Checkmate

Blaine Broussard

Wofford College  
429 North Church Street  
Spartanburg, SC 29303-3663  
912-308-8794

broussardbm@email.wofford.edu

**ABSTRACT**

In this paper we present an algorithm that can calculate the number of spaces a queen can attack on a board, given a certain set of obstacles. The problem is from hackerrank’s “Queen’s Attack II” The algorithm is linear time with a constant space complexity.

**Keywords**

“Cardinal Directions”, “linear time”, “chessboard”

# Introduction

This is a paper regarding a queen put in place on an n x n chessboard. The queen can be anywhere on the board. Obstacles can be added to the board to block the queen. A queen cannot move past an obstacle, limiting the number of spaces she can move. The output should be the total number of spaces the queen can attack.

# Problem Statement

# In the problem, a single queen is placed on an n x n chessboard. The rows are numbered 1 through n, bottom to top and the columns are numbered 1 through n left to right. The queen’s position can be described as an ordered pair (R, C). R is the row the queen is on and c is the column number. The queen can move though standard chess rules, up, down, left, right, or diagonally in any direction.

# Up to n^2 – 1 obstacles may be placed on the board, also described in ordered pairs. The goal is to print the number of squares that the queen can move to. In this example, it is 13.

**Inputs:**

* **n x n chessboard.**
* **k obstacles.**
* **R C, the position of the queen in rows, columns.**
* **P and Q, the obstacles on the board.**

**Output:**

* **number of squares the queen can safely attack.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **6** |  | O |  |  |  |  |
| **5** |  |  |  |  |  |  |
| **4** |  |  |  | O |  |  |
| **3** |  | Q |  |  | O |  |
| **2** |  |  |  |  |  |  |
| **1** |  | O |  |  |  |  |

**\***R, C is row, column of the queen. In this case it would be (3,2). P, Q is in the same format

# Analysis

The queen can move horizontally, diagonally, and vertically, meaning there are 8 different movement directions (hereafter called paths) she can move across. Let’s use cardinal directions:

* North = Vertical Upwards
* South = Vertical Downwards
* West = Horizontal Left
* East = Horizontal Right
* Northeast = Diagonal right upwards
* Southeast = Diagonal right downwards
* Southwest = Diagonal left downwards
* Northwest = Diagonal left upwards

As far as obstacles are concerned, we only care about the closest obstacle along a certain “path”. Secondarily, we only care about obstacles that are along the queen’s path. Therefore, the only obstacles that matter are the closest ones in a certain cardinal direction.

An obstacle is in the queen’s path if it shares certain properties. Say for example the queen is located at (4,3), and an obstacle is located at (6,5). We can determine that the obstacle is in the queen’s diagonal path by subtracting the respective rows and column. 4-6 is -2, and 3-5 is -2. If the two numbers are the same, this means the obstacle is in a diagonal path. We can determine the distance away by taking the absolute value of the difference. Abs (-2) is 2, so the obstacle is 2 squares away. This means there is (2-1) spaces the queen can attack before running into this obstacle.

We can do a similar thing with both horizontal and vertical obstacles. If an obstacle shares the same column with the queen, it is either above or below her. If it shares the same row, it is either to the left or right. Subtracting the difference in either columns or rows and subtracting 1 will result in the distance in squares between the queen and an obstacle.

To reiterate, we only care about the closest obstacle. Obstacles that are further away down a “path” than a previous obstacle have no bearing on the output.

A brute force approach would create a two-dimensional grid and then move the queen to determine the absence or existence of an obstacle. The queen could move in the 8 directions, and the results examined. This would be quadratic time, which is far too slow.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **6** |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |
| **4** |  |  |  |  |  |  |
| **3** |  | Q |  |  | O | O |
| **2** |  |  |  |  |  |  |
| **1** |  |  |  |  |  |  |

\*The obstacle in (3,6) does not matter, because the queen will be stopped by the obstacle at (3,5).

# Algorithm

The algorithm I am using to solve this problem could be considered an iterative algorithm. It is Θn time, n being the number of obstacles given.

The algorithm examines the obstacles in reference to the queen’s position. We only need to know the first obstacle in a certain path, so the algorithm throws out obstacles further than a previous obstacle.

The algorithm is iterative, it goes through the list of obstacle ordered pairs. First, it checks if the obstacle is on one of the queen’s paths, a requirement for altering the output. Next it checks if the obstacle is closer than a previously identified obstacle. If both are true, then the algorithm updates the moveable spaces of the queen in that direction to the distance between the queen and the current obstacle.

At the end of the loop, if a certain direction has no obstacles, the algorithm adds up the distance between the queen and the edge of the board in all directions without obstacles. Finally, the algorithm adds up the total number of moveable spaces for an output.

The algorithm passes all the hacker rank tests.

## Pseudocode

algorithm Queenattack(*n*)

**Output:**

* **An integer number of squares the queen can attack.**

Map <- Map(direction, pair)

*Direction* <- (*N,NE,E,SE,S,SW,W,NW*)

For dir in *direction* do

Map[dir] <-(0,0)

For i in 1 to m:

If obstacle [i] lies along direction and obstacle[i] is closer to map[d] than map[d] then map[d] <- obstacle[i]

Moves <- 0

For D in *directions* do

Moves <- moves + number of squares between (R,C) and end of map[d]

Return moves

## Time and Space Complexity

The algorithm is linear time in the number of obstacles. The more obstacles, the longer it takes to run. Its space complexity is constant. The map contains the obstacle values we care about.

# Example

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **6** |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |
| **4** |  |  |  |  | O |  |
| **3** |  | Q |  | O | O |  |
| **2** |  | O |  |  |  |  |
| **1** |  |  |  | O |  |  |

Here is a sample problem. The Queen is in position (3,2). The obstacle set is {(2,2), (1,4), (3,5), (4,5), (3,4)}.

(2,2) is south of the queen. south is zero.

(1,4) is southeast of the queen, southeast is 2-1, so 1

(3,5) is east. East is 2

(4,5) does not fit any criteria

(3,4) is east, East is 1

South=0

East = 1

Southeast = 1

Now the program calculates distance from the queen to the edge of the board in the remaining directions.

North = 3

West = 1

Southwest = 1

Northeast = 3

Northwest = 1

The answer should be 11.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **6** |  |  |  |  |  |  |
| **5** |  |  |  |  |  |  |
| **4** |  |  |  |  | O |  |
| **3** |  | Q |  | O | O |  |
| **2** |  | O |  |  |  |  |
| **1** |  |  |  | O |  |  |

This is correct if you count all the grey spaces.

# Implementation

The program is written in c++ because of its easily accessible for and if operators. It also uses vectors to hold the obstacles. The program would work equally well in c with arrays. The program contains a for loop with 8 nested if statements, and 8 statements after that. Currattack is both calculated and returned at the end of the function.

The number of spaces between the queen and an obstacle is calculated differently depending on which direction we are dealing with. For the south direction, for example, the number of squares between a south obstacle and the queen is the obstacle’s row – queen’s row – 1.

The north and south directions check if the obstacle is in the same column as the queen, and whether it is a greater of lesser row. The west and east do the same except the opposite. The distance between the queen and the obstacle is stored in a variable such as *South* or *North*.

Diagonals are calculated by taking the absolute value of the row of and queen minus the row of the obstacle and making sure it is equal to the absolute value of the difference between the queen’s column and the obstacles column. The distance between the two is stored

in *Northwest* for example. If the iteration encounters a lower distance between the queen and an obstacle, the variable is updated.

At the end, if the Variable which holds the distance for each direction has not changed, the distance between the queen and the edge of the board in that direction is calculated, and the variable is updated to that value.

Finally, the direction variables are tallied up and returned.

# ConclusionS

This is an extremely efficient algorithm and is rather simple. By defining the algorithm in terms of the obstacles, we eliminate unnecessary computation. The obstacles only eliminate paths the queen can go across, limiting the total number of available spaces.

# REFERENCES

1. HackerRank: 2020. https://www.hackerrank.com/. Accessed 2020-04-08.
2. <http://dx.doi.org/10.1016/j.jss.2005.05.030>.
3. Spector, A. Z. 1989. Achieving application requirements. In *Distributed Systems*, S. Mullender, Ed. ACM Press Frontier Series. ACM, New York, NY, 19-33. DOI= <http://doi.acm.org/10.1145/90417.90738>.