Algorithms

Homework 3 Report

자유전공학부 김지원
2019-11563

1. Describe how your algorithm works and how you made your algorithms as efficiently as possible.

   First, the iterative algorithm iterates from board[1][1] to board[N][N] and places queens on each cell. When placing on each cell, do a safety check with *is_cell_safe* function. This function checks this cell is a tree/ queen/ blank. Then, if it's a blank, it checks if this cell is safe from the other queen's attack. If it is safe, it places a queen here and checks the next cells. If the number of placed queens equals N, it increments 'result' and does back tracking by removing the lastly placed queen and try placing it from the next cell. Also, if the iteration reaches the dead end, by reaching board[N+1][1], it also does backtracking with the same method. There is a special case in which the algorithm places the last queen at board[N][N]. Since this is one of the possible solution, it has to pop one element, but since this is also the last possible state given N-1 queens placed, it needs to pop one more element.

   Second, the recursive algorithm uses *solveQueens* function to increment the static 'result' variable. The function has three inputs; row number, column number, and the number of queens placed so far. It increments 'result' and returns if queensPlaced equals N or if the matrix has reached a dead end. Otherwise, it calls *is_cell_safe* function to check if this cell is available. If it is safe, it places the queen in this cell and calls itself again this time with the next row and column number, and the number of queens incremented by 1. After, it unmarks the current cell to do back tracking to look for other solutions and calls itself again with the next row, column number but with the same number of queens placed.

   This algorithm uses additional data structures to run as efficiently as possible. It first uses row[], column[], diag[], anti_diag[] arrays to store the number of queens in that certain row/ column/ diagonal/ anti-diagnoal. Diagonal address is determined by (row) – (column) + N since all the cells on the diagonal have the same number. Also, anti-diagonal address is determined by (row) + (column) with the same reason. This makes it able to check **in constant time** if there are any queens present threatening this cell. This process is done in *get_queens* function. This function returns characters each 'r', 'c', 'd', and 'a' if any of the above array value is positive (means a queen is present). It checks for trees blocking the queen only to the row/ column/ diag/ anti-diag with a queen so it's more efficient than checking all 4 ways. Also, since the iteration starts at [1][1] and proceeds to [N][N], the **checking only is done upper or the left part** to reduce the time.

2. Compare the running time of iterative backtracking altorithm and that of recursive backtracking algorithm. (unit : ms)

    a. Different values of **N**, **K**

N	Iterative Backtracking	Recursive Backtracking
4	2.04	1.02
5	6.24	4.52
6	15.63	8.41
9	578.32	462.13
11	46752.12	49383.23

Time is calculated by measuring each iterative and recursive algorithm with same input. Also, With a certain N, both algorithms are run on different numbers of K from 0 to 3 and averaged by the end since different number of trees affect the running time. Each number is an average of 20 test practices each. (The recursive algorithm run faster after 3-4 first trials so the test cases are measured from the 4^{th}.)

As we can see, the recursive algorithm is faster than the iterative algorithm. Iterative algorithm takes more time when N is small. This is assumed to be due to the overhead of little comparisons and additional work done in iterative algorithm. However, as the N grows larger, iterative backtracking tends to be faster than recursive backtracking due to a lower overhead of function calls and stack management. Iterative backtracking uses an explicit stack to manage the search state which provides better control over the backtracking process and reduce the memory and performance impact of recursive function calls. While recursive backtracking is more concise and intuitive to implement, it relies on function calls and the associated call stack so this might impact the performance, especially for large search spaces or deep recursion levels.

3. Environment of the program, how to execute the program.

    Open jdk – 19

    Run the program by compiling the program ($ javac Main.java). Execute it with 3 command arguments. First argument with the version, second and last argument with each input and output file paths (ex.) $ java Main 1 ./input/1.in ./output/1.out)