

Homework #1

1/5/23

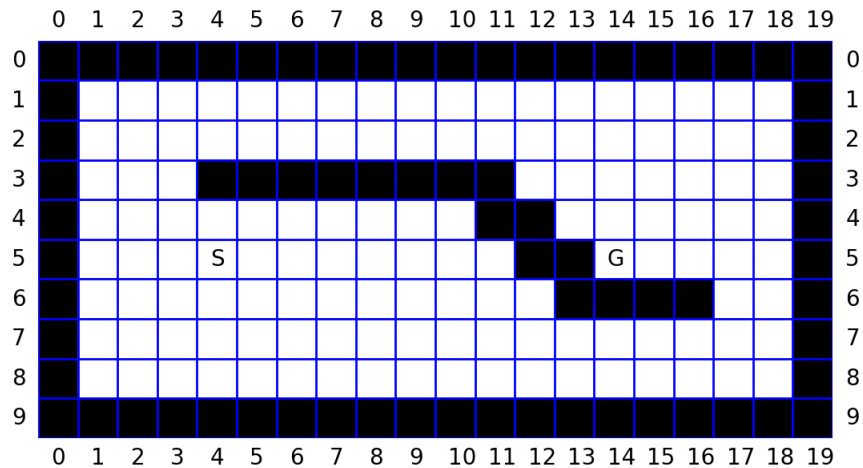
due Wednesday 1/10/23 11:59pm

To start us out, I want to explore the classic and fundamental discrete planning algorithms: Dijkstra and A*. This should get us thinking about the concepts, flex our programming muscles slightly, and be useful for many later algorithms.

As always, **please submit results together with explanations and code**, so we can appreciate your understanding.

Problem 1-3 (Discrete Grid Planning Problems):

For the following three problems, consider the following discrete world and imagine a simple robot that can move either up, down, left, or right (not diagonal, keeping it as simple as possible). It covers an 11x17 grid, with *start* and *goal* positions marked.



The small size should mean that our code doesn't have to be particularly efficient. Practically, you are welcome to code this however you like. But you may want to look at the skeleton code `gridplanner_skeleton.py` (posted on Canvas) which prepares the nodes and creates a visualization. The code section to edit is marked by "FIXME" in the `planner()` function.

As you code, be aware of the following list operations (using the `onDeck` list as an example):

<code>item = onDeck.pop(0)</code>	Pop (remove and return) the first element.
<code>onDeck.remove(item)</code>	Remove the given item.
<code>onDeck.append(item)</code>	Append the given item at the end.
<code>bisect.insort(onDeck, item)</code>	Insert the given item, so as to keep the list ordered.

In all cases, please (i) show a screen shot, with the final path (start to goal) clearly marked, (ii) report the final path cost, and (iii) give the number of states that the algorithm processed to find the solution (marked DONE, not including remaining on-deck states).

Problem 1 (Dijkstra’s considering only the number of steps) - 20 points:

Let’s begin with the simplest case, where the cost at each node is simply the number of steps taken to get there. Please implement Dijkstra’s algorithm with this assumption.

You should see the algorithm growing the tree outward radially, covering nearly the entire space, until it reaches the goal. If you examine the grid, you’ll see the path around the top is indeed shorter and should be solution the algorithm finds.

As with all problems, please submit the relevant code section (the `planner()` function) as well as a screenshot (Matplotlib has a “save-figure” button), the cost, and number of processed nodes.

Problem 2 (Dijkstra’s with move-dependent cost) - 20 points:

Please update the code to consider a cost to travel between two nodes as

$$\text{cost} = 5 * |\text{row}_1 - \text{row}_2| + 1 * |\text{column}_1 - \text{column}_2|$$

A single step moving up/down costs 5 while a single step moving left/right retains a cost of 1. With the varying costs, we now have to explicitly make sure the on-deck queue remains ordered.

As you might guess, the path around the top now incurs a higher cost as it requires more vertical movement. It is now optimal to take the lower route.

Please again submit your findings.

Problem 3 (A* estimating the remaining cost to the goal) - 22 points:

As the final step, we convert to A*. Please update the algorithm to additionally consider a cost-to-go estimate. The on-deck queue should now be sorted by the total predicted path cost, being the sum of cost to reach and estimated cost to go. You should see improvements in how well the planner pushes towards the goal.

This works for both (above) cost metrics, but to keep things simple, let’s return to cost being simply the number of steps (Problem 1). This also returns the optimal path to taking the upper route.

As such, the most logical metric to estimate the cost to go is the Manhattan distance:

$$d_{\text{Manhattan}}(n_1, n_2) = |\text{row}_1 - \text{row}_2| + |\text{column}_1 - \text{column}_2|$$

Please try 3 levels of aggressiveness, setting the cost-to-go estimate \hat{c}_{togo} to

- (a) $\hat{c}_{\text{togo}} = 1 \cdot d_{\text{Manhattan}}(\text{node}, \text{goal})$
- (b) $\hat{c}_{\text{togo}} = 2 \cdot d_{\text{Manhattan}}(\text{node}, \text{goal})$
- (c) $\hat{c}_{\text{togo}} = 10 \cdot d_{\text{Manhattan}}(\text{node}, \text{goal})$

As such, nodes far away from the goal are increasingly penalized and the search focuses more aggressively on driving to the goal.

Once again, please submit the findings for all cases as well as the code for the A* planner.

Problem 4 (Inverting the Search) - 10 points:

To gain a little extra perspective, please re-run Problem 3, parts (a) and (d), but for grid #2. That is, re-run A^* with a cost-to-go estimate of 1x or 10x the Manhattan distance.

This challenge is pretty typical of a robot trying to get from a hallway to a set of coordinates inside a particular room. Not knowing (in advance) where the doors are, the search algorithm explores many intermediate rooms looking for a passageway.

Next repeat while flipping (inverting) the search: Growing the search tree from the goal to the start. Practically, simply swap the start and goal spaces. Notice that is much easier to “grow out of a room”, meaning leaving a room and heading down hall, than the original direction?

In general, depending on the planning problem, it can be very helpful to build a search tree “backwards”.

Please simply report the number of states processed in these cases.

Problem 5 (High-Dimensional State Space = Sokoban) - 24 points:

I would love to explore a more complex case, in particular a higher dimensional state space. For this, the game Sokoban is ideal. It again operates on a simple 2D grid, like the previous problems. But it asks a robot (or human-like character in most games) to push boxes to designated goal location. As part of the rules, you can only push a single box, not two at a time. Try playing a game, for example at

<https://www.mathsisfun.com/games/sokoban.html>

This particular game is grid #9 in the code, so you can compare your solution.

As the robot and each box have their own (x, y) or (row, column) coordinates, the DOFs or dimensionality of the problem is $2(1 + N_{\text{boxes}})$. So even a small grid can see millions of possible states. More precisely, if the grid has N_{spaces} spaces, the total number of possible states is

$$N_{\text{states}} = N_{\text{spaces}} \binom{N_{\text{spaces}} - 1}{N_{\text{boxes}}} = \frac{N_{\text{spaces}}!}{(N_{\text{spaces}} - N_{\text{boxes}} - 1)! N_{\text{boxes}}!}$$











That said, the same algorithms work. We just need to consider

- Given the large number of possible states, it is not feasible to pre-allocate the entire graph. Instead, as the search tree grows, it has to instantiate the nodes on-the-fly.
- Each node corresponds to single state, which combines the coordinates of the robot and all boxes into a high-dimensional vector. So the code needs to clearly distinguish grid spaces from states/nodes.
- Because the state space is very complex, there is no good cost-to-go estimate. And as the cost is simply the number of steps taken, we can use the most basic algorithm (equivalent to Problem 1) with a FIFO on-deck queue.
- The code also does not need to track the **seen** and **done** flags. By design, un-**seen** nodes do not exist yet. And **done** nodes have been pushed off the on-deck queue.

So while the basic algorithm is simpler than before, the overhead of dealing with all the pieces means even in my best implementation, I needed 200 lines of code. So we are providing a very large skeleton in `sokoban_skeleton.py`. And asking you to update ~10 lines of code.

We are asking you to implement the state transition function: given a particular (current) state (the location of the robot and all boxes), and an action/direction (moving up/down/left/right), what is the resulting (next) state? If the movement is not possible, this should return `None` instead. See also the instructions in the skeleton code and the following illustration.

If the robot tries to move to the right, the following things can happen: It may encounter nothing, a wall, or a box, which in turn can encounter nothing, a wall, or another box:

Current	Next	Outcome	
		Just moving, no box	Only robot grid space changed
		Blocked by wall	no change
		Pushing the box	Robot and box grid space changed
		Blocked by wall	no change
		Blocked by double box	no change

→ Direction of (attempted) movement

If everything is working, you should see the test results as noted in the code. And the general results should match those given at the beginning of the code. Please try the different grids.

Please submit the code for the state transition function, as well as the data for grid #5: the number of solutions steps and the nodes created during the search.

Problem 6 OPTIONAL (Backwards Sokoban) - BONUS 20 points:

If you are having fun and have time remaining, consider the following challenge: As we saw in Problem 4 and in the data in the top of the Sokoban code, it can be very helpful to search backwards, rooting the tree at the goal. For Sokoban, often a 10x reduction in number of nodes visited! Can you update the code to search backwards?

Beyond rooting the search tree at the goal, two big things need to happen:

- The above state transition or `children()` function now needs to consider what the previous states could have reached the current state. The pushing action is not easily inverted. Can you construct an equivalent illustration of potential previous to current state transitions?
- There are now numerous root nodes for the search tree. Note that the robot could be next to any of the boxes at the goal. So we have to initialize the on-deck queue with all possible goal nodes.

Let us know if you are thinking about this and would like any further guidance.

Problem 7 (Time Spent) - 4 points:

Approximately how much time did you spend on this set? Any major obstacles?

And, going forward, we are always going to ask: if you are handing in late, would you prefer to use penalty-free late hours (default) or take a 10 points/day penalty (we are happy to change later).