Backtracking

Backtracking is a refinement of the brute-force approach, which systematically searches for a solution to a problem among all available options. It is suitable for scenarios where there is a set of options available at each step, and we must choose one from these. After a choice is made, there is a new set of options for the next step. This procedure is repeated over and over until we reach a final state.

Conceptually, all options compose a tree structure. Leaves in the tree correspond to solution states, some of which might be the final acceptable state, but others might not. The backtracking algorithm traverses this tree recursively, from the root down and in depth-first order.

When it reaches a leaf that corresponds to a non-acceptable state, it backtracks to continue the search for another leaf by revoking the most recent choice and tries out the next option. If it runs out of options, it revokes again and tries another choice at that node. If it ends up at the root with no options left, there are no acceptable states to be found.

As shown in Figure 4-5, there are two options available at each step. The backtracking algorithm starts at the root node A, and it selects the node B from the two options. It chooses D for the second step, and the state is not acceptable. Therefore, it backtracks to node B and then selects the next option and advances to node E. It reaches a non-acceptable state again, and it has to return back to node B again. Since it has tried all options on node B, it returns back to the parent node A. It selects the next option of node A, which is node C. It then selects node F. It stops traversal since it reaches an acceptable state.

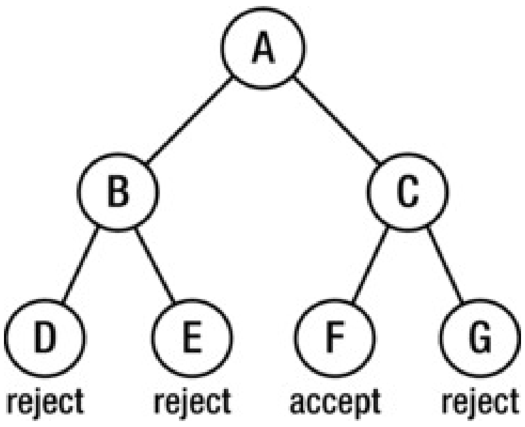

Figure 4-5. *Tree structure to visualize options for certain problems. There are two options available at each step.*

Usually the backtracking algorithm is implemented with recursion. When it reaches a node, it tries all options for the next step recursively if the state is not rejected.

String Path in Matrix

Question 30 How do you implement a function to check whether there is a path for a string in a matrix of characters? It moves to left, right, up, and down in a matrix, and a cell for a step. The path can start from any cell in a matrix. If a cell is occupied by a character of a string on the path, it cannot be occupied by another character again.

For example, the matrix below with three rows and four columns has a path for the string "bcced" (as highlighted in the matrix). It does not have a path for the string "abcb", because the first "b" in the string occupies the "b" cell in the matrix, and the second "b" in the string cannot enter the same cell again.

a *b c* e

s f *c* s

a *d e* e

It is a typical problem that can be solved by the backtracking algorithm. First, we choose any entry in the matrix to start a path. Suppose the character value in an entry is *ch* and the entry is for the i^{th} node on the path. If the i^{th} character in the string is also *ch*, it tries all options available at that entry for the next step. All entries except those on boundaries have four options. It continues until all characters in the string have been found on a path.

The path of a string can be defined as a stack because of the recursive nature of the backtracking algorithm. When the algorithm enters a rejecting state where the character on the path is not the same as the corresponding character in the string, it returns back to the preceding state and pops a character off the stack.

Because a path can enter an entry in the matrix only once at most, a set of Boolean flags are defined to mark whether an entry has been occupied by a previous step on the path.

The solution can be implemented with recursion, as depicted in Listing 4-14.

Listing 4-14. *C++ Code for String Paths*

```
bool hasPath(char* matrix, int rows, int cols, char* str) {
 if(matrix == NULL || rows < 1 || cols < 1 || str == NULL)
     return false;

 bool *visited = new bool[rows * cols];
 memset(visited, 0, rows * cols);

 int pathLength = 0;
 for(int row = 0; row < rows; ++row) {
        for(int col = 0; col < cols; ++col) {
            if(hasPathCore(matrix, rows, cols, row, col, str, pathLength, visited))
                    return true;
        }
    }

    delete[] visited;

    return false;
}

bool hasPathCore(char* matrix, int rows, int cols, int row, int col, char* str, int&
pathLength, bool* visited) {
    if(str[pathLength] == '\0')
        return true;

    bool hasPath = false;
    if(row >= 0 && row < rows && col >= 0 && col < cols
            && matrix[row * cols + col] == str[pathLength]
            && !visited[row * cols + col]) {
        ++pathLength;
        visited[row * cols + col] = true;

        hasPath = hasPathCore(matrix, rows, cols, row, col - 1, str, pathLength, visited)
                || hasPathCore(matrix, rows, cols, row - 1, col, str, pathLength, visited)
                || hasPathCore(matrix, rows, cols, row, col + 1, str, pathLength, visited)
                || hasPathCore(matrix, rows, cols, row + 1, col, str, pathLength, visited);

        if(!hasPath) {
            --pathLength;
            visited[row * cols + col] = false;
        }
    }

    return hasPath;
}
```

When an entry with index (`row` , `col`) is not a rejecting state, the function `hasPathCore` has a try at the four neighbors with index (`row` , `col` - 1), (`row` - 1, `col`), (`row` , `col` + 1), and (`row` + 1, `col`). It continues until it reaches the final acceptable state when the statement `str[length] == '\0'` gets `true` , which indicates that all characters in the string have been matched in a path.

Source Code:

`030_StringPath.cpp`

Test Cases:

- Functional cases: A 2-D matrix has/does not have a path for a string
- Boundary cases: (1) A 2-D matrix has only a row, a column, or even an element. (2) A path for a string occupies all elements in a matrix.

Robot Move

Question 31 A robot starts at cell (0, 0) of a grid with *m* rows and *n* columns. It can move to the left, right, up, and down, and moves one cell for a step. It cannot enter cells where the digit sum of the row index and column index are greater than a given *k*.
For example, when *k* is 18, the robot can reach cell (35, 37) because 3+5+3+7=18. However, it cannot reach cell (35, 38) because 3+5+3+8=19 and that is greater than *k*. How many cells can the robot reach?

Similar to the preceding problem, a grid can also be viewed as a 2-D matrix, where all cells except those on boundaries have four neighbors.
The robot starts to move from the cell (0,0). When it reaches a cell (*i*,*j*), we check whether it is a valid move according to the digits sum of the indexes. If the move is valid, we continue to move to the four neighboring cells (*i*,*j*-1), (*i*-1,*j*), (*i*,*j*+1), and (*i*+1,*j*). Therefore, the problem can be solved with the recursive code in Listing 4-15.

Listing 4-15. *C++ Code for Moving Robots*

```
int movingCount(int threshold, int rows, int cols){

    bool *visited = new bool[rows * cols];

    for(int i = 0; i < rows * cols; ++i)

        visited[i] = false;


    int count = movingCountCore(threshold, rows, cols, 0, 0, visited);


    delete[] visited;


    return count;

}


int movingCountCore(int threshold, int rows, int cols, int row, int col, bool* visited){

    int count = 0;

    if(check(threshold, rows, cols, row, col, visited)) {

        visited[row * cols + col] = true;


        count = 1 + movingCountCore(threshold, rows, cols, row - 1, col, visited)

                + movingCountCore(threshold, rows, cols, row, col - 1, visited)

                + movingCountCore(threshold, rows, cols, row + 1, col, visited)

                + movingCountCore(threshold, rows, cols, row, col + 1, visited);

    }


    return count;

}
```

Similar to the previous problem, a set of Boolean flags is defined in order to avoid duplicated moves.
The function `check` in Listing 4-16 checks whether a move to the cell (`row` , `col`) is valid, and the function `getDigitSum` finds the sum of all digits of a given number.

Listing 4-16. *C++ Code for Moving Robots*

```
bool check(int threshold, int rows, int cols, int row, int col, bool* visited){

    if(row >=0 && row < rows && col >= 0 && col < cols

        && getDigitSum(row) + getDigitSum(col) <= threshold

        && !visited[row* cols + col])

        return true;


    return false;

}


int getDigitSum(int number){

    int sum = 0;

    while(number > 0){

        sum += number % 10;

        number /= 10;

    }


    return sum;

}
```

Source Code:

```
031_RobotMove.cpp
```

Test Cases:

- Functional cases: Counting mov es on a normal grid
- Boundary cases: Counting mov es on a grid with only a row, a column, or ev en an element