ITCS 6150 Project 2 Report

N-Queens Problem Formulation

The N-Queens problem deals with placing N-Queens on a board sized NxN in such a way that they cannot attack one another. We are solving this problem using several variants of Hill Climbing Search. We want to support Hill Climbing without Sideways move, with Sideways move, and both of those in the context of Random Restart. Sideways moves allows queens to be moved to spaces with the same number of queens attacking them, if there is no better place to move any queens. The Sideways Move variant uses Hashmaps to store the positions the queens have been in, since we don't want to move sideways into a position we have been in before, and hashmaps are an efficient way of accomplishing that. Both versions need to use priority queues for sorting which of the queens has the highest number of queens attacking it. I chose to write the program in Python, specifically Python3, which has a built-in dictionary structure which uses hashmapping, and a module called heapq that enables an efficient priority queue.

We want to find the success and failure rates for both Hill Climbing without and with Sideways move, the average steps on a success of both, and the average steps on fail of both. We also want to show four search sequences from random initial configurations for both. For random restart, we want to find the average number of restarts required without sideways move, and the average number of steps required with and without sideways move. We do this by running each 1000 times by default, though this can be changed by changing the value of the 'iterations' variable in the part of the code responsible for final user interaction.

Program Structure

Helper Functions

Problem setup

The function makeArray takes in a value n, and makes a 2d array of zeroes of size nxn. This allows us to make empty arrays to put through the heuristic function very easily.

The function makeQueens takes in a value n, and generates n queens, such that there is one queen per row, each on a random column on their row. This was the method of setup recommended during the Week 4 lecture.

Hill Climb Helpers

The function isAttacking takes in a queen, and the indices for the square we are checking if the queen is attacking. It returns a 1 if the queen is attacking the square, and a 0 if it is not. This is used during the process of the hill climb algorithm to check if a single queen is attacking a square, since we want to lower the heuristic value h of the squares we are considering if they are being attacked by the queen we are considering moving. This way all squares will be considered equally even if they are attacked by the queen we are moving.

The function bestMove takes in the 2d heuristic array, the indices representing the queen qi and qj, and the array of all queens. It checks to see which move is the best move by checking for the lowest heuristic of all the unoccupied squares. It uses the queens array to check if a square is occupied. It uses the isAttacking helper to lower the h value of the squares it is considering if that square is under attack by the queen. If the queen is already in the best position, it returns None for the position. Otherwise, it returns the best possible position for that queen.

The function sidewaysFinder takes in the 2d heuristic array, the indices of the queen qi and qj and the queens array. In a very similar manner to the bestMove function, it finds all the squares with the same h value as the square the queen is on, and places them in an array, which it then returns. This is useful for the sideways move version of hill climb.

Result Display

The printArray function is a helper that takes in a 2d array of any size, and prints it row by row so that it can be displayed to the user.

The function showQueens takes in the size of the board n, and the array of queens, and places the queens in an array, then prints the array. Unoccupied spaces are listed as zeroes, and occupied spaces are listed as ‘Q’. It uses the printArray function to do so.

The printPath function uses the showQueens function to print an array of queens arrays so that they can be displayed to the user as the list of steps taken to reach a solution or dead end. This function is used only in the user-interactable portion of the program for printing out the list of steps.

Heuristic Function

The populateAttacking function takes in a 2d array of size nxn and an array of n queens. It populates the array with the number of queens attacking on each square. It uses the functions incrementRow, incrementColumn, and incrementDiags to do so by running each queen in the queens array through each of the three functions. It then decrements the space the queen is sitting on by 2 to account for repeated increments by the other functions.

The incrementRow function takes in the 2d array and index i, and increments every value in the row i. It does this by incrementing every value in array[i]. This increments all the spaces that are attacked by the queen horizontally.

The incrementColumn function takes in the 2d array and index j, and increments the entire column j. It does this by going through each row of the array and incrementing the block in that row with index j (representing the column id). This increments all the spaces that are attacked by the queen vertically.

The incrementDiags function takes in the 2d array and the indices qi and qj representing the position of the queen that we are incrementing for. It then goes through the entire 2d array and increments all the values with indices such that i+j is equal to qi+qj, and all values with indices such that i-j is equal to qi-qj. This pattern increments all the spaces that are attacked by the queen diagonally.

Hill Climb Functions

The first Hill Climb Function, hillClimbSearch, takes in the parameter n, and an optional parameter queens, which can be used if we want to test how the different versions of the program solve a specific layout. If we are not given an array of queens, it uses makeQueens to make one randomly. It stores the state of the queens array in the path array, so that we can print it later to complete one of the tasks of the assignment. It also uses a counter called ‘steps’ to store the number of moves made to get to the end. It then enters a loop which it only exits by returning. With each instance of the loop, it reinitializes the heuristic array and stores the values using the populateAttacking heuristic function, so that when queens are moved, the heuristic changes for the next step. It either returns when it has found a solution, or when it has exhausted them by checking every queen for good moves. It uses a priority queue represented by a heapq heap in

order to prioritize which queens to check for moves first by how bad the h value of the square they are sitting on is. If all of the queens are on squares with h==0, a solution has been found, so we return the current position of the queens representing the solution, the value of the counter 'steps' and the path it took to find the solution. If we are not at a solution, it checks for good moves for each of the queens starting with the queen with the worst h in the priority queue using the bestMove function. If none of these queens have good moves, but they aren't all on squares with h==0, the algorithm returns None, representing the absence of a solution, the value of the counter 'steps', and the path it took before failing. If one of the queens does have a good move, it moves the queen, stores the new position of the queens in the 'path' array and increments the steps counter.

The second Hill Climb Function, hillClimbSearchSideways, works exactly like the hillClimbSearch function with a few exceptions. It also takes in another parameter, called maxSidewaysMoves, which allows us to choose the maximum number of consecutive sideways moves before failure. It uses the counter sidewaysMovesCount to count the number of consecutive sideways moves. If a good move is made, this value is reset to zero. It also stores all the states of the queens array that it has visited so that sideways moves will not cause the algorithm to loop. It stores these using a hashmap represented by a python dictionary. Also, instead of ending when no good moves are found, it instead checks every queen for sideways moves (using sidewaysFinder), putting them in an array of possible sideways moves. Once this array is populated, it chooses one of the sideways moves randomly, moves the queen accordingly, increments the sideways moves counter, and starts over. If there are no sideways moves, the algorithm fails and it returns None in place of the queens array, representing no solution, the path taken to get there, and the number of steps taken. If we have reached our limit

for sideways moves, the algorithm fails and it returns None in place of the queens array, representing no solution, the path taken to get there, and the number of steps taken. By this, the algorithm only fails if there are no possible good or sideways moves OR if it exceeds the maximum number of consecutive sideways moves.

User Interactable Code

The user interactable code first takes in the n value from the user, and the number of allowed consecutive sideways moves from the user. It suggests 100 for the consecutive sideways moves as this is a good number for the 8-queens problem. It then finds all of the values that the project description requires us to find and reports them to the user. It uses the ‘iterations’ variable to determine the number of times it runs each algorithm to get an average. This code is also where the entirety of the “Random Restart” code is. The random restart code uses the two Hill Climb functions (which both start with a random setup by default) to search until it finds a solution.

Results of Experiments

Over 1000 Iterations each:

Regular Hill Climb Search Results:

Success Rate: 14.499999999999998%

Failure Rate: 85.5%

Average Steps On Success: 4.475862068965517

Average Steps On Fail: 3.3321637426900583

Sideways Hill Climb Search Results (100 consecutive sideways moves max):

Success Rate: 96.6%

Failure Rate: 3.4000000000000057%

Average Steps On Success: 13.493788819875776

Average Steps On Fail: 16.323529411764707

Random-restart Hill Climbing Results:

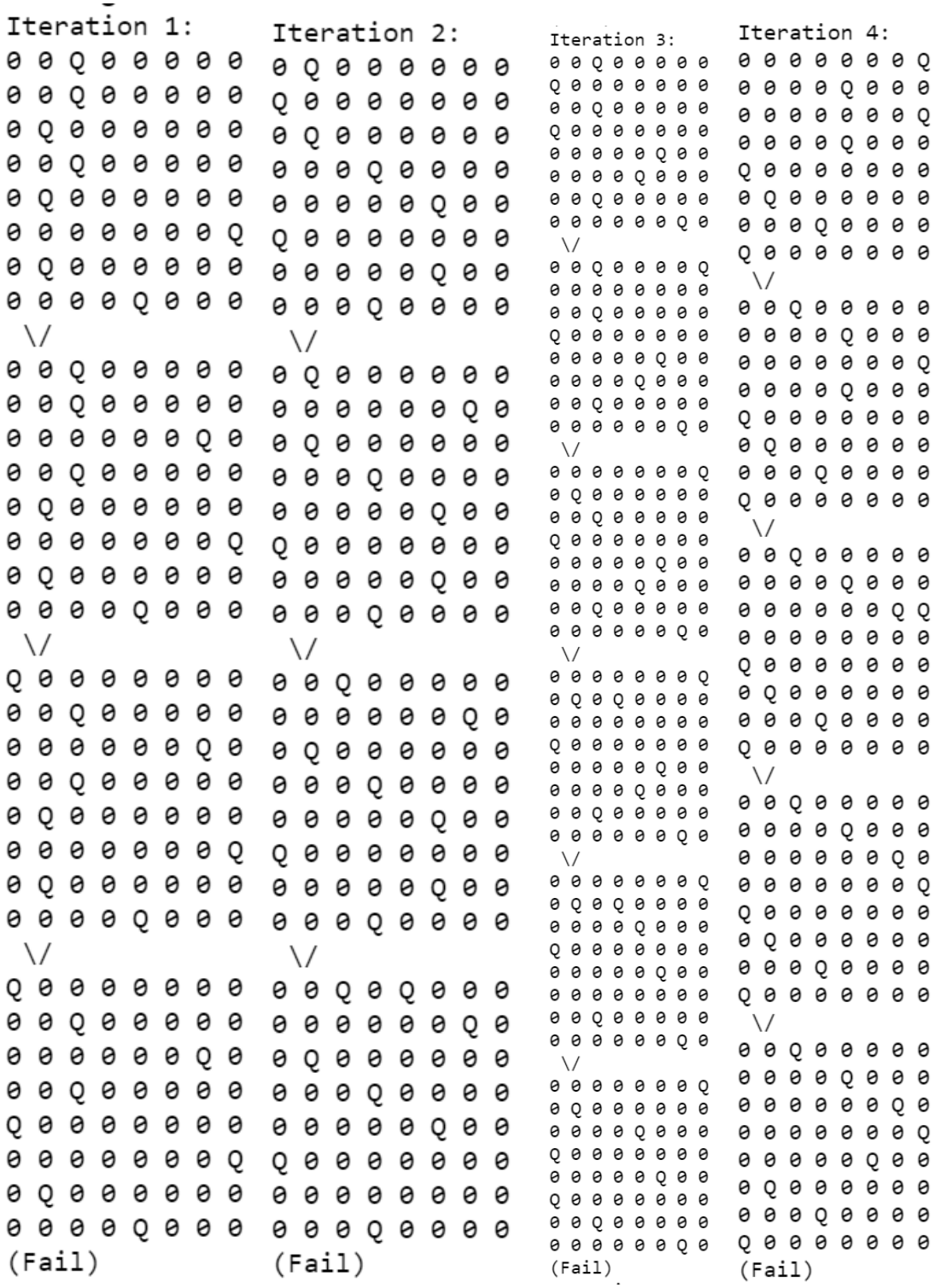
Average number of Random Restarts per iteration (Regular): 6.675

Average number of Steps per iteration (Regular): 23.22

Average number of Random Restarts per iteration (Sideways): 1.022

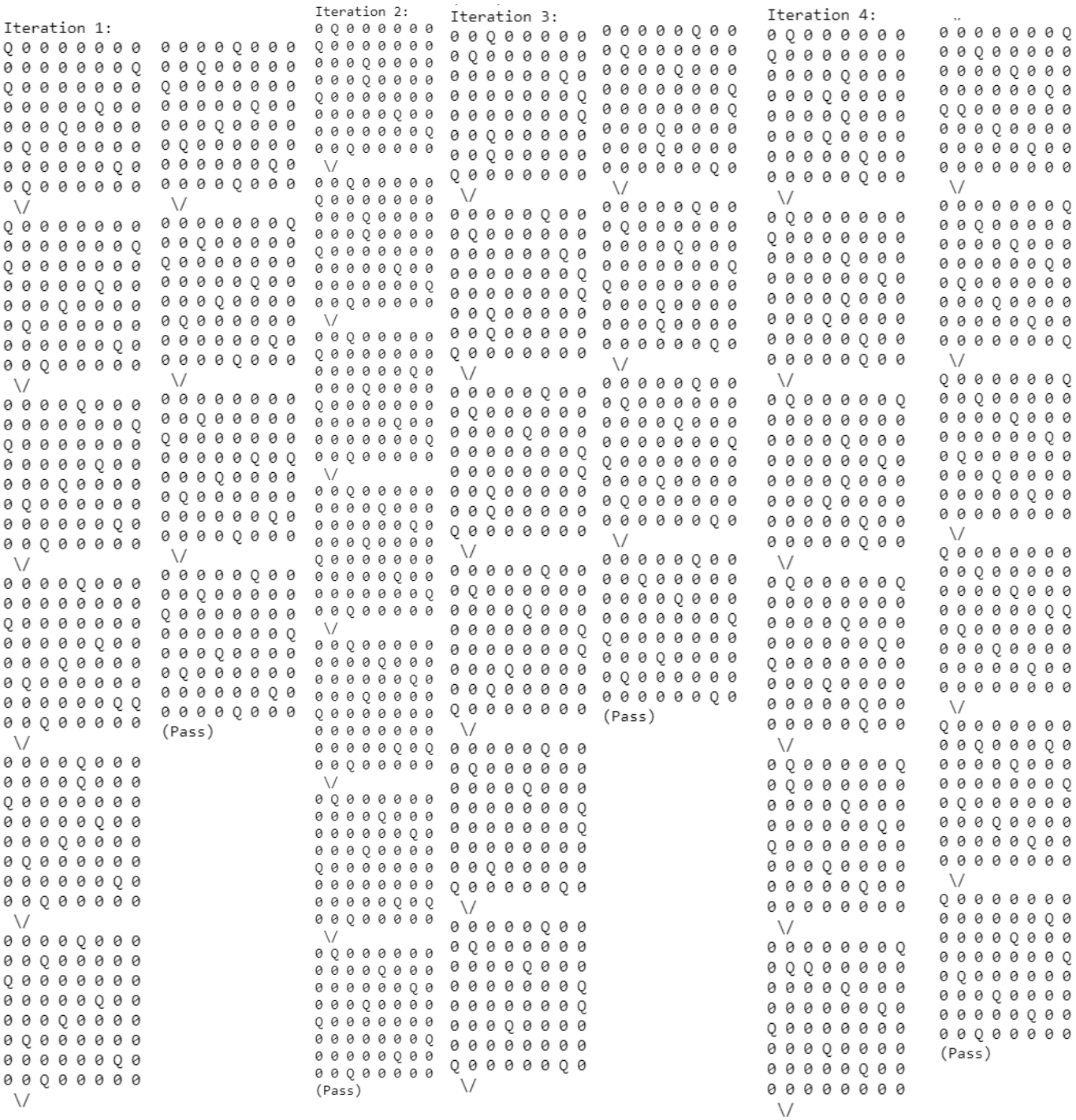
Average number of Steps per iteration (Sideways): 14.46

4 Random Initial State paths without sideways move:



As you can see, it failed all four times, which can be expected with a 14% success rate.

4 Random Initial State paths with sideways move:



As you can see, the sideways move function passed all four times, which is to be expected with its high success rate, but it also took longer paths than without sideways move.

Results Summary

The results of the experiments showed the expected tradeoff between the Hill Climb algorithm with and without sideways move. The sideways move algorithm increased success rates and reduced failure rates dramatically, but also increased the number of steps taken in both the success and failure case. The random restart always succeeds, but it takes different numbers of restarts depending on which base algorithm it uses. The sideways moves algorithm reduces both the average number of restarts, and the number of moves required to find a solution in comparison to the no-sideways-moves version of the algorithm. This is because of the dramatically higher success rate that the sideways-moves version claims compared to its less significant increase in number of steps taken.