**Algorithms**

**Dijkstra’s Algorithm**

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices.

It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.

Dijkstra's algorithm is often considered to be the most straightforward algorithm for solving the shortest path problem.

**Google Maps** uses the Dijkstra algorithm to show the shortest distance between the source and destination.

**Airlines** use Dijkstra’s algorithm to plan flight paths that minimize fuel consumption and reduce travel time.

**Algorithm:**

1. Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.
2. Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.
3. For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.
4. We are now done with the current vertex, so we mark it as visited. A visited vertex is not checked again.
5. Go back to step 2 to choose a new current vertex and keep repeating these steps until all vertices are visited.
6. In the end we are left with the shortest path from the source vertex to every other vertex in the graph.

A diagram of a triangle with numbers and lines

Description automatically generated with medium confidence

Output: 0 4 12 19 21 11 9 8 14

Explanation: The distance from 0 to 1 = 4.

The minimum distance from 0 to 2 = 12. 0->1->2

The minimum distance from 0 to 3 = 19. 0->1->2->3

The minimum distance from 0 to 4 = 21. 0->7->6->5->4

The minimum distance from 0 to 5 = 11. 0->7->6->5

The minimum distance from 0 to 6 = 9. 0->7->6

The minimum distance from 0 to 7 = 8. 0->7

The minimum distance from 0 to 8 = 14. 0->1->2->8

**Time Complexity**: O(V2), V as the number of vertices in our graph

**Disadvantages:**

1. Dijkstra's algorithm does not work for graphs with negative edges
2. inefficiency on Dense Graphs
3. High Memory Usage
4. Slower on Unweighted Graphs

**Graph Traversal:**

To traverse a Graph means to start in one vertex, and go along the edges to visit other vertices until all vertices, or as many as possible, have been visited.

The two most common ways a Graph can be traversed are:

Depth First Search (DFS)

Breadth First Search (BFS)

**DFS** is usually implemented using a **stack** or recursion (which utilizes the call stack), while **BFS** is usually implemented using a **queue**.

**Depth First Search**

Depth First Search is said to go "deep" because it visits a vertex, then an adjacent vertex, then that vertex's adjacent vertex, and so on, and in this way, the distance from the starting vertex increases for each recursive iteration.

Low Memory Usage as it stores only the nodes on the current path in the stack

Efficient for Deep Solutions.

DFS can be used to detect cycles in a graph and to find connected components in undirected graphs.

DFS is ideal for problems involving backtracking, such as solving mazes, and puzzles, or exploring all possible configurations in combinatorial problems.

**Algorithm**:

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 that aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

**Time Complexity**: O(V + E), where V is the number of nodes and E is the number of edges.

**Disadvantages**:

1. If the graph has cycles or is infinite, DFS may get stuck in an infinite loop, especially if it revisits the same nodes repeatedly.
2. DFS does not guarantee the shortest path to a solution
3. Recursive implementations of DFS can lead to a stack overflow if the graph is very deep or if the recursion depth is too large.

**Breadth-First Search**

Breadth First Search visits all adjacent vertices of a vertex before visiting neighboring vertices to the adjacent vertices. This means that vertices with the same distance from the starting vertex are visited before vertices further away from the starting vertex are visited.

To build an index by search index

Path finding algorithm

For GPS navigation

**Algorithm:**

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

The algorithm works as follows:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue 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 back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

**Time Complexity**: O (V + E), where V is the number of nodes and E is the number of edges.

**Disadvantages:**

1. Managing the queue used in BFS can be challenging, especially when the queue grows large in width-heavy graphs. This can lead to performance bottlenecks and increased processing time, particularly in environments with limited computational resources.
2. In environments with limited memory, such as embedded systems or mobile devices, the high memory usage of BFS can be a critical disadvantage, making it less suitable compared to DFS
3. complex when dealing with multiple sources.

**Divide and Conquer**

breaking down a complex problem into smaller, more manageable parts, solving each part individually, and then combining the solutions to solve the original problem.

A diagram of a number

Description automatically generated with medium confidence

Merge sort is a classic example of a divide-and-conquer sorting algorithm.

**Divide and Conquer Applications**

* Binary Search
* Merge Sort
* Quick Sort
* Strassen's Matrix multiplication
* Karatsuba Algorithm

**Algorithm:**

Here are the steps involved:

1. **Divide**: Divide the given problem into sub-problems using recursion.
2. **Conquer**: Solve the smaller sub-problems recursively. If the subproblem is small enough, then solve it directly.
3. **Combine:** Combine the solutions of the sub-problems that are part of the recursive process to solve the actual problem.

divide each subpart recursively into two halves until you get individual elements.A diagram of numbers and arrows

Description automatically generated

combine the individual elements in a sorted manner. Here, **conquer** and **combine**A diagram of numbers and symbols

Description automatically generated

**Time Complexity:**

T(n) = aT(n/b) + f(n),

where,

n = size of input

a = number of subproblems in the recursion

n/b = size of each subproblem. All subproblems are assumed to have the same size.

f(n) = cost of the work done outside the recursive call, which includes the cost of dividing the problem and cost of merging the solutions

**Disadvantages:**

**Additional Memory Usage**: Many divide-and-conquer algorithms require additional memory to store intermediate results or subproblems.

**Overkill for Simple Problems**: For small or simple problems, divide-and-conquer may introduce unnecessary complexity and overhead compared to more straightforward, iterative approaches.

**Implementation Complexity**: Divide-and-conquer algorithms can be more challenging to implement correctly, particularly for problems where dividing the problem or merging the results is non-trivial.

**Fractional Knapsack Problem**

The items are broken to maximize the profit. The problem in which we break the item is known as a Fractional knapsack problem.

The fractional knapsack problem can be solved optimally using a **greedy algorithm**

The greedy approach guarantees an **optimal solution** for the fractional knapsack problem. This means that the algorithm will always yield the maximum possible value that can be carried in the knapsack.

the fractional knapsack allows for the inclusion of fractions of items. This flexibility means that the algorithm can more easily **maximize the total value**.

The fractional knapsack model closely mirrors **many real-world scenarios** where items can be divided, such as allocating resources or **investments**, making it highly practical.

**Algorithm:**

* Consider all the items with their weights and profits mentioned respectively.
* Calculate Pi/Wi of all the items and sort the items in descending order based on their Pi/Wi values.
* Without exceeding the limit, add the items into the knapsack.
* If the knapsack can still store some weight, but the weights of other items exceed the limit, the fractional part of the next time can be added.
* Hence, giving it the name fractional knapsack problem.

**Time Complexity:** O(N \* log N)

**Disadvantages:**

Not Suitable for Indivisible Items

**Lack of Realism in Some Contexts**: In some situations, the ability to take fractions of items may oversimplify the problem and lead to solutions that aren't practical or feasible in real-life scenarios.

**May Not Generalize to Other Problems**: The reliance on a greedy approach means that the techniques used to solve the fractional knapsack problem don't necessarily apply to others.

My [GitHub](https://github.com/manojreddy24/people_tech) link contains documentation and practiced problems https://github.com/manojreddy24/people\_tech