# Chapter 14: Backtracking

DR.S.SRIDHAR

ASSOCIATE PROFESSOR

DEPARTMENT OF IST

ANNA UNIVERSITY

# objectives

- Basics of backtracking
- Basics of *N*-queen problem using backtracking
- Basics of sum of subsets problem using the backtracking approach
- Overview of vertex colouring problem and its significance
- Identifying a Hamiltonian cycle in a graph using the backtracking approach
- Generating permutations using backtracking
- Finding convex hull using the backtracking approach

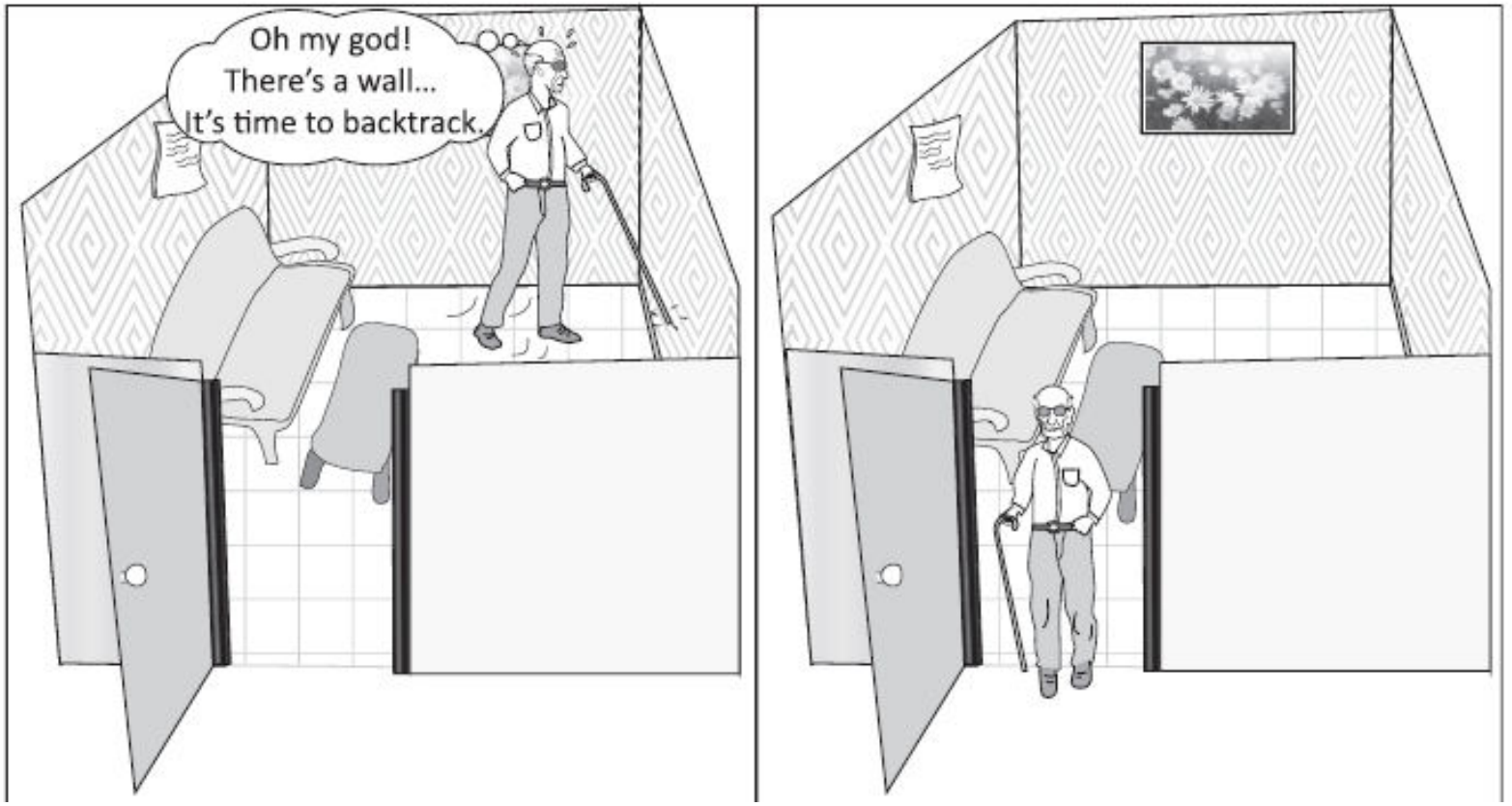
# Backtracking is useful

***Enumeration problems*** In an enumeration problem, all solutions are listed for a given problem.

***Decision problems*** In a decision problem, a solution is given in terms of yes/or no.

***Optimization problems*** In optimization problems, optimal solutions are required, which maximize or minimize the given objective function as per the constraints of the given problem.

# What is backtracking?



# What backtracking is about!

Backtracking is a depth-first search, with some bounding functions. Bounding functions or validity functions represent the constraints of the given problem. First, the backtracking process defines a solution vector as $n$-tuple vector $(x_1, x_2, \ldots, x_n)$ for the given problem. Here $n$ is the number of components of the solution vector and each $x_i$, where $i$ ranges from 1 to $n$, represents a partial solution. These partial solution components $x_i$ are generated based on the concept of constraints.

# Stages of backtracking

The backtracking approach involves two stages: generation of a state-space tree and exploring the state-space tree.

# Some Terminologies

***Answer states*** These are solution states where the path from the root to the leaf defines the solution of the problem.

***Live node*** A node that has been generated already but is yet to generate the children is called a live node.

***E-node*** A node that is under consideration and is in the process of being generated is called an e-node.

***Dead node*** A node that is already explained and cannot be considered for further searches is called a dead node.

# Searching for a goal

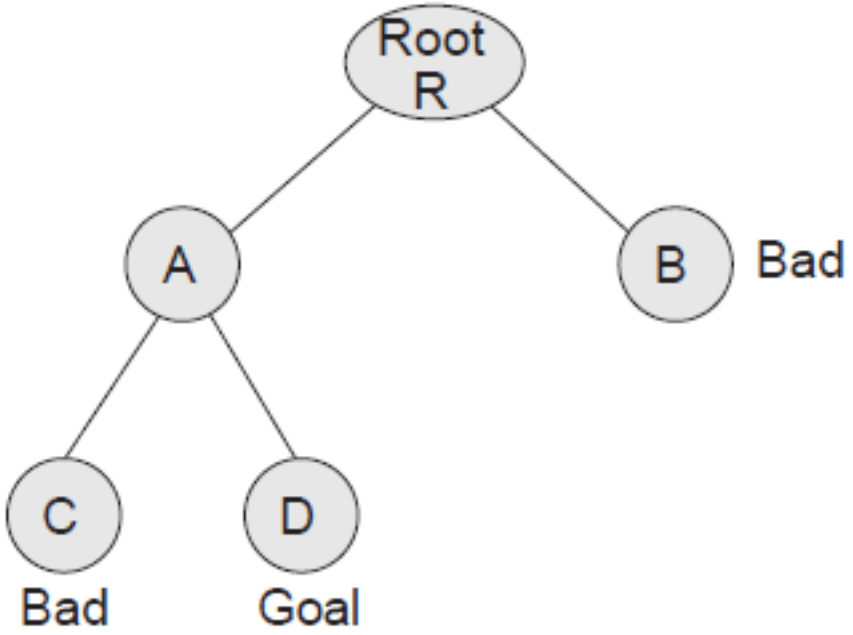



**Fig. 14.1** Illustrative example of backtracking

# Search Procedure

1. Search starts at root R. Here, there are two choices A and B. Pick choice A.
2. At node A, the two choices are C and D.
3. Move to choice C. The choice is bad. So backtrack to A.
4. A is already explored, so move to D.
5. D is the goal state and report success.
6. Backtrack to A and the root R. Explore B. Since B is bad, it does not require processing. Hence, terminate the search.

# Informal algorithm

**Step 1:** While there are many choices left out, perform Step 2.
**Step 2:** Generate a state-space tree using the DFS approach.
**2a:** Check the next configuration using bounding functions.
**2b:** If the solution is promising then
if the solution is obtained then
print the solution
else
Backtrack to the parent of the node and try again
**Step 3:** End.

# Formal algorithm

**Algorithm try(u)**

```
%% Input: node u, starts with root of the state-space tree
%% Output: Result of the problem
Begin
    if (promising(u)) then
```

# Formal algorithm

```
        if (u is a goal) then
           print the solution
        else
          for each v, v ∈ child(u) do
                try(v)
          End for
        End if
    End if
End
```

# Alternate view – Try and Expand

**Algorithm tryexpand(u)**

```
%% Input: Node u, starts with root of the state-space tree
%% Output: Result of the problem
Begin
    Generate children v of node u
     for each v, v∈child(u) do
         if (promising(u)) then
            if (u is a goal) then
              print the solution
            else
              tryexpand(v)
            End if
         End if
     End for
End
```

# Complexity Analysis

For example, if $c_1$ children are encountered for the first component of the solution vector, $c_2$ children for the second component, and so on, then the number of children encountered for the solution of the random vector is given by the relation $T(n) = 1 + c_1 + c_1c_2 + \cdots + c_1c_2 \ldots c_n$. These estimates can be selected many times, and their average can be chosen as the estimation of the actual size of the tree.

# N-Queen Problem

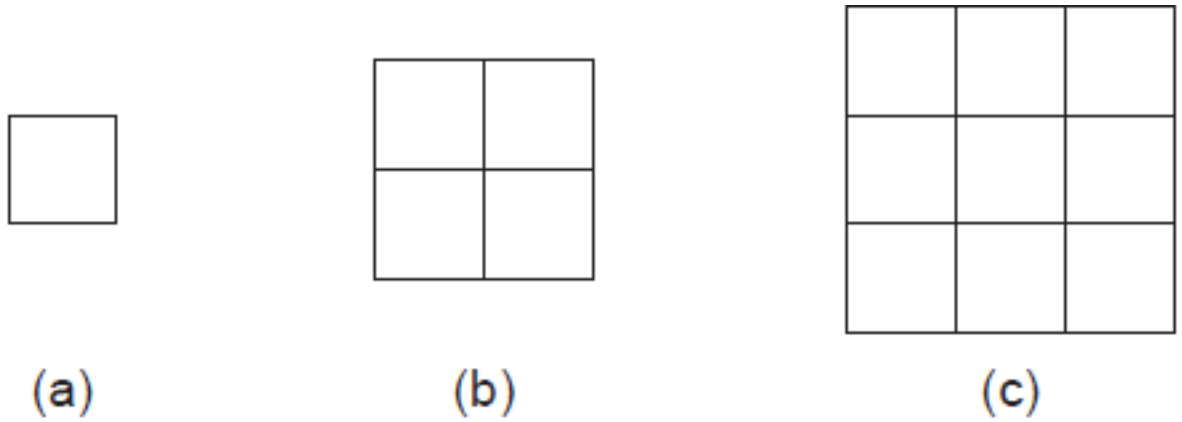



**Fig. 14.3** Boards for queen problems where queens cannot be placed in non-attacking position (a) One-queen problem (b) Two-queen problem (c) Three-queen problem

# Example of 4-Queen Problem

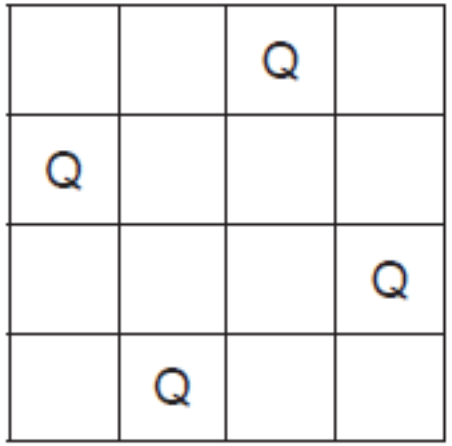



**Fig. 14.4** One possible solution to 4-queen problem

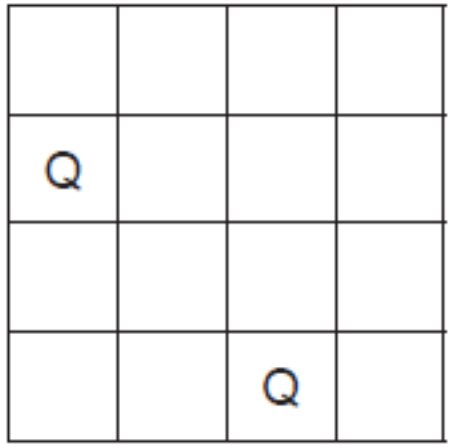



**Fig. 14.5** Attacking position of two queens

# Promising Node

1. col($i$) = col($k$). This condition shows that queens are in the same row in respective columns.
2. col($i$) − col($k$) = $i$ − $k$ or col($i$) − col($k$) = $k$ − $i$; in other words, column($i$) − column($k$) = abs($k$ − $i$) for checking whether the queens are in diagonal positions.

# Informal algorithm

**Step 1:** Start from the first column, check row by row, and place a queen.
**Step 2:** Move on to the next column and place the next queen.
**Step 3:** Check if the placement of queen is safe.
**Step 4:** If the placement of queen is safe, check if *N* queens are placed. If yes, then print the solution; else remove the last queen that is placed and backtrack.
**Step 5:** If solution is not found, then report an error.

# Formal algorithm

**Algorithm queen(i)**

```
%% Input: Queen i
%% Output: Placement of queen i given by col(i)
   Begin
      if promising(i) then      (Promising function is a bounding function)
           if (i == n) then
                   print col[1] .. col[n]
           end if
      else
           for j = 1 to n do                        %% all columns j
```

# Formal algorithm

```
                col[i + 1] = j
                queen[i +1];
            end for
        end if
End
```

# State space Tree

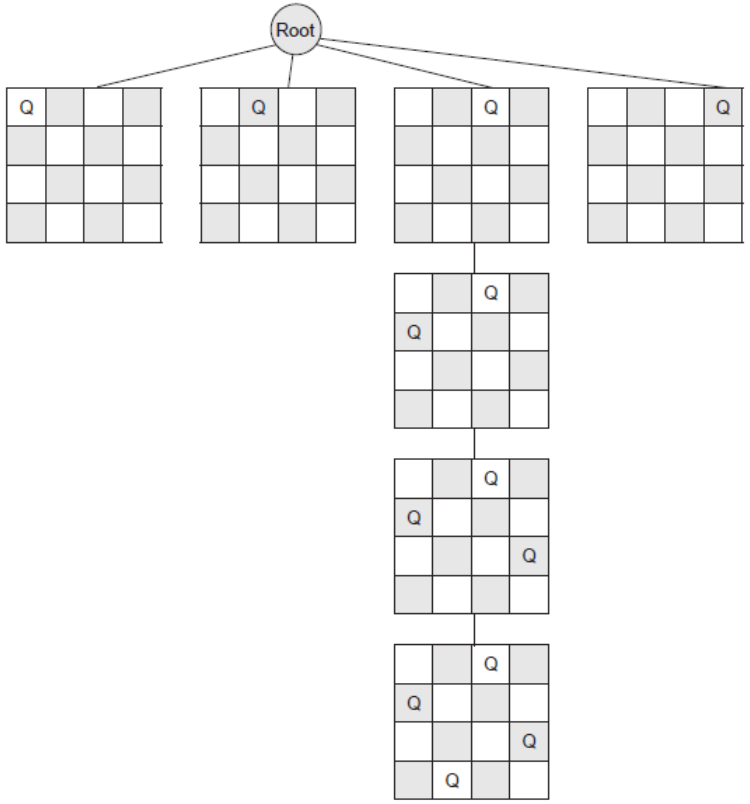



**Fig. 14.7** State-space tree for 4-queen problem

# Formal algorithm

**Algorithm promising(i)**

```
%% Input: Queen i
%% Output: Status about the feasibility of the placement of queen i as true or false
  Begin
```

```
   flag = true
   for k = 1 to i - 1 do
        if (col[i] = col[k]) then           %% rows are same
               if ((|col[i] -col[k]| = |i - k|) or (col[i] == col[k])) then
                      flag = false
               End if
        End if
   End for
   return (flag)
  End
```

# Complexity analysis

The argument for a 4-queen problem (Fig. 14.7) can be extended to an 8-queen problem as well. It can be seen that the number of nodes that will be generated are $1+4+4^2+\cdots+4^4 = \frac{4^{4+1}-1}{4-1} = \frac{4^5-1}{3}$. However, the constraint that no two queens in an attacking position reduces these constraints to $1+4+4\times3+4\times3\times2+4\times3\times2\times1$ promising nodes. In general, $1 + n + n(n-1) + n(n-1)(n-2) + \cdots + n!$ promising nodes are possible. This number of nodes can be used as an upper bound to assess the worst-case behaviour of the algorithm.

# Sum of subsets

Given $n$ positive items with weights $wi$ $\{w_1, w_2,\ldots,w_n\}$ and a positive integer $W$, the problem is about finding all the combinations of items whose sum of weights amounts to $W$. The weights are usually in an ascending order of magnitude and unique. The solution of this problem is often expressed as a solution vector $X$, where the inclusion of an item is indicated by '1' and exclusion by '0'. The following example illustrates a problem of sum of subsets.

# Example state space Tree

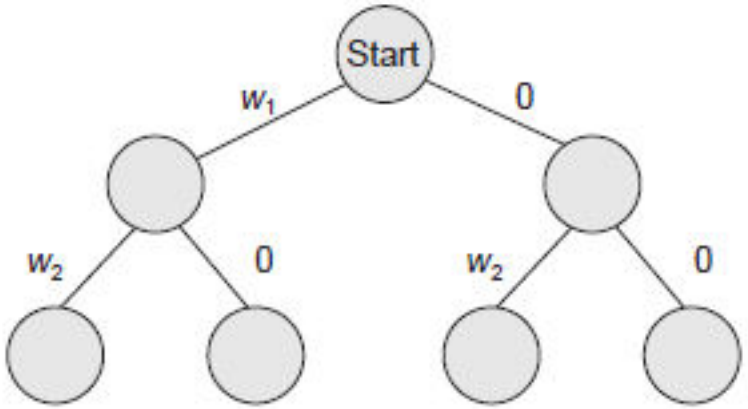



**Fig. 14.8** State-space tree

# Informal Algorithm

**Step 1:** Let the weight be the sum of accumulated weights of all items.
**Step 2:** If the weight of all items accumulated equals $W$, then print the solution. Otherwise, perform Step 3.
**Step 3:** Try the following steps for every branch of the state-space tree:
- **3a:** Add the item to the next level and update the weight.
- **3b:** Exclude the item and update the weight.
- **3c:** Go to Step 2.

**Step 4:** End.

# Formal algorithm

**Algorithm sum-of-subsets(i, weight, total, W)**

```
%% Input: Item i, weight is the weight of items added so far and total is the
%% total that is still available, W is the integer weight for which one wishes
%% to find the subsets of items whose sum must be equal to W, X is the vector
%% whose values are 0 or 1 that indicate the inclusion or exclusion of the items
%% Output: Items whose sum equals W
Begin
    if promising_sum-of-subsets (i)
         if (weight == W) then
                  print items X[1 .. i]
         end if
    else
         (X[i + 1] = 1   %% Include the item
         sum-of-subsets (i + 1, weight + w_{i+1}, total - w_{i+1}, W)
         X[i + 1] = 0    %% Exclude the item
         sum-of-subsets(i + 1, weight, total - w_{i+1}, W)
    End if
End
```

# Promising node

**Algorithm promising_sum-of-subsets(i)**

```
%% Input: Item i
%% Output: Status about the feasibility of including item as part of the subset
Begin
    flag = true
    if ((weight + total ≥ W) and (weight ==W) or (weight + wi+1 ≤ W)) then
      flag = false
    end if
    return flag
End
```

# Complexity analysis

***Complexity Analysis***

It can be observed that the generated state-space tree is a binary tree. At every stage, a node generates two children nodes. Therefore, the number of nodes that will be generated equals $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$. This leads to a time complexity of sum of subsets of $O(2^n)$.

# M-colouring problem

**Step 1:** While there are untried configurations of assigning colours to vertices, perform Step 2.
**Step 2:** Generate the next configuration by assigning arbitrary colours to the vertices.
- **2a:** Test the assignment of colours to vertices.
- **2b:** If the assignment is promising then report success and break.
- **2c:** Go to Step 1.

**Step 3:** End.

# Example

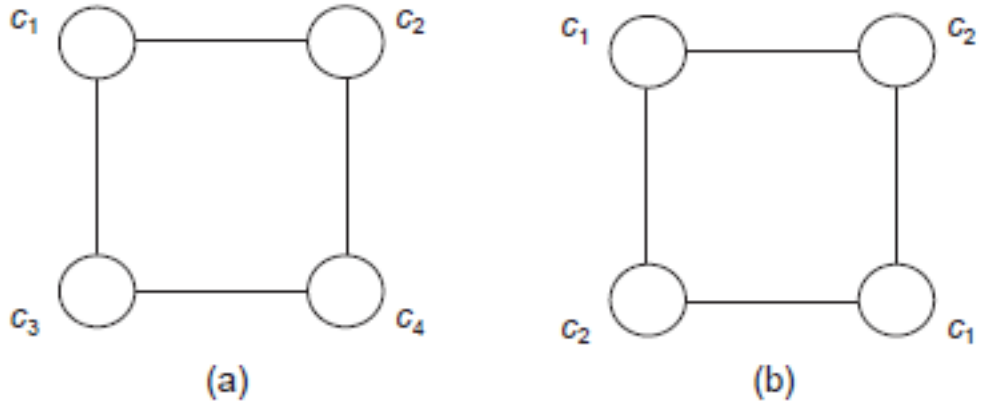



**Fig. 14.11** Ways of assigning colour (a) Non-optimal (b) Optimal

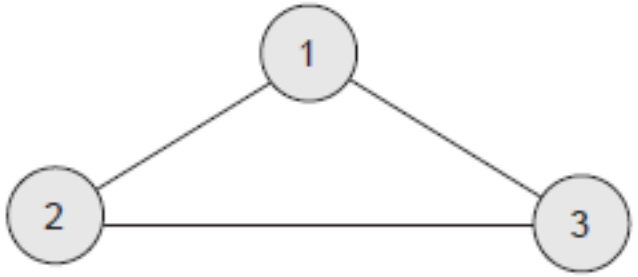



**Fig. 14.12** Example of complete graph $K_i$

# State space Tree

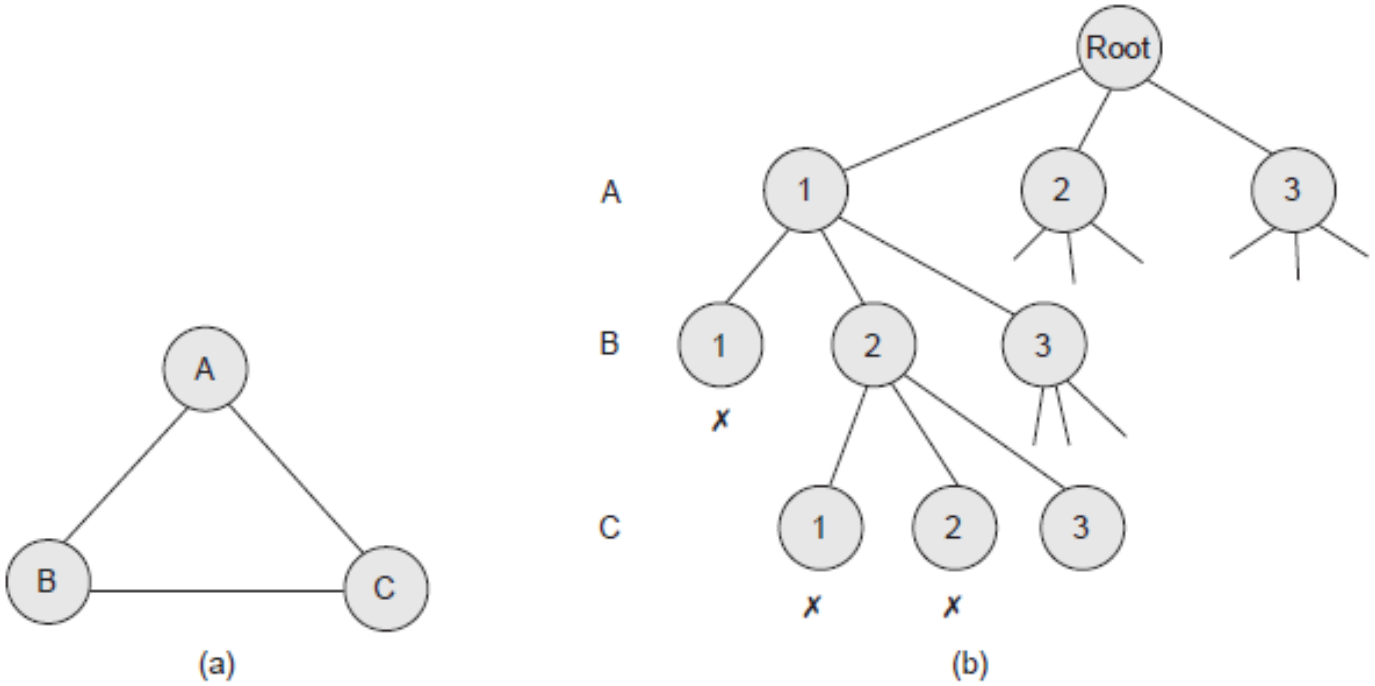



**Fig. 14.14** M-colouring problem (a) Original graph (b) Portion of state-space tree where only feasible colouring schemes are shown

# Formal algorithm

**Algorithm colouring(i)**

```
%% Input: Node i
%% Output: Colours of vertices of graph G, that is, array colour(1 … n)
Begin
    if promising_colouring(i) then
         if (i == n) then    %% If all vertices are coloured then print solution
              print colour[1 ··· n]
         else
            j = 1
            while (j <= M) do          %% for all colours M do
                colour(i + 1) = j      %% Assign colour j to node i + 1
                colouring(j + 1)
               j = j + 1
            End while
         End if
    End if
End
```

# Promising node

**Algorithm promising_colouring(i)**

```
%% Input: Node i
%% Output: Status of colouring of node i
Begin
   flag = true
```

# Formal algorithm

```
    for j = 1 to i - 1 do
        if (vertices i and j are neighbours) then    %% can be checked using
                                                     %% adjacency matrix
            if (colour(i) == colour(j)) then         %% Check colours of
                                                     %% nodes i and j
               flag = false
            End if
        End if
    End for
    return (flag)
End
```

# Complexity analysis

***Complexity Analysis***

The number of nodes in a state-space tree would be $1 + M + M^2 + \cdots + M^n = \frac{M^{n+1} - 1}{M - 1}$. Therefore, it can be concluded that the algorithm is exponential. Here $M$ is the number of colours.

# Hamiltonian circuit Problem

Hamiltonian cycle is a cycle that traverses all the vertices of a given graph G exactly once and ends at the starting vertex. The input for a Hamiltonian circuit problem is an undirected graph. The Hamiltonian problem involves checking if a Hamiltonian cycle is present in a given graph G or not. The following graph (Fig. 14.15) illustrates the presence of a Hamiltonian graph in G.

It can be verified that a Hamiltonian cycle 4-3-1-2-4 exists in Fig. 14.15.

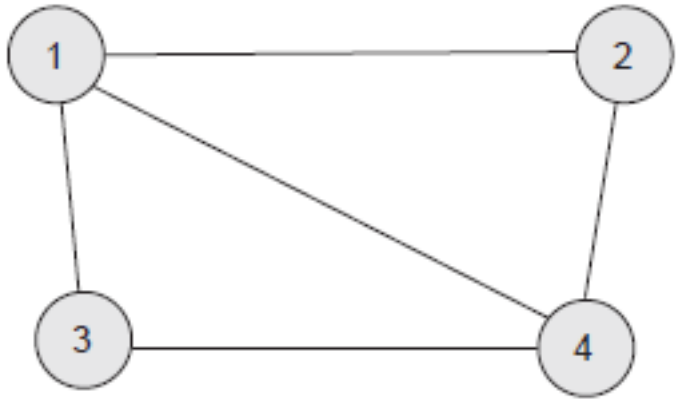



**Fig. 14.15** Example of Hamiltonian cycle

# Constraints

The constraints for a Hamiltonian problem are as follows:

1. The $i$th vertex in the path must be adjacent to the $(i-1)$th vertex in any path.
2. The starting vertex and the $(n-1)$th vertex should be adjacent.
3. The $i$th vertex cannot be one of the first $(i-1)$ vertices.

# Informal algorithm

**Step 1:** While there are untried configurations of travelling of vertices, perform Step 2.
**Step 2:** Generate the next configuration by visiting the next adjacent node.
**2a:** Test the visited path.
**2b:** If the travelled path is promising then report success and break.
**2c:** Go to Step 1.
**Step 3:** End.

# Formal algorithm

**Algorithm Hamiltonian(i)**

```
%% Input: Node i—initially the starting node
%% Output: Hamiltonian cycle if exists
Begin
  if promising_Hamiltonian(i) then
       if (i == n – 1) then       %% If all vertices are covered
                                  %% then print solution
          print v[0] ··· v[n – 1]
       end if
  else
       j = 2                      %% Starts from node 2 as 1 is
                                  %% the starting node
       while (j <= n) do          %% for all vertices do
          v[i] = j                %% Assign vertex j
          Hamiltonian(i + 1)
          j = j + 1
       End while
  End if
End
```

# Promising node

**Algorithm promising_Hamiltonian(i)**

```
%% Input: Node i
%% Output: Status of colouring of node i
Begin
    flag = true
    for j = 1 to i – 1 do
        if (vertices v_i and v_j are neighbours) then      %% Check for distinctness of
                                                           %% vertices
           flag = false
        End if
    End for
        if (vertices v_i and v_i-1 are neighbours) then    %% Check last and first
                                                           %% vertices are neighbours
             flag = true
        else
            flag = false
        End if
   Return(flag)
End
```

# Complexity analysis

***Complexity Analysis***

The number of nodes in the state-space tree algorithm is given as follows:

$$1+(n-1)+(n-1)^2+\cdots+(n-1)^{n-1}$$

$$=\frac{(n-1)^n-1}{n-2}$$

Therefore, the algorithm is an exponential algorithm.

# Generating Permutations

**Step 1:** While there are untried configurations of permutations, perform Step 2.
**Step 2:** Generate the next permutation by adding an element.
- **2a:** Test the generated permutation.
- **2b:** If the generated permutation is correct then report success and break.
- **2c:** Otherwise, backtrack and go to Step 1.

**Step 3:** End.

# State space tree

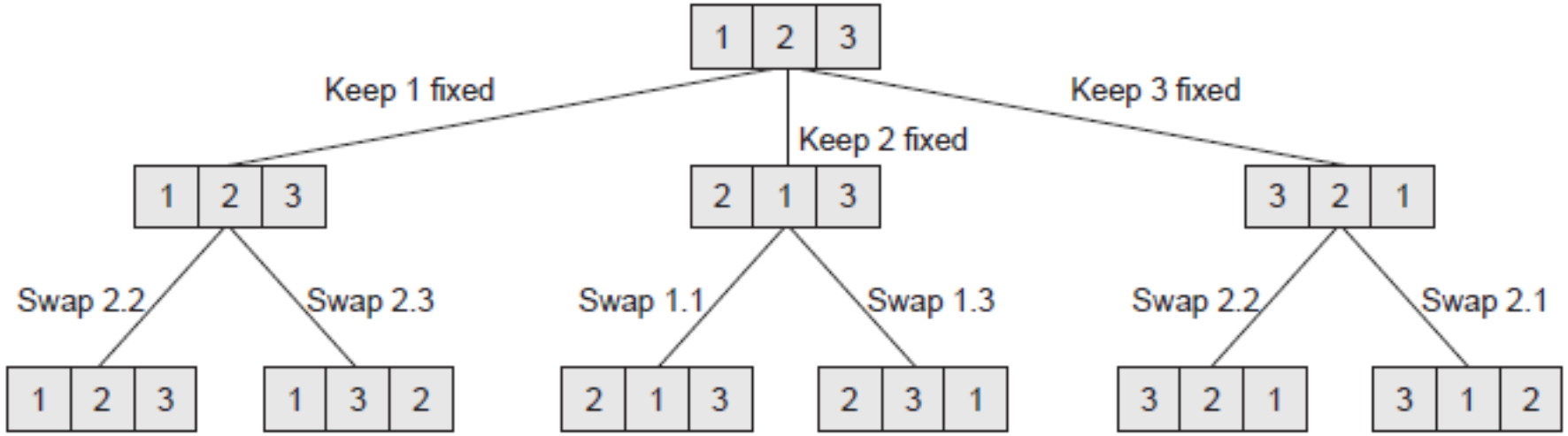



**Fig. 14.18** State-space tree for permutation of {1, 2, 3}

# Formal algorithm

**Algorithm permutate(i)**

```
%% Input: Input item A with n elements A[1 .. n]
%% Output: List of permutations
Begin
    if (i == n) then
        Display A(1 .. n)
    else
        for j = i to n do                  %% from i to n
           A[j] ↔ A[i])                    %% swap elements A[i] and A[j]
           Permutate(i + 1)                %% Permutate the rest
           A[j] ↔ A[i])                    %% Swap to restore the original position
        End for
    End if
End
```

# Complexity Analysis

If $n = 1$, it can be observed that no permutations are required. For the rest of the elements when $n \geq 2$, $nt_{n-1}$ permutations are required. The recurrence equation for permutation is thus given as follows:

$$T(n) = \begin{cases} 0 & \text{if } n = 0 \\ nt_{n-1} & \text{for } n \geq 2 \end{cases}$$

By solving this equation, one can find that the complexity of this algorithm is $\Theta(nn!)$.

# Informal algorithm

The informal algorithm is given as follows:

**Step 1:** Select anchor point $p_o$ by picking a point from the given $n$ points. The anchor point has the minimum $y$-coordinate among all $n$ points.

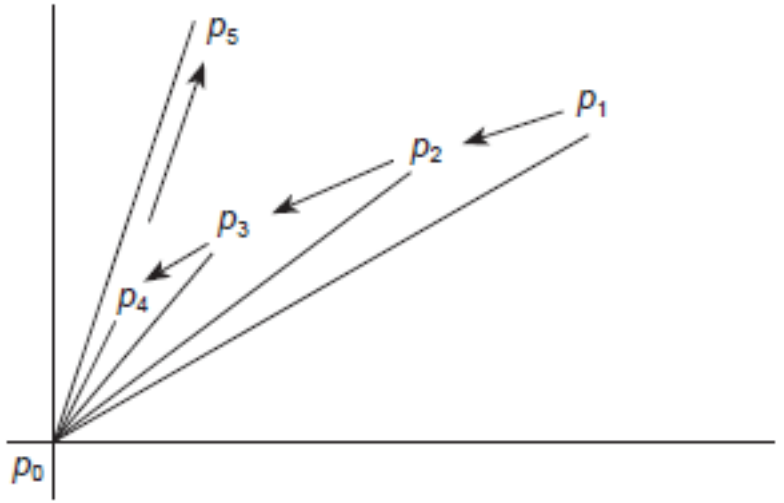



**Fig. 14.20** Graham scan of four points

# Informal algorithm..

**Step 2:** Sort the remaining points lexicographically using a polar angle. Let the other points on set $A$ be $[1, 2, \ldots, m]$, where $m = n - 1$.

**Step 3:** Remove the interior points of the hull. This is done by checking the points in the counter-clockwise manner.

**Step 4:** For each new point $p_k$, perform the following steps:

- **4a:** If $p_k$ forms a left turn with the last two points of the stack, then push the point onto the stack.
- **4b:** Otherwise, pop the last point off the stack.

**Step 5:** If anchor point $p_0$ is encountered, then stop the process as the stack has all the vertices of the convex hull.

**Step 6:** End.

# procedure

**Case 1 (left turn)** In this case, point $p_k$ is pushed onto the stack and the sweep line moves to the next point.

**Case 2 (right turn)** In the case of a right turn, point $p_j$ is popped off the stack. Figure 14.20 illustrates this point.

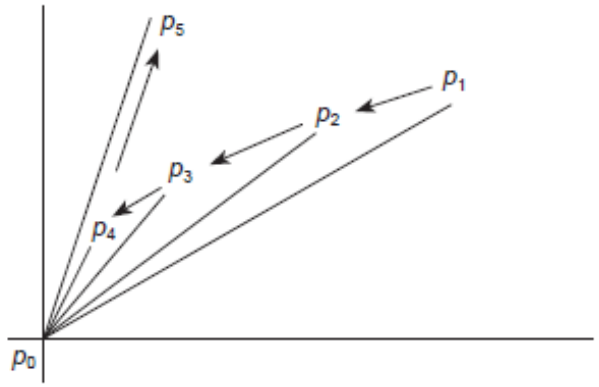



**Fig. 14.20** Graham scan of four points

# Formal algorithm

**Algorithm Graham(A[1 .. n])**

```
%% Input: A set of n points
%% Output: Convex hull
Begin
     Sort n points and choose anchor point p_0 with minimum y-coordinate
     Sort all points except p_0 lexicographically using polar angle and
     rename it p_1 … p_n
     Push p_0, p_1, and p_2 on stack
     k = 3
     while (k ≤ n − 1) do
        b = pop(S)
        a = pop(S)
        if (a, b, and p_k form a left turn) then
            push(S, p_k); k = k + 1
        else
            v = pop(S)
            k = k + 1
            if (v = p_0) then
                break
            end if
        end if
     end while
End
```

# Complexity analysis

***Complexity Analysis***

This algorithm involves sorting that takes $\theta(n \log n)$ time. Checking a right or a left turn takes a constant (i.e., $\theta(1)$) time. Checking all points would take $\theta(n)$ time. Therefore, the total complexity of a Graham scan algorithm is $\max\{\theta(n), \theta(1), \theta(n \log n)\}$, that is, $\theta(n \log n)$ time.

# Glossary

## ■ GLOSSARY ■

**Backtracking** A design technique used for solving optimization problems

**Boundary function** A criteria function for checking the validity of a vector

**Explicit constraint** The rule that explicitly forces the solution vector to take specific values

**Fixed tuple** A state-space tree that is formed by a set of binary decisions

**Graham scan** A technique of finding a convex hull by scanning and verifying all points in an incremental fashion

**Hamiltonian cycle** A cycle that starts from a vertex, visits all other vertices only once, and returns back to the starting vertex

# Glossary

**Hamiltonian path** A path that starts from a vertex, visits all other vertices only once, and returns back to the starting vertex

**Implicit constraint** The rule that limits the generation or processing of a tuple

***M*-colouring problem** A problem of colouring a vertex of a graph such that no two adjacent vertices share a common colour

**Solution vector** An *n*-tuple vector that represents *n* choices

**State-space tree** A tree that represents problem states of various stages

**Sum of subsets** A problem, given *n* positive items and a positive integer *W*, of finding a subset of integers whose sum equals *W*

**Variable tuple** A state-space tree that is formed by variable components