**Discrete Math Problem Solution in C++**

1. Find Relations R1 and R2 on a Set

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> A = {1, 2, 3, 4};

vector<pair<int, int>> R1, R2;

for (int a : A) {

for (int b : A) {

if (b % a == 0)

R1.emplace\_back(a, b);

if (a <= b)

R2.emplace\_back(a, b);

}

}

cout << "Relation R1 (a divides b): ";

for (auto p : R1) cout << "(" << p.first << "," << p.second << ") ";

cout << "\nRelation R2 (a <= b): ";

for (auto p : R2) cout << "(" << p.first << "," << p.second << ") ";

}

1. Find Relation and Matrix Representation

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> A = {1, 2, 3};

vector<int> B = {1, 2};

vector<pair<int, int>> R;

int matrix[3][2] = {0};

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

for (int j = 0; j < B.size(); j++) {

if (A[i] > B[j]) {

R.emplace\_back(A[i], B[j]);

matrix[i][j] = 1;

}

}

}

cout << "Relation R (a > b): ";

for (auto p : R) cout << "(" << p.first << "," << p.second << ") ";

cout << "\nMatrix Representation:\n";

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

for (int j = 0; j < B.size(); j++)

cout << matrix[i][j] << " ";

cout << endl;

}

}

1. Graph Coloring (Welsh-Powell Algorithm)

#include <iostream>

#include <vector>

#include <algorithm>

using namespace std;

class Graph {

int V;

vector<vector<int>> graph;

public:

Graph(int v) : V(v), graph(v, vector<int>(v, 0)) {}

void addEdge(int u, int v) {

graph[u][v] = 1;

graph[v][u] = 1;

}

vector<int> welshPowellColoring() {

vector<int> degree(V), color(V, -1);

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

degree[i] = count(graph[i].begin(), graph[i].end(), 1);

vector<int> order(V);

iota(order.begin(), order.end(), 0);

sort(order.begin(), order.end(), [&](int a, int b) {

return degree[a] > degree[b];

});

int currentColor = 0;

for (int u : order) {

if (color[u] == -1) {

color[u] = currentColor;

for (int v : order) {

if (color[v] == -1) {

bool canColor = true;

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

if (graph[v][k] && color[k] == currentColor)

canColor = false;

if (canColor) color[v] = currentColor;

}

}

currentColor++;

}

}

return color;

}

};

int main() {

Graph g(5);

g.addEdge(0, 1); g.addEdge(0, 2);

g.addEdge(1, 3); g.addEdge(1, 4);

g.addEdge(2, 3); g.addEdge(3, 4);

vector<int> coloring = g.welshPowellColoring();

cout << "Vertex Colors: ";

for (int c : coloring) cout << c << " ";

}

1. Floyd-Warshall Algorithm

#include <iostream>

#include <vector>

#include <limits>

using namespace std;

const int INF = numeric\_limits<int>::max();

void floydWarshall(vector<vector<int>>& graph) {

int V = graph.size();

vector<vector<int>> dist = graph;

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

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

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

if (dist[i][k] != INF && dist[k][j] != INF)

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);

for (const auto& row : dist) {

for (int val : row)

cout << (val == INF ? "INF" : to\_string(val)) << " ";

cout << endl;

}

}

int main() {

vector<vector<int>> graph = {

{0, 3, INF, INF},

{INF, 0, 2, INF},

{INF, INF, 0, 1},

{8, INF, INF, 0}

};

floydWarshall(graph);

}

1. Matrix Union and Intersection

#include <iostream>

#include <vector>

using namespace std;

vector<vector<int>> matrix\_union(const vector<vector<int>>& A, const vector<vector<int>>& B) {

int rows = A.size(), cols = A[0].size();

vector<vector<int>> result(rows, vector<int>(cols));

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

for (int j = 0; j < cols; ++j)

result[i][j] = A[i][j] | B[i][j];

return result;

}

vector<vector<int>> matrix\_intersection(const vector<vector<int>>& A, const vector<vector<int>>& B) {

int rows = A.size(), cols = A[0].size();

vector<vector<int>> result(rows, vector<int>(cols));

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

for (int j = 0; j < cols; ++j)

result[i][j] = A[i][j] & B[i][j];

return result;

}

void print\_matrix(const vector<vector<int>>& M) {

for (const auto& row : M) {

for (int val : row) cout << val << " ";

cout << endl;

}

}

int main() {

vector<vector<int>> MR1 = {{1,0,1},{1,0,0},{0,1,0}};

vector<vector<int>> MR2 = {{1,0,1},{0,1,1},{1,0,0}};

cout << "MR1 ∪ MR2:\n";

print\_matrix(matrix\_union(MR1, MR2));

cout << "\nMR1 ∩ MR2:\n";

print\_matrix(matrix\_intersection(MR1, MR2));

}

1. Newton Forward Interpolation

#include <iostream>

#include <vector>

using namespace std;

vector<vector<double>> forward\_difference(const vector<double>& y) {

int n = y.size();

vector<vector<double>> table(n, vector<double>(n));

for (int i = 0; i < n; i++) table[i][0] = y[i];

for (int j = 1; j < n; j++)

for (int i = 0; i < n - j; i++)

table[i][j] = table[i + 1][j - 1] - table[i][j - 1];

return table;

}

double newton\_forward(const vector<double>& x, const vector<double>& y, double value) {

double h = x[1] - x[0];

double u = (value - x[0]) / h;

vector<vector<double>> table = forward\_difference(y);

double result = y[0];

double u\_term = 1;

double fact = 1;

for (int i = 1; i < x.size(); i++) {

u\_term \*= (u - (i - 1));

fact \*= i;

result += (u\_term / fact) \* table[0][i];

}

return result;

}

int main() {

vector<double> x = {1911, 1921, 1931, 1941, 1951, 1961};

vector<double> y = {12, 15, 20, 27, 39, 52};

double year\_to\_predict = 1946;

cout << "Estimated population in " << year\_to\_predict << ": "

<< newton\_forward(x, y, year\_to\_predict) << endl;

}

1. Newton Backward Interpolation

#include <iostream>

#include <vector>

using namespace std;

vector<vector<double>> backward\_difference(const vector<double>& y) {

int n = y.size();

vector<vector<double>> table(n, vector<double>(n));

for (int i = 0; i < n; i++) table[i][0] = y[i];

for (int j = 1; j < n; j++)

for (int i = n - 1; i >= j; i--)

table[i][j] = table[i][j - 1] - table[i - 1][j - 1];

return table;

}

double newton\_backward(const vector<double>& x, const vector<double>& y, double value) {

int n = x.size();

double h = x[1] - x[0];

double u = (value - x[n - 1]) / h;

vector<vector<double>> table = backward\_difference(y);

double result = y[n - 1];

double u\_term = 1;

double fact = 1;

for (int i = 1; i < n; i++) {

u\_term \*= (u + (i - 1));

fact \*= i;

result += (u\_term / fact) \* table[n - 1][i];

}

return result;

}

int main() {

vector<double> x = {1, 2, 3, 4, 5, 6, 7, 8};

vector<double> y = {1, 8, 27, 64, 125, 216, 343, 512};

double x\_to\_predict = 7.5;

cout << "Estimated value at f(" << x\_to\_predict << "): "

<< newton\_backward(x, y, x\_to\_predict) << endl;

}

1. Newton Divided Difference

#include <iostream>

#include <vector>

using namespace std;

vector<vector<double>> divided\_difference\_table(const vector<double>& x, const vector<double>& y) {

int n = x.size();

vector<vector<double>> table(n, vector<double>(n));

for (int i = 0; i < n; i++) table[i][0] = y[i];

for (int j = 1; j < n; j++)

for (int i = 0; i < n - j; i++)

table[i][j] = (table[i+1][j-1] - table[i][j-1]) / (x[i+j] - x[i]);

return table;

}

double newton\_interpolation(const vector<double>& x, const vector<double>& y, double value) {

vector<vector<double>> table = divided\_difference\_table(x, y);

double result = table[0][0];

double term = 1;

for (int i = 1; i < x.size(); i++) {

term \*= (value - x[i - 1]);

result += term \* table[0][i];

}

return result;

}

int main() {

vector<double> x = {4, 5, 7, 10, 11, 13};

vector<double> y = {48, 100, 294, 900, 1210, 2028};

double value = 15;

cout << "Interpolated value at " << value << " is: "

<< newton\_interpolation(x, y, value) << endl;

}

1. Lagrange Interpolation

#include <iostream>

#include <vector>

using namespace std;

double lagrange\_interpolation(const vector<double>& x\_values, const vector<double>& y\_values, double x) {

double result = 0;

int n = x\_values.size();

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

double term = y\_values[i];

for (int j = 0; j < n; j++) {

if (j != i)

term \*= (x - x\_values[j]) / (x\_values[i] - x\_values[j]);

}

result += term;

}

return result;

}

int main() {

vector<double> x = {5, 6, 9, 11};

vector<double> y = {12, 13, 14, 16};

double value = 10;

cout << "Interpolated value at x = " << value << " is y = "

<< lagrange\_interpolation(x, y, value) << endl;

}

10. Bisection Method

#include <iostream>

#include <cmath>

using namespace std;

double f(double x) {

return x\*x - 4\*x - 10;

}

double bisection(double a, double b, double tol = 1e-6) {

if (f(a) \* f(b) >= 0) {

cout << "Invalid interval.\n";

return -1;

}

double c;

while ((b - a) / 2.0 > tol) {

c = (a + b) / 2.0;

if (f(c) == 0.0)

return c;

else if (f(a) \* f(c) < 0)

b = c;

else

a = c;

}

return (a + b) / 2.0;

}

int main() {

double root = bisection(-2, -1.5);

if (root != -1)

cout << "Root is: " << root << endl;

}

11. False Position Method

#include <iostream>

#include <cmath>

using namespace std;

double f(double x) {

return x\*x - x - 2;

}

double false\_position(double a, double b, double tol = 1e-6, int max\_iter = 100) {

if (f(a) \* f(b) >= 0) {

cout << "Invalid interval.\n";

return -1;

}

double c;

for (int i = 0; i < max\_iter; i++) {

c = (a \* f(b) - b \* f(a)) / (f(b) - f(a));

if (abs(f(c)) < tol)

return c;

if (f(c) \* f(a) < 0)

b = c;

else

a = c;

}

return c;

}

int main() {

double root = false\_position(1, 3);

if (root != -1)

cout << "Root found: " << root << endl;

}