Algorithms and Data Structures: Homework #13

Due on May 11, 2020 at 23:00

Henri Sota

Problem 13.1

Consider the n queens problem as discussed in the lecture. Solve the adapted version of the problem in which instead of placing n queens on an $n \times n$ chessboard, you need to place n horses, which do not threaten each other. n is a natural number and the only input of the problem. Your implementation should print all possible placements for a given input n.

My implementation of the backtrack solution to the problem of placing n horses on a $n \times n$ chessboard works by recursively finding empty or unmarked positions to place horses on the board. Every time such a position is found, a horse is placed and its possible attack positions are marked onto the board. The board is then copied to continue solving a similar problem with $n - 1$ horses, but starting from the position where a horse was just placed. The solution therefore tries to place horses in empty or unmarked positions until it runs out of horses to place or positions to place a horse.

```cpp
void solve(std::vector<std::vector<char>> board, int n, int xInit, int yInit) {
    // Check if n is 0 (All possible horses have been placed correctly)
    if (n == 0) {
        // Solution has been found, therefore print the board
        // Comment line in case only number of solutions is wanted
        printBoard(board);

        solutions++;
        return;
    }
    // Iterate through the board starting from position (xInit, yInit) until
    // position (boardSize - 1, boardSize - 1)
    for (int i = xInit; i < (int) board.size(); ++i) {
        for (int j = yInit; j < (int) board.size(); ++j) {
            // Check if position (i, j) is empty or not under attack on board
            if (checkPosition(board, i, j)) {
                // Copy board to try to find solutions for placing a horse
                // on position (i, j)
                std::vector<std::vector<char>> testBoard = board;
                // Place horse on the empty position
                testBoard[i][j] = 'H';
                // Place positions under attack after placing the new horse
                setAttackPositions(testBoard, '*', i, j);
                // Find solutions (if any) to place the leftover n - 1
                // horses in the positions after (i, j)
                solve(testBoard, n - 1, i, j);
            }
        }
        // Set the yInit to 0 after checking the first row (and other rows),
        // in order to all leftover positions
        yInit = 0;
    }
}
```

Listing 1: n Horses on a n x n chessboard Backtracking Solution Implemented in C++

Implementation can be found in the file "nHorsesBacktracking.cpp", inside the "nHorsesBacktracking" directory. Line 108 can be commented out in order for the program to only print the number of solutions or the output can be dumped into a file for simplicity.

Problem 13.2

Use the textbook "Introduction to Algorithms" by Cormen et al. or other equivalent resources to learn about the Rabin-Karp algorithm.

a) Explain the algorithm using an example (different from the example in the textbook) and explain the steps which would be performed by the algorithm.

Rabin-Karp is a string-searching algorithm that makes use of hashing to find matches between the pattern and the substrings of the the text. It starts by calculating the hash value of the pattern with length M and the hash value of the substring of length M starting at the first position of the text, $T[0 \dots M-1]$. It checks if the hash values match, and if they do, it performs character-by-character comparison (as Naive String Search performs on each iteration) to find out if the pattern matches the substring, and if they don't, it shifts onto the next substring, by only requiring a constant time operation to calculate the new hash value of the next substring, $T[1 \dots M]$. The algorithm performs this check until the last substring with length M, substring $T[N-M \dots N]$.

The most important part of the algorithm is the hash function, which is responsible for minimizing the amount of collisions created by spurious hits, false matches due to the match of the hash value between substring and pattern.

Example:
Setting text to be *saabo* and setting the pattern to be *aab*

$$text = \text{saabo} \qquad pattern = \text{aab}$$

Length of the pattern, m, is 3 and length of the text, n, is 5. The difference $n-m$ is 2. Setting $d = 256$ and choosing a q to be a prime number, 97 (25th prime number). Calculating h using the following formula:

$$h \equiv d^{m-1} (\bmod q) = 256^2 (mod 97) = 61$$

Calculating the hash value for the pattern and the first substring with length m using formulas for 3 iterations:

$$p = (dp + P[i]) \bmod q$$
$$t_0 = (dt_0 + T[i]) \bmod q$$

First Iteration	$p = 0$	$t = 18$
Second Iteration	$p = 0$	$t = 49$
Third Iteration	$p = 1$	$t = 31$

The loop that iterates through all possible contiguous substrings with length m will iterate 3 times in total, checking for the substring starting at position 0, 1, and 2. During the first iteration, the hash values do not match therefore, the new hash value for the coming substring will be calculated using the formula:

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q$$

where t_s represents the old hash value, $T[s+1]$ represents the ASCII value of the first digit of the old substring and $T[s+m+1]$ represents the ASCII value of the digit to be added onto the substring. This calculation $t_{s+1} = (256*(31-115*61)+98) \bmod 97 = 1$ (after normalizing by adding the positive value q). Therefore, on the next iteration, the hash values of the pattern and the new substring s_1 will match and the algorithm will check character-by-character if pattern appears at this shifted position. In this case it does and the algorithm outputs that it has found a match on shifted position 1.

Moving onto the next substring and performing the same procedure to calculate the new hash value for this position, $(256 * (31 - 97 * 61) + 111) \bmod 97$, yields a negative value which after adding q is 76. The hash values do not match and therefore no other match is found. The algorithm terminated and found only one match of the pattern "aab" inside the text "saabo".

b) Implement the algorithm.

```cpp
void rabinKarpMatcher(std::string T, std::string P, int d, int q) {
    int n = T.length();
    int m = P.length();

    int h = 1;
    // Calculate h to the value of the high-order digit position of an m-digit
    // window
    for (int i = 0; i < m - 1; ++i)
        h = (h * d) % q;

    int p = 0;
    int t = 0;

    // Iterate through the length of the patter and the first substring from
    // text, T[0 ... m - 1]
    for (int i = 0; i < m; ++i) {
        // Calculate hash value for the pattern and the first substring
        p = (d * p + P[i]) % q;
        t = (d * t + T[i]) % q;
    }

    // Iterate through all possible contiguous substring from starting position
    // 0 to n - m
    for (int s = 0; s <= (n - m); ++s) {
        // Check if hash values match for the pattern and substring
        if (p == t) {
            int j;
            // Check character-by-character match of pattern and substring
            for (j = 0; j < m; ++j)
                // Stop checking in case mismatch occurs
                if (T[s + j] != P[j])
                    break;
            // Check if mismatch occured on the m + 1th character (beyond size)
            if (j == m)
                // Meaning check went through and no mismatch occured
                // Therefore pattern and substring match
                std::cout << "Pattern occurs with shift " << s << std::endl;
        }
        // Calculate new hash based on the old hash
        if (s < (n - m)) {
            t = (d * (t - (T[s] * h)) + T[s + m]) % q;
            // Check if new hash value became negative
            if (t < 0)
                // Add a number equal to modulo q to make it positive and
                // maintain remainder value
                t += q;
        }
    }
}
```

Listing 2: Rabin Karp Algorithm Implementation in C++

Implementation can be found in the file "rabinKarp.cpp", inside the "rabinKarp" directory. Customized input can be given by uncommenting line 21.