Comprehensive Guide to Algorithms in C

Zak
GitHub: https://github.com/arrhenius975

January 4, 2025

Contents

Dedicated to my one and only lovely life-partner, best-friend and the love of my life @r0s3-V3lv3t (https://github.com/r0s3-V3lv3t)

1 Introduction

This document serves as a comprehensive guide to implementing classic algorithms in C. It covers graph algorithms and dynamic programming techniques, providing ideas, mathematical foundations, and C implementations.

2 Graph Algorithms

2.1 Bellman-Ford Algorithm

2.1.1 Idea

The Bellman-Ford algorithm finds the shortest path from a single source to all vertices in a graph, even with negative edge weights. It relaxes all edges $|V| - 1$ times, where V is the number of vertices.

2.1.2 Problem Definition

Given a weighted graph $G = (V, E)$, where V is the set of vertices and E is the set of edges, find the shortest path from a source vertex to all other vertices.

2.1.3 Implementation in C

Listing 1: Bellman-Ford Algorithm in C

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <limits.h>

#define INF INT_MAX

// Structure to represent an edge in the graph
typedef struct Edge {
    int src, dest, weight;
} Edge;

// Bellman-Ford Algorithm
void bellmanFord(int vertices, int edges, Edge edgeList[], int
   source) {
    int distance[vertices];

    // Step 1: Initialize distances from the source to all
       vertices as infinite
    for (int i = 0; i < vertices; i++) {
        distance[i] = INF;
    }
    distance[source] = 0;

    // Step 2: Relax all edges |V| - 1 times
    for (int i = 1; i <= vertices - 1; i++) {
```

```c
        for (int j = 0; j < edges; j++) {
            int u = edgeList[j].src;
            int v = edgeList[j].dest;
            int weight = edgeList[j].weight;
            if (distance[u] != INF && distance[u] + weight < distance[v]) {
                distance[v] = distance[u] + weight;
            }
        }
    }

    // Step 3: Check for negative-weight cycles
    for (int j = 0; j < edges; j++) {
        int u = edgeList[j].src;
        int v = edgeList[j].dest;
        int weight = edgeList[j].weight;
        if (distance[u] != INF && distance[u] + weight < distance[v]) {
            printf("Graph contains a negative weight cycle\n");
            return;
        }
    }

    // Print the calculated shortest distances
    printf("Vertex Distance from Source %d:\n", source);
    for (int i = 0; i < vertices; i++) {
        if (distance[i] == INF) {
            printf("Vertex %d: INF\n", i);
        } else {
            printf("Vertex %d: %d\n", i, distance[i]);
        }
    }
}

int main() {
    int vertices = 5;
    int edges = 8;

    Edge edgeList[] = {
        {0, 1, -1}, {0, 2, 4}, {1, 2, 3},
        {1, 3, 2}, {1, 4, 2}, {3, 2, 5},
        {3, 1, 1}, {4, 3, -3}
    };

    int source = 0;
    bellmanFord(vertices, edges, edgeList, source);

    return 0;
}
```

2.2 Dijkstra's Algorithm

2.2.1 Idea

Dijkstra's algorithm finds the shortest path from a single source to all vertices in a graph with non-negative weights. It uses a priority queue for efficient selection of the next vertex with the shortest distance.

2.2.2 Problem Definition

Given a weighted graph $G = (V, E)$, find the shortest path from a source vertex to all other vertices.

2.2.3 Implementation in C

Listing 2: Dijkstra's Algorithm in C

```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define INF INT_MAX
#define MAX_VERTICES 100

// Find the vertex with the minimum distance value
int minDistance(int dist[], bool visited[], int vertices) {
    int min = INF, minIndex;

    for (int v = 0; v < vertices; v++) {
        if (!visited[v] && dist[v] <= min) {
            min = dist[v];
            minIndex = v;
        }
    }

    return minIndex;
}

// Dijkstra's Algorithm
void dijkstra(int graph[MAX_VERTICES][MAX_VERTICES], int
   vertices, int source) {
    int dist[vertices];        // Distance array
    bool visited[vertices];    // Visited set

    // Initialize all distances as INF and visited[] as false
    for (int i = 0; i < vertices; i++) {
        dist[i] = INF;
        visited[i] = false;
    }
    dist[source] = 0; // Distance of source vertex is 0

    // Find shortest path for all vertices
```

```c
    for (int count = 0; count < vertices - 1; count++) {
        // Pick the minimum distance vertex
        int u = minDistance(dist, visited, vertices);
        visited[u] = true;

        // Update dist[] of adjacent vertices
        for (int v = 0; v < vertices; v++) {
            if (!visited[v] && graph[u][v] && dist[u] != INF &&
                dist[u] + graph[u][v] < dist[v]) {
                dist[v] = dist[u] + graph[u][v];
            }
        }
    }

    // Print the calculated shortest distances
    printf("Vertex Distance from Source %d:\n", source);
    for (int i = 0; i < vertices; i++) {
        if (dist[i] == INF) {
            printf("Vertex %d: INF\n", i);
        } else {
            printf("Vertex %d: %d\n", i, dist[i]);
        }
    }
}

int main() {
    int vertices = 9;
    int graph[MAX_VERTICES][MAX_VERTICES] = {
        {0, 4, 0, 0, 0, 0, 0, 8, 0},
        {4, 0, 8, 0, 0, 0, 0, 11, 0},
        {0, 8, 0, 7, 0, 4, 0, 0, 2},
        {0, 0, 7, 0, 9, 14, 0, 0, 0},
        {0, 0, 0, 9, 0, 10, 0, 0, 0},
        {0, 0, 4, 14, 10, 0, 2, 0, 0},
        {0, 0, 0, 0, 0, 2, 0, 1, 6},
        {8, 11, 0, 0, 0, 0, 1, 0, 7},
        {0, 0, 2, 0, 0, 0, 6, 7, 0}
    };

    int source = 0;
    dijkstra(graph, vertices, source);

    return 0;
}
```

2.3 Prim's Algorithm

2.3.1 Idea

Prim's algorithm constructs a Minimum Spanning Tree (MST) for a connected graph by starting from an arbitrary vertex and growing the MST one edge at a time.

2.3.2 Problem Definition

Given a weighted graph $G = (V, E)$, construct an MST.

2.3.3 Implementation in C

Listing 3: Prim's Algorithm in C

```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

#define INF INT_MAX
#define MAX_VERTICES 100

// Find the vertex with the minimum key value
int minKey(int key[], bool mstSet[], int vertices) {
    int min = INF, minIndex;

    for (int v = 0; v < vertices; v++) {
        if (!mstSet[v] && key[v] < min) {
            min = key[v];
            minIndex = v;
        }
    }

    return minIndex;
}

// Print the constructed MST
void printMST(int parent[], int
   graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    printf("Edge\tWeight\n");
    for (int i = 1; i < vertices; i++) {
        printf("%d - %d\t%d\n", parent[i], i,
           graph[i][parent[i]]);
    }
}

// Prim's Algorithm
void prims(int graph[MAX_VERTICES][MAX_VERTICES], int vertices) {
    int parent[vertices];       // Array to store the MST
    int key[vertices];          // Key values used to pick
       minimum weight edges
    bool mstSet[vertices];      // Set of vertices included in
       the MST

    // Initialize all keys as infinite and mstSet[] as false
    for (int i = 0; i < vertices; i++) {
        key[i] = INF;
        mstSet[i] = false;
    }
```

```c

    key[0] = 0;        // Include the first vertex in the MST
    parent[0] = -1;    // First node is the root

    // Construct the MST
    for (int count = 0; count < vertices - 1; count++) {
        int u = minKey(key, mstSet, vertices);
        mstSet[u] = true;

        for (int v = 0; v < vertices; v++) {
            if (graph[u][v] && !mstSet[v] && graph[u][v] <
               key[v]) {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }

    printMST(parent, graph, vertices);
}

int main() {
    int vertices = 5;
    int graph[MAX_VERTICES][MAX_VERTICES] = {
        {0, 2, 0, 6, 0},
        {2, 0, 3, 8, 5},
        {0, 3, 0, 0, 7},
        {6, 8, 0, 0, 9},
        {0, 5, 7, 9, 0}
    };

    prims(graph, vertices);

    return 0;
}
```

2.4 Kruskal's Algorithm

2.4.1 Idea

Kruskal's algorithm constructs an MST by sorting all edges by weight and adding them to the MST if they do not form a cycle.

2.4.2 Problem Definition

Given a weighted graph $G = (V, E)$, construct an MST.

2.4.3 Implementation in C

Listing 4: Kruskal's Algorithm in C

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Edge {
    int src, dest, weight;
} Edge;

typedef struct Graph {
    int V, E;
    Edge* edges;
} Graph;

typedef struct Subset {
    int parent, rank;
} Subset;

// Create a graph
Graph* createGraph(int V, int E) {
    Graph* graph = (Graph*)malloc(sizeof(Graph));
    graph->V = V;
    graph->E = E;
    graph->edges = (Edge*)malloc(E * sizeof(Edge));
    return graph;
}

// Find set of an element
int find(Subset subsets[], int i) {
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

// Union of two sets
void Union(Subset subsets[], int x, int y) {
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    if (subsets[xroot].rank < subsets[yroot].rank) {
        subsets[xroot].parent = yroot;
    } else if (subsets[xroot].rank > subsets[yroot].rank) {
        subsets[yroot].parent = xroot;
    } else {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges by weight
int compare(const void* a, const void* b) {
    Edge* a1 = (Edge*)a;
    Edge* b1 = (Edge*)b;
```

```c
    return a1->weight > b1->weight;
}

// Kruskal's Algorithm
void kruskal(Graph* graph) {
    int V = graph->V;
    Edge result[V];
    int e = 0;
    int i = 0;

    qsort(graph->edges, graph->E, sizeof(graph->edges[0]),
       compare);

    Subset* subsets = (Subset*)malloc(V * sizeof(Subset));
    for (int v = 0; v < V; v++) {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    while (e < V - 1 && i < graph->E) {
        Edge nextEdge = graph->edges[i++];
        int x = find(subsets, nextEdge.src);
        int y = find(subsets, nextEdge.dest);

        if (x != y) {
            result[e++] = nextEdge;
            Union(subsets, x, y);
        }
    }

    printf("Edge\tWeight\n");
    for (int j = 0; j < e; j++) {
        printf("%d - %d\t%d\n", result[j].src, result[j].dest,
           result[j].weight);
    }

    free(subsets);
}

int main() {
    int V = 4, E = 5;
    Graph* graph = createGraph(V, E);

    graph->edges[0] = (Edge){0, 1, 10};
    graph->edges[1] = (Edge){0, 2, 6};
    graph->edges[2] = (Edge){0, 3, 5};
    graph->edges[3] = (Edge){1, 3, 15};
    graph->edges[4] = (Edge){2, 3, 4};

    kruskal(graph);

```

```c
    free(graph->edges);
    free(graph);
    return 0;
}
```

3 Dynamic Programming Algorithms

3.1 Huffman Coding

3.1.1 Idea

Huffman Coding is used for lossless data compression. It builds a binary tree based on the frequencies of characters, ensuring minimal encoding cost.

3.1.2 Problem Definition

Given a set of characters and their frequencies, construct a binary tree for optimal encoding.

3.1.3 Implementation in C

Listing 5: Huffman Coding in C

```c
#include <stdio.h>
#include <stdlib.h>

#define MAX_SIZE 100

typedef struct Node {
    char data;
    unsigned freq;
    struct Node *left, *right;
} Node;

// Min-Heap structure
typedef struct MinHeap {
    unsigned size;
    unsigned capacity;
    Node** array;
} MinHeap;

Node* createNode(char data, unsigned freq) {
    Node* temp = (Node*)malloc(sizeof(Node));
    temp->data = data;
    temp->freq = freq;
    temp->left = temp->right = NULL;
    return temp;
}

MinHeap* createMinHeap(unsigned capacity) {
```

```c
    MinHeap* minHeap = (MinHeap*)malloc(sizeof(MinHeap));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array = (Node**)malloc(minHeap->capacity *
        sizeof(Node*));
    return minHeap;
}

void swapNodes(Node** a, Node** b) {
    Node* t = *a;
    *a = *b;
    *b = t;
}

void minHeapify(MinHeap* minHeap, int idx) {
    int smallest = idx;
    int left = 2 * idx + 1;
    int right = 2 * idx + 2;

    if (left < minHeap->size && minHeap->array[left]->freq <
        minHeap->array[smallest]->freq)
        smallest = left;

    if (right < minHeap->size && minHeap->array[right]->freq <
        minHeap->array[smallest]->freq)
        smallest = right;

    if (smallest != idx) {
        swapNodes(&minHeap->array[smallest],
            &minHeap->array[idx]);
        minHeapify(minHeap, smallest);
    }
}

Node* extractMin(MinHeap* minHeap) {
    Node* temp = minHeap->array[0];
    minHeap->array[0] = minHeap->array[minHeap->size - 1];
    --minHeap->size;
    minHeapify(minHeap, 0);
    return temp;
}

void insertMinHeap(MinHeap* minHeap, Node* node) {
    ++minHeap->size;
    int i = minHeap->size - 1;

    while (i && node->freq < minHeap->array[(i - 1) / 2]->freq) {
        minHeap->array[i] = minHeap->array[(i - 1) / 2];
        i = (i - 1) / 2;
    }

```

```
    minHeap->array[i] = node;
}

void printCodes(Node* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top + 1);
    }

    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top + 1);
    }

    if (!(root->left) && !(root->right)) {
        printf("%c: ", root->data);
        for (int i = 0; i < top; ++i)
            printf("%d", arr[i]);
        printf("\n");
    }
}

void buildHuffmanTree(char data[], int freq[], int size) {
    Node *left, *right, *top;
    MinHeap* minHeap = createMinHeap(size);

    for (int i = 0; i < size; ++i)
        minHeap->array[i] = createNode(data[i], freq[i]);
    minHeap->size = size;

    while (minHeap->size != 1) {
        left = extractMin(minHeap);
        right = extractMin(minHeap);

        top = createNode('$', left->freq + right->freq);
        top->left = left;
        top->right = right;

        insertMinHeap(minHeap, top);
    }

    int arr[MAX_SIZE], topIndex = 0;
    printCodes(minHeap->array[0], arr, topIndex);
}

int main() {
    char arr[] = {'a', 'b', 'c', 'd', 'e', 'f'};
    int freq[] = {5, 9, 12, 13, 16, 45};
    int size = sizeof(arr) / sizeof(arr[0]);

    buildHuffmanTree(arr, freq, size);
```

```c

    return 0;
}
```

3.2 Longest Common Subsequence (LCS)

3.2.1 Idea

LCS finds the longest subsequence common to two sequences. It uses dynamic programming to solve the problem efficiently.

3.2.2 Problem Definition

Given two sequences X and Y, find the longest subsequence present in both.

3.2.3 Implementation in C

Listing 6: Longest Common Subsequence in C

```c
#include <stdio.h>
#include <string.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

void LCS(char* X, char* Y, int m, int n) {
    int L[m + 1][n + 1];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++) {
            if (i == 0 || j == 0)
                L[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                L[i][j] = 1 + L[i - 1][j - 1];
            else
                L[i][j] = max(L[i - 1][j], L[i][j - 1]);
        }
    }

    int index = L[m][n];
    char lcs[index + 1];
    lcs[index] = '\0';

    int i = m, j = n;
    while (i > 0 && j > 0) {
        if (X[i - 1] == Y[j - 1]) {
            lcs[index - 1] = X[i - 1];
            i--;
            j--;
            index--;
```

```c
        } else if (L[i - 1][j] > L[i][j - 1])
            i--;
        else
            j--;
    }

    printf("LCS of %s and %s is %s\n", X, Y, lcs);
}

int main() {
    char X[] = "AGGTAB";
    char Y[] = "GXTXAYB";

    int m = strlen(X);
    int n = strlen(Y);

    LCS(X, Y, m, n);

    return 0;
}
```

3.3 Matrix Chain Multiplication

3.3.1 Idea

Matrix Chain Multiplication minimizes the number of scalar multiplications required to multiply a chain of matrices.

3.3.2 Problem Definition

Given an array of dimensions $p[]$, find the minimum number of multiplications required to multiply the matrices.

3.3.3 Implementation in C

Listing 7: Matrix Chain Multiplication in C

```c
#include <stdio.h>
#include <limits.h>

void matrixChainOrder(int p[], int n) {
    int m[n][n];

    for (int i = 1; i < n; i++)
        m[i][i] = 0;

    for (int len = 2; len < n; len++) {
        for (int i = 1; i < n - len + 1; i++) {
            int j = i + len - 1;
            m[i][j] = INT_MAX;

```

```c
            for (int k = i; k <= j - 1; k++) {
                int cost = m[i][k] + m[k + 1][j] + p[i - 1] *
                   p[k] * p[j];
                if (cost < m[i][j])
                    m[i][j] = cost;
            }
        }
    }

    printf("Minimum number of multiplications is %d\n", m[1][n -
       1]);
}

int main() {
    int arr[] = {10, 30, 5, 60};
    int n = sizeof(arr) / sizeof(arr[0]);

    matrixChainOrder(arr, n);

    return 0;
}
```

3.4 Activity Selection Problem

3.4.1 Idea

The Activity Selection Problem selects the maximum number of non-overlapping activities using a greedy approach.

3.4.2 Problem Definition

Given *start*[] and *finish*[], select the maximum subset of activities such that no two overlap.

3.4.3 Implementation in C

Listing 8: Activity Selection Problem in C

```c
#include <stdio.h>

void activitySelection(int start[], int finish[], int n) {
    printf("Selected activities are:\n");

    int i = 0;
    printf("Activity %d (Start: %d, Finish: %d)\n", i + 1,
       start[i], finish[i]);

    for (int j = 1; j < n; j++) {
        if (start[j] >= finish[i]) {
            printf("Activity %d (Start: %d, Finish: %d)\n", j +
               1, start[j], finish[j]);
```

```
            i = j;
        }
    }
}

int main() {
    int start[] = {1, 3, 0, 5, 8, 5};
    int finish[] = {2, 4, 6, 7, 9, 9};
    int n = sizeof(start) / sizeof(start[0]);

    activitySelection(start, finish, n);

    return 0;
}
```

4 Conclusion

This document provides a structured guide to understanding and implementing various algorithms using C. The approaches are categorized into graph algorithms and dynamic programming, covering essential techniques to solve complex computational problems.

5 References

- **Introduction to Algorithms**, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Chapters: Graph Algorithms, Dynamic Programming.
- **Algorithms**, Robert Sedgewick and Kevin Wayne. Chapters: Graph Processing, Greedy Algorithms.