

Searching Algorithms

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. In this tutorial, we are mainly going to focus upon searching in an array. When we search an item in an array, there are two most common algorithms used based on the type of input array.

- **Linear Search:** It is used for an unsorted array. It mainly does one by one comparison of the item to be search with array elements. It takes linear or $O(n)$ Time.
- **Binary Search:** It is used for a sorted array. It mainly compares the array's middle element first and if the middle element is same as input, then it returns. Otherwise it searches in either left half or right half based on comparison result (Whether the mid element is smaller or greater). This algorithm is faster than linear search and takes $O(\log n)$ time.

1- Interpolation Search

The Interpolation Search is an improvement over [Binary Search](#) for instances, where the values in a sorted array are uniformly distributed. Interpolation constructs new data points within the range of a discrete set of known data points. Binary Search always goes to the middle element to check. On the other hand, interpolation search may go to different locations according to the value of the key being searched. For example, if the value of the key is closer to the last element, interpolation search is likely to start search toward the end side. To find the position to be searched, it uses the following formula.

arr[] ==> Array where elements need to be searched

x ==> Element to be searched

lo ==> Starting index in arr[]

hi ==> Ending index in arr[]

$$pos = lo + \left\lceil \frac{(x - arr[lo]) * (hi - lo)}{(arr[hi] - arr[Lo])} \right\rceil$$

Time Complexity:

Average case: $O(\log^2(\log^2 n))$

If the elements are uniformly distributed (evenly spaced), the algorithm is highly efficient and outperforms binary search.

Worst case: $O(n)$

This occurs when the data is non-uniformly distributed, such as in exponentially increasing keys or clustered data, causing the search to degrade to linear time.

Best Case: $O(1)$

Achieved if the target value is found at the first estimated probe position.

Auxiliary Space Complexity: $O(1)$

The algorithm requires only a constant amount of extra memory to store index pointers and variables during the search process.

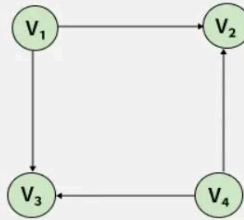
Graph Algorithms

Graph is a non-linear data structure like [tree data structure](#). A Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The vertices are connected with each other through edges.

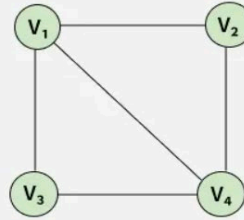
- The limitation of tree is, it can only represent hierarchical data. For situations where nodes or vertices are randomly connected with each other other, we use Graph.
- Example situations where we use graph data structure are, a social network, a computer network, a network of locations used in GPS and many more examples where different nodes or vertices are connected without any hierarchic or constraint on structure.

The following images show different types of graphs that see when solving graph problems

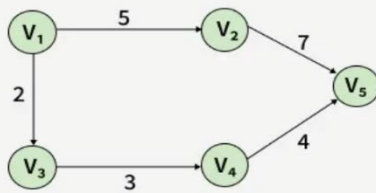
Directed Graph



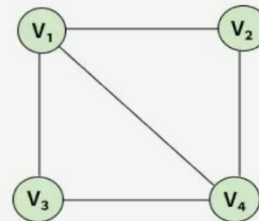
Undirected Graph



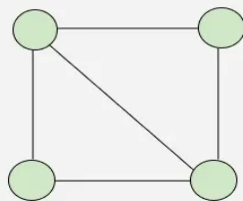
Weighted Graph



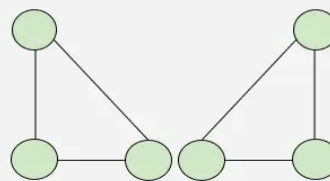
Unweighted Graph



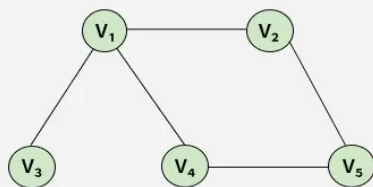
Connected Graph



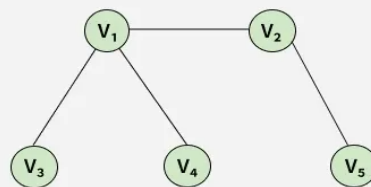
Disconnected Graph



Cyclic Graph



Acyclic Graph



A **Graph** is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E)**.

Representations of Graph

Here are the two most common ways to represent a graph : For simplicity, we are going to consider only **unweighted graphs** in this post.

1. Adjacency Matrix
2. Adjacency List

Adjacency Matrix Representation

An adjacency matrix is a way of representing a graph as a boolean matrix of (0's and 1's).

Let's assume there are n vertices in the graph So, create a 2D matrix `adjMat[n][n]` having dimension $n \times n$.

Greedy Algorithms

Greedy algorithms are a class of algorithms that make locally optimal choices at each step with the hope of finding a global optimum solution.

- At every step of the algorithm, we make a choice that looks the best at the moment. To make the choice, we sometimes sort the array so that we can always get the next optimal choice quickly. We sometimes also use a priority queue to get the next optimal item.
- After making a choice, we check for constraints (if there are any) and keep picking until we find the solution.
- Greedy algorithms do not always give the best solution. For example, in coin change and 0/1 knapsack problems, we get the best solution using Dynamic Programming.

1- Breadth First Search or BFS for a Graph

Breadth First Search (BFS) is a graph traversal algorithm that starts from a source node and explores the graph level by level. First, it visits all nodes directly adjacent to the source. Then, it moves on to visit the adjacent nodes of those nodes, and this process continues until all reachable nodes are visited.

- BFS is different from [DFS](#) in a way that closest vertices are visited before others. We mainly traverse vertices **level by level**.
- Popular graph algorithms like [Dijkstra's shortest path](#), [Kahn's Algorithm](#), and [Prim's algorithm](#) are based on BFS.
- BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an **unweighted graph** and many more problems.

Time Complexity: $O(V + E)$, The for loop ensures BFS starts from every unvisited vertex to cover all components, but the visited array ensures each vertex and edge is processed only once, keeping the total time complexity to be linear.

Auxiliary Space: $O(V)$, using a queue to keep track of the vertices that need to be visited.

Applications of BFS in Graphs

BFS has various applications in graph theory and computer science, including:

- Shortest Path Finding
- Cycle Detection
- Connected Components
- Network Routing

2- Depth First Search or DFS for a Graph

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices **one by one**. When we traverse an **adjacent vertex**, we completely finish the traversal of all vertices reachable through that adjacent vertex. This is similar to **Preorder Traversal of Binary Tree**, where we

first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.

Time complexity: $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph.

Auxiliary Space: $O(V + E)$, since an extra visited array of size V is required, And stack size for recursive calls to dfsRec function.

3- Dijkstra's Algorithm

Given a **weighted undirected** graph and a source vertex . We need to find the shortest path distances from the source vertex to all other vertices in the graph.

- Use of priority Queue

The priority queue always selects the node with the smallest current distance, ensuring that we explore the shortest paths first and avoid unnecessary processing of longer paths. Dijkstra's algorithm always picks the node with the minimum distance first. By doing so, it ensures that the node has already checked the shortest distance to all its neighbors. If this node appears again in the priority queue later, we don't need to process it again, because its neighbors have already checked the minimum possible distances

Time Complexity: $O((V+E)*\log V)$, Where E is the number of edges and V is the number of vertices.

Auxiliary Space: $O(V+E)$