For this assignment, we were asked to implement the program as 3 separate versions. This report will first focus on commentary related to the specific versions, then focus on more the more general content in a “version-agnostic” section.

# Version 1

## Path Planning to the Box

This is the version where the path planning is implemented. My version of the path planning works like this: First, the code will find what side of the box the robot should be targeting. This is determined by looking at where we want the box to be pushed, and targeting the side of the box that’s in the opposite direction as that.
Once the target side is determined, the process of moving the robot to the box begins. This is handled in the moveRobotNextToBox function.
First, a target location is determined. This can be the target side of the box but usually it will be a position that puts the robot in a straight shot to the target side of the box. The target position is always a straight line away from the robot’s current position.
The robot will move towards the target position and when it makes it there, will perform the same process again of finding a new target position and moving towards it.

At first, this path-finding algorithm would first prioritize moving vertically, then move horizontally. However, this caused problems as some situations would result in the robot traveling through the box to get to the intended side of it.
To remedy this, I prioritized vertical travel first if the target side of the box was on the north or south ends, and horizontal travel first if the target side of the box was on the east or west ends.

This solved this box collision problem except for one specific case: if a robot started out already a straight line away from the box, and the target side of the box was on the opposite side that the robot was, the path-finding algorithm would still make the robot walk through the box!

Here’s how I handled that specific exception: if the robot had not yet moved and it was in this situation, move in a direction that puts you out of the box’s direct line of sight. This would make the next two target positions form an L-shape like usual and avoid the collision.

## Box Pushing to the Door

The robot reaches the target side of the box and begins pushing it. The robot pushes the box to the target position that was originally used to determine the target side of the box. If the box is not in the same location as its respective door at the end, perform the process again, starting with path-finding the robot to the new target side of the box. Once the box is pushed onto the door, the function ends.

This part of the program was implemented without any real issues.

# Version 2

Version 2 is very similar to version 1. Instead of running the robotFunc function iteratively for each robot, they are all run at the same time through the use of multithreading. Previously, calls to displayGridPane() and displayStatePane() had to be done at certain points in the execution of the function, but these were removed now that it was being handled by the main thread.

Writing to the file was also given mutex lock protection now that multiple sources could potentially write to it at the same time, causing issues.

# Version 3

Version 3 added the ability for objects to be aware of other objects. If a robot or box tries to move to a new space, it will not do so until that space is no longer occupied by an existing object.

This was handled by creating a grid of mutex locks equivalent in size and dimensions to the main grid.

If an object was occupied a space on the main grid, the associated element on the mutex grid would be lock and they would unlock it when they left.

# Version-agnostic

## Input Parameters

The input parameters were converted to ints using the atoi() method. If a number outside of the 1 to 3 range is entered for the doors parameter, the program will spit out a message telling you to enter within range. Besides that, no safety mechanisms or exceptions were written for bad input.

## Generating the objects on the grid

Doors were the first to be generated. As was specified in the PDF, I simply provided a random position for each of these without consideration for overlap with other doors.

Secondly, the boxes were generated. The spawn of these boxes excludes the edges of the grid.

While I am really not a fan of this, I used a re-roll style method to avoid overlap with doors and other boxes--if the generated box happens to spawn in the same position as another object, it will simply be generated again.

This same “re-roll” method was applied when generating robot positions.

The reason I used this style in spite of its downsides is because the alternative I had in mind seemed complicated to implement and like it’d be much more intensive than what would normally come of the re-reroll method. This alternative style would have determined the number of open spots in the grid,

picked a random number with the number of open spots as the max, and found the respective open spot and used that.

## Output of the program

All file output was handled by constructing a string and passing it into a writeToFile function. This function is protected by a mutex lock, so only one thread will be able to write to it at a time.

## Limitations of the program

As previously stated, the program can only handle certain input without crashing, as the only form of input validation is a range checker on the number of doors.

If the density of objects on the grid is very high, it might take a long time to generate, as the program will be continually picking random positions until it finds one that's available.

As the robots will only move onto a space if it's not already occupied and the robots do not change their path in response to this, deadlock can happen. The likelihood of this increases the denser the number of objects is on the grid.

The thread list size is not dynamically allocated, so running the program with more than 256 robots will result in unintended behavior.

## Difficulties I ran into

Having to resort to the "re-roll" method of determining a random position for some of the object was not something I wanted to do but went for simply due to a lack of many better options I could think of.

I was planning on dynamically allocating the array of threads, however no matter what I did it would always result in the program terminating without an active exception.

The series of mutex locks for the grid was intended to be a 2D array as well, however strange behavior was occurring at the locks for any element in column 0, so I resorted to a 1D array instead.

Other difficulties I ran into were discussed earlier in the report, like having to find a way to avoid traveling over a box.

## Extra Credit & later changes

Sometime after completing the original version of my assignment, I made some modifications to my program. RobotFunc and its associated programs have been reorganized into a clearer and more concise format. The path-finding algorithm was also updated to be more in line with how it works in the assignment specs. In addition to this I have added some deadlock handling.

When a robot is moving independently, it will check if there is an object in the space it plans to move to. If an object is detected, it will move in an alternate direction. The robot will still adhere to a path generated by the path-finding algorithm, so this doesn't do a stupendous job at resolving deadlock since it will try to take the same path it just did, but it helps in a small way by creating an opening for another

robot to get through that a program without this functionality would not be able to resolve. A side-effect of this behavior is that deadlocked robots may move back and forth on the grid instead of just staying still.

A robot stepping away from its route to fix deadlock is locked behind a mutex lock. This is because having one deadlocked robot move out of the way at a time instead of having multiple deadlocked robots moving out of the way at a time is more likely to actually resolve the issue.