1] Implement depth first search algorithm and Breadth First Search algorithm, Use an undirected graph and develop a recursive algorithm for searching all the vertices of a graph or tree data structure.

What is Depth-first search?

- Depth-first search is a recursive algorithm used to traverse tree or graph data structure.
- It is called depth-first search because it starts with root node and follows each path to its greatest depth before moving to next path.
- It uses stack data structure i.e. LIFO(Last In First Out).
- Time Complexity is O(V+E)
  where, V = Vertices,
  E = Edges

Depth First Search (DFS)

In this tutorial, you will learn about depth first search algorithm with examples and pseudocode. Also, you will learn to implement DFS in C, Java, Python, and C++.

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

Depth First Search Algorithm

A standard DFS implementation puts each vertex of the graph into one of two categories:

1. Visited
2. Not Visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

The DFS algorithm works as follows:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

BFS algorithm

In this article, we will discuss the BFS algorithm in the data structure. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unexplored nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

There are many ways to traverse the graph, but among them, BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node.

- BFS can be used to find the neighboring locations from a given source location.
- In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
- BFS can be used in web crawlers to create web page indexes. It is one of the main algorithms that can be used to index web pages. It starts traversing from the source page and follows the links associated with the page. Here, every web page is considered as a node in the graph.
- BFS is used to determine the shortest path and minimum spanning tree.

- BFS is also used in Cheney's technique to duplicate the garbage collection.
- It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Algorithm

The steps involved in the BFS algorithm to explore a graph are given as follows -

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

(waiting state)

[END OF LOOP]

This code is an implementation of a tree data structure and includes functions for performing depth-first search (DFS) and breadth-first search (BFS) on the tree. It also includes functions to search for a specific node in the tree using DFS and BFS.

Let's go through the code step by step:

1. The code begins by including the necessary libraries: `iostream`, `queue`, and `vector`. These libraries are required for input/output operations, queue implementation, and vector data structure respectively.

2. The `TreeNode` class is defined, representing a node in the tree. It has the following member variables:

- `val`: A character representing the value of the node.
- `child`: A vector of `TreeNode` pointers representing the children of the node.
- `noc`: An integer representing the number of children for the node.
- `flag`: A boolean flag used for marking nodes during search operations.

The class also has a constructor that takes the value of the node (`val`) and the number of children (`N`) as input. Inside the constructor, it prompts the user to enter the child nodes for the current node by taking input for each child's value and the number of children it has. It creates a new `TreeNode` object for each child and adds it to the `child` vector.

3. The `searchFlag` variable is declared outside the class. It is a boolean flag used during search operations to determine if the key has been found.

4. The `searchDFS` function is defined. It performs a depth-first search starting from a given `root` node and searches for a `key` character in the tree. It uses recursion to traverse the tree in a depth-first manner. If the `root` node is not null and the `searchFlag` is false (key not found yet), it prints the value of the current node (`root->val`). If the current node's value is equal to the `key`, it sets the `searchFlag` to true and returns. Otherwise, it recursively calls `searchDFS` for each child of the current node.

5. The `DFS` function is defined to perform a depth-first search on the tree. It starts from a given `root` node and prints the values of nodes in a depth-first manner. It uses recursion to traverse the tree. If the `root` node is not null, it prints the value of the current node and recursively calls `DFS` for each child of the current node.

6. The `BFS` function is defined to perform a breadth-first search on the tree. It starts from a given `root` node and prints the values of nodes in a breadth-first manner. It uses a queue data structure to keep track of the nodes to visit. It initializes the queue with the `root` node, sets the `flag` of the `root` node to true, and enters a loop that continues until the queue is empty. Inside the loop, it dequeues a node from the front of the queue, prints its value, and enqueues its children (whose `flag` is false) into the queue.

7. The `searchBFS` function is defined to search for a `key` character in the tree using breadth-first search. It starts from a given `root` node and performs a breadth-first search, looking for the `key`. It uses a queue to keep track of the nodes to visit. It initializes the queue with the `root` node and enters a loop that continues until the queue is empty. Inside the loop, it dequeues a node from the front of the queue, marks it as visited (`flag = true`), prints its value, and checks if its value is equal to the `key`.

If the key is found, it returns. If not, it enqueues the children of the current node into the queue.

8. In the `main` function, the user is prompted to enter the value of the root node and the total number of children for the root node. Based on this input, a `TreeNode` object is created with the given root value and number of children.

9. The `DFS` function is called with the `root` node to perform a depth-first search and print the values of nodes in a depth-first manner.

10. The `BFS` function is called with the `root` node to perform a breadth-first search and print the values of nodes in a breadth-first manner.

11. The user is prompted to enter a destination character.

12. The `searchDFS` function is called with the `root` node and the destination character to search for the destination using depth-first search. It prints the path taken in the tree to reach the destination.

13. The `searchBFS` function is called with the `root` node and the destination character to search for the destination using breadth-first search. It prints the path taken in the tree to reach the destination.

14. Finally, the program returns 0 to indicate successful execution.

Overall, the code allows you to create a tree, perform DFS and BFS on the tree, and search for specific nodes using DFS and BFS algorithms.

2] Implement A star Algorithm for any game search problem.

What is A* Search Algorithm?
A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.
Why A* Search Algorithm?
Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.
And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.
What A* Search Algorithm does is that at each step it picks the node according to a value-‘**f**’ which is a parameter equal to the sum of two other parameters – ‘**g**’ and ‘**h**’. At each step it picks the node/cell having the lowest ‘**f**’, and process that node/cell.
We define ‘**g**’ and ‘**h**’ as simply as possible below
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’ which are discussed in the later sections.
Algorithm
We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
```

```
    put the starting node on the open

    list (you can leave its f at zero)

3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search

        ii) else, compute both g and h for successor
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

          successor.f = successor.g + successor.h

        iii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor

        iV) if a node with the same position as
            successor  is in the CLOSED list which has
            a lower f than successor, skip this successor
            otherwise, add  the node to the open list
     end (for loop)

    e) push q on the closed list
    end (while loop)
```

The given code is an implementation of the 8-puzzle problem using the A* search algorithm. The 8-puzzle is a sliding puzzle that consists of a 3x3 grid with 8 numbered tiles and one empty space. The goal is to arrange the tiles in a specific order by sliding them into the empty space.

Let's go through the code step by step:

1. The code includes the necessary libraries `iostream`, `vector`, and `algorithm`.

2. The code defines a struct named `node` that represents a state in the puzzle. It contains the following members:

   - `hr`: Heuristic value for the state.
   - `sq`: 2D vector representing the current puzzle state.
   - `r` and `c`: The row and column position of the empty space in the puzzle.

3. The code defines a comparison function `comp` that compares two `node` pointers based on their heuristic values. This function is used for sorting the nodes in the A* search algorithm.

4. The code declares a global variable `level` initialized to 0. This variable is used for tracking the level or depth of the search.

5. The code defines a recursive function `Eight` that implements the A* search algorithm. It takes the following parameters:

   - `ol`: A reference to a vector of `node` pointers, which represents the open list.
   - `goal`: A 2D vector representing the goal state of the puzzle.
   - `r` and `c`: The row and column position of the empty space in the current puzzle state.

6. The `Eight` function begins by printing the current level, heuristic value, and the current puzzle state.

7. If the current puzzle state is the goal state, the program exits.

8. The function generates new states by swapping the empty space with adjacent tiles (up, down, left, and right). For each possible swap, it calculates the heuristic value by counting the number of misplaced tiles in the new state compared to the goal state.

9. A new `node` is created for each new state, and its members are initialized accordingly.

10. The new node is added to the open list (`ol`).

11. The current node is removed from the open list.

12. The open list is sorted based on the heuristic values using the `comp` function.

13. The `Eight` function is recursively called with the new open list, goal state, and the position of the empty space in the next node.

14. The `main` function is defined.

15. A variable `count` is initialized to 0. It represents the number of misplaced tiles in the initial state compared to the goal state.

16. Two vectors, `start` and `goal`, are defined to store the initial and goal states of the puzzle, respectively. The user is prompted to input the values for these vectors.

17. The position of the empty space (`r` and `c`) in the initial state is determined.

18. The number of misplaced tiles (`count`) is calculated by comparing the initial and goal states.

19. A new `node` is created for the initial state and added to the open list.

20. The `Eight` function is called with the open list, goal state, and the position of the empty space in the initial state.

21. The program terminates.

Overall, the code uses the A* search algorithm to solve the 8-puzzle problem by exploring possible states and selecting the ones with the lowest heuristic values. It continues this process until it reaches the goal state or exhausts all possible states.

3] Implement Greedy search algorithm for any of the following application: I. Selection Sort

***Selection sort** is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.*

The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.

The given code is an implementation of the selection sort algorithm in C++. It sorts an array of integers in ascending order.

Let's go through the code step by step:

1. The code starts by including the necessary header files, iostream, which allows input and output operations.

2. The function `selection_sort` is defined. It takes an integer array `arr` and the size of the array `n` as parameters.

3. Inside the `selection_sort` function, there are two nested for loops. The outer loop iterates from 0 to `n-1` (excluding the last element of the array). This loop selects one element at a time as the minimum element.

4. At the beginning of each iteration of the outer loop, the variable `min` is initialized with the current value of `i`, indicating that the minimum element is assumed to be at index `i`.

5. The inner loop starts from `i` and iterates till `n-1`. It compares each element with the current minimum element (`arr[min]`). If a smaller element is found, the index of that element is stored in `min`.

6. After the inner loop completes, the minimum element in the remaining unsorted portion of the array is found. The minimum element is then swapped with the element at index `i` using a temporary variable `temp`.

7. The code prints the current state of the array after each pass of the outer loop. It displays the elements in the array separated by spaces.

8. Once the outer loop completes, the array is sorted in ascending order.

9. After sorting, the sorted array is printed by iterating through the elements and displaying them.

10. The `main` function is defined, which is the entry point of the program.

11. Inside the `main` function, the user is prompted to enter the size of the array.

12. An integer array `arr` of size `n` is declared.

13. The user is then prompted to enter the elements of the array one by one, and the elements are stored in the array using a for loop.

14. The original array is printed before sorting.

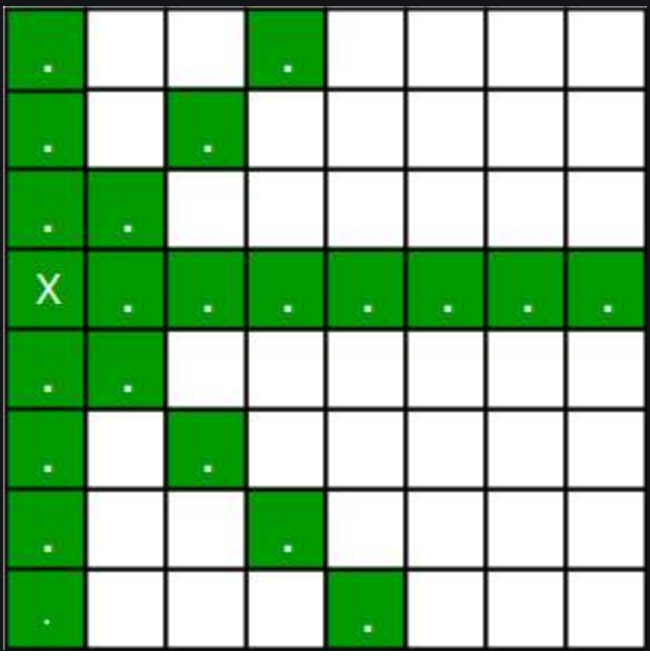
15. The `selection_sort` function is called with the array and its size as arguments.

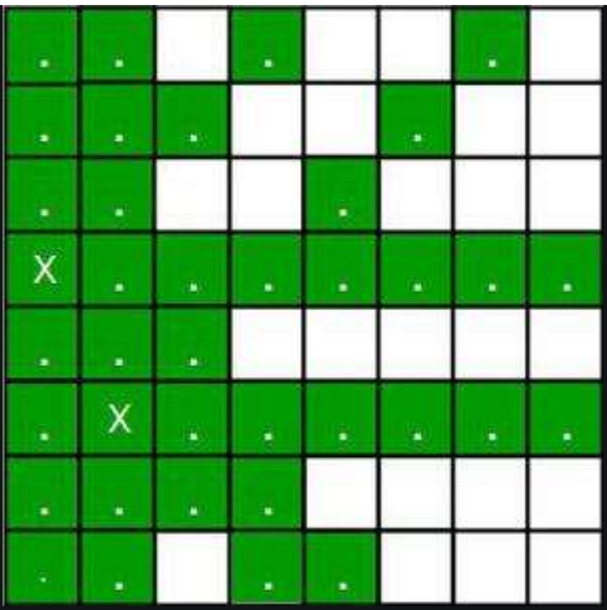
16. Finally, the program returns 0, indicating successful execution.

The commented code at the end of the file is an alternative example that directly provides the size and elements of the array within the `main` function instead of taking user input. The purpose is to show an example of the code's usage with specific values for demonstration purposes.

4] Implement a solution for a Constraint Satisfaction Problem using Branch and Bound and Backtracking for n-queens problem or a graph coloring problem

The **N queens puzzle** is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal.
The backtracking Algorithm for N-Queen is already discussed here. In a backtracking solution, we backtrack when we hit a dead end. ***In Branch and Bound solution, after building a partial solution, we figure out that there is no point going any deeper as we are going to hit a dead end***.
Let's begin by describing the backtracking solution. "The idea is to place queens one by one in different columns, starting from the leftmost column. When we place a queen in a column, we check for clashes with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as part of the solution. If we do not find such a row due to clashes, then we backtrack and return false."





Placing 1st queen on (3, 0) and 2nd queen on (5, 1)

1. For the 1st Queen, there are total 8 possibilities as we can place 1st Queen in any row of first column. Let's place Queen 1 on row 3.
2. After placing 1st Queen, there are 7 possibilities left for the 2nd Queen. But wait, we don't really have 7 possibilities. We cannot place Queen 2 on rows 2, 3 or 4 as those cells are under attack from Queen 1. So, Queen 2 has only 8 – 3 = 5 valid positions left.
3. After picking a position for Queen 2, Queen 3 has even fewer options as most of the cells in its column are under attack from the first 2 Queens.

We need to figure out an efficient way of keeping track of which cells are under attack. In previous solution we kept an 8-by-8 Boolean matrix and update it each time we placed a queen, but that required linear time to update as we need to check for safe cells.
Basically, we have to ensure 4 things:
1. No two queens share a column.
2. No two queens share a row.
3. No two queens share a top-right to left-bottom diagonal.
4. No two queens share a top-left to bottom-right diagonal.

Number 1 is automatic because of the way we store the solution. For number 2, 3 and 4, we can perform updates in O(1) time. The idea is to keep **three Boolean arrays that tell us which rows and which diagonals are occupied**.

Lets do some pre-processing first. Let’s create two N x N matrix one for / diagonal and other one for \ diagonal. Let’s call them slashCode and backslashCode respectively. The trick is to fill them in such a way that two queens sharing a same /diagonal will have the same value in matrix slashCode, and if they share same \diagonal, they will have the same value in backslashCode matrix.
For an N x N matrix, fill slashCode and backslashCode matrix using below formula –
slashCode[row][col] = row + col
backslashCode[row][col] = row – col + (N-1)

This code solves the N-Queens problem, which is a classic problem in computer science where you have to place N queens on an NxN chessboard such that no two queens threaten each other. The code uses a backtracking algorithm to find all possible solutions.

Now, let's go through the code step by step:

1. The code starts by including the necessary header files, `iostream`, and `iostream`, and then declares the `std` namespace.

2. Two global variables are declared: `count` to keep track of the number of solutions found, and `grid`, which represents the chessboard.

3. The `display_solution` function is defined. It takes the size of the chessboard `n` as a parameter and prints the current configuration of the chessboard. It iterates over each cell of the grid and prints its value.

4. The `isSafe` function is defined to check if it is safe to place a queen in a given position. It takes the column `col`, row `row`, and size of the chessboard `n` as parameters. It checks three conditions: whether there is a queen in the same column, whether there is a queen in the upper left diagonal, and whether there is a queen in the upper right diagonal. If any of these conditions are true, it returns false indicating that it is not safe to place a queen in that position. Otherwise, it returns true.

5. The `solve` function is defined. It takes the size of the chessboard `n` and the current row `row` as parameters. It uses recursion and backtracking to find all possible solutions. If the current row is equal to `n`, it means all the queens have been placed successfully, so it calls the `display_solution`

function and returns true. Otherwise, it tries to place a queen in each column of the current row. If it is safe to place a queen in a particular column, it marks that position in the `grid` array as 1 and makes a recursive call to solve for the next row (`row+1`). If the recursive call returns true, it means a solution has been found, so it sets `res` to true. If the recursive call returns false, it means there is no solution from that position, so it backtracks by setting the grid position to 0. Finally, it returns the value of `res`, indicating whether a solution was found or not.

6. The `main` function is defined. It starts by taking input from the user for the number of queens `n` they want to place. It initializes the `grid` array by setting all its elements to 0.

7. It calls the `solve` function with `n` and 0 as arguments and stores the return value in the `res` variable.

8. It then prints the total number of possible positions to place the queens, which is stored in the `count` variable.

9. If `res` is false, it means no solution was found, so it prints "Solution does not exist."

10. Finally, the code returns 0 to indicate successful program execution.

In summary, the code uses backtracking and recursion to solve the N-Queens problem and prints all possible solutions.