1) **Define the agent problem**

Action Space (A) - All possible actions are left, right, up, down (L,R,U,D) as long as not moving through the walls or box.

Percept Space (P) - The agent will always know the exact state of the map e.g. boxes, targets and the maze, it will not change so there is nothing the agent cannot perceive.

Since the problem is fully observable, the percept space is the same as the state space:

$$P = S$$

State Space (S) - The state space is defined as the position of the boxes and the player in the maze, in addition to the area reachable to the agent.

World Dynamics/Transition Function ($T : S \times A \rightarrow S0$) - This game is deterministic, the agent knows exactly where it will be after performing an action

Perception Function ($Z : S \rightarrow P$) - Since P = S, there is no need to consider this function.

Utility Function ($U : S \rightarrow R$) - The cost of this action is 1 per step.

2) What type of agent is it? (i.e. discrete /continuous, fully / partially observable, deterministic / non-deterministic, static / dynamic)? Please explain your selection

(Discrete, fully observable, deterministic, static)

Discrete - since the player can only move one grid a time; the action space of the agent is not continuous.

Fully observable - since the percept space of the agent is the entire map (including every box, target and the player itself).

Deterministic - since the agent knows exactly where it will be after performing an action; the world dynamics is a function.

Static - For each state before the player moves, the map doesn't change its state.

4) Heuristic

The heuristic used for the A* algorithm was by mapping a box to a target by using the smallest Manhattan distance (MD). This is a good heuristic, as the agent will be "encouraged" to explore branches which move the boxes closer to their targets, as these will have the highest priority (lowest f(n)) on the PQ. This heuristic is admissible, as the agent must push the boxes these distances to the targets and is the minimum distance assuming no obstacles. Thus, this heuristic will never overestimate and $h(n) <= h^*(n)$.

Another heuristic that implemented was MD of mapping a box to a target, in addition to the MD of the agent to the nearest box (If a player is touching the box this distance is zero). This is similar to the previous heuristic; however, this heuristic also "encourages" the player to move towards the nearest box - a requirement to be able to push a box towards a target. This is also an admissible heuristic, as the minimal distance involves the agent moving to a box in order for the Sokoban to push it towards a goal.

Testing these two different heuristics provided interesting results, especially when changing the weighting of the MD of box to target and agent to box for the second approach.

5) Compare A* and uniform cost search

Test case: 1box_m2

A star:
Time required = 0.07700037956237793
Explored states = 195

Container max size = 55
Number of steps to the goal = 26
The value of i: 557

Uniform cost search:
Time required = 0.07999038696289062
Explored states = 290
Container max size = 71
Number of steps to the goal = 26
The value of i: 743

A comparison between A star and uniform cost search shows a marginal improvement in run time (~4%) and a nearly 50% (48.7%) decrease in explored states from the A* algorithm. This is before deadlock detection implementation. For fringes, there were 55, and 71 for UCS and A* respectively, and a total of 195 and 290 explored nodes for UCS and A* respectively.

Criteria	
A	The number of nodes generated
B	The number of nodes on the fringe when the search terminates
C	The number of nodes on the explored list (if there is one) when the search terminates
D	The run time of the algorithm (e.g. in units such as mins:secs)

Testcase: 1box_m2

Criteria	UCS	A*
A	361	106
B	71	55
C	290	51
D	0.07999s	0.077000s

A notable finding was the implementation of greedy first best search:

	Greedy
A	134

B	65
C	69
D	0.0100s

Greedy best first search, while exploring more states than A* found the optimal solution the fastest (requiring 12.5% the runtime of UCS). However, on more difficult test cases, while it found a solution it was unable to find the most optimal path.

Explanation of results

A* search returns fewer nodes generated, as it explores less states. A* can bias its search using it's heuristic, allowing "encouragement" of certain actions of the agent. In case of a heuristic that uses the Manhattan distance between a box and target, a node/state that moves a box closer to the target will be of higher priority on the PQ, and that branch will be explored first. Thus, A* will have a lower number of states explored, as it will not visit states the heuristic deems to be moving further away from the goal state. Since A* has less nodes to explore, it is able to reduce its run time.

5) Deadlocks

To find deadlocks, the obstacle map is iterated to find corners. Corners that are deadlocked can be defined as any cell that does have a wall, or target in it, in addition to having three or more walls in adjacent squares (up, down, left, right) or have two adjacent walls that do not share the same x and y coordinates (Such as a wall above and below, left and right). Furthermore, along each of the corners found to be deadlocks that share the same x or y coordinate, the edges can be marked as deadlocks under certain conditions. Along an edge, a cell is a deadlock if there is no target or wall along the edge, or there is no hole wall such that a box can be pushed through. To determine whether there is a wall or target, the edge cell is checked. The adjacent cells in the same shared corner coordinate (If two corners share a y coordinate, then check y + 1 and y – 1) is also checked to make sure a wall is present, and there is not a hole that can be used to move the box from the edge (Off the shared corner coordinate).

An alternative somewhat advanced approach to dead lock detection could involve iterating over each of the boxes and following a "pulling" method. The pulling method involves the Sokoban pulling a box throughout the whole map and recording the positions that the box visits. The positions that were not visited by the box are marked as dead lock positions, as if the Sokoban cannot pull a box into a certain position, then the Sokoban is also unable to push a box from that position (Assuming the position is not a target). This method would find every possible dead lock position, including difficult to find dead locks such as "stacked" boxes, where a box cannot be pushed due to its path being blocked by another box.