Lab 07: Implementing Backtracking Algorithms

ICT 2202: Algorithm Analysis Design and Lab

Submitted By

Group 1

Tahshin Islam Tuhin(ID: 1957)

Nobin Chandra(ID: 1958)

Md. Shaon Khan(ID: 1984)

Submitted To

Professor Fahima Tabassum, PhD

Institute of Information Technology

Jahangirnagar University

Submission Date: 09/12/2025

Overview:

Algorithms based on the backtracking method play an important role in solving problems that require exploring multiple possibilities under given constraints. Backtracking works by constructing a solution step by step and immediately abandoning any partial solution that violates the problem conditions, thereby reducing unnecessary computations. The 8-Queen Problem applies backtracking to place queens on a chessboard row by row, ensuring that no two queens attack each other in the same row, column, or diagonal. When a conflict occurs, the algorithm backtracks to try alternative placements. In the Combination Sum Problem, backtracking is used to generate all valid combinations of numbers that add up to a target value by recursively including or excluding elements and pruning paths that exceed the target sum. The Word Break Problem uses backtracking to determine whether a given string can be segmented into meaningful words from a dictionary by exploring different prefix partitions and reverting when a dead end is reached. Similarly, the m-Coloring Problem demonstrates backtracking in graph theory by assigning colors to vertices such that no two adjacent vertices share the same color, backtracking whenever a coloring choice violates the constraint. Together, these problems illustrate how backtracking efficiently explores the solution space, making it an effective technique for solving combinational and constraint-satisfaction problems in computer science.

Problem -01: 8- Queen Problem

Algorithm-

1. Start with an empty n × n chessboard.
2. Begin from the first row (row = 0).
3. Try to place a queen in every column of the current row.
4. Before placing a queen, check whether the position is safe:
    - No queen in the same column above.
    - No queen in the upper left diagonal.
    - No queen in the upper right diagonal.
5. If the position is safe, place the queen and move to the next row.
6. If no safe position is found in a row, remove the previously placed queen (backtracking) and try the next column.
7. Repeat the process until queens are placed in all rows.
8. When all queens are placed, the solution is obtained.

Explanation:

Time Complexity: O The algorithm uses backtracking, which tries many possible positions.For an n × n board, the time complexity is exponential.

Worst-case time complexity: O(n!)

Space Complexity: Total space complexity: $O(n^2)$

Code:

```
#include <bits/stdc++.h>
using namespace std;
bool isSafe(vector<vector<int>>& mat, int
row, int col) {
   int n = mat.size();
   int i, j;
   for (i = 0; i < row; i++)
      if (mat[i][col])
         return false;
   for (i = row-1, j = col-1; i >= 0 &&
       j >= 0; i--, j--)
      if (mat[i][j])
         return false;
   for (i = row-1, j = col+1; j < n &&
       i >= 0; i--, j++)
      if (mat[i][j])
         return false;
   return true;
}
bool placeQueens(int row,
vector<vector<int>>& mat) {
   int n = mat.size();
   if(row == n) return true;
   for(int i = 0; i < n; i++){
      if(isSafe(mat, row, i)){
         mat[row][i] = 1;
         if(placeQueens(row + 1, mat))
            return true;
         mat[row][i] = 0;
      }
   }
   return false;
}
vector<vector<int>> queens() {
   int n = 8;
   vector<vector<int>> mat(n, vector<int>(n,
0));
   placeQueens(0, mat);

   return mat;
}
int main() {
   vector<vector<int>> res = queens();
   for (auto v: res) {
      for (auto square: v) {
         cout << square << " ";
      }
      cout << endl;
   }
}
```

1. Output

1 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 1

0 0 0 0 0 1 0 0

0 0 1 0 0 0 0 0

0 0 0 0 0 0 1 0

0 1 0 0 0 0 0 0

0 0 0 1 0 0 0 0

Problem- 02 : Combinational Sum

Algorithm:

1. Start at index 0 with remaining sum remSum = target and an empty current combination cur.
2. If remSum == 0, record cur as a valid combination and return.
3. If remSum < 0 or index is beyond the array, return (dead end).
4. Choice A (take arr[index]): append arr[index] to cur, call recursively with the same index and remSum - arr[index]. After returning, remove the last element (backtrack).
5. Choice B (skip arr[index]): call recursively with index + 1 (do not change remSum).
6. Continue until all combinations are explored.

Explanation:

Time Complexity: $O(2^n)$ (where n is the number of elements in the array)

Space Complexity: O(n) (for recursion stack, excluding output storage)

Code:

```
#include <bits/stdc++.h>
using namespace std;
void makeCombination(vector<int> &arr, int
remSum, vector<int> &cur,
                     vector<vector<int>> &res, int
index) {
   if (remSum == 0) {
      res.push_back(cur);
      return;
   }
   if (remSum < 0 || index >= arr.size())
      return;
   cur.push_back(arr[index]);
   makeCombination(arr, remSum -
arr[index], cur, res, index);

   cur.pop_back();
   makeCombination(arr, remSum, cur, res,
index + 1);
}
```

```
vector<vector<int>>
targetSumComb(vector<int> &arr, int target)
{
   vector<int> cur;
   vector<vector<int>> res;
   makeCombination(arr, target, cur, res, 0);

   return res;
}
int main() {
   vector<int> arr = {1,2,3,4};
   int target = 5;
   cout<<"Input : arr={1,2,3,4},
target=5"<<endl;
   vector<vector<int>> res =
targetSumComb(arr, target);
   for (vector<int> &v : res) {
      for (int i : v) {
         cout << i << " ";
      }
      cout << endl;
   }
}
```

2. Output:

Input: arr = {1,2,3,4}, target = 5

1 1 1 1 1

1 1 1 2

1 1 3

1 2 2

1 4

2 3

Problem – 03: Word Break Problem

Algorithm (Word Break II)**:**

Inputs: string s of length n, dictionary dict (list)

1. wordBreak(s, dict):
    i. Initialize res = [], curr = "".
    ii. Call backtrack(start = 0).
2. backtrack(start):
    i. If start == n: push curr into res and return.
    ii. For end = start+1 to n:
        a. Let word = s.substr(start, end-start).
        b. If word is in dict:
            1. Save prev = curr.
            2. If curr non-empty, append a space to curr.
            3. Append word to curr.
            4. Recurse backtrack(end).
            5. Restore curr = prev (backtrack).
3. After recursion finishes, return res.

Explanation:

Time Complexity:

- The algorithm tries all possible ways to split the string.
- For a string of length n, there can be many possible partitions.

$O(2^n)$

exponential, because the function explores many possible word combinations.

Space Complexity: O(n) + output size

Code:

```
#include <bits/stdc++.h>
using namespace std;
bool isInDict(string &word, vector<string> &dict) {
   for (string &d : dict) {
      if (d == word) return true;
   }
   return false;
}
void wordBreakHelper(string &s, vector<string>
&dict,
                string &curr, vector<string> &res,
                int start) {

   if (start == s.length()) {
      res.push_back(curr);
      return;
   }
   for (int end = start + 1; end <= s.length(); ++end)
{
      string word = s.substr(start, end - start);
      if (isInDict(word, dict)) {
         string prev = curr;

         if (!curr.empty())
            curr += " ";

         curr += word;
         wordBreakHelper(s, dict, curr, res, end);
         curr = prev;
         }
    }
}
```

```
vector<string> wordBreak(string s,
vector<string> &dict) {
   vector<string> res;
   string curr;
   wordBreakHelper(s, dict, curr, res,
0);
   return res;
}

int main() {
   string s = "nobinroy";
   cout<<"Input : s=\"nobinroy\",
dict={\"nob\", \"in\",
\"nobin\",\"roy\",\"abc\"}"<<endl;
   vector<string> dict = {"nob", "in",
"nobin","roy","abc"};

   vector<string> result = wordBreak(s,
dict);

   for (string sentence : result) {
      cout << sentence << endl;
   }

}
```

3. Output:

Input : s="nobinroy", dict={"nob", "in", "nobin","roy","abc"}

nob in roy

nobin roy

Problem – 04: m Coloring Problem.

Algorithm:

1. Input: number of vertices V, list of edges, number of colors m.
2. Build adjacency list adj from the edge list (so neighbor checks are O(deg(v))).
3. Initialize an array color[V] with 0 (0 = uncolored).
4. Backtracking function solve(v):
   - If v == V, all vertices are colored → return true (solution found).
   - For each color c from 1 to m:
     - Check isSafe(v, c) — verify no neighbor of v already has color c.
     - If safe:
       - Assign color[v] = c.
       - Recursively call solve(v+1). If it returns true, propagate true upward.
       - Otherwise backtrack: set color[v] = 0 and try next color.
   - If no color works for v, return false (trigger backtracking).
5. Start with solve(0). If true, color[] holds a valid assignment; otherwise no m-coloring exists with m colors.

Explanation:

Time Complexity: Worst case: the algorithm tries up to m choices for each of the V vertices → O(m^V)

Space Complexity: Total space: O(V + E). If you count the color array and stack together: O(V + E) (or simply O(V + E)).

Code:

```
#include <bits/stdc++.h>
using namespace std;

bool isSafe(int v, int col, const vector<vector<int>>&
adj, const vector<int>& color) {
    for (int nbr : adj[v]) {
        if (color[nbr] == col) return false;
    }
    return true;
}

bool solve(int v, int m, const vector<vector<int>>&
adj, vector<int>& color) {
    int n = color.size();
    if (v == n) return true;

    for (int c = 1; c <= m; c++) {
        if (isSafe(v, c, adj, color)) {
            color[v] = c;
            if (solve(v + 1, m, adj, color)) return true;
            color[v] = 0;
        }
    }
    return false;
}

bool graphColoring(int V, const
vector<vector<int>>& edges, int m) {
    vector<vector<int>> adj(V);
    for (auto &e : edges) {
        adj[e[0]].push_back(e[1]);
        adj[e[1]].push_back(e[0]);
    }

    vector<int> color(V, 0);

    if (!solve(0, m, adj, color)) {
        cout << "false\n";
        return false;
    }

    cout << "true\nColor
assignment:\n";
    for (int i = 0; i < V; i++) {
        cout << "Vertex " << i << " ->
Color " << color[i] << "\n";
    }
    return true;
}
int main() {
    int V = 4;
    vector<vector<int>> edges = {{0,
1}, {0, 2},{0,3}, {1, 3}, {2, 3}};
    int m = 3;

    graphColoring(V, edges, m);
}
```

4. Output:

input : V=4, edges={{0, 1}, {0, 2},{0,3}, {1, 3}, {2, 3}}, m=3

true

Color assignment:

Vertex 0 -> Color 1

Vertex 1 -> Color 2

Vertex 2 -> Color 2

Vertex 3 -> Color 3

Discussion:

From the experiments performed in this lab, it can be concluded that backtracking is a powerful and flexible algorithmic technique for solving combinational and constraint-satisfaction problems. Through the implementation of problems such as the 8-Queen Problem, Combination Sum, Word Break, m-Coloring, and Rod Cutting, we observed how backtracking systematically explores all possible solution paths while pruning those that violate problem constraints. This approach significantly reduces unnecessary computations compared to brute-force methods.

Each problem highlighted a different application of backtracking: the 8-Queen and m-Coloring problems focused on enforcing positional and adjacency constraints, the Combination Sum and Rod Cutting problems demonstrated recursive decision-making with inclusion and exclusion choices, and the Word Break problem illustrated string partitioning under dictionary constraints. Despite solving different kinds of problems, all implementations followed the same core backtracking principle of trial, constraint checking, and reversal (backtracking).

Although backtracking algorithms often have exponential time complexity in the worst case, they remain highly effective for problems with manageable input sizes and can be further optimized using techniques such as memoization, pruning, and heuristic ordering. Overall, this lab successfully demonstrated the practical importance of backtracking and its role in solving complex real-world problems in computer science, especially where multiple choices and constraints must be handled efficiently.