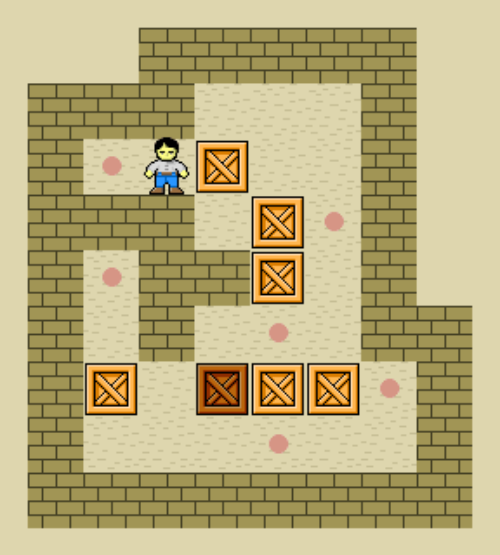
ECE 473 Homework 4



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.

You can work on this problem in a group of two or three if you have declared that group on Blackboard (each of you must designate the same group to qualify) by the deadline provided there (March 12, 1:30pm). Each group must designate a single student to submit the finished code and all members are responsible for the entire homework and will get the same grade. At any point during the homework period, any student may request to be removed from their group and work as an individual (all code written to that point must be shared with the group before departing), and the professor will approve such departures individually after consulting the group members. Group departures are discouraged, and will be disallowed if the departing member has not clearly created a reasonable share of the code to that point (unless all group members agree about the departure).

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. Note that the assignment below refers to grader performance standards that are not yet set. Until you receive the grader, make the program as effective as possible so as to be ready for the grader standards when we provide them. We will also specify later how to submit your solutions.

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 graded 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 or tested by any grader provided later.

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

```
python 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

```
python sokoban.py 1 ucs
```

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

Note: For the following problems, your implementation of the described features should demonstrate improvement in run time in some levels of `levels.txt`. You may assume that each problem is worth roughly the same points.

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

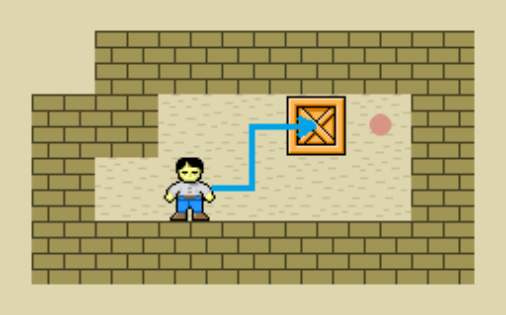
You can type

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

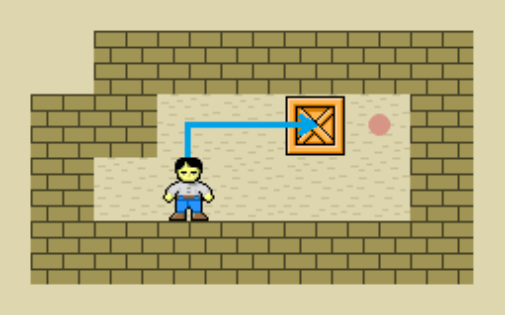
to invoke dead end detection.

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 may need to modify the `solve_sokoban` function as well to account for the change in the action sequence returned by the search algorithm.

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,

```
python sokoban.py 1 f
```

allows you to solve level 1 using the compressed action search.

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

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

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 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 an average of 2 extra-credit exam points per student, with higher rank in % solved leading to a larger share of the points. No student's grade will be lowered as a result of other students receiving extra credit.