Backtracking

Objectives

- Introduction to Backtracking
- State Space Tree Representation
- N Queen problem using backtracking
- Sum of Subset using Backtracking
- Graph Coloring using Backtracking

Introduction

- Backtracking can be defined as a general algorithmic technique that considers **searching every possible combination** in order to solve an optimization problem.
- It is a recursive technique.
- It generates a state space tree for all possible solutions.
- It traverse the state space tree in the depth first order.
- So, in a backtracking we attempt solving a sub-problem, and if we don't reach the desired solution, then undo whatever we did for solving that sub-problem, and try solving another sub-problem.
- All the solutions require a set of constraints divided into two categories: explicit and implicit constraints.

The N - Queen Problem

- The N - queen is the problem of placing N chess queens on an $N \times N$ chessboard so that, no two queens attack each other.
- Two queens of same row, same column or the same diagonal can attack each other.

Two Queens can attack each other if we assume that one Queen is at (i, j) that is at i^{th} row and j^{th} col. and another is at (k, l) that is k^{th} row and l^{th} column. Then they can attack each other if

$i = k$ *same row*

$j = l$ *same col.*

$abs(i - k) = abs(j - l)$ *same diagonal*

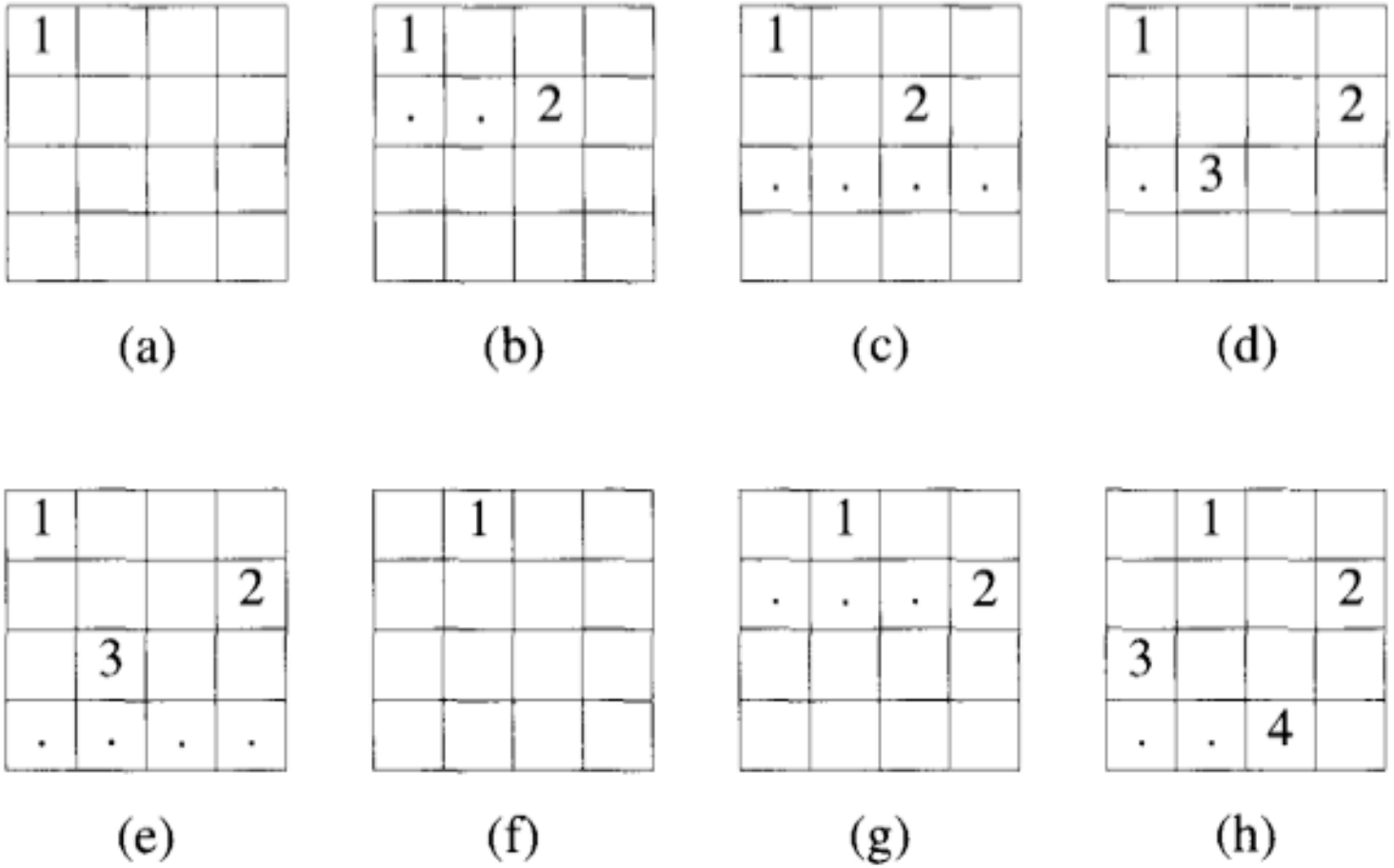

Fig. 1: Example of a backtrack solution to a 4-queens problem

```
1   Algorithm Place(k, i)
2   // Returns true if a queen can be placed in kth row and
3   // ith column. Otherwise it returns false. x[ ] is a
4   // global array whose first (k − 1) values have been set.
5   // Abs(r) returns the absolute value of r.
6   {
7       for j := 1 to k − 1 do
8           if ((x[j] = i) // Two in the same column
9               or (Abs(x[j] − i) = Abs(j − k)))
10                  // or in the same diagonal
11              then return false;
12      return true;
13  }
```

Algorithm 1: Can a new Queen be placed?

```
1   Algorithm NQueens(k, n)
2   // Using backtracking, this procedure prints all
3   // possible placements of n queens on an n × n
4   // chessboard so that they are nonattacking.
5   {
6       for i := 1 to n do
7       {
8           if Place(k, i) then
9           {
10              x[k] := i;
11              if (k = n) then write (x[1 : n]);
12              else NQueens(k + 1, n);
13          }
14      }
15  }
```

Algorithm 2: All Solutions to the n-queens problems

SUM OF SUBSETS

Suppose we are given n distinct positive numbers (usually called weights) and we desire to find all combinations of these numbers whose sums are m. This is called the *sum of subsets* problem.

- For a node at level i the left child corresponds to $x_i = 1$ and the right to $x_i = 0$.
- The bounding functions we use, therefore,

  $B_k(x_1, \ldots, x_k) = true$ iff

  $$\sum_{i=1}^{k} w_i\, x_i + \sum_{i=k+1}^{n} w_i \geq m$$

  and $\sum_{i=1}^{k} w_i\, x_i + w_{k+1} \geq m$

```
1   Algorithm SumOfSub(s, k, r)
2   // Find all subsets of w[1 : n] that sum to m. The values of x[j],
3   // 1 ≤ j < k, have already been determined. s = Σ_{j=1}^{k-1} w[j] * x[j]
4   // and r = Σ_{j=k}^{n} w[j]. The w[j]'s are in nondecreasing order.
5   // It is assumed that w[1] ≤ m and Σ_{i=1}^{n} w[i] ≥ m.
6   {
7       // Generate left child. Note: s + w[k] ≤ m since B_{k-1} is true.
8       x[k] := 1;
9       if (s + w[k] = m) then write (x[1 : k]); // Subset found
10          // There is no recursive call here as w[j] > 0, 1 ≤ j ≤ n.
11      else if (s + w[k] + w[k + 1] ≤ m)
12          then SumOfSub(s + w[k], k + 1, r - w[k]);
13      // Generate right child and evaluate B_k.
14      if ((s + r - w[k] ≥ m) and (s + w[k + 1] ≤ m)) then
15      {
16          x[k] := 0;
17          SumOfSub(s, k + 1, r - w[k]);
18      }
19  }
```

Algorithm 3: Recursive backtracking algorithm for sum of subsets problem

Fig.2 shows the portion of the state space tree generated by function SumofSub while working on the instance $n = 6, m = 30.$ and $w[1:6] = \{5,10,12,13,15,18\}$. The rectangular nodes list the values of s, k and r on each of the calls to SumOfSub. Circular nodes represent points at which subsets with sums m are printed out. At nodes $A, B,$ and C the output is respectively $(1,1,0,0,1), (1,0,1,1),$ and $(0,0,1,0,0,1)$.

Note that the tree of Fig.2 contains only 23 rectangular nodes. The full state space tree for $n = 6$ contains $2^6 - 1 = 63$ nodes from which calls could be made.

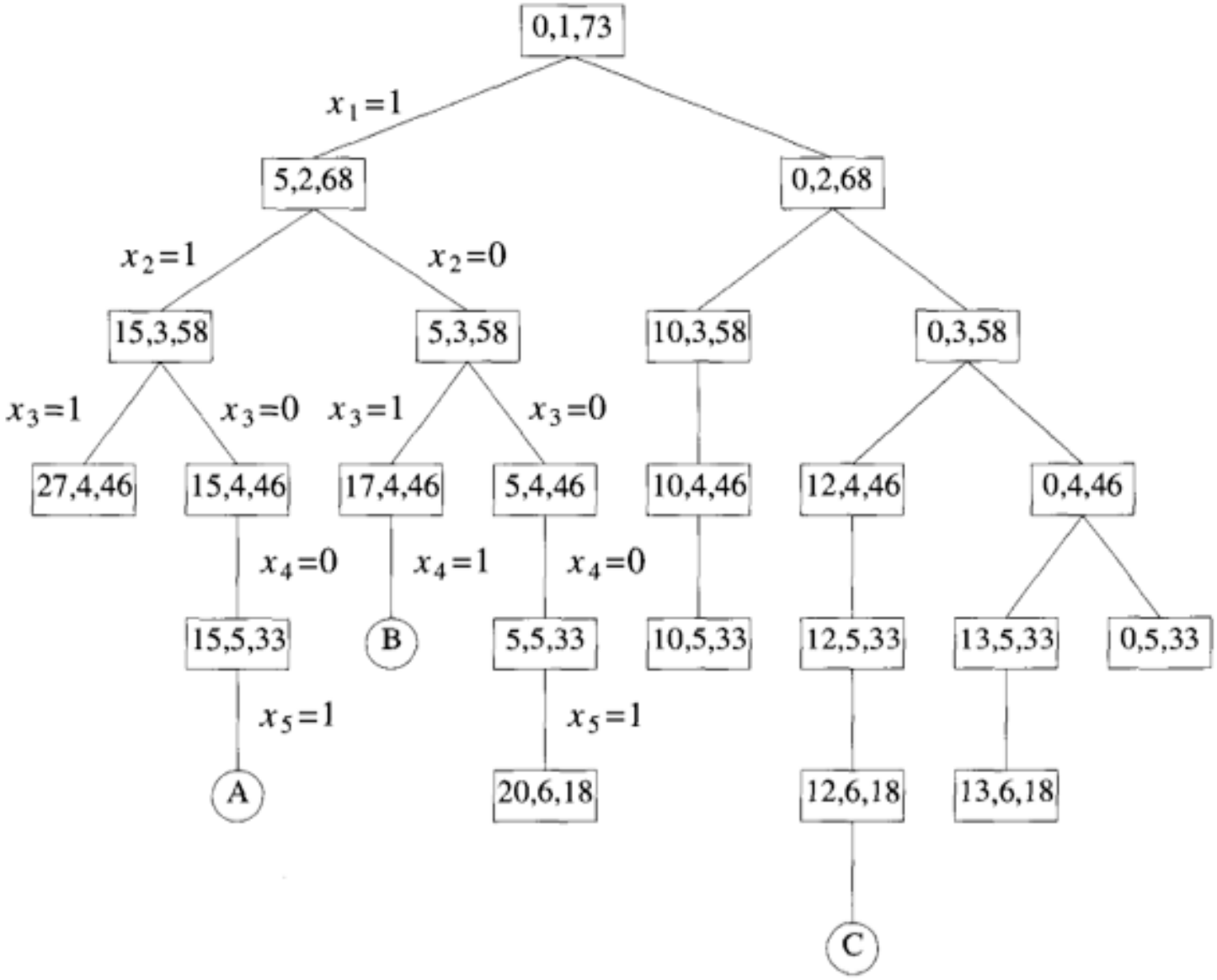

Fig.2: Portion of state space tree generated by SumOfSub

Graph Coloring

- Let G be a graph and m be a given positive integer. We want to discover the nodes of G can be colored in such a way that no two adjacent nodes have the same color yet only m colors are used. This is termed as $m - colorability\ decision$ problem.
- Note: if d is the degree of the given graph, then it can be colored with $d + 1$ colors.
- The $m - colorability\ optimization$ problem asks for the smallest integer m for the graph G can be colored. This integer is referred to as *chromatic number* of the graph.

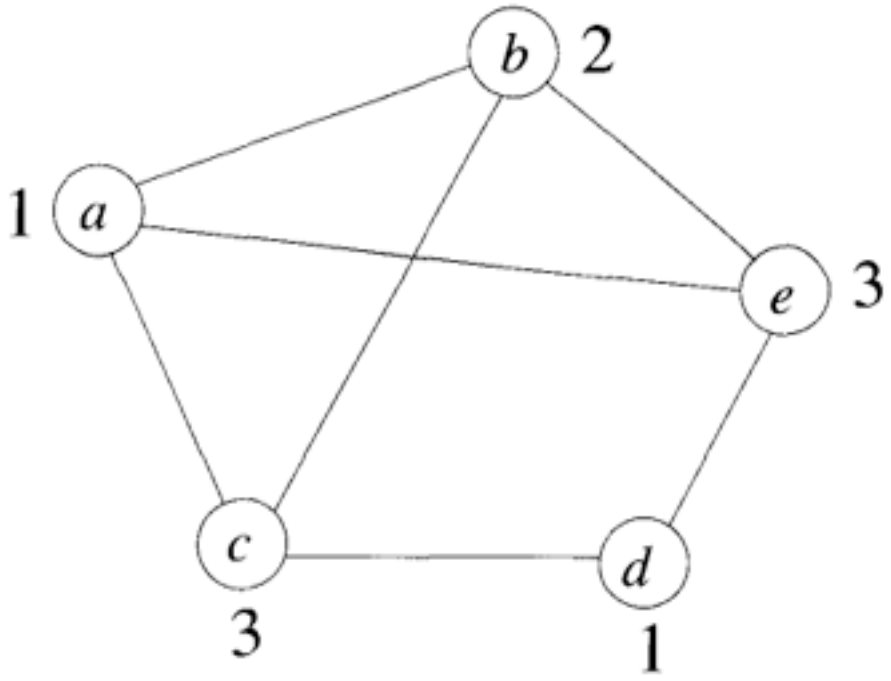

Fig.3: An example graph and its coloring

The given graph can be colored with three colors 1, 2 and 3. The color of each node is indicated next to it. It can also be seen that three colors are needed to color this graph and hence this graph's chromatic number is 3.

```
1   Algorithm mColoring(k)
2   // This algorithm was formed using the recursive backtracking
3   // schema. The graph is represented by its boolean adjacency
4   // matrix G[1 : n, 1 : n]. All assignments of 1, 2, ..., m to the
5   // vertices of the graph such that adjacent vertices are
6   // assigned distinct integers are printed. k is the index
7   // of the next vertex to color.
8   {
9       repeat
10      {// Generate all legal assignments for x[k].
11          NextValue(k); // Assign to x[k] a legal color.
12          if (x[k] = 0) then return; // No new color possible
13          if (k = n) then     // At most m colors have been
14                              // used to color the n vertices.
15              write (x[1 : n]);
16          else mColoring(k + 1);
17      } until (false);
18  }
```

Algorithm 4: Finding all m-colorings of a graph

```
1   Algorithm NextValue(k)
2   // x[1], ..., x[k − 1] have been assigned integer values in
3   // the range [1, m] such that adjacent vertices have distinct
4   // integers. A value for x[k] is determined in the range
5   // [0, m]. x[k] is assigned the next highest numbered color
6   // while maintaining distinctness from the adjacent vertices
7   // of vertex k. If no such color exists, then x[k] is 0.
8   {
9       repeat
10      {
11          x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
12          if (x[k] = 0) then return; // All colors have been used.
13          for j := 1 to n do
14          {   // Check if this color is
15              // distinct from adjacent colors.
16              if ((G[k, j] ≠ 0) and (x[k] = x[j]))
17              // If (k, j) is and edge and if adj.
18              // vertices have the same color.
19                  then break;
20          }
21          if (j = n + 1) then return; // New color found
22      } until (false); // Otherwise try to find another color.
23  }
```

Algorithm 5: Generating a next color

The recursion continues for all the vertex of the graph, trying the different colors.

If no color is safe and not all the nodes are filled, it will backtrack and try a different color.

For small problem, the backtracking approach gives the optimal result. However graph coloring problem is NP hard.

Time complexity: $O(m^n)$ where m is the number of colors and n is the number of vertices of the graph.