Name : Sonal Dilip Gholap

UID : 2021600020

Branch : CSE-AIML

Batch : B

Experiment : 7

Aim : Experiment on Backtracking Algorithms

- **Objectives :**

1. To study Backtracking Algorithms like Sum of Subset problem and N-Queen problem.
2. To implement Sum of Subset problem and N-Queen problem using backtracking.
3. To derive and analyze the time complexity of Sum of Subset problem and N-Queen problem.

- **Theory :**

1. Backtracking is a problem-solving technique that involves building a solution incrementally, one step at a time, and testing each partial solution to see if it satisfies the problem constraints. If a partial solution violates a constraint, the algorithm backtracks to the previous step and tries a different option.
2. Backtracking is often used to solve combinatorial problems, such as finding all possible permutations or subsets of a set, or finding a path through a maze.
3. The backtracking algorithm terminates when either a valid solution is found, or all possible options have been explored without finding a valid solution.
4. Backtracking algorithms have exponential time complexity in the worst case. This is because the number of possible solutions that need to be explored grows exponentially with the size of the input.
5. Every constraint satisfaction problem which has clear and well-defined constraints on any objective solution, that incrementally builds candidate to the solution and abandons a candidate (“backtracks”) as soon as it determines that the candidate cannot possibly be completed to a valid solution, can be solved by Backtracking.

6. Backtracking is a fundamental technique in computer science and is used in many areas of research, including artificial intelligence, operations research, and computer graphics. It is a powerful tool for solving complex problems that involve searching a large space of possible solutions.

- **Sum of Subset problem**

Pseudocode for Sum of Subset problem

```
subsetSum(arr, targetSum, currSubset, currSum, currIndex) :

    if currSum == targetSum then do
        for num in currSubset:
            print num
        return

    if currIndex >= arr.size() then
        return

    currSubset.push_back(arr[currIndex])
    subsetSum(arr, targetSum, currSubset, currSum + arr[currIndex], currIndex + 1)

    currSubset.pop_back()
    subsetSum(arr, targetSum, currSubset, currSum, currIndex + 1)
```

Derivation of Time Complexity

The time complexity of the subsetSum function depends on the number of recursive calls made and the time taken for the operations inside the function.

In this function, there are two recursive calls for each element in the input array. Therefore, the total number of recursive calls made is 2^n, where n is the number of elements in the input array.

For each recursive call, there are two operations performed: adding an element to the current subset (push_back) and removing an element from the current subset (pop_back). Both of these operations have a constant time complexity of O(1).

Therefore, the overall time complexity of the subsetSum function is O(2^n), where n is the number of elements in the input array. This is because the number of recursive calls grows exponentially with the size of the input array.

Code :

```
#include <iostream>
#include <vector>

using namespace std;

void subsetSum(vector<int>& arr, int targetSum, vector<int>& currSubset, int currSum, int
currIndex) {
    // Base case: if the current sum is equal to the target sum, print the current subset
    if (currSum == targetSum) {
        cout << "Subset with sum " << targetSum << ": ";
        for (int num : currSubset) {
            cout << num << " ";
        }
        cout << endl;
        return;
    }

    // If the current index exceeds the size of the array, backtrack
    if (currIndex >= arr.size()) {
        return;
    }

    // Consider the current element and backtrack
    currSubset.push_back(arr[currIndex]);
    subsetSum(arr, targetSum, currSubset, currSum + arr[currIndex], currIndex + 1);

    // Exclude the current element and backtrack
    currSubset.pop_back();
    subsetSum(arr, targetSum, currSubset, currSum, currIndex + 1);
}

int main() {

    int n;
    cout<<"Enter the length of the array: ";
    cin>>n;
    vector<int> arr;
```

```
    cout<<"Enter elements of the array: ";
    for(int i=0; i<n; i++){
        int a;
        cin>>a;
        arr.push_back(a);
    }

    int targetSum;
    cout<<"Enter target sum: ";
    cin>>targetSum;
    vector<int> currSubset;
    subsetSum(arr, targetSum, currSubset, 0, 0);
    return 0;
}
```

Output :

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptoo
ls-1.15.1-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In
-at2rzbz0.kii' '--stdout=Microsoft-MIEngine-Out-eoiszaol.3vd' '--stderr=Microsoft-MIEngine-Err
or-czqw2rri.cwb' '--pid=Microsoft-MIEngine-Pid-1jtukvmi.0jv' '--dbgExe=C:\Program Files (x86)\
mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Enter the length of the array: 5
Enter elements of the array: 1 3 4 5 6
Enter target sum: 9
Subset with sum 9: 1 3 5
Subset with sum 9: 3 6
Subset with sum 9: 4 5
PS C:\C++ Learning Course>
```

- **N-Queen problem**

Pseudocode for N-Queen problem

isSafe(board, row, col, n):

```
for i from 0 to col-1 do:
    if board[row][i] == 1:
        return false
```

```
    for i from row to 0 do:
        for j from col to 0 do:
            if board[i][j] == 1:
                return false

    for i from row to n-1 do:
        for j from col to 0 do:
            if board[i][j] == 1:
                return false

    return true
```

solveNQueens(board, col, n):

```
    if col == n then:
        return true

    for i from 0 to n-1 do:
        if isSafe(board, i, col, n) then:
            board[i][col] = 1

            if solveNQueens(board, col+1, n) then:
                return true

            board[i][col] = 0

    return false
```

Derivation of Time Complexity

Recurrence relation for N-Queen

$T(n) = nT(n-1) + O(n^2)$

$T(n-1) = (n-1)T(n-2) + O(n^2)$

$T(n-2) = (n-2)T(n-3) + O(n^2)$

$T(n-k) = (n-k)T(n-k-1) + O(n^2)$

Solving further,

$T(n) = n(n-1)T(n-2) + (n+1)O(n^2)$

$= n(n-1)(n-2)T(n-3) + [n(n-1)+n+1]O(n^2)$

$= O(n!)$

Code :

```cpp
#include <iostream>
#include <vector>

using namespace std;

// Helper function to check if a queen can be placed in the current row and column
bool isSafe(vector<vector<int>>& board, int row, int col, int n) {
    // Check if there is a queen in the same row
```

```
    for (int i = 0; i < col; i++) {
        if (board[row][i]) {
            return false;
        }
    }

    // Check if there is a queen in the upper left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j]) {
            return false;
        }
    }

    // Check if there is a queen in the lower left diagonal
    for (int i = row, j = col; i < n && j >= 0; i++, j--) {
        if (board[i][j]) {
            return false;
        }
    }

    // If no queen is found to be attacking the current cell, return true
    return true;
}

// Function to solve the n-queen problem using backtracking
bool solveNQueens(vector<vector<int>>& board, int col, int n) {
    // Base case: all queens are placed, return true
    if (col == n) {
        return true;
    }

    // Try placing a queen in each row of the current column
    for (int i = 0; i < n; i++) {
        if (isSafe(board, i, col, n)) {
            // Place the queen in the current cell
            board[i][col] = 1;

            // Recursively solve the subproblem by placing the remaining queens
            if (solveNQueens(board, col + 1, n)) {
                return true;
            }

            // If no solution is found, remove the queen from the current cell
            board[i][col] = 0;
        }
    }

    // If no queen can be placed in the current column, return false
    return false;
}
```

```cpp
// Function to print the solution
void printSolution(vector<vector<int>>& board, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            cout << board[i][j] << " ";
        }
        cout << endl;
    }
}

int main() {
    cout<<"Enter N : ";
    int n;
    cin>>n;
    vector<vector<int>> board(n, vector<int>(n, 0));
    if (solveNQueens(board, 0, n)) {
        printSolution(board, n);
    } else {
        cout << "No solution found." << endl;
    }
    return 0;
}
```

Output :

```
PS C:\C++ Learning Course>  & 'c:\Users\Sonal Dilip Gholap\.vscode\extensions\ms-vscode.cpptools-1.15.1-win32-x64\debugAdapters\bin\WindowsDebugLauncher.exe' '--stdin=Microsoft-MIEngine-In-yktzaopx.ukt' '--stdout=Microsoft-MIEngine-Out-hqq0akwc.sbg' '--stderr=Microsoft-MIEngine-Error-dmqreswv.urg' '--pid=Microsoft-MIEngine-Pid-s0nxjgr2.s4b' '--dbgExe=C:\Program Files (x86)\mingw-w64\i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gdb.exe' '--interpreter=mi'
Enter N : 4
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
PS C:\C++ Learning Course>
```

Result :

1. Time complexity of Sum of Subset problem – **O(2^n)**
2. Time complexity of N-Queen problem – **O(n!)**

Conclusion :

1. Backtracking is a brute-force search technique that tries out all possible configurations of a problem until a correct solution is found.
2. Backtracking is a powerful technique for solving problems that require exhaustive search of a solution space.
3. Backtracking is a powerful algorithmic technique, but it can be expensive and time-consuming, especially for complex problems.