Assignment 1 on Search

Due Oct 10 by 12p.m. **Points** 100

Table of Contents

Warning

We know that solutions to this or related problems may exist online. *Do not use these solutions, as this would be plagiarism.* To earn marks on this assignment, you must develop your own solutions.

Also, please consider the following important points:

- *Do NOT add any non-standard imports*. All imports already in the starter code must remain. When in doubt, post a question on Piazza.
- *Do NOT modify any provided code*. Modifying the provided code may cause you to *fail* all the tests.
- Submit your program on MarkUs and run the provided tests *well before* the deadline. The fact that your code runs on your own system but not on MarkUs is *NOT* a legitimate reason for a regrade request.
- The provided tests on MarkUs are an *extremely basic* sanity check. Passing the provided tests only means that your program runs without errors; it does *NOT* mean that you will receive full marks on the assignment.

The Sokoban Puzzle

The goal of this assignment will be to implement a program to solve Sokoban puzzles, as shown in Figure 1. Sokoban is a puzzle game where a robot aims to push multiple boxes into storage spaces in a warehouse, following some rules.

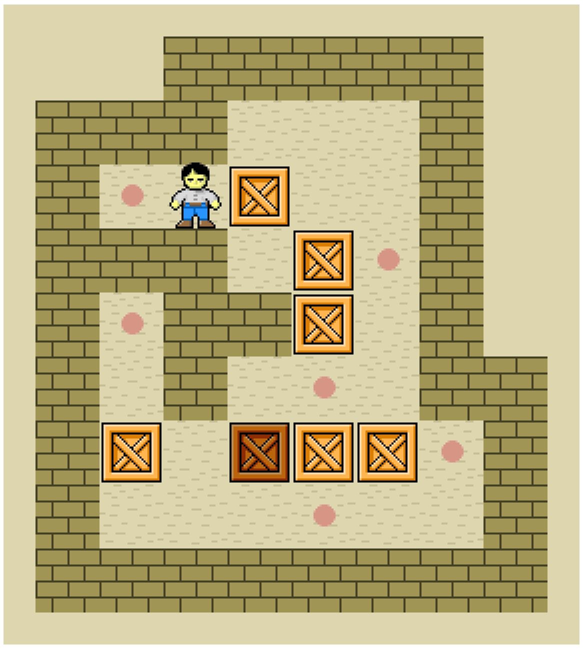

Figure 1: An animated example of a Sokoban puzzle

Sokoban can be played online at https://www.sokobanonline.com/play. We recommend that you familiarize yourself with the game before proceeding. We will give a formal description of the puzzle in the next section.

The rules of Sokoban are described below. In our version of Sokoban, there may be more than one robot in each puzzle.

- The puzzle is on a grid with N columns and M rows.
- The puzzle may have multiple robots, boxes, storage locations, and obstacles (walls).
    - The number of boxes is equal to the number of storage locations.
- Each robot can move up, down, left, or right into an empty space.
- A robot can push a box into an empty space. That is, if a robot moves to the location of a box, the box will move one square in the same direction.
    - The boxes can only be pushed, not pulled, by robots.
    - Only one robot can move at a time.
    - Only one box can be moved at a time.
    - A box can only be moved one unit at a time.
    - A robot cannot push more than one box at a time. If two boxes are next to each other, a robot cannot move one box toward the other.
    - Neither robots nor boxes can pass through other robots, other boxes, or the walls.
- Each movement has a cost of one, regardless of whether a robot is pushing a box or not.
- The goal of the robots is to move all the boxes such that every box is in a storage location.

Your Tasks

You will implement two search algorithms to solve Sokoban puzzles: Depth-first search and A* search.

You will also implement two heuristic functions for A* search: the Manhattan distance heuristic function and an advanced heuristic function of your invention.

The Manhattan Distance Heuristic Function

Manhattan Distance: The name "Manhattan distance" comes from how people navigate Manhattan in New York City. As you may know, Manhattan is a grid. To get from one place to another in Manhattan, you must travel horizontally or vertically through the grid. The Manhattan distance between two locations is the sum of the horizontal and vertical distances between the two locations.

We can derive the Manhattan Distance heuristic function as follows. First, we relax the Sokoban puzzle by allowing all the items (e.g. robots, boxes, and obstacles) to overlap. Then, we can solve the relaxed puzzle optimally by moving each box over other items to the closest storage location. The cost of this optimal solution would be the sum of the Manhattan distances between each box and its closest storage location.

An Advanced Heuristic Function

You will design and implement one advanced heuristic function, satisfying the two requirements below.

1. Your advanced heuristic function must dominate the Manhattan distance heuristic function. This ensures that A* search with the advanced heuristic function is more efficient than A* search with the Manhattan distance heuristic function.
2. Your advanced heuristic function must be admissible. This ensures that A* search with the advanced heuristic function finds the optimal solution.

Starter Code

To support you in completing this assignment, we have provided two files to get you started: *board.py* and *solve_starter.py*. Below, we provide a brief explanation of the important elements. The files contain detailed documentation. Please read through the documentation carefully before getting started.

The focus of this assignment is to practice implementing search algorithms. Therefore, it is not essential that you write code to implement the Sokoban puzzle. For this reason, we have provided code that implements the Sokoban puzzle in *board.py*.

- The *Board* class keeps track of important information on the board, such as the dimensions of the board and the locations of robots, boxes, storage, and obstacles.
- The *State* class is a wrapper for the Board class. It stores additional information such as the f value, state depth, and parent state.
- The zero heuristic function returns zero for every state. You can use this function to test A* search before implementing other heuristic functions.

- The *read_from_file* function reads the contents of the given file, creates a Board using the file contents, and returns the Board. See the *Input File Format* section below for a description of the input file format.
- The *__str__* function in the Board class converts the board to a readable string representation. You can use this function to write a board to the console or a file.

Moreover, we have also provided code to run your program with the provided command line parameters in *solve_starter.py*.

- *solve_puzzle(board, algo, hfn)* takes a board, a search algorithm, and a heuristic function and solves the puzzle using the specified algorithm.
- *__main__* is the main function that reads in command line parameters and runs the functions to solve a Sokoban puzzle.

The rest of the *solve.py* file contains several empty functions you must implement. The two key functions below correspond to the two search algorithms.

- *dfs(init_board, hfn)* is the depth-first search implementation.
- *a_star(init_board, hfn)* is the A* search implementation.

In addition, you must implement several other helper functions described below. In theory, these helper functions are not strictly necessary. You could put all the code in the *dfs* and *a_star* functions. However, we created these helper functions for several reasons.

1. Breaking down a large program into a set of smaller functions is a *good practice* in software development. Doing this makes your program easier to read and understand.
2. The helper functions provide a *roadmap* to complete this assignment *incrementally in small steps*. For instance, you can start by implementing *is_goal, get_path,* and *get_successors*. Then, your *dfs* and *a_star* implementations can use the *get_successors* function.
3. Also, you can *test the helper functions individually using unit tests* before testing the complete search algorithm implementations. Testing the smaller functions is easier and allows you to catch and fix bugs quickly.
4. Finally, breaking down the program into multiple functions allows us to evaluate the functions in your program using unit tests. This way, even if you cannot complete the entire program by the deadline, you can still receive marks for any functions (including the helper functions) that you have implemented correctly.

The helper functions that you must implement are described below. Please *do NOT change the provided function signatures*. Changing them may cause you to *fail all the tests.*

- *is_goal(state)* is a helper function determining whether a state is a goal.
- *get_path(state)* is a helper function that returns a path from the initial state to the given state. You can use this to generate the solution after finding a goal state.
- *get_successors(state)* is a helper function that returns a list of successor states of the given state.

- *heuristic_basic(board)* is the Manhattan distance heuristic function.
- *heuristic_advanced(board)* is your advanced heuristic function.

Input File Format

An example of an input file is given below. This is also the input file format expected by the read_from_file function.

```
sokobanonline_1_1
8
8
  ###
  #.#
  # ####
###? ?.#
#. ?a###
####?#
   #.#
   ###
```

We will explain the input file format in detail below.

- The first line is the name of the puzzle. This name can be arbitrary.
- The second line is the width of the puzzle (i.e. the number of columns).
- The third line is the height of the puzzle (i.e. the number of rows).
- The next few lines encode the actual puzzle.

The characters we use to encode different elements of the puzzle are as follows.

- **#** (the hashtag) denotes a wall or obstacle.
- Characters for empty storage points:
  - **.** (the dot) denotes a storage point that is NOT covered by a box or a robot.
- Characters for boxes:
  - **?** (the question mark) denotes a box that is NOT at a storage point.
  - ***** (the asterisk) denotes a box that is at a storage point.
- Characters for robots:
  - Any lowercase letter (e.g. a, b, c) denotes a robot that is NOT at a storage point.
  - Any uppercase letter (e.g. A, B, C) denotes a robot that is at a storage point.

Running Your Program

Once you have implemented and tested the individual functions, you can run your program to solve a Sokoban puzzle.

For example, if you would like to solve a Sokoban puzzle in *sokoban1.txt* using *A* search* with *the Manhattan heuristic function* and save the solution in *solution1.txt*, you would run the following command in a terminal.

```
python3 solve.py --algorithm a_star --heuristic basic --inputfile sokoban1.txt --outputfile solution
1.txt
```

Similarly, if you would like to solve a Sokoban puzzle in *sokoban2.txt* using *depth-first search* and save the solution in *solution2.txt*, you would run the following command.

```
python3 solve.py --algorithm dfs --inputfile sokoban2.txt --outputfile solution2.txt
```

A summary of the command line parameters is provided below.

Command Line Parameters

Parameter	Type	Required?	Description
`--inputfile`	str	required	The file that contains the puzzle.
`--outputfile`	str	required	The file that contains the solution to the puzzle.
`--algorithm`	str	required	The search algorithm. The choices are dfs and a_star.
`--heuristic`	str	optional	The heuristic function used by any heuristic search algorithm. The choices are zero, basic, and advanced.

The Optional Early Feedback Deadlines

To encourage you to start working on the assignments early and tackle them incrementally, we have introduced *two optional early feedback deadlines* for each assignment. This assignment is available for three weeks. We will provide some extra unit tests during the first two weeks. If you make a submission during either week, you can get additional feedback from these extra unit tests. Note that these extra unit tests will be *exclusively* available during their respective weeks and will no longer be available in the third week. These optional early feedback deadlines are not worth any marks.

We created these optional early feedback deadlines for the following reasons.

- *Encourage you to start the assignments early*: If you start early, you can take advantage of these extra unit tests, which won't be available later on.
- *Help you tackle the assignment incrementally*: We break down the assignment into smaller tasks so that the task for each week is more manageable.
- *Provide extra support*: The extra unit tests make it easier to debug and test your program by identifying problems in your program.
- *Avoid adding extra stress*: The early deadlines are not worth any marks. Therefore, if you are busy with other obligations and cannot take advantage of the early deadlines, you are not missing out on any marks, and you can still potentially get full marks on the assignments.

Submission, Test Availability and Grading Scheme

We have set up three projects on MarkUs: A1-Week1, A1-Week2 and A1. The first two projects (A1-Week1 and A1-Week2) contain the extra unit tests for the optional early deadlines. They are not worth any marks.

To earn marks for this assignment, you must *complete solve_starter.py*, *change its name to solve.py* and *submit it to the A1 project on MarkUs*.

The table below contains detailed information regarding the available tests and the grading scheme. We have also included a few important notes below the table. Please read them carefully!

MarkUs Test Availability and Grading Scheme

Category	**Test Descriptions**	**A1 - Week 1** **Sep 19 - 26**	**A1 - Week 2** **Sep 26 - Oct 3**	**A1** **Sep 19 - Oct 10**	**Marks**
Helper Functions	is_goal	2 public tests		2 public tests + 3 hidden tests = 5 tests	5
	get_path	1 public test		1 public test + 4 hidden tests = 5 tests	5
	get_successors	1 public test		1 public test + 9 hidden tests = 10 tests	10
DFS	dfs solution is valid		1 public test	1 public test + 4 hidden tests = 5 tests	25
A* + basic heuristic	heuristic_basic is correct		1 public test	1 public test + 4 hidden tests = 5 tests	10
	a_star + heuristic_basic solution is valid		1 public test	1 public test + 4 hidden tests = 5 tests	10
	a_star + heuristic_basic solution is optimal		1 public test	1 public test + 4 hidden tests = 5 tests	10
A* + advanced heuristic	heuristic_advanced is admissible			1 public test + 4 hidden tests = 5 tests	5
	heuristic_advanced dominates heuristic_basic			1 public test + 4 hidden tests = 5 tests	5

Category	Test Descriptions	A1 - Week 1 Sep 19 - 26	A1 - Week 2 Sep 26 - Oct 3	A1 Sep 19 - Oct 10	Marks
	a_star + heuristic_advanced solution is valid			1 public test + 4 hidden tests = 5 tests	5
	a_star + heuristic_advanced solution is optimal			1 public test + 4 hidden tests = 5 tests	5
	a_star + heuristic_advanced is more efficient than a_star + heuristic_basic			5 hidden tests = 5 tests	5

Let us explain a few important things below.

(1) The public tests available during week 1 or week 2 are different from those available in the final assignment on MarkUs.

(2) A solution path is valid if it satisfies the conditions below:

- The first state is the initial state.
- The last state is a goal state.
- Every state is a successor of the previous state.

(3) Some unit tests will run a search algorithm (DFS or A*) to solve a Sokoban puzzle. Whenever this happens, your program must *terminate in under 3 minutes to earn marks*.

To help you gauge the efficiency of your program, we have provided one puzzle with moderate difficulty below. Our A* search implementation (with the basic Manhattan distance heuristic function) can solve the puzzle below in about 1 minute (or 60 seconds).

```
sokobaninfo_nabo_cosmos_3
8
9
######
#    #
# ##?###
# . *  #
##  * A#
 #  ?  #
 # #*###
 #   #
 #####
```