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;

queue<int> queue;

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();

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

if (!visited[\*i]) {

visited[\*i] = true;

queue.push(\*i);

}

}

}

}

int main() {

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);

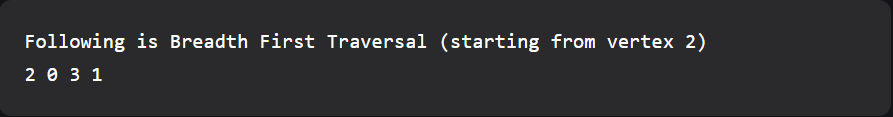
cout << "Following is Breadth First Traversal "

<< "(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.

// C++ program to print DFS traversal from

#include <bits/stdc++.h>

using namespace std;

class Graph {

public:

map<int, bool> visited;

map<int, list<int> > adj;

// function to add an edge to graph

void addEdge(int v, int w);

// DFS traversal of the vertices

// reachable from v

void DFS(int v);

};

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

{

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

}

void Graph::DFS(int v)

{

// Mark the current node as visited and

// print it

visited[v] = true;

cout << v << " ";

// Recur for all the vertices adjacent

// to this vertex

list<int>::iterator i;

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

if (!visited[\*i])

DFS(\*i);

}

// Driver's code

int main()

{

// Create a graph given in the above diagram

Graph g;

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 Depth First Traversal"

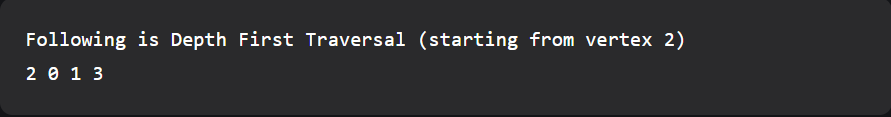
" (starting from vertex 2) \n";

// Function call

g.DFS(2);

return 0;

}



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.

// C++ implementation of above approach

#include <bits/stdc++.h>

using namespace std;

// graph

vector<vector<int> > graph;

// map to store cost of edges

map<pair<int, int>, int> cost;

// returns the minimum cost in a vector( if

// there are multiple goal states)

vector<int> uniform\_cost\_search(vector<int> goal, int start)

{

// minimum cost upto

// goal state from starting

// state

vector<int> answer;

// create a priority queue

priority\_queue<pair<int, int> > queue;

// set the answer vector to max value

for (int i = 0; i < goal.size(); i++)

answer.push\_back(INT\_MAX);

// insert the starting index

queue.push(make\_pair(0, start));

// map to store visited node

map<int, int> visited;

// count

int count = 0;

// while the queue is not empty

while (queue.size() > 0) {

// get the top element of the

// priority queue

pair<int, int> p = queue.top();

// pop the element

queue.pop();

// get the original value

p.first \*= -1;

// check if the element is part of

// the goal list

if (find(goal.begin(), goal.end(), p.second) != goal.end()) {

// get the position

int index = find(goal.begin(), goal.end(),

p.second) - goal.begin();

// if a new goal is reached

if (answer[index] == INT\_MAX)

count++;

// if the cost is less

if (answer[index] > p.first)

answer[index] = p.first;

// pop the element

queue.pop();

// if all goals are reached

if (count == goal.size())

return answer;

}

// check for the non visited nodes

// which are adjacent to present node

if (visited[p.second] == 0)

for (int i = 0; i < graph[p.second].size(); i++) {

// value is multiplied by -1 so that

// least priority is at the top

queue.push(make\_pair((p.first +

cost[make\_pair(p.second, graph[p.second][i])]) \* -1,

graph[p.second][i]));

}

// mark as visited

visited[p.second] = 1;

}

return answer;

}

// main function

int main()

{

// create the graph

graph.resize(7);

// add edge

graph[0].push\_back(1);

graph[0].push\_back(3);

graph[3].push\_back(1);

graph[3].push\_back(6);

graph[3].push\_back(4);

graph[1].push\_back(6);

graph[4].push\_back(2);

graph[4].push\_back(5);

graph[2].push\_back(1);

graph[5].push\_back(2);

graph[5].push\_back(6);

graph[6].push\_back(4);

// add the cost

cost[make\_pair(0, 1)] = 2;

cost[make\_pair(0, 3)] = 5;

cost[make\_pair(1, 6)] = 1;

cost[make\_pair(3, 1)] = 5;

cost[make\_pair(3, 6)] = 6;

cost[make\_pair(3, 4)] = 2;

cost[make\_pair(2, 1)] = 4;

cost[make\_pair(4, 2)] = 4;

cost[make\_pair(4, 5)] = 3;

cost[make\_pair(5, 2)] = 6;

cost[make\_pair(5, 6)] = 3;

cost[make\_pair(6, 4)] = 7;

// goal state

vector<int> goal;

// set the goal

// there can be multiple goal states

goal.push\_back(6);

// get the answer

vector<int> answer = uniform\_cost\_search(goal, 0);

// print the answer

cout << "Minimum cost from 0 to 6 is = "

<< answer[0] << endl;

return 0;

}



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

}

// C++ program to create target string, starting from

// random string using Genetic Algorithm

#include <bits/stdc++.h>

using namespace std;

// Number of individuals in each generation

#define POPULATION\_SIZE 100

// Valid Genes

const string GENES = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP"\

"QRSTUVWXYZ 1234567890, .-;:\_!\"#%&/()=?@${[]}";

// Target string to be generated

const string TARGET = "Sovon Mitro";

// Function to generate random numbers in given range

int random\_num(int start, int end)

{

int range = (end-start)+1;

int random\_int = start+(rand()%range);

return random\_int;

}

// Create random genes for mutation

char mutated\_genes()

{

int len = GENES.size();

int r = random\_num(0, len-1);

return GENES[r];

}

// create chromosome or string of genes

string create\_gnome()

{

int len = TARGET.size();

string gnome = "";

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

gnome += mutated\_genes();

return gnome;

}

// Class representing individual in population

class Individual

{

public:

string chromosome;

int fitness;

Individual(string chromosome);

Individual mate(Individual parent2);

int cal\_fitness();

};

Individual::Individual(string chromosome)

{

this->chromosome = chromosome;

fitness = cal\_fitness();

};

// Perform mating and produce new offspring

Individual Individual::mate(Individual par2)

{

// chromosome for offspring

string child\_chromosome = "";

int len = chromosome.size();

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

{

// random probability

float p = random\_num(0, 100)/100;

// if prob is less than 0.45, insert gene

// from parent 1

if(p < 0.45)

child\_chromosome += chromosome[i];

// if prob is between 0.45 and 0.90, insert

// gene from parent 2

else if(p < 0.90)

child\_chromosome += par2.chromosome[i];

// otherwise insert random gene(mutate),

// for maintaining diversity

else

child\_chromosome += mutated\_genes();

}

// create new Individual(offspring) using

// generated chromosome for offspring

return Individual(child\_chromosome);

};

// Calculate fitness score, it is the number of

// characters in string which differ from target

// string.

int Individual::cal\_fitness()

{

int len = TARGET.size();

int fitness = 0;

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

{

if(chromosome[i] != TARGET[i])

fitness++;

}

return fitness;

};

// Overloading < operator

bool operator<(const Individual &ind1, const Individual &ind2)

{

return ind1.fitness < ind2.fitness;

}

// Driver code

int main()

{

srand((unsigned)(time(0)));

// current generation

int generation = 0;

vector<Individual> population;

bool found = false;

// create initial population

for(int i = 0;i<POPULATION\_SIZE;i++)

{

string gnome = create\_gnome();

population.push\_back(Individual(gnome));

}

while(! found)

{

// sort the population in increasing order of fitness score

sort(population.begin(), population.end());

// if the individual having lowest fitness score ie.

// 0 then we know that we have reached to the target

// and break the loop

if(population[0].fitness <= 0)

{

found = true;

break;

}

// Otherwise generate new offsprings for new generation

vector<Individual> new\_generation;

// Perform Elitism, that mean 10% of fittest population

// goes to the next generation

int s = (10\*POPULATION\_SIZE)/100;

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

new\_generation.push\_back(population[i]);

// From 50% of fittest population, Individuals

// will mate to produce offspring

s = (90\*POPULATION\_SIZE)/100;

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

{

int len = population.size();

int r = random\_num(0, 50);

Individual parent1 = population[r];

r = random\_num(0, 50);

Individual parent2 = population[r];

Individual offspring = parent1.mate(parent2);

new\_generation.push\_back(offspring);

}

population = new\_generation;

cout<< "Generation: " << generation << "\t";

cout<< "String: "<< population[0].chromosome <<"\t";

cout<< "Fitness: "<< population[0].fitness << "\n";

generation++;

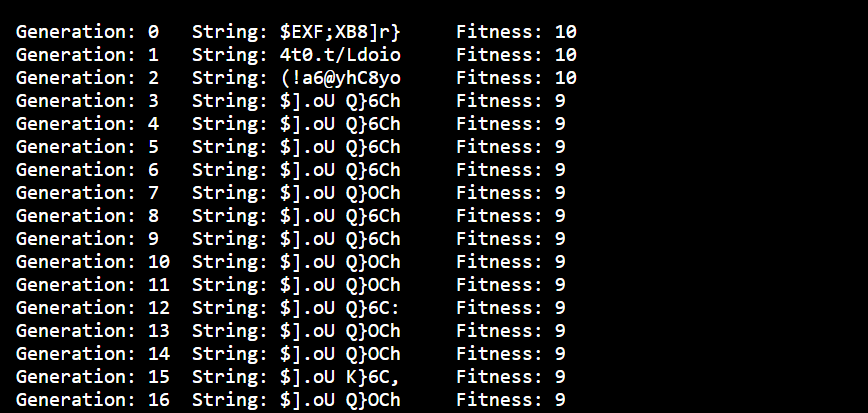
}

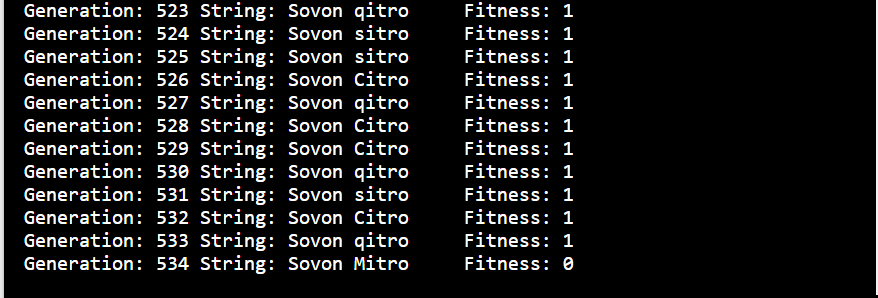
cout<< "Generation: " << generation << "\t";

cout<< "String: "<< population[0].chromosome <<"\t";

cout<< "Fitness: "<< population[0].fitness << "\n";

}





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.