Shreya Wagley wagleyshreya@gmail.com

Algorithm Analysis

- search() function:

Overview:
The function uses a breadth-first search algorithm to explore all possible moves and states from the current state, keeping track of the explored states in a vector to avoid duplicates. The function stops searching and prints a failure message if the goal state is not reached within 101 steps, or if all possible states have been explored. If the goal state is reached, the function prints a success message and the number of steps taken to reach the goal state.

- The simplified code for the algorithm:

```
function search (block, destRow, destCol){
   create a queue called "frontier"
   add the initial state to "frontier"
   create a vector called "explored"
   set step = 0

   while "frontier" is not empty and "step" < 100{
      current_state = get the next state from "frontier"

      if the current state is the goal state (block is at destRow, destCol){
         print goal achieved in n steps
         print the current state's board
         push the block down in the current state
         return
      }

      print current step number
      print the current state's board
      push the block down in the current state

      (1)valid_moves = get all valid moves from the current state

      for each move in valid_moves{
         if move is not in "explored" {
            add move to "frontier"
            add move to "explored"
         }
       }
        increment step
    }
    print failure message
}
```

Shreya Wagley wagleyshreya@gmail.com

- Analysing the time complexity for the search() function:

The initialization of the queue, frontier, and explored vector take constant time, O(1).

The while loop executes for a maximum of 100 steps, so it has a constant upper bound of O(100).

The time complexity of getting the next state from the queue and checking if it is the goal state takes constant time, O(1).

The time complexity of printing the current state and step number takes constant time, O(1).

The time complexity of getting the valid moves from the current state depends on the number of blocks in the state. Since there are k blocks in the worst case, where k = n^2 - n, the time complexity is
O(k), or O(n^2 - n).

The nested for loop that checks if the move has already been explored has a worst-case time complexity of O(k), or O(n^2 - n).
Adding the move to the frontier and explored vectors takes constant time, O(1).

- The complexity of my search algorithm in Big-O:

Since a normal queue is used, the time complexity of adding a state to the queue is O(1). Similarly, removing a state from the queue is also O(1). Checking if a state has already been explored requires iterating over the list of explored states, which takes O(k). Generating valid moves takes O(k) as well.

Therefore, the overall time complexity of the search algorithm in the worst case is O(k^2), or O(n^4 - 2n^3 + n^2) further simplified to O(n^4) by removing the lower-order terms n^2 and -2n^3. In practice, the algorithm may find the goal state before exploring the entire search space, resulting in a lower time complexity.

- # AStarSearch() function

Overview:
The function uses a priority queue to store the states to be explored, with priority based on the total cost of the state (the number of steps taken plus a heuristic value). The function explores states until it reaches the goal state or until it has explored all possible states within a specified limit. It then prints the final state and the number of steps taken to reach the goal state or the reason for termination if no goal state is found.

- The simplified code for the algorithm:

```
function AStarSearch(block, destRow, destCol){
   create a priority queue called "frontier" to hold the states to be explored
   add the initial state to "frontier" with a priority of 0
   create a set called "explored" to store explored states

   set step = 0

   while "step" < 101 and "frontier" is not empty{
      current_state = get the next state with the lowest priority from "frontier"

      if the current state is the goal state (block is at destRow, destCol){
         print goal achieved in n steps
         print the current state's board
         push the block down in the current state
         return
      }
       print current step number
       print the current state's board
       push the block down in the current state
       (1)valid_moves = get all valid moves from the current state

       for each move in valid_moves{
           calculate the heuristic value for the move
           h = heuristic value for the move (using (2)Manhattan Distance)
           f = total cost for the move (step + h)

           if move is not in "explored"{
              add move to "frontier" with a priority of "f"
              add move to "explored"
           }
           increment step
        }
        print failure message
    }
```

- Analysing the time complexity for the AStarSearch() function:

The initialization of the priority queue and set takes O(1) time.

The while loop iterates a maximum of 101 times, which is a constant value. Therefore, it takes O(1) time.

Within the while loop, getting the next state from the queue takes O(log n) time because of the pop() operation of the priority queue.

Checking if the current state is the goal state takes O(1) time.

Printing the current state and step number takes O(1) time.

Generating the valid moves from the current state takes O(b) time, where b is the branching factor of the problem.

Checking each valid move takes O(log n) time because of the insert() operation of the set.

Adding the move to the frontier with the calculated priority takes O(log n) time because of the push() operation of the priority queue.

Adding the move to the explored states takes O(log n) time because of the insert() operation of the set.

- The complexity of my search algorithm in Big-O:

In the worst-case scenario where the number of blocks is n^2 - n, the branching factor of the problem can be approximated as 2, since each block can move in up to two directions (left column, right column). Thus, the time complexity of the algorithm can be approximated as O(2^d log n), where d is the depth of the optimal solution.

In practice, the algorithm may find the goal state before exploring the entire search space, resulting in a lower time complexity.

Shreya Wagley wagleyshreya@gmail.com

Appendix:

(1) valid_moves function:

```
function valid_moves() returns a vector of States{
   valid_moves = an empty vector of States

   for all blocks on the grid{
      row = row index of block
      col = col index of block

      if block can move left{

         new_state = a copy of the current state
         new_state.move block to left
         valid_moves.add(new_state)

      }
      if block can move right{

         new_state = a copy of the current state
         new_state.move block to right
         valid_moves.add(new_state)
       }
    return valid_moves vector
}
```

(2) Manhattan distance function:

```
function Manhattan distance(block, destRow, destCol) returns an integer:

   blockRow = row index of block
   blockCol = col index of block

   distance = | (blockRow - destRow) |+ | (blockCol - destCol) |

   return distance
}
```