Alex Webster - Technical Document + Code
Jessie Lu - Technical Document
Hatim Belhadjhamida - Technical Document + Code

N-Queens Problem

Introduction

This report summarizes the process and results of which a solution to the N-Queens problem was developed. This specific version of this N-Queens problem required a solution such that no two queens shared a row, column or diagonal, and that 'n' or the dimensions of the board ranged from 4 to 1,000,000 inclusively. There were various maximum runtimes depending on the size of the board, with 30 min being the time limit on the 'Large Difficulty' upper bounds of n. To minimize run-time to solve this problem, various algorithmic approaches were attempted and improved upon. Improvements focused on but were not limited to changes made to the method of which the chess board was initialized, and how conflict-data was stored. Results showed that using the minimum-conflict method to placing the queens on the chess board and using multiple encoded lists resulted in the lowest run-time.

1 - Process

When we first attempted the N-Queens problem we wanted to keep it simple, and clean. Our first iteration of the algorithm was very slow, however we knew there was room for improvement. In an attempt to make our algorithm more efficient we focused on how the chess board was initialized, and the queen's conflict data structure efficiency.

1.1 - Data Structures

We realized the representation of the csp data structure would be critical to reaching desired performance metrics. Our first attempt was a simple list structure where list[column] = row. This required constant recalculation of conflicting variables because it provided no information of the current states conflicts (the csp constraints). This launched an investigation into data structures that would be more efficient.

Graph

To increase performance we next choose a graph data structure where the vertices represented queen position and edges were conflicts.

$$csp = G(V, E),\ V[column] = row \text{ and } E[column] = set(\text{columns that can attack column})$$

The intuition being that, through graph edges we could maintain the current conflicts with little overhead. In addition, using a set for the edges we could have O(1) insert, delete and search operations when looking for conflicts and maintaining structural integrity. We later realized that this was an ineffective way of representing conflicts, as it was unnecessary to know the specific queen's that

conflicted. We required a structure that allowed simple checks to calculate conflicts at any position on the board.

Multiple Encoded Lists - List representing queens on all attack vectors **(Final structure)**

After careful consideration, we realized the number of attacking vectors (rows, columns, forward and backwards diagonals) increases with linear time. This meant our space complexity is reasonable and we could encode the indexes of lists in a way that we can find the amount of conflicts at any position in constant time (with a constant time overhead to keep integrity). Specifics about the structure are commented in the code, but for summary we have 3 lists:

1. A list for conflicts on each row
2. A list for conflicts on each forward diagonal (\)
3. A list for conflicts on each backward diagonal (/)

1.2 - Board Initialization

What we failed to realize until our final iteration is the importance of the chess board initialization. In our first attempt we placed the queen's on separate columns, and randomized their placement on the rows. It was very simple and easy to implement, however it hurt our algorithm's performance by a significant amount. This is caused by the small percentage of board configurations that converge to the global maximum of 0 conflicts instead of some local minimum (Figure 1). Once we noticed these effects on the performance we knew we had to make improvements.

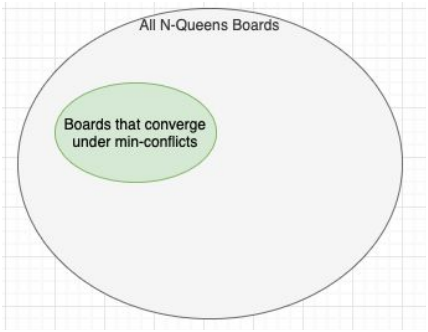

Figure 1 - The green represents boards that reach solutions under min-conflicts vs all possible board initialization states. This shows a random algorithm simply isn't enough and a greedy approach is required.

After much discussion we agreed to minimize the conflict on each queen placement during initialization. This would entail finding the least conflicting position between the rows on each column and placing the queen. Once this was implemented we immediately saw a drastic increase in performance. Before it took our algorithm around 30 seconds to compute a chess board of size 10,000. After our improvements it took 30 seconds to compute a size of 100,000.