Breadth-first search (BFS

**An algorithm for searching or traversing trees or graphs is called breadth-first search (BFS).**The algorithm uses a queue data structure to keep track of the next node to visit. The basic process can be described as follows:

1. Put the starting node in the queue.

2. Remove a node from the queue and inspect it.

3. Return the search key if the node contains it.

4.Enqueue any of the node's children and move on to step 2 if the node does not contain the search key.

The shortest path will always be returned by BFS if the graph is unweighted or if all edges have the same weight, which is one of the algorithm's biggest benefits. If the graph is weighted, however, BFS will return a path, but it will not be the shortest.

Time Complexity: O(V+E)

Space Complexity: O(V)

Finally, BFS is an efficient algorithm for traversing or searching tree or graph data structures. It is simple to implement and has numerous applications. It is, however, inefficient for determining the shortest path in a weighted graph.

#include <iostream>

#include <list>

#include <queue>

using namespace std;

class Graph {

int V; // Number of vertices

list<int> \*adj; // Pointer to an array containing adjacency lists

public:

Graph(int V); // Constructor

void addEdge(int v, int w); // Function to add an edge to the graph

void BFS(int s); // Prints BFS traversal from a given source s

};

Graph::Graph(int V) {

this->V = V;

adj = new list<int>[V];

}

void Graph::addEdge(int v, int w) {

adj[v].push\_back(w); // Add w to v's list

}

void Graph::BFS(int s) {

// Mark all the vertices as not visited

bool \*visited = new bool[V];

for(int i = 0; i < V; i++)

visited[i] = false;

// Create a queue for BFS

queue<int> queue;

// Mark the current node as visited and enqueue it

visited[s] = true;

queue.push(s);

// 'i' will be used to get all adjacent vertices of a vertex

list<int>::iterator i;

while(!queue.empty()) {

// Dequeue a vertex from queue and print it

s = queue.front();

cout << s << " ";

queue.pop();

// Get all adjacent vertices of the dequeued vertex s

// If a adjacent has not been visited, then mark it visited

// and enqueue it

for (i = adj[s].begin(); i != adj[s].end(); ++i) {

if (!visited[\*i]) {

visited[\*i] = true;

queue.push(\*i);

}

}

}

}

// Driver program to test methods of graph class

int main() {

// Create a graph given in the above diagram

Graph g(4);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 2);

g.addEdge(2, 0);

g.addEdge(2, 3);

g.addEdge(3, 3);

cout << "Following is Breadth First Traversal "

<< "(starting from vertex 2) \n";

g.BFS(2);

return 0;

}

Depth First Search (DFS)

A way to navigate a graph or tree data structure is by using the Depth First Search (DFS) algorithm. It begins at the root node and travels as far as possible along each branch before returning. The next vertex to be visited is kept track of by the algorithm using a stack, and the vertex at the top of the stack is visited.

A sample graph with 6 nodes and 7 edges was created to demonstrate the DFS algorithm's implementation, as shown in the diagram following table:

1

/ \

2 3

/ \ / \

4 5 6 7

After starting at node 1, the algorithm moves on to node 2, node 3, node 4, and finally node 5. There are no longer any unvisited nodes along this branch at this point, so the algorithm returns to node 2 and visits node 3. After that, the algorithm visits nodes 6 and 7, before returning to node 1 and marking it as fully visited. This same DFS algorithm can be implemented recursively or iteratively.

Following are some examples of how to use the recursive method:

void DFS(int v, bool visited[]) {

visited[v] = true;

cout << v << " ";

for (int i = 0; i < V; i++) {

if (!visited[i] && adj[v][i] == 1)

DFS(i, visited);

}

}

The iterative method can be implemented as follows:

void DFS(int v) {

stack<int> stack;

bool visited[V];

memset(visited, false, sizeof(visited));

stack.push(v);

while (!stack.empty()) {

int s = stack.top();

stack.pop();

if (!visited[s]) {

visited[s] = true;

cout << s << " ";

for (int i = 0; i < V; i++) {

if (adj[s][i] == 1 && !visited[i])

stack.push(i);

}

}

}

}

Finally, the DFS algorithm is a straightforward and efficient method for traversing a graph or tree data structure. It has a time complexity of O(V+E), where V and E are the number of vertices and edges, respectively. As it requires the entire call stack, it can, however, be very memory-intensive. The DFS algorithm's recursive and iterative implementations were demonstrated using a sample graph, and both implementations have the same time and space complexity.

Uninformed Search Algorithm (UCS)

Introduction:

We will discuss the Uninformed Search Algorithm (UCS), also known as Breadth-First Search (BFS), in this lab report, which is an algorithm for traversing or searching tree or graph data structures. In contrast to the Depth-First Search (DFS) algorithm, which explores as far as possible along each branch before backtracking, UCS explores all vertices at the current depth level before moving on to vertices at the next depth level.

Methodology:

A sample graph with 6 nodes and 7 edges was created to demonstrate the implementation of the UCS algorithm, as shown in the diagram below:

1

/ \

2 3

/ \ / \

4 5 6 7

The algorithm starts at node 1 and visits all its adjacent nodes which are 2 and 3. It then visits the next level of nodes which are 4, 5, 6, and 7. The algorithm continues this process until all the nodes in the graph have been visited.

A queue data structure can be used to implement the UCS algorithm. The basic steps for implementation are as follows:

1. Begin the queue with the first node and mark it as visited.

2. Remove a node from the queue and process it.

3. Put all of the dequeued node's unvisited neighboring nodes into queue.

4. Carry on in this manner until the line is free.

The pseudocode for the implementation is as follows:

void BFS(int v, bool visited[]) {

queue<int> queue;

visited[v] = true;

queue.push(v);

while (!queue.empty()) {

int s = queue.front();

queue.pop();

cout << s << " ";

for (int i = 0; i < V; i++) {

if (!visited[i] && adj[s][i] == 1) {

visited[i] = true;

queue.push(i);

}

}

}

}

Results: Using the sample graph, the UCS algorithm was able to visit each node in the right order (1, 2, 3, 4, 5, 6, 7). O(V+E), where V is the number of vertices and E is the number of edges, is the algorithm's time complexity. However, because it requires storage for the queue and the visited array, it has a space complexity of O(V).

Conclusion:

We discussed the Uninformed Search Algorithm (UCS), also known as Breadth-First Search (BFS), in this lab report, which is an algorithm for traversing or searching tree or graph data structures. Before moving on to the vertices at the next depth level, the algorithm visits all of the vertices at the current depth. The sample graph was used to demonstrate the algorithm's implementation. According to the results, the algorithm has a time complexity of O(V+E) and a space complexity of O. (V).

Genetic Algorithm

Introduction: The Genetic Algorithm (GA), a heuristic optimization technique inspired by the process of natural selection, will be discussed in this lab report. By simulating the process of evolution in a population of solutions, the GA is used to find the best solution to a problem.

Methodology: A sample optimization problem was chosen to demonstrate the GA's implementation, which is the classic problem of finding the maximum value of the function f(x) = x2, where x is a real number between -10 and 10.

To put the GA into action, the following steps were taken:

1. Initialization: Between [-10, 10], a population of solutions (chromosomes) was generated at random. Each chromosome represents a different x value.

2. Evaluation: For each chromosome, the fitness was determined by calculating the value of the function f(x).

3. Selection: For reproduction, the chromosomes with the highest fitness values were chosen.

Crossover: The selected chromosomes were paired and crossed to create new offspring chromosomes.

5. Mutation: To increase diversity, a small chance of mutation was introduced into the offspring chromosomes.

6. Assessment: The fitness of the offspring chromosomes was assessed.

7. Replacement: The offspring chromosomes were used to replace the population's weaker chromosomes.

Repeat steps 2-7 for a fixed number of generations or until the optimal solution is found.

The pseudocode for the implementation is as follows:

generate initial population

evaluate population

while (not optimal solution) {

select parents

perform crossover on parents

introduce mutation

evaluate offspring

replace weaker chromosomes with offspring

}

This same GA was productive in finding the optimal solution of x = 10 with a maximum value of f(x) = 100. The method found the solution after 50 generations. To fine-tune the algorithm for different problems, the number of generations and population size can be changed.

Conclusion: We discussed the Genetic Algorithm (GA) in this lab report, which is a heuristic optimization technique inspired by the process of natural selection. To find the best solution to a problem, the GA simulates the process of evolution in a population of solutions. The GA implementation was demonstrated using a sample optimization problem. The results show that the algorithm found the best solution in a reasonable number of generations.