CP468

Final Project (N Queens)

Group 7

Zongyang Li (lixx7236@mylaurier.ca, 180272360)
Jeetindra Dhori (dhor6000@mylaurier.ca, 180726000)
Khushi Satra (satr3100@mylaurier.ca, 203383100)
Xiaocong Lian (lian7210@mylaurier.ca, 191717210)
Rini Perencsik (pere4930@mylaurier.ca, 191504930)

December 11, 2022

Outline

Discussion of design choices

N-queens representation

To represent a state in N-queens, it was our goal to represent all the necessary information in as little space possible. We fixed a given queen to each of the N columns. Therefore, a 1xN vector was used such that each element represents the row number in that given column. More specifically, if the 1xN vector was named puzz, the queen in the ith column would be placed in the puzz[i]th row. A 1 dimensional vector was chosen as opposed to an NxN binary representation of N-queens in order to be more space efficient.

To represent each possible conflict, three data structures were used:

- Row vector (1xN) such that the ith element represents the number of queens in row i
- Diag1 vector (1x2N-1) such that the ith element represents the number of queens in the ith diagonal in the set of diagonals beginning at the top right corner
- Diag2 vector (1x2N-1) such that the ith element represents the number of queens in the ith diagonal in the set of diagonals beginning at the top left corner

We chose this design because we felt that it was able to represent all the necessary information about a queen's positioning and conflicts in the least amount of space. These four data structures are used together through an object oriented programming design. We took an OOP approach to the problem so that we could easily call methods to reassign queens at any point in the program. Using an OOP design also helped in our organization of the code through the use of classes.

For an example of a given state, we have that the queen in column i, row puzz[i] is in row conflict with row[puzz[i]] number of queens, diagonal # 1 conflict with diag1[i+puzz[i]] number of queens, diagonal #2 conflict with diag1[i-puzz[i]+N] number of queens, and column conflict with 0 queens.

Since N-queens is framed as a constraint satisfaction problem (CSP), these three data structures essentially store constraint information about each queen. If a queen is not violating any row, diag1, and diag2 constraints then the queen will have a value of 1 at its corresponding position in each of these vectors.

An N-queens is solved when the entire diag1, diag2, and row vectors have no values above 1.

Initial state creation

The initial state was chosen by a greedy assignment process that assigns a queen to a row with the minimum number of conflicts. This was done so to reduce the number of steps that the min conflicts algorithm had to run.

Language choice

We initially wrote the program in python and were able to solve up to N = 100,000. However, we realized that for N=1M, the program would take several days to complete. As a result, we rewrote the program in C++ which greatly improved run time.

Optimization challenges

A challenge we faced throughout the entire project was efficiency in time and space, especially time. We initially wrote the program in python, but we did not believe that it would solve N=1M in a reasonable amount of time, so we rewrote the program in C++ to solve this challenge. The second challenge was that it was taking too many steps (and thus too much time) to find a solution for large N. We knew that it should take an average of 50 steps to find a solution, independent of N size, so we were initially confused on why it was taking so many steps. After further research, we found that it takes an average of 50 steps after a good initial state had been found. Therefore, we wrote a method that would build a good initial state so that a smaller number of steps (closer to average of 50) was required to solve the problem.

Results of test runs for n = 10, 100, 10000, 100000, 1000000

Machine specs used to generate results

Hardware Overview:

Model Name:	MacBook Pro
Model Identifier:	MacBookPro17,1
Chip:	Apple M1
Total Number of Cores:	8 (4 performance and 4 efficiency)
Memory:	8 GB
System Firmware Version:	6723.81.1
Serial Number (system):	FVFFG9NVQ05D
Hardware UUID:	930C556E-166F-5542-9866-0E06127FD45D
Provisioning UDID:	00008103-00182920118A001E
Activation Lock Status:	Disabled

N = 10

```
N: 10
Initial state created in 0.000146 seconds
After initial state creation, solution found in 0.001758 seconds
Total duration: 0.001918 seconds
Number of steps: 44
```

N = 100

```
N: 100
Initial state created in 0.002099 seconds
After initial state creation, solution found in 0.004255 seconds
Total duration: 0.006371 seconds
Number of steps: 113
```

N = 10,000

```
Assigned 9000
Initial state created in 3.02511 seconds
After initial state creation, solution found in 0.017272 seconds
Total duration: 3.0424 seconds
Number of steps: 42
```

N = 100,000

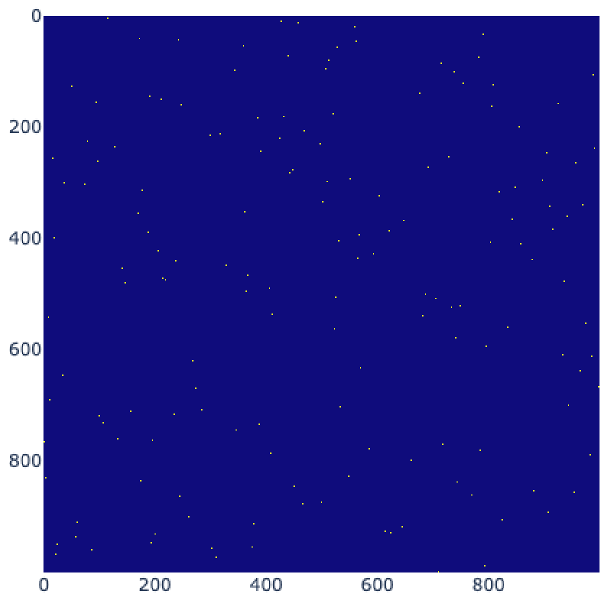
```
Initial state created in 298.04 seconds
After initial state creation, solution found in 0.496 seconds
Total duration: 298.536 seconds
Number of steps: 142
```

N = 1,000,000

```
Assigned 999000
Initial state created in 29742.5 seconds
After initial state creation, solution found in 5.0688 seconds
Total duration: 29747.6 seconds
Number of steps: 104
1
```

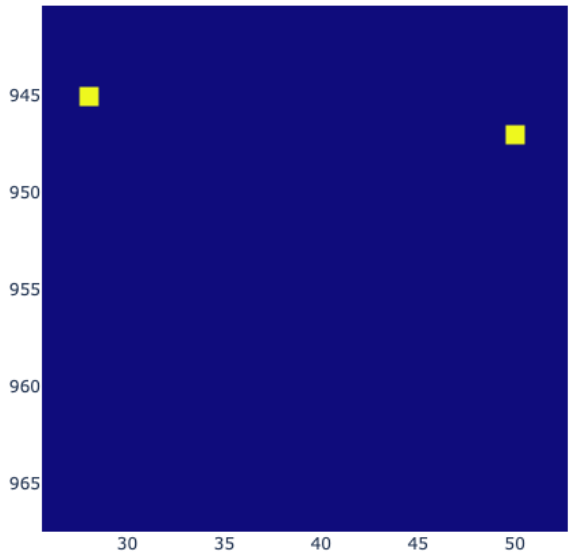
To see the actual output of the program, view the output_N.txt files for a list of row numbers that the queen in column i is assigned to. When looking at the amount of time to create a good initial state for the min conflicts algorithm, we can see that the time increases relatively quickly with N. The time it takes to create a good initial state is low at 0.000153 seconds for N=10 and very high at 29742.5 seconds (8.2 hrs) for N = 1M. This increase in time makes sense since the search for an optimal state is O(N^2) since for every column (xN), we iterate through every row (xN) to find the one with the minimum number of conflicts, which itself takes constant time. However, we can see that once a good initial state has been reached, the actual time it takes to find a solution is relatively quick, even for a large N (5 seconds for 1M). This is because the time it takes to find a solution is O(max_steps*N) = O(N). This linear increase in time is shown in the images above. Amazingly, after a good initial state has been built, there is research that shows that only an average of 50 steps is needed and it becomes independent of problem size. This appears to be true for our program as well seeing that N=1M does not require particularly more steps than any other N. Since the jump from 100,000 to 1M was more than 8 hours, we assume that the program we wrote cannot solve for N larger than 1M because the initial state creation takes too long. However, if a good initial state was passed to the min conflicts algorithm, then it can find a solution in a few seconds, regardless of N size.

Graphical representation



For a graphical representation of a solution found with our code, we decided to create a multichannel image data using a graphing python library named plotly. The image above is the graphical representation of a solution our code found for n = 1,000. The yellow squares represent a queen's placement on the board and a blue square represents an empty square. To create this image, we converted the program's output of a list of the row numbers for each queen in a column to a 2D nxn list such that 0 represents an empty space and 1 represents the presence of a queen at position i,j on the chess board.

This poster is also interactive. If you suspect that two points have been placed on the same row, column, or diagonal, you can zoom into those points to verify. For example, let's say that we suspect that the two points near 0,1000 are on the same row. After zooming in using the interactive feature, we can see that they are actually on different rows:



How to compile code

The steps to running the code for N-queens are as follows:

1. Specify the output file that you would like the results written to in the main. The default is ‘output.txt’
2. Compile n_queens.cpp to create an executable file named n_queens.exe
3. Run the executable file n_queens.exe
4. The program will wait until you specify an n as user input in console after ‘N:’
5. The program output will be in ‘output.txt’ which will have a list of the row number that each queen is in for a given column

Note: if you do not have stdc++.h, copy the contents of this code and place inside a folder named ‘bits’ and place that folder in the path where C++ libraries are stores in your computer

The steps to running the graphical representation of a solution are as follows:

1. compile and run C++ program which will create a output_n.txt’ file with the a list of row positions for each column’s queen needed to create the visual
2. upload the file ‘CP468_project_poster.ipynb’ into Google colab
3. upload the grid.txt file into your google drive
4. Run each code chunk in ‘CP468_project_poster.ipynb’

Code

```
#include <bits/stdc++.h>
#include <fstream>
```

```
#include <random>
#include <ctime>

using namespace std;

class NQueenCSP {
   public:
   // Size of puzzle and max_steps for min conflicts algo
   int N, max_steps;
   // Will store how many queens are in each diagonal and row
   vector<int> diag1;
   vector<int> diag2;
   vector<int> row;
   // How the puzzle is represented - puzz[i] = the row # of the queen in col i
   vector<int> puzz;
   mt19937 rng;

   std::clock_t start;
   float is_creation_duration;

   NQueenCSP(int _N, int _MS): N(_N), max_steps(_MS) {
       /*!
       Attributes
       ----------
       puzz: int vector
           where each element represents the row number in that given column
       row: int vector
           where the ith element represents the number of queens in row i
       diag1: int vector
           where the ith element represents the number of queens in the ith diagonal
in the set of diagonals beginning
           at the top right corner
       diag2: int vector
           where the ith element represents the number of queens in the ith diagonal
in the set of diagonals beginning
           at the top left corner



       Methods
       -------
       init_cnts()
```

```
        initializes puzz, row, diag1, diag2 to all 0s since C++ wont allow us to do
it in the constructor
    conflicted()
        returns a vector of queens in conflict in the current step
    solved()
        determines if the n-queens state is a solution
    n_conflicts(int i, int j)
        determines the number of unique conflict types for a given queen
    upd_vals(int i, int j, bool subtract)
        updates diag1, diag2, and row upon adding or removing a queen from a column
    select_col_to_change()
        randomly selects a queen to change the position of and removes that queen
from column
    reassign(int i)
        assigns queen to the row position with the least number of conflicts
    min_conflicts()
        runs min conflicts local search algorithm on the created initial state
    print_grid()
        prints an NxN grid of solution
    */

    // start timer for building initial state
    start = std::clock();
    init_cnts();
    // min conflicts is very sensitive to initial state
    // so we build initial state by assigning each queen to the row
    // that has the least conflicts so far
    diag1[0]=1;
    diag2[0 + N - 1]=1; // have to add N - 1 because i-j can be as small as -(N-1)
    row[0]=1;
    for(int i = 1; i < N; i++) {
        reassign(i);
        if (i%1000 == 0) cout << "Assigned " << i << '\n';
    }
    is_creation_duration =  std::clock() - start;
    cout << "Initial state created in " <<  (std::clock() - start) / (double)
CLOCKS_PER_SEC << " seconds" << endl;
  }

  void init_cnts() {
    /*!
    Description
```

```
        ----------
        Initializes row, diag1, diag2, puzz to all 0s to be called in constructor
        */

        // There are 2*n-1 diagonals for both sets of diagonals (top down and bottom
up)
        // there also n queens and n rows
        for(int i = 0; i < 2*N-1; i++) {
            diag1.push_back(0);
            diag2.push_back(0);
            if (i < N) row.push_back(0);
            if (i < N) puzz.push_back(0);
        }
        // seed the random number generator
        rng.seed(0);
    }

    vector<pair<int,int> > conflicted(){
        /*!
        Description
        ----------
        Determines all the queens in conflict in the current step

        Returns
        ----------
        ret: a vector of integer pairs
            list of conflicted queens where in a pair<i, j>, i is the column and j is
the row of queen
        */

        vector<pair<int,int> > ret;
        for(int i = 0; i < N; i++) {
            int j = puzz[i];
            // if there is more than one queen in either of the diagonals this queen is
in
            // or the row it is in, this queen is conflicted so we add it to the list
            if ((diag1[i+j] > 1)|(diag2[i-j+N-1] > 1)|(row[j] > 1))
ret.push_back(make_pair(i,j));
        }

        return ret;
    }
```

```
  bool solved() {
      /*!
      Description
      ----------
      If the list of conflicted positions is empty, the puzzle is solved

      Returns
      ----------
      true: if a valid n-queens assignment has been found
      false: otherwise
      */
      return conflicted().empty();
  }

  int n_conflicts(int i, int j) {
      /*!
      Description
      ----------
      Determines the number of unique conflict types for a queen in column i, row j

      Returns
      ----------
      integer: number of unique conflict types for a queen
      */


      // Conflicts that come from the same direction (same diag or row) are only
counted once
      // so we take advantage of the fact that for an int, (bool)i = True if i > 0
and 0 if i == 0
      // and also that bools are really just 0s and 1s so can be treated as such in
c++
      return (bool)diag1[i+j] + (bool)diag2[i-j+N-1] + (bool)row[j];
  }

  void print_grid(){
      /*!
      Description
      ----------
      writes an NxN grid of solution where Q represents a queen and _ represents a
blank space to file grid_vix.txt
```

```
        */
        ofstream fh;
        fh.open("grid_viz.txt", ios::trunc);
        for(int col = 0; col < N; col++) {
            for (int row = 0; row < N; row++) {
                vector<int>::iterator itr = find(puzz.begin(), puzz.end(), col);
                int index = distance(puzz.begin(), itr);
                if (index == row){
                    fh << "Q ";
                } else {
                    fh << "_ ";
                }

                }
            fh << endl;
            }
        fh.close();
        }


    void upd_vals(int i, int j, bool subtract) {
        /*!
        Description
        ----------
        Adds 1 to the queens diag1, diag2, and row upon adding a new queen (chosen
strategically by reassign())
        Subtratcs 1 to the queens diag1, diag2, and row upon removing an old queen
(chosen randomly by select_col_to_change())
        */

        diag1[i+j] += !subtract - subtract;
        diag2[i-j+N-1] += !subtract - subtract;
        row[j] += !subtract - subtract;
    }

    int select_col_to_change() {
        /*!
        Description
        ----------
        Takes a randomly conflicted queen and removes it from the board, and then
return swhich queen we chose
```

```
        Returns
        ----------
        integer: column number of queen to be changed
        */

        // rng() produces a big number (bigger than 1000000)
        // so if we take rng() % vector.size(), we will get a random number in the
range 0 .. vector.size()-1
        // - a random index of vector
        vector<pair<int,int> > conf = conflicted();
        pair<int,int> ch = conf[rng() % (int)conf.size()];
        upd_vals(ch.first, ch.second, 1);
        return ch.first;
    }

    void reassign(int i) {
        /*!
        Description
        ----------
        Determines the row number for the queen at column i which has the minimum
number of conlicts
        Updates diag1, diag2, row upon calling upd_vals and updates puzz internally
        */

        // Find the minimum number of conflicts that is achievable
        int mini = N+1;
        for(int j = 0; j < N; j++) {
            mini = min(mini, n_conflicts(i, j));
        }
        // Make vector of all rows for this col that achieve this minimum
        vector<int> choices;
        for(int j = 0; j < N; j++) {
            if (n_conflicts(i,j) == mini) choices.push_back(j);
        }
        // Randomly pick one and add it to the board
        int new_row = choices[rng() % (int)choices.size()];
        upd_vals(i,new_row,0);
        puzz[i]=new_row;
    }

    bool min_conflicts() {
        /*!
```

```
        Description
        ----------
        Runs min conflicts algorithm which makes assignments based on the minimum
number of clinflicts

        Returns
        ----------
        true: if a solution was found within max_steps number of steps
        false: if a solution was not found within max_steps number of steps (does not
mean it is unsolvable)
        */

        // While we have done less than max_steps iterations and the puzzle is not
solved,
        // run the min_conflicts algorithm
        for(int x = 0; x < max_steps; x++) {
            if (solved()) {
                cout << "After initial state creation, solution found in " <<
(std::clock() - is_creation_duration) / (double) CLOCKS_PER_SEC<< " seconds" << endl;
                cout << "Total duration: " << std::clock() / (double) CLOCKS_PER_SEC <<
" seconds"<< endl;
                cout << "Number of steps: " << x << endl;
                return 1;
            }
            int i = select_col_to_change();
            reassign(i);
            // print the state at each step
            // for(int i = 0; i < N-1; i++) {
            //     cout << puzz[i];
            // }
            // cout << endl;
        }
        return 0;
    }
};

int main() {
    cout << "N: ";
    int N; cin >> N;
    // Make instance of class with desired N (read from command line)
    NQueenCSP nqcsp = NQueenCSP(N, 1000);
    // Ensure that we have solved the puzzle (this line will output 1 if so)
```

```
    cout << nqcsp.min_conflicts() << endl;
    // Log our solved configuration to text file 'output.txt' in json format for list
    ofstream f;
    f.open("output.txt", ios::trunc);
    f << '[';
    for(int i = 0; i < N-1; i++) {
        f << nqcsp.puzz[i] << ',';
    }
    f << nqcsp.puzz[N-1] << ']';
    f.close();
    // only uncomment for n <= 100, otherwise use graphical poster representation tool
in google colab
    //nqcsp.print_grid();
}
```