\rangle, \langle x_1, y_1\rangle, \dots, \langle x_n, y_n\rangle]$, where $\langle x_0, y_0\rangle$ is the square inhabited by our character and $\langle x_1, y_1\rangle$ to $\langle x_n, y_n\rangle$ are the squares inhabited by the crates numbered from 1 to n. The initial state s_0 for our example would be represented by $s_0 = [\langle 0, 2\rangle, \langle 0, 1\rangle]$.*

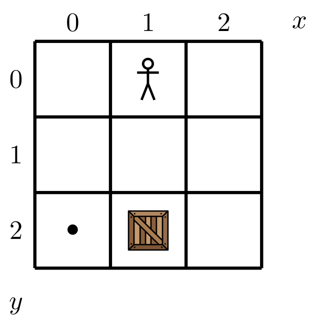
Figure 1b shows the application of breadth-first search with duplicate checking on this example, where successors were generated (i.e. added to the search tree) following the move order: walk/push right, walk/push down, walk/push left, walk/push up. Note that we can never push and walk in the same direction at the same time. The number in circles denotes the order of expansion, where a node counts as expanded when its successors are generated, or it was identified as the goal state. Duplicate states are technically not generated and are crossed out in this depiction. The search terminates after the goal state $[\langle 0, 1\rangle, \langle 0, 0\rangle]$ is expanded (successors for the goal are not generated!). The found solution is pushing the crate up.



(a) Level for exercise 1 (a).



(b) Level for exercise 1 (b).



(c) Level for exercise 1 (c).

Figure 2: Sokoban levels for exercise 1. Crate numbering given by the numbers in blue.

[1] https://www.mathsisfun.com/games/sokoban.html

1 Blind Search (10 Points)

In the following, you will use both breadth-first search and depth-first search to solve Sokoban puzzles. In all of these following tasks, successors must be generated according to the action order: walk/push right, walk/push down, walk/push left, walk/push up (ignoring inapplicable moves). Of course, successors should also be ordered accordingly from left to right in the search tree.

(a) Consider the Sokoban level shown in Figure 2a. The initial state is given by $s_0 = [\langle 0, 1\rangle, \langle 1, 0\rangle]$.

Draw the search tree generated by *breadth-first search without duplicate checking*. Draw directed edges between search nodes and their successors and label them with the move that generated them (see Figure 1b). Similarly, annotate nodes by their expansion number. Draw and mark nodes representing duplicate states by crossing them out, but do not consider them for expansion. Stop when a state is selected for expansion in which all crates have arrived at their intended destination and the level is completed. How can we complete the level according to the search? Write down the sequence of actions (e.g. move left $\rightarrow$ push right $\rightarrow \dots$). (3 Points)

(b) Consider the Sokoban level shown in Figure 2b. The initial state is given by $s_0 = [\langle 0, 2\rangle, \langle 0, 1\rangle, \langle 1, 1\rangle]$.

Draw the search tree generated by *breadth-first search with duplicate checking*. Proceed as in exercise (a), but cross out nodes that are already part of the search tree and do not expand them further (see state $[\langle 0, 2\rangle, \langle 0, 1\rangle]$ in Figure 1b). How can we complete the level according to the search? Write down the sequence of actions. (3.5 Points)

(c) Consider the Sokoban level shown in Figure 2c. The initial state is given by $s_0 = [\langle 1, 0\rangle, \langle 1, 2\rangle]$.

Draw the search tree generated by *depth-first search*. *Only generate one successor at a time and continue the search from this successor. Generate the next successor only in case search backtracks (see lecture slides).* Draw edges between search nodes and those successors that depth-first search has generated and label them with the move that generated them. Annotate nodes by their expansion number. Stop when a state is selected for expansion in which all crates have arrived at their intended destination and the level is completed. How can we complete the level according to the search? (3 Points)

(d) Depth-first search without duplicate checking in general does not guarantee to actually find a solution. What about Sokoban puzzles in particular? Briefly explain your answer. (0.5 Points)

2 Jupyter Notebook: Backpropagation (10 Points)

In this exercise, you will implement backpropagation with Numpy. Please see **assignment05.ipynb** for detailed instructions.

Submission Details:

Upload your submission to our CMS until *December 21, 2022 at 23:59*. Late submissions will not be graded! The submission should be uploaded by exactly **one** team member. Make sure that your submission contains the name and matriculation number of each team member. Submit your solution **a single *pdf* file** with your answers to exercises 1 and the Jupyter Notebook (**just the *ipynb* file**) containing your solution to exercise 2.