Questions

Compare the amount of space used and discuss the effects on running time if instead of a separate circular queue, a queue link was included in position structures and the queue was implemented as a singly linked list.
In the context of the problem, the amount of space occupied by the backing array of the queue is negligible, so operating space — `O(n)` — doesn't differ between the two options. As for running time, queue operations in both only set a few pointers with each operation, so both the linked list and the array-backed operation are same in operating time — `O(1)`.

Compare the worst-case time and average case time of implementing hash table buckets in the standard way as a singly linked list compared to representing a bucket with a binary search tree.
A binary tree would require that the items being hashed to the hash table have some sort of way being being compared to on a binary basis, which, for a position structure, could be the absolute time the position was generated. However, while insertion and deletion would be fast (`O(n)`). To build a binary search, in the **worst case scenario** would be `O(n^2)` time, should the data arrive presorted, as the number of comparisons needed to insert an item would scale linearly with each additional member in the list — so to add a third item, two comparisons would need to be done, fourth item, 3, and so on an so forth. However, in the average case time, a proof by induction can show that `O(nlog(n))` time to build a tree.

The number of items that ended up in the hash table may exceed the number of buckets. Describe how you would implement closed hashing to address this issue, and compare the time and space with the open hashing used for this assignment.
I'd expand the size of the array should the number of items exceed the number of buckets in the array, and copy the original array to a new one, one that's pro. The copying would operate in `O(n)` space and time, where `n` is the size of the original array.

Although it would not be needed for the Panama Canal puzzle, one can imagine that for more complex puzzles and more complex distance metrics, if a shorter path is discovered to a position already in the queue, one may want to promote that position to the front of the queue. Describe how you would provide the promote operation with both array and linked list implementations of a queue. Discuss the time and space used.
Array Implementation: The position would be moved to the front of the array. The operation is in `O(n)` time and space, as the entire array would have to shifted over, so memory would have to be allocated for moving the array.

Linked List Implementation: The position would just be moved to the front of the list. The operation is in `O(1)` time and space, as the operation just requires changing `next` and `previous` of 4 nodes. Because the operation's memory use is independent of the size (the operation only operates a small number of nodes each time), the space used is constant.

After a solution is found, it can be visited in reverse order by following the back pointers to the start position. Two approaches to printing the solution in correct order are to push pointers to the positions on a stack as the back pointers are followed (and then when the start position is reached pop positions off the stack in correct order), or to reverse the back pointers as there are traversed

(and then follow the back pointers once the start position is reached). Compare these two approaches in terms of time and space, ease of programming, and issues such as not knowing the length of the solution in advance.
To ensure a stack (if it is implemented by an array) isn't overrun by the new contents, it would require knowing the size of the solution in advance. In terms of time, there isn't a clear benefit of using a stack as opposed to following and reversing a linked list, as both would require passing over the solution twice, hence operating in `O(2n)` time. However, there is no difference in space usage — once the solution is copied over toe the results stack, the original linked list can be deleted, and the net space usage would remain mostly equal, assuming the back array is of negligible size compared to the results. Space usage, under the previous assumptions, is therefore `O(n)`. As for programming ease, the traversing the linked list backwards is slightly easier, by sheer virtue of not writing a stack implementation. The difference, however, is slight, but writing less code is considered preferable by the author.

When a position X is dequeued that came from a position Y, does your code see that it just came from that direction or does it generate position Y and discover that it is already in the hash table. Discuss the pros and cons of these two alternatives.
My code looks dequeues a position, and checks if the recently dequeued position inside the hash table. I chose this approach due to the simplicity of the implementation. The alternative is a more complex solution, but requires less operations to check. This trade off is in programmer efficiency vs. machine efficiency. Due the high volatility of the codebase, I chose in favor of programmer efficiency.

Consider the Traffic Jam Puzzle shown below. If one move is defined as moving one piece one unit of distance in one direction, discuss how you would generalize your program to solve this puzzle. In addition, if one move is defined as putting your finger on a piece and pushing it one or two units (either on a straight line path or a around a corner), discuss how your program could be further generalized to find a solution of a minimal number of moves.
The current implementation of the puzzle assumes that a single tile fits in a single point in an array. That is, a single tile fits in a single point `Array[x][y]`. To generalize this puzzle, the board will have to account for the variable size of a tile, and therefore, the limited types of movements available to it. Each tile would be be implemented as a structure that includes information about the dimensions of a tile as well as its current orientation.

What was the most difficult part of this assignment; are there any software tools you did not have that would have made it easier? Would collaboration with others have helped, if not why, if so, how would you have organized it?
One of the most difficult parts of the assignment was the C programming language itself. Xcode's debugging tools made working with C a far more pleasant experience, as the any sort of non-segmentation fault errors are caught while writing code. In addition, it's absurdly easy to catch any sort of runtime errors with breakpoints and `po` command in the debugger.

Collaboration would've made writing and debugging some data structures significantly easier. While the TAs have excellent availability, there's a fundamentally different dynamic at play, as it's a one-sided, student-teacher relationship. When working with other students, there's a sense that you're in this together, and that alone would've made a huge difference in morale, especially as finals arrive. As for organization, we'd collaborate over GitHub, and split writing each data structure between us, working on a part that each one of us feels most confident in.