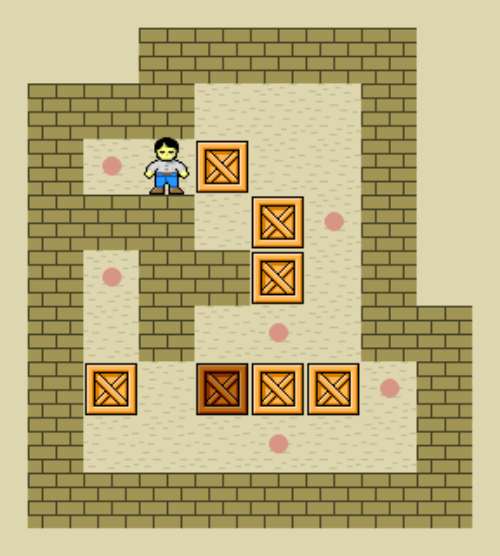
ECE 473 Project 2

Final project deadline: Friday, April 1, 12:00 Noon (no late penalty until Saturday noon hard deadline)

Team selection deadline: Thursday, March 10, 12:00 Noon (no late penalty until Friday noon hard deadline)

Introduction

In this homework, we ask you to extend our provided code for automated Sokoban puzzle solution using state space search. The code we provide formulates a simple state space and can solve some Sokoban problems we provide using uniform-cost search, also provided. In addition, we provide the code for A* search, but we do not provide any heuristic function.



In this homework we are primarily interested in finding puzzle solutions, without regard to the length/cost of the solution. Many Sokoban puzzles will be completely unsolvable by the code we distribute (and the solution code and even any known code) and we are interested in that issue, not in optimizing the number of steps to solve.

The homework involves both coding and design of puzzle levels to illustrate features of the code. Your submission will be in two files, `sokoban.py` and `design.txt`, containing a modified and extended version of the code we provided, and a set of puzzle levels you invented, respectively.

This assignment will be graded on the project grading criteria specified in the syllabus. As such, grade effects come to you either from failing the minimum standard specified below, or placing in the top 20% of those teams passing that standard in our competition. In each case the grade effect is to move your final grade across a single mini-grade-boundary (includes +/- boundaries). See the final section of this assignment for a few more details.

Note that a significant part of this assignment is developing a complete understanding of the code we are providing; the code is not written to be instructional but to be functional, and reverse engineering an understanding of this sort of code is an important computer-engineering skill. You are free to modify any part of it in direct service of the assignments below, but creative modifications, especially with purposes outside the suggestions below, should be checked with the TA before inclusion in your solution.

Please note that you will be evaluated on levels that are not provided and will not be checked by any grader we provide you. It is not sufficient to perform well only on the levels provided to you now or with any grader provided later.

Team selection

Note: We have added a new line to the team selection submission described here. The first line must be your team's code name for any leaderboard we may manage to offer.

You must select your team by noon Friday March 4. Anyone not submitting a team selection by that time will be expected to work alone. Please submit a text file containing your team's code name for the leaderboard on the first line, followed by your team members' Purdue email handles (omit *@purdue.edu*), one per line, with the team leader on the first line. All team members must submit the same team with the same code name and team leader or the team will be rejected. For example, our course leadership team (omitting UTAs for brevity here) would be described, as follows:

```
miracle-workers
givan
gong123
jkim17
```

We may process these submissions with a script; please follow the prescribed format exactly.

Instructions on the code

Some sample Sokoban maps are provided to you in `levels.txt`. You can refer to the following table to interpret the symbols for each map tile.

symbol	meaning
`#`	wall
`@`	player
`+`	player on target
`$`	box
`*`	box on target
`.`	target
`(space)`	floor

To run the code, you will need to provide certain command line arguments. You can type

```
python3 sokoban.py -h
```

to see a full list of available arguments. The `algorithm` argument can take any of the values in the following table. You will learn more about what each algorithm does as you proceed further in this homework.

`algorithm`	description
`ucs`	basic UCS
`a`	uncompressed actions with A* using basic heuristic
`a2`	uncompressed actions with A* using `heuristic2`
`f`	compressed actions with UCS
`fa`	compressed actions with A* using basic heuristic
`fa2`	compressed actions with A* using `heuristic2`

For example, you can type

```
python3 sokoban.py 1 ucs
```

to run the solver on level 1 in `levels.txt` using basic UCS.

Problems

Note: For the following problems, your implementation of the described features should demonstrate improvement in run time in some levels of `levels.txt`.

Note: You are free to add helper functions or make any changes anywhere in the code, but you **should not** modify the function signature of any provided functions except to add arguments with default values. You **should not** modify the command line interface except to add tagged arguments with default values.

Problem 1: Dead end detection

The Sokoban domain contains many states where there is no path to a goal. For instance, if a box is pushed to a wall corner that is not a target location, then there is no way to move the box out and hence it is a dead end. For problem one you must modify the code in `sokoban.py` to detect enough dead end states to pass the grader performance standard.

If the `expand` function refuses to produce any descendant for the dead end states, we can greatly reduce the search space. The dead end detection is not meant to be exhaustive; being able to detect every dead end state is not necessary. In fact, this is both very difficult to achieve and very expensive to compute so it may actually hurt us. You should try to detect as many dead end states as possible while keeping the detection algorithm cheap.

You can type

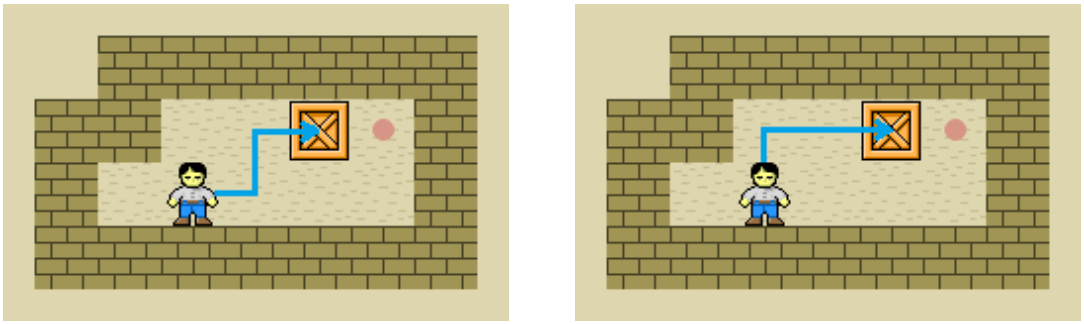
```
python3 sokoban.py 1 ucs -d
```

to invoke dead end detection.

Hint: Precomputing certain quantities could be very helpful: you should probably not be doing some kinds of dead-end detection over and over on each state, but once for the entire problem map.

Problem 2: Action compression

In the provided simple state space, the available actions are {`up`,`down`,`left`,`right`}, which corresponds to the single step movement of the player in Sokoban. However, certain sequences of actions have the same effect on the Sokoban puzzle and can be treated as the same "large step". For instance, the two action sequences shown in Fig. 1 have exactly the same effect on the puzzle: both of them result in pushing the box to the right by one step.



(a) `rur` (b) `urr`

Figure 1: The two action sequences are equivalent.

Generally speaking, for the sake of solving Sokoban, we care more about the movement of the *boxes*, rather than the movement of the *player*. Instead of finding the available moves for the player, we can focus on finding the available box pushes in each state. In this way, we effectively compress many equivalent action sequences into one larger action. In Fig. 1, the available box pushes are "push the box to the right" and "push the box to the left".

For this problem, modify the code in `sokoban.py` to implement the above-mentioned action compression in a new search problem named `SokobanProblemFaster`, which inherits directly from the provided `SokobanProblem`. Redefine the `expand` function in the derived class so that it overrides the definition in the base class. You need to modify the `solve_sokoban`

function as well to account for the change in the action sequence returned by the search algorithm. (The second value of the triple that the `solve_sokoban` function returns should be a list of actions, that is `u/d/l/r` characters.)

Note that a compressed action can be assigned a cost equal to its number of base actions, or a cost that is always one. These choices lead to different behaviors you can explore, as the latter considers all solutions with the same number of box pushes at the same cost, regardless of how long a trip the person takes between pushes.

Use `f` for the command line argument `algorithm` to run the Sokoban solver with your implementation of `SokobanProblemFaster`. For instance,

```
python3 sokoban.py 1 f -d
```

allows you to solve level 1 using the compressed action search, together with dead end detection.

Hint: You may ask the `expand()` function to append states that involves only on those actions that results in changes of box positions.

Problem 3: A* with a simple admissible heuristic

Even with action compression and dead-end detection, UCS cannot solve many Sokoban puzzles in a reasonable amount of time. Let's see how A* could help.

In the lecture example, we've seen that an admissible heuristic for the 8-puzzle is the sum of Manhattan distances[1] between each pair of block and its destination. However, unlike the 8-puzzle, we do not know the correspondence between the boxes and the targets in Sokoban, i.e. which box goes to which target location. Despite this fact, propose a way to build a simple **admissible** heuristic function for Sokoban that can be computed quickly, based on Manhattan distance.

Implement your proposed heuristic in the `heuristic` function in `sokoban.py`.

Problem 4: A* with a better heuristic

In the previous problem, we required the heuristic function to be admissible so that A* is guaranteed to find the shortest solution (the solution with the least number of box moves). However, for very complicated Sokoban puzzles, we are happy if the program can find any solution at all, even if the solution found is not the optimal one. It is worthwhile to use a more complicated heuristic that is not always admissible, as long as it gives a better chance of finding a solution in a reasonable amount of time.

Bear in mind that a complicated heuristic could be prohibitively expensive to compute, and thus may not improve the run time although it might reduce the number of states visited. There is a trade-off between having a good heuristic to guide the search to the goal, and having a fast heuristic that is cheaper to compute. **It is vital that your heuristic be**

[1] Manhattan distance between two points (x_1, y_1) and (x_2, y_2) is defined as $|x_1 - x_2| + |y_1 - y_2|$.

computable very efficiently, possibly leveraging some precomputed problem-wide information. You should build a heuristic that improves the search in most cases.

Propose a heuristic that improves performance well enough to solve all the provided levels, with some much faster than the other methods above, and implement it in `heuristic2`. (Explain your heuristic clearly in comments near your code.)

Problem 5: Design your own Sokoban maps

Design artificial (probably easy and not fun for humans) Sokoban puzzle levels to clearly illustrate the advantages of each of the above code developments. Specifically, design a level `p5-level-1` for which UCS is slow without dead-end detection, but fast with dead-end detection. Secondly, design a level `p5-level-2` that is slow for UCS with dead-end detection but fast with compressed actions added. Thirdly, design a level `p5-level-3` that remains slow with compressed actions and dead-end detection, but becomes fast when the simple heuristic is added and A* employed. Fourthly, design a level `p5-level-4` that requires all of the code improvements to become fast.

For these levels, achieve as much contrast as you can between "slow" and "fast". Ideally, a fast solution is under 10 seconds, or perhaps under 60 seconds and a slow solution unfinished at 300 seconds. In any case "fast" should be no more than half the runtime of "slow".

Submit your four levels in `design.txt`, and include a brief explanation why each level achieves the stated goals. Make sure the levels are formatted in the same way as the provided examples so that they are readable by `sokoban.py`, and named `p5-level-1`, `p5-level-2`, `p5-level-3`, and `p5-level-4`. Explanations should be on comment lines (which start with the `'` characters) with their corresponding levels.

(Optional) You may include a fifth level `p5-contest` that illustrates all the features of your program, especially if you add features beyond what we require above. For this level, your explanation is particularly important, because those with good explanations as to why the level is interesting (what algorithmic features are explored) are most likely to be selected for inclusion in the competition. We don't want you to build your algorithms specifically for some obscure difficult level and then include that level; we are interested in general algorithmic features that enable solving the provided levels.

We will select some of students' `p5-contest` levels into the Sokoban contest based on the explanations, and include more problems of our own. The top 20% of teams (ranked by percent of problems solved within a somewhat generous time limit) will be awarded the grade rewards specified for project performance in the syllabus.

Note: For the competition, we will invoke your program using `c` for the command line argument `algorithm`, i.e.

```
python3 sokoban.py <level> c
```

You can do something as simple as interpreting `c` as `fa2 -d` in the code, or something more complicated. You should briefly explain your contest approach as comments in the code, but almost anything goes as far as algorithmic and design ideas. We're not interested in

straight-up speedups (like clever use of libraries to speed up what we are already doing) and most creative things should get our approval before being submitted.

What you will submit

You will submit the following machine-readable files. This project has no written component beyond the requirement that the files contain comments as follows:

- **sokoban.py** must have comments at the top of the file naming any algorithmic technique you believe you are using (including making up a brief meaningful phrase for any innovation you have added that you found important for performance and indicating that it is your invention).
- **design.txt** must include a brief explanation why each level achieves the stated goals. Make sure the levels are formatted in the same way as the provided examples so that they are readable by `sokoban.py`, and named `p5-level-1`, `p5-level-2`, `p5-level-3`, and `p5-level-4`. Explanations should be on comment lines (which start with the `'` characters) with their corresponding levels.

Your code must be written entirely by your team members; no outside code can be used for this project. If you read outside code to understand an algorithm, you must put it aside before working with your own code. Also, you are responsible to understand how everything you submit is designed to work.

Minimum standard

To meet the minimum standard for this project to not hurt your grade you must at least:

1. Do a solid job on problems 1-3, achieving at least a factor of 0.7 time speedup in each case on every hidden problem we have reserved for this purpose, with the improved version finishing in faster than one minute. (This standard is subject to loosening if we determine there is a need, but do not rely on this.)
2. Provide the first three novel map levels as specified in problem 5, demonstrating your solutions to problems 1-3. You must get the same speedup on these levels as just described.

We are providing (or will modify the distribution zip file soon to provide) a minimum standard grader that will evaluate your code on this criterion for any file of map levels — use level names as shown in design.txt for this purpose. Only levels for problems 1 to 3 (e.g. named p1-* or p5*-1 for p1) will be noticed by this grader code `hw11_min_std_grader.py` to see if your problem solutions pass. Use the `-h` flag to see the command line interface for this grader. This grader is new code this semester and please watch Piazza for any announced revisions to the distribution. Note that you cannot evaluate the minimum standard using this grader without providing the three map levels requested by problem 5, though you can do this one level/problem at a time as the grader will run whatever levels it finds.

As the deadline nears (at least the last 7 days, watch Piazza), we will offer minimum standard evaluation on submitted code against our hidden problems. We will also evaluate your submitted map designs for problem 5 minimum standard requirements. We will do this evaluation at least once a day on newly submitted code until you pass. Once you pass that evaluation, you do not need to worry about the minimum standard any further.

Competition

We are providing some map levels for you to test your code on in `levels.txt`. We may provide more as the deadline approaches closer. We are reserving other levels to use in testing your code for the competition. Submissions will be ranked first on how many problems are solved within the time deadline of the problem. Your code can condition on this deadline if you wish. Submissions that solve the same number of problems will be ranked by total time used in the solved problems. We will try to provide some leaderboard ranking submissions during the last week, but we are not certain we can make this work due to the long runtime needed to evaluate this kind of program. It is possible we will run just one problem to determine the top few submissions and then rank those on a larger set of problems