Experiment No 3: Implement QuickSort algorithm to sort elements

QuickSort: A Brief Overview

QuickSort is a sorting algorithm that follows these key steps:

1. **Pivot Selection:** Choose a pivot element from the array.
2. **Partitioning:** Rearrange the array so that elements smaller than the pivot are on the left, and elements greater than the pivot are on the right.
3. **Recursion:** Recursively apply QuickSort to the left and right subarrays.
4. **Combine:** Combine the sorted subarrays to obtain the final sorted array.

How QuickSort Works

1. Pivot Selection:
   - The choice of pivot significantly affects QuickSort's performance.
   - Common pivot selection strategies include:
     - Selecting the first or last element as the pivot.
     - Randomly choosing an element as the pivot.
     - Using the middle element as the pivot.
2. Partitioning:
   - We rearrange the array such that all elements less than the pivot are on the left, and all greater elements are on the right.
   - This process is done by maintaining two pointers—one for elements smaller than the pivot and another for elements greater than the pivot.
3. Recursion:
   - Apply QuickSort recursively to the left and right subarrays.
   - The pivot element is already in its correct position after partitioning.
4. Combine:
   - Combine the sorted subarrays to obtain the fully sorted array.

Illustration

Let's consider an example array: arr[] = {10, 80, 30, 90, 40}. Here's how QuickSort operates step by step:

1. Initial Partitioning:
   - Choose the rightmost element (90) as the pivot.
   - Rearrange the array: {10, 80, 30, 40, 90}.
2. Recursive Calls:
   - Apply QuickSort to the left subarray {10, 80, 30, 40} and the right subarray {90}.
   - The left subarray further partitions into {10, 30} and {80, 40}.
3. Final Merge:
   - Combine the sorted subarrays: {10, 30, 40, 80, 90}.

Implementation in C++

```
#include <bits/stdc++.h>

using namespace std;


int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int i = low - 1;
   for (int j = low; j < high; j++) {
      if (arr[j] < pivot) {
         i++;
         swap(arr[i], arr[j]);
      }
   }
   swap(arr[i + 1], arr[high]);
   return i + 1;
}


void quickSort(int arr[], int low, int high) {
   if (low < high) {
      int pi = partition(arr, low, high);
      quickSort(arr, low, pi - 1);
      quickSort(arr, pi + 1, high);
   }
}


// Example usage:
int main() {
```

```
    int arr[] = {10, 80, 30, 90, 40};
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    // The sorted array is now in 'arr'
    return 0;
}
```

Why QuickSort?

- **Efficiency:** QuickSort has an average time complexity of **O(n log n)**.
- **In-Place Sorting:** It sorts the array in-place without requiring additional memory.
- **Widely Used:** QuickSort is widely used due to its simplicity and efficiency.

Experiment No 4: Implement Mini-max and Binary search algorithm

Let's explore the Mini-max algorithm and the Binary Search algorithm, along with their implementations in C++. I'll also provide information about their time complexities.

Mini-max Algorithm

The Mini-max algorithm is commonly used in decision-making and game theory to find the optimal move for a player, assuming that the opponent also plays optimally. It's widely employed in two-player turn-based games like Tic-Tac-Toe, Chess, and Backgammon.

Here's how the Mini-max algorithm works:

1. Players:
   - The two players are called the **maximizer** and the **minimizer**.
   - The maximizer aims to get the highest score, while the minimizer tries to minimize the score.
2. Board Evaluation:
   - Each board state has an associated value.
   - If the maximizer has the upper hand, the score tends to be positive; if the minimizer has the advantage, it tends to be negative.
   - These values are calculated using game-specific heuristics.
3. Backtracking Approach:
   - The algorithm explores all possible moves, backtracks, and makes decisions.
   - It assumes that the opponent plays optimally.

Implementation in C++

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

int minimax(int depth, int nodeIndex, bool isMax, int scores[], int h) {
  if (depth == h)
    return scores[nodeIndex];

  if (isMax)
    return max(minimax(depth + 1, nodeIndex * 2, false, scores, h),
           minimax(depth + 1, nodeIndex * 2 + 1, false, scores, h));
```

```
    else
        return min(minimax(depth + 1, nodeIndex * 2, true, scores, h),
                   minimax(depth + 1, nodeIndex * 2 + 1, true, scores, h));
}

int log2(int n) {
    return (n == 1) ? 0 : 1 + log2(n / 2);
}

int main() {
    int scores[] = {3, 5, 2, 9, 12, 5, 23, 23};
    int n = sizeof(scores) / sizeof(scores[0]);
    int h = log2(n);
    int res = minimax(0, 0, true, scores, h);
    cout << "The optimal value is: " << res << endl;
}
```

The above code demonstrates the Mini-max algorithm for a given set of scores1.

Binary Search Algorithm

The Binary Search algorithm efficiently finds the position of a target value within a sorted array. It repeatedly divides the search interval in half, making it faster than linear search. Binary search works in logarithmic time complexity (O(log N)).

How Binary Search Works

1. Divide the search space into two halves by finding the middle index.
2. Compare the middle element with the target value.
3. If the key is found, terminate the process.
4. Otherwise, choose the appropriate half (left or right) for the next search.
5. Repeat until the key is found or the search space is exhausted.

Implementation in C++ (Iterative)

```
#include <bits/stdc++.h>
using namespace std;
int binarySearch(int arr[], int low, int high, int x) {
   while (low <= high) {
      int mid = low + (high - low) / 2;
      if (arr[mid] == x)
         return mid;
      if (arr[mid] < x)
         low = mid + 1;
      else
         high = mid - 1;
   }
   return -1;
}
int main() {
   int arr[] = {2, 3, 4, 10, 40};
   int x = 10;
   int n = sizeof(arr) / sizeof(arr[0]);
   int result = binarySearch(arr, 0, n - 1, x);
   (result == -1) ? cout << "Element is not present in array"
                  : cout << "Element is present at index " << result;
   return 0;
}
```

Experiment No 5: Implement Dijkstra algorithm for single source shortest path.

Dijkstra's Shortest Path Algorithm

Dijkstra's algorithm, developed by Dutch computer scientist Edsger W. Dijkstra in 1956, is a powerful method for finding the shortest path from a single source vertex to all other vertices in a weighted graph. It's widely used in various applications, including computer networks, map systems, and transportation planning.

Purpose and Use-Cases

The need for Dijkstra's algorithm arises when we want to find the shortest distance or path between two points. Some common use-cases include:

1. Routing in Computer Networks: Dijkstra's algorithm helps determine the optimal path for data packets to travel from one network node to another.
2. Map Systems (e.g., Google Maps): It calculates the shortest route between a starting point and a destination, considering real-world road networks.

How Dijkstra's Algorithm Works

1. Initialization:
   - Mark the source vertex with a current distance of 0 and all other vertices with infinity.
   - Maintain a set of visited vertices and an array to store the tentative distances.
2. Greedy Approach:
   - Iteratively select the unvisited vertex with the smallest tentative distance.
   - Update the distances of its neighbors if a shorter path is found.
3. Backtracking:
   - Once the algorithm completes, backtrack from the destination vertex to the source vertex to find the actual path.

Pseudocode for Dijkstra's Algorithm

Python

```
function dijkstra(G, S):
    Initialize distance[] and previous[] arrays
    Add source vertex S to the priority queue Q with distance 0

    while Q is not empty:
```

```
    U = Extract MIN from Q
    for each unvisited neighbor V of U:
      tempDistance = distance[U] + edge_weight(U, V)
      if tempDistance < distance[V]:
        distance[V] = tempDistance
        previous[V] = U

  return distance[], previous[]
```

AI-generated code. Review and use carefully. More info on FAQ.

Implementation in C++

Below is a simple implementation of Dijkstra's algorithm in C++:

```
#include <bits/stdc++.h>

using namespace std;


void dijkstra(vector<vector<pair<int, int>>>& graph, int source) {
  int count = graph.size();
  vector<bool> visited(count, false);
  vector<int> distance(count, INT_MAX);
  vector<int> previous(count, -1);

  distance[source] = 0;
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
  pq.push({0, source});

  while (!pq.empty()) {
    int u = pq.top().second;
    pq.pop();
```

```
            visited[u] = true;

            for (auto& neighbor : graph[u]) {
                int v = neighbor.first;
                int weight = neighbor.second;
                if (!visited[v] && distance[u] + weight < distance[v]) {
                    distance[v] = distance[u] + weight;
                    previous[v] = u;
                    pq.push({distance[v], v});
                }
            }
        }

        // Print the shortest distances and paths
        // ...
    }

    int main() {
        // Construct the graph
        // ...
        return 0;
    }
```

Conclusion

Dijkstra's algorithm efficiently finds the shortest path in graphs with non-negative edge weights. Its simplicity and effectiveness make it a valuable tool in various domains. Next time you need to find the optimal route, consider the elegance of Dijkstra's algorithm!

Experiment No 6: Implement Kruskal and Prim's algorithms to find the minimum spanning tree.

Kruskal's Algorithm

Overview

Kruskal's algorithm is a greedy algorithm that constructs an MST by iteratively adding edges with the lowest weight. It ensures that no cycles are formed during this process.

Steps for Kruskal's Algorithm

1. **Sort Edges**: Arrange all edges in ascending order of their weights.
2. **Initialize MST**: Start with an empty set (the MST).
3. **Add Edges**: Add edges to the MST one by one, ensuring that no cycles are formed.
4. **Repeat**: Continue until the MST includes all vertices.

Pseudocode for Kruskal's Algorithm

Python

```
KRUSKAL(G):
    A = ∅
  Sort all edges from low weight to high
  For each edge (u, v) in G.E:
        if FIND-SET(u) ≠ FIND-SET(v):
            A = A ∪ {(u, v)}
    UNION(u, v)
  return A
```

AI-generated code. Review and use carefully. More info on FAQ.

Implementation in C++

Below is a simple implementation of Kruskal's algorithm in C++:

```
#include <bits/stdc++.h>
```

```
using namespace std;

class Graph {
private:
    vector<pair<int, pair<int, int>>> edges;
    int V;

public:
    Graph(int V) : V(V) {}
    void add_edge(int u, int v, int w) {
        edges.push_back({w, {u, v}});
    }
    // Other helper functions...
};

// Union-Find functions...
// Kruskal's algorithm implementation...

int main() {
    Graph g(6);
    g.add_edge(0, 1, 4);
    g.add_edge(0, 2, 4);
    // Add other edges...
    // Call Kruskal's algorithm...
    return 0;
}
```

Prim's Algorithm

Overview

Prim's algorithm is another greedy approach to finding an MST. It starts with a single vertex and keeps adding edges with the lowest weight until all vertices are included.

Steps for Prim's Algorithm

1. **Initialize MST**: Start with an empty spanning tree.
2. **Choose Vertex**: Select an arbitrary vertex as the starting point.
3. **Add Edges**: Repeatedly add the edge with the minimum weight that connects the tree to a new vertex.
4. **Repeat**: Continue until all vertices are included.

Pseudocode for Prim's Algorithm

Python

```
PRIM(G):
    T = ∅
  U = {1} # Start with an arbitrary vertex
    while U ≠ V:
        Find the minimum weight edge (u, v) such that u ∈ U and v ∈ V
- U
        T = T ∪ {(u, v)}
        U = U ∪ {v}
  return T
```

AI-generated code. Review and use carefully. More info on FAQ.

Implementation in C++

Below is a simple implementation of Prim's algorithm in C++:

```
#include <bits/stdc++.h>
using namespace std;
class Graph {
private:
    vector<vector<pair<int, int>>> adj_list;
    int V;
public:
    Graph(int V) : V(V) {
        adj_list.resize(V);
    }
    void add_edge(int u, int v, int w) {
        adj_list[u].push_back({v, w});
        adj_list[v].push_back({u, w});
    }
    // Other helper functions...
};
// Priority queue functions...
// Prim's algorithm implementation...
```

```
int main() {
    Graph g(6);
    g.add_edge(0, 1, 4);
    g.add_edge(0, 2, 4);
    // Add other edges...
    // Call Prim's algorithm...
    return 0;
}
```

Conclusion

Both Kruskal's and Prim's algorithms efficiently find minimum cost spanning trees. Kruskal's algorithm focuses on sorting edges, while Prim's algorithm starts with a single vertex. Depending on the problem context, choose the most suitable algorithm for constructing an MST! ▯▯

Experiment No 7: Implement Bellman Ford algorithm to find the shortest path for a multistage graph

The Bellman-Ford algorithm is a versatile shortest path algorithm that can handle both weighted and unweighted graphs. Unlike Dijkstra's algorithm, which works only with non-negative edge weights, Bellman-Ford can also handle graphs with negative edge weights. Additionally, it has the capability to detect negative cycles, making it a powerful tool for various scenarios.

How Bellman-Ford Works

1. **Initialization:**
   - Start with a single source vertex and initialize the distance to all other vertices as **infinite.**
   - Set the distance of the source vertex to itself as **zero.**
2. **Relaxation of Edges:**
   - The algorithm iteratively relaxes edges by finding new paths that are shorter than the previously estimated paths.
   - For each edge (u, v) in the graph, if distance[u] + weight(u, v) < distance[v], update distance[v] and set previous[v] to u.
3. **Repeat Step 2:**
   - Repeat the relaxation process for all vertices **N-1 times**, where N is the number of vertices in the graph.
4. **Negative Cycle Detection:**
   - After N-1 iterations, perform one more relaxation.
   - If any vertex's distance changes during this additional relaxation, it indicates the existence of a **negative cycle** in the graph.

Pseudocode for Bellman-Ford Algorithm

Python

```
def bellman_ford(graph, src):
    # Step 1: Initialize distance array
    dist = [float("Inf")] * len(graph)
    dist[src] = 0

    # Step 2: Relax edges N-1 times
    for _ in range(len(graph) - 1):
```

```
    for u, v, w in graph:
      if dist[u] != float("Inf") and dist[u] + w < dist[v]:
        dist[v] = dist[u] + w

  # Step 3: Detect negative cycle
  for u, v, w in graph:
    if dist[u] + w < dist[v]:
      print("Error: Negative Cycle Exists")
      return

  return dist
```

AI-generated code. Review and use carefully. More info on FAQ.

Applications of Bellman-Ford Algorithm

- **Routing protocols** in computer networks.
- **Arbitrage detection** in financial markets.
- **Resource allocation** in transportation networks.
- **Path planning** in robotics and autonomous vehicles.

Drawback

- Bellman-Ford is **slower** than Dijkstra's algorithm due to its iterative nature.
- However, its ability to handle negative edge weights and detect negative cycles makes it a valuable tool in certain scenarios.

Remember that while Bellman-Ford is powerful, it's essential to choose the right algorithm based on the specific problem requirements and graph characteristics.

Experiment No 8: Implement Bellman Ford algorithm to find the shortest path for a multistage graph

The Floyd-Warshall algorithm, named after its creators Robert Floyd and Stephen Warshall, is a fundamental algorithm in computer science and graph theory. It efficiently finds the shortest paths between all pairs of nodes in a weighted graph. Unlike Dijkstra's algorithm, which focuses on single-source shortest paths, Floyd-Warshall considers all vertices simultaneously.

How Floyd-Warshall Works

1. **Initialization:**
   - Create a matrix A[0] of dimensions n x n, where n is the number of vertices.
   - Initialize each cell A[i][j] with the distance from vertex i to vertex j.
   - If there is no direct path from i to j, set the cell value to infinity.
2. **Dynamic Programming Updates:**
   - For each intermediate vertex k (from 1 to n), update the matrix A[k] using the following rule:
     - A[i][j] = min(A[i][j], A[i][k] + A[k][j])
     - This means considering the possibility of going through vertex k to improve the path from i to j.
3. **Repeat Step 2:**
   - Continue updating the matrix for all intermediate vertices.
4. The final matrix A[n] contains the shortest paths between all pairs of vertices.

Pseudocode for Floyd-Warshall Algorithm

Python

```
def floyd_warshall(graph):
    n = len(graph)
    A = [[float("inf") if i != j else 0 for j in range(n)] for i in range(n)]

    for u, v, w in graph:
        A[u][v] = w

    for k in range(n):
        for i in range(n):
            for j in range(n):
```

```
            A[i][j] = min(A[i][j], A[i][k] + A[k][j])

    return A
```

AI-generated code. Review and use carefully. More info on FAQ.

Real-World Applications

- Navigation systems: Finding optimal routes between locations.
- Transit planning: Determining efficient bus or train routes.
- Game development: Pathfinding algorithms for game characters.
- Network routing: Routing packets in computer networks.

Remember that Floyd-Warshall works for both directed and undirected graphs but does not handle graphs with negative cycles. It's a powerful tool for solving complex connectivity problems!

Experiment No 9: Implement 0/1 knapsack problem

The 0/1 Knapsack Problem is a classic optimization problem in computer science and dynamic programming. Given a set of items, each with a weight and a value, the goal is to select a subset of items to maximize the total value while ensuring that the total weight does not exceed a given capacity.

Problem Statement

Given:

- **N** items, each with a weight **wt[i]** and a value **val[i]**.
- A knapsack with a maximum weight capacity **W**.

The task is to find the maximum value that can be obtained by selecting some items to fit into the knapsack, subject to the constraint that the total weight of the selected items does not exceed W.

Approach: Dynamic Programming

We can solve the 0/1 Knapsack Problem using dynamic programming. The key idea is to build a table (often called a memoization table) to store intermediate results for subproblems. Here's the pseudocode for the dynamic programming approach:

Python

```
def knapSack(W, wt, val, n):
    # Base case: If no items or no capacity left, return 0
    if n == 0 or W == 0:
        return 0

    # If the weight of the nth item exceeds the capacity, exclude it
    if wt[n - 1] > W:
        return knapSack(W, wt, val, n - 1)

    # Otherwise, consider two cases:
    # 1. Include the nth item
    # 2. Exclude the nth item
```

```
    return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),
               knapSack(W, wt, val, n - 1))

# Example usage
profit = [60, 100, 120]
weight = [10, 20, 30]
W = 50
n = len(profit)
print("Maximum value:", knapSack(W, weight, profit, n))
```

AI-generated code. Review and use carefully. More info on FAQ.

Output:
Maximum value: 220

Time Complexity

The time complexity of the recursive solution is exponential (O(2^N)), but we can optimize it using dynamic programming. The dynamic programming approach has a time complexity of O(N*W), where N is the number of items and W is the maximum weight capacity of the knapsack.

Conclusion

The 0/1 Knapsack Problem is a fundamental problem with practical applications in resource allocation, project scheduling, and more. Dynamic programming provides an efficient way to solve it and find the optimal solution

Experiment No 10: Implement Bellman Ford algorithm to find the shortest path for a multistage graph

The Traveling Salesman Problem (TSP) is a classic optimization problem where a salesperson needs to visit a set of cities exactly once and return to the starting city, minimizing the total distance traveled. It's a challenging NP-hard problem with no known polynomial-time solution.

Here's a simple implementation of the TSP using a brute-force approach in Python:

Python

```
def traveling_salesman(graph, start):
    n = len(graph)
    vertex = [i for i in range(n) if i != start]
    min_path = float("inf")

    # Generate all (n-1)! permutations of cities
    for perm in itertools.permutations(vertex):
        current_pathweight = 0
        k = start
        for i in perm:
            current_pathweight += graph[k][i]
            k = i
        current_pathweight += graph[k][start]
        min_path = min(min_path, current_pathweight)

    return min_path

# Example usage
graph = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
```

```
start_city = 0
print("Minimum path weight:", traveling_salesman(graph, start_city))
```

AI-generated code. Review and use carefully. More info on FAQ.

Output:
Minimum path weight: 80

In this implementation:

1. We consider city 1 as the starting and ending point (since the route is cyclic).
2. We generate all possible permutations of cities (excluding the starting city).
3. For each permutation, we calculate the total path weight.
4. We keep track of the minimum path weight encountered.

The time complexity of this approach is O(n!), where n is the number of vertices in the graph. The auxiliary space is O(n) due to the vector used to store vertices.

Remember that this approach is not efficient for large graphs, but it serves as a straightforward way to understand the problem. More advanced techniques like dynamic programming and heuristics are used for practical solutions. ⊠⊠

Experiment No 11: Implement N-Queen Problem

The **N-Queen problem** is a classic example of a backtracking algorithm. The goal is to place **N chess queens** on an **N×N chessboard** in such a way that no two queens can attack each other (i.e., they are not in the same row, column, or diagonal). Let's dive into the solution.

Approach:

Start in the **leftmost column**.
If all queens are placed, return true.
For each row in the current column:
Check if the queen can be safely placed in that row (no clashes with already placed queens).
If safe, mark this [row, column] as part of the solution and recursively check if placing the queen here leads to a solution.
If it does, return true.
If not, unmark this [row, column], backtrack, and try other rows.
If no valid solution is found after trying all rows, return false.

Recursive Tree:

For better visualization, consider the recursive tree for the **4-Queen problem**:

```
Q . . .
. . . Q
. Q . .
. . Q .
```

Example Code (C++):

```
#include <bits/stdc++.h>
#define N 4
using namespace std;

void printSolution(int board[N][N]) {
   for (int i = 0; i < N; i++) {
      for (int j = 0; j < N; j++)
         cout << (board[i][j] ? "Q " : ". ");
      cout << "\n";
   }
}

bool isSafe(int board[N][N], int row, int col) {
   int i, j;
```

```
    for (i = 0; i < col; i++)
      if (board[row][i])
        return false;
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--)
      if (board[i][j])
        return false;
    for (i = row, j = col; j >= 0 && i < N; i++, j--)
      if (board[i][j])
        return false;
    return true;
  }

  bool solveNQUtil(int board[N][N], int col) {
    if (col >= N)
      return true;
    for (int i = 0; i < N; i++) {
      if (isSafe(board, i, col)) {
        board[i][col] = 1;
        if (solveNQUtil(board, col + 1))
          return true;
        board[i][col] = 0;
      }
    }
    return false;
}

bool solveNQ() {
  int board[N][N] = {{0, 0, 0, 0},
                     {0, 0, 0, 0},
                     {0, 0, 0, 0},
                     {0, 0, 0, 0}};
  if (!solveNQUtil(board, 0)) {
    cout << "No solution exists.\n";
    return false;
  }
  printSolution(board);
```

```
    return true;
}

int main() {
    solveNQ();
    return 0;
}
```

Experiment No 12: Implement N-puzzle Problem

The **N-puzzle problem** (also known as the **15-puzzle** or **8-puzzle**) is a classic sliding puzzle where you need to rearrange numbered tiles on a square grid to reach a specific goal configuration. The goal is to move the tiles from their initial arrangement to a final arrangement by sliding them into the empty space.

In the context of the **15-puzzle**, the puzzle consists of **15 numbered tiles** on a **4x4 grid**, with one tile left empty. The objective is to rearrange the tiles to match the final configuration.

Branch and Bound Algorithm for the 15-Puzzle:

1. **Problem Description**:
   - Given a 4x4 grid with 15 numbered tiles (from 1 to 15) and one empty space.
   - The goal is to reach a specific arrangement of tiles by sliding them into the empty space.
2. **Branch and Bound Approach**:
   - Branch and Bound is an optimization technique that combines backtracking and pruning to efficiently explore the search space.
   - It aims to find the optimal solution by avoiding unnecessary exploration of subproblems.
   - Here's how it works for the 15-puzzle:
3. **State Representation**:
   - Represent the current state as a 4x4 grid, where each tile is labeled with its number (1 to 15) and the empty space is represented as 0.
4. **Cost Function**:
   - Define a cost function for each state:
     - Let `f(x)` be the length of the path from the root to state `x` (the number of moves made so far).
     - Let `h(x)` be the number of non-blank tiles not in their goal position (the number of misplaced tiles).
     - The cost function is then `c(x) = f(x) + h(x)`.
5. **Branching and Exploration**:
   - Generate child states by sliding tiles into the empty space (up, down, left, or right).
   - Explore the child states in a BFS-like manner.
   - Prune sub-trees that cannot lead to an optimal solution (i.e., if `c(x)` exceeds the current best solution).
6. **Backtracking and Dead Nodes**:
   - If a state cannot lead to a better solution, mark it as a "dead node."
   - Backtrack when necessary to explore other possibilities.
7. **Heuristic Function (h(x))**:
   - An algorithm is available to approximate `h(x)` (the number of moves needed to transform state `x` to a goal state).
8. **Implementation**:
   - Implement the Branch and Bound algorithm in your preferred programming language.

- Use the cost function to guide exploration and pruning.

Remember that the 15-puzzle has a large state space, so efficient pruning is crucial for solving it optimally. Good luck with your implementation! □

Experiment No 13: Implement naïve string matching problem

The **naïve string matching algorithm** (also known as the **brute-force string matching algorithm**) is a straightforward approach for finding all occurrences of a given pattern within a text. Let's dive into the details and provide an example implementation in C++.

Naïve String Matching Algorithm:

1. **Problem Statement:**
   - Given a text string T of length n and a pattern string P of length m, find all occurrences of P within T.
   - You may assume that n > m.
2. **Approach:**
   - Slide the pattern P over the text T one character at a time.
   - Check if the characters match at each position.
   - If a match is found, slide the pattern by 1 again to check for subsequent matches.
3. **Algorithm Implementation (C++):**

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

void search(const string& pat, const string& txt) {
    int M = pat.size();
    int N = txt.size();
    for (int i = 0; i <= N - M; i++) {
        int j;
        for (j = 0; j < M; j++) {
            if (txt[i + j] != pat[j])
                break;
        }
        if (j == M)
            cout << "Pattern found at index " << i << endl;
    }
}

int main() {
    string txt = "AABAACAADAABAAABAA";
    string pat = "AABA";
    search(pat, txt);
    return 0;
}
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 13
```

31. **Time Complexity:**
    - Best Case: O(n) when the pattern is found at the very beginning of the text (or very early on).
    - Worst Case: O(n^2) when the pattern doesn't appear in the text at all or appears only at the very end.
    - In the worst case, for each position in the text, the algorithm may need to compare the entire pattern against the text.
32. **Space Complexity:**
    - Auxiliary Space: O(1) (no extra space needed for data structures)

Experiment No 14: Implement KMP algorithm for string matching

The **Knuth-Morris-Pratt (KMP) algorithm** is a powerful string-searching algorithm that efficiently finds occurrences of a given pattern within a larger text string. It avoids unnecessary character comparisons by utilizing information from previously matched characters. Let's delve into the details and provide an example implementation in C++.

Knuth-Morris-Pratt (KMP) Algorithm:

1. **Problem Statement:**
   - Given a text string T of length n and a pattern string P of length m, find all occurrences of P within T.
   - You may assume that n > m.
2. **Key Idea:**
   - The KMP algorithm improves upon the naive approach by avoiding redundant character comparisons.
   - It constructs a **Longest Proper Prefix which is also Suffix (LPS)** array for the pattern.
3. **Algorithm Steps:**
   - Preprocess the pattern P to create the LPS array.
   - Slide the pattern over the text, comparing characters:
     - If a mismatch occurs at position j in the pattern, use the LPS array to determine the next position to compare.
     - Update j based on the LPS value.
     - Continue until the entire pattern is matched or a mismatch occurs.
4. **Example Implementation (C++):**

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

void computeLPS(const string& pat, vector<int>& lps) {
    int m = pat.size();
    int len = 0;  // Length of the previous longest prefix suffix
    lps[0] = 0;

    int i = 1;
    while (i < m) {
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0)
                len = lps[len - 1];
            else {
                lps[i] = 0;
```

```
            i++;
        }
    }
}
void search(const string& pat, const string& txt) {
    int m = pat.size();
    int n = txt.size();
    vector<int> lps(m, 0);
    computeLPS(pat, lps);

    int i = 0, j = 0;
    while (i < n) {
        if (pat[j] == txt[i]) {
            i++;
            j++;
        }
        if (j == m) {
            cout << "Pattern found at index " << i - j << endl;
            j = lps[j - 1];
        } else if (i < n && pat[j] != txt[i]) {
            if (j != 0)
                j = lps[j - 1];
            else
                i++;
        }
    }
}

int main() {
    string txt = "AABAACAADAABAABA";
    string pat = "AABA";
    search(pat, txt);
    return 0;
}
```

Output:

```
Pattern found at index 0
Pattern found at index 9
Pattern found at index 12
```

Time Complexity:

- O(n + m) in the worst case.

Space Complexity:

- O(m) for the LPS array.

The KMP algorithm efficiently handles cases where many matching characters are followed by a mismatch, making it a valuable tool for string searching. Feel free to adapt this approach to other programming languages as needed. Happy coding!