Fundamentals of Data Structures

Projects 3: Dijkstra Sequence

Author:

Date: 2025/03/10

2024-2025 Spring & Summer Semester

Table of Contents

Chapter 1: Introduction

1.1) Problem Description

Dijkstra's algorithm is one of the very famous greedy algorithms.

It is used for solving the single source shortest path problem which gives the shortest paths from one particular source vertices to all the other vertices of the given graph. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later.

In this algorithm, a set contains vertices included in shortest path tree is maintained. During each step, we find one vertices which is not yet included and has a minimum distance from the source, and collect it into the set. Hence step by step an ordered sequence of vertices, let's call it Dijkstra sequence, is generated by Dijkstra's algorithm.

1.2) Background

Dijkstra's algorithm maintains a set of vertices whose shortest distance from the source is already determined. In each step, it:

- Selects the vertices with minimum distance from the unprocessed set
- Updates distances of adjacent vertices if a shorter path is found through the selected vertices
- Adds the selected vertices to the processed set

This creates what we call a "Dijkstra sequence" - the order in which vertices are selected. For a given graph, multiple valid Dijkstra sequences may exist depending on how ties are broken when selecting minimum-distance vertices.

In this project, we analyze whether a given sequence of vertices could be a valid Dijkstra sequence for a specific graph.

Chapter 2: Algorithm Specification

2.1) Structure of Graph

The graph is represented as an adjacency list, where each vertices has a list of its adjacent vertices and their respective weights. The graph is undirected, meaning that edges have no direction.

```
Struct Graph
├── Number of vertices
├── Number of edges
└── Array of verticles
    ├── Verticle 1
    │   ├── Verticle index
    │   └── Linked list of edges
    │       ├── The connected verticle
    │       ├── Edge weight
    │       └── Next verticle
```

```
    ├── Verticle 2
    ...
```

2.2) Dijkstra Sequence Verification Algorithm

To verify if a given sequence is a valid Dijkstra sequence, the algorithm simulates Dijkstra's shortest path algorithm and checks if the vertices could be selected in the given sequence.

In Dijkstra algorithm, we simply maintain a set of vertices whose shortest distance from the source is already determined. In each step, we select the vertices with minimum distance from the unprocessed set, update distances of adjacent vertices if a shorter path is found through the selected vertices, and add the selected vertices to the processed set.

During each step, we select the vertices with minimum distance from the unprocessed set. Thus our mission is to verify whether each vertices in the given sequence is the closest vertice.

```
Algorithm isDijkstraSequence(graph, source, sequence):
    distances = {node: infinity for node in graph}
    distances[source] = 0

    priority_queue = new PriorityQueue()
    priority_queue.pushBack(source)

    for item in sequence:

        item_index = priority_queue.is_min(item)
        if(item_index != -1) return false

        current_dist, current_node = priority_queue.popFront(item_index)
        if current_dist > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node]:
            new_dist = current_dist + weight

            if new_dist < distances[neighbor]:
                distances[neighbor] = new_dist
                priority_queue.pushBack(neighbor, new_dist)
```

If each vertice in the given sequence pass the test, we can verify that the given sequence is a valid Dijkstra sequence.

2.3) Priority Queue Implementation

To efficiently implement Dijkstra's algorithm, we can use a priority queue to select the vertices with the minimum distance. The queue is implemented as a binary min-heap based on distance values.

- Each element in the queue contains:
  - A distance value (dis)
  - A vertices index (v)
- The priority queue supports operations like:
  - pushBack: Adds a new element while maintaining heap property
  - popFront: Removes and returns the minimum element
  - isHeapTop: Checks if a given vertices is at the top of the heap. Notice that there could be multiple vertices with the same distance, so we need to check if the given vertices is the minimum among them.

2.4) Time and Space Complexity

- Time Complexity:
  - Each vertices is processed once: $O(V)$
  - Each edge is processed once: $O(E)$
  - For each vertices, we may need to insert into a priority queue: $O(\log V)$
  - For each step, we need to check if the given vertices is the minimum among them: avereage $O(1)$, worst $O(V)$, pop index of the priority queue: $O(\log V)$
  - Total time complexity: average $O(V \log V + E)$, worst $O(V^2 + E)$
- Space Complexity:
  - $O(V)$ for the distance array, visited array, and priority queue
  - $O(V + E)$ for the graph

This implementation balances efficiency with code simplicity, focusing on correctness for the verification task rather than optimizing the traditional Dijkstra algorithm implementation.

Chapter 3: Testing Results

3.1) Functional Test

1. For the case given in the problem.

Input	**Output**
5 7 1 2 2 1 5 1 2 3 1 2 4 1 2 5 2 3 5 1 3 4 1 4 5 1 3 4 2 5 3 1 2 4	YES YES YES NO

2 3 4 5 1 3 2 1 5 4	

2. Customly constructed dataset

I generated a random connected map with given V and E, and then I generated several random sequence of vertices. Some of them are valid Dijkstra sequences, while others are not. I used the algorithm to verify the sequences.

For example:

Input	**Output**
5 8 1 3 4 1 5 7 1 4 4 1 2 2 2 4 10 2 3 7 3 4 1 3 5 5 7 1 2 3 4 5 1 2 4 3 5 1 2 4 5 3 1 5 2 4 3 1 5 4 3 2 1 2 5 3 4 1 4 5 2 3	YES YES NO NO NO NO NO

Input	**Output**
6 9 1 3 2 1 2 3 1 4 9 2 6 8 2 5 6 3 5 3 4 6 5 4 5 5 5 6 1 6 1 3 2 5 6 4 1 5 4 2 6 3 1 4 2 3 5 6 1 5 6 2 4 3	YES NO NO NO NO NO

1 6 3 2 5 4 1 2 5 6 4 3	

The result generally match the type of sequence during build. The algorithm can correctly verify the Dijkstra sequence.

3. For minimum cases

Input	**Output**
2 1 1 2 1 2 1 2 2 1	YES YES

3.2) Performance Test

V	**E**	**T (Ticks)**
10	12	1.692308
20	25	3.041510
50	55	6.266667
100	110	9.133333
200	215	19.911100
500	520	53.153333
1000	1050	133.306667
5000	5100	1559.021000
10000	10120	5118.998890

The table above shows the testing results of the algorithm with different sizes of graph. The first column is the number of vertices, the second column is the number of edges, and the third column is the time taken to execute the algorithm in ticks.

Chapter 4: Analysis and Comments

Using Matplotlib, we could plot the tick taken to execute the four versions of searching algorithms.

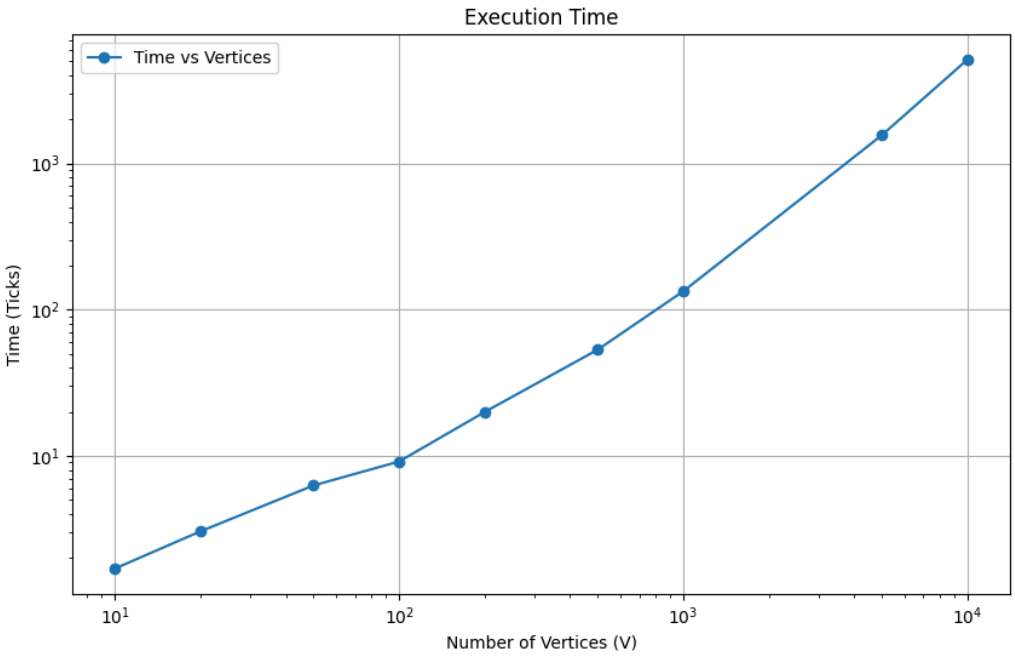

Figure 1: “Execution time”

The figure use log scale axis. From above we can see that the algorithm is above linear, but below quadratic. Thus we could verify the time complexity is $O(V \log V)$.

The algorithm is efficient for the given problem.

Appendix: Source Code (in C)

File CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.10)

set(CMAKE_C_COMPILER "gcc")

project(Dijkstra_Sequence)

include_directories(${CMAKE_CURRENT_SOURCE_DIR}/includes)

file(GLOB SOURCES ${CMAKE_CURRENT_SOURCE_DIR}/src/*.c)

add_executable(main ${SOURCES} main.c)
```

File main.c:

```
#include <stdio.h>
#include "includes/graph.h"
#include <stdlib.h>

int main(){
    int nv,ne;
```

```
    scanf("%d %d",&nv,&ne);
    // Create a graph with nv vertices
    GraphPtr g = createGraph(nv);
    for(int i = 0; i < ne; i++){
        int u,v,weight;
        scanf("%d %d %d",&u,&v,&weight);
        // Undirected graph: add edges in both directions
        addEdge(g,u,v,weight);
        addEdge(g,v,u,weight);
    }
    int cases;
    scanf("%d",&cases);
    int *seq = (int*)malloc(nv * sizeof(int));
    for (int i = 0; i < cases; i++){
        // Read the sequence of vertices
        for(int j = 0;j < nv;j++){
            scanf("%d",&seq[j]);
        }
        // Check if the sequence is a Dijkstra sequence
        bool res = isDijkstraSequence(g,seq);
        if (res == true)
            // If the sequence is a Dijkstra sequence, print "YES"
            puts("YES");
        else puts("NO");
    }
    // Free the allocated memory
    destroyGraph(g);
    return 0;
}
```

File includes/bool.h

```
#pragma once

// Define a boolean type for C
typedef enum Boolean { false, true } bool;
```

File includes/graph.h

```
#pragma once
#include "bool.h"

// GraphNode structure
typedef struct GraphNodes{
    int data;
    struct GraphEdges *firstEdge; // pointer to the first edge of the node
} GraphNodes,*GraphNodesPtr;

// GraphEdges structure
// Each edge connects two nodes and has a weight
typedef struct GraphEdges{
```

```
    int v;
    int weight;
    struct GraphEdges *next; // pointer to the next edge in the list
} GraphEdges,*GraphEdgesPtr;

// Graph structure
typedef struct Graph{
    int Nv,Ne; // Nv = number of vertices, Ne = number of edges
    GraphNodes * nodes; // array of nodes
} Graph,*GraphPtr;

GraphPtr createGraph(int n);   // create a graph with n vertices
void destroyGraph(GraphPtr G); // destroy the graph and free memory
void addEdge(GraphPtr G, int u, int v, int weight);  // add an edge to the graph
bool isDijkstraSequence(GraphPtr G, int *seq); // check if the sequence is a
Dijkstra sequence
```

File src/graph.c

```
#include "../includes/graph.h"
#include "../includes/priority_queue.h"
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>

GraphPtr createGraph(int n){
    // Create a graph with n vertices
    GraphPtr G = (GraphPtr)malloc(sizeof(Graph));
    G->Nv = n;
    G->Ne = 0;
    // Allocate memory for the nodes array
    G->nodes = (GraphNodesPtr)malloc(n * sizeof(GraphNodes));
    for(int i = 0; i < n; i++){
        // Initialize each node
        G->nodes[i].data = i+1;
        G->nodes[i].firstEdge = NULL;
    }
    return G;
}

void destroyGraph(GraphPtr G){
    // Destroy the graph and free memory
    for(int i = 0; i < G->Nv; i++){
        GraphEdgesPtr edge = G->nodes[i].firstEdge;
        // Free each edge in the list
        while(edge != NULL){
            GraphEdgesPtr temp = edge;
            edge = edge->next;
            free(temp);
```

```
        }
    }
    // Free the nodes array
    free(G->nodes);
    free(G);
}

void addEdge(GraphPtr G, int u, int v, int weight){
    // Check if the vertice index are valid
    if(u < 1 || u > G->Nv || v < 1 || v > G->Nv) return;
    GraphEdgesPtr newEdge = (GraphEdgesPtr)malloc(sizeof(GraphEdges));
    // Initialize the new edge
    newEdge->v = v-1;
    newEdge->weight = weight;
    // Chain the new edge to the existing edges of the node
    newEdge->next = G->nodes[u-1].firstEdge;
    G->nodes[u-1].firstEdge = newEdge;
    // Update the number of edges in the graph
    G->Ne++;
}

bool isDijkstraSequence(GraphPtr G, int *seq){
    int *visited = (int *)malloc(G->Nv * sizeof(int));
    int *dist = (int *)malloc(G->Nv * sizeof(int));
    // Initialize visited and dist arrays
    for (int i = 0; i < G->Nv; i++) {
        visited[i] = 0;
        dist[i] = INT32_MAX;
    }
    PriorityQueuePtr pq = createPriorityQueue(G->Nv);
    int cur = 0;
    // Push the first element of the sequence into the priority queue
    pushBack(pq, (Point){0, seq[cur] - 1});
    dist[cur] = 0;
    for(; cur < G->Nv; cur++) {
        // Clear visited elements from the priority queue
        while(!isEmpty(pq) && visited[pq->data[1].v]) popFront(pq, 0);
        // Check if the current element is the top of the priority queue
        int index = isHeapTop(pq, seq[cur] - 1);
        if( index == -1) {
            // If the current element is not the top of the priority queue, free
memory and return false
            free(visited);
            free(dist);
            return false;
        }
        Point u = popFront(pq, index);
        if(visited[u.v]) continue;
        visited[u.v] = 1;
        // Get the edges of the current node
        GraphEdgesPtr edge = G->nodes[u.v].firstEdge;
```

```
        while(edge != NULL){
            int v = edge->v;
            // Check if the edge is valid and not visited
            if(!visited[v] && dist[u.v] + edge->weight < dist[v]){
                dist[v] = dist[u.v] + edge->weight;
                // Push the new distance and verticle into the priority queue
                pushBack(pq, (Point){dist[v], v});
            }
            edge = edge->next;
        }
    }
    free(visited);
    free(dist);
    return true;
}
```

File includes/priority_queue.h

```
#pragma once
#include "bool.h"
#define TEMPLATE(T) typedef T ElementType; \
                    typedef T * ElementTypePtr;

// Point structure
typedef struct Point{
    int dis, v; // dis = distance, v = vertex
} Point;

TEMPLATE(Point)

typedef struct PriorityQueue{
    ElementTypePtr data; // Array to store the elements of the priority queue
    int size;  // Current number of elements in the priority queue
    int capacity; // Maximum capacity of the priority queue
}PriorityQueue, *PriorityQueuePtr;

PriorityQueuePtr createPriorityQueue(int capacity); // Create a priority queue
with a given capacity
void destroyPriorityQueue(PriorityQueuePtr pq); // Destroy the priority queue and
free memory
void resizePriorityQueue(PriorityQueuePtr pq, int newCapacity); // Resize the
priority queue to a new capacity
void pushBack(PriorityQueuePtr pq, ElementType value); // Add an element to the
priority queue
ElementType popFront(PriorityQueuePtr pq, int index); // Remove and return the
given index element of the priority queue
bool isEmpty(PriorityQueuePtr pq); // Check if the priority queue is empty
void printPriorityQueue(PriorityQueuePtr pq); // Print the elements of the
priority queue
bool Compare(ElementType a, ElementType b); // Compare two elements
```

```
int isHeapTop(PriorityQueuePtr pq, int data); // Check if the top element of the
priority queue is a heap
```

File src/priority_queue.c

```
#include "../includes/priority_queue.h"
#include <stdio.h>
#include <stdlib.h>

PriorityQueuePtr createPriorityQueue(int initialCapacity) {
    // Create a priority queue with a given initial capacity
    PriorityQueuePtr pq = (PriorityQueuePtr)malloc(sizeof(PriorityQueue));
    pq->data = (ElementType *)malloc((initialCapacity + 1) * sizeof(ElementType));
    pq->size = 0;
    pq->capacity = initialCapacity;
    return pq;
}

void destroyPriorityQueue(PriorityQueuePtr pq) {
    // Destroy the priority queue and free memory
    free(pq->data);
    free(pq);
}

void resizePriorityQueue(PriorityQueuePtr pq, int newCapacity) {
    // Resize the priority queue to a new capacity
    pq->data = (ElementType *)realloc(pq->data, (newCapacity + 1) *
sizeof(ElementType));
    pq->capacity = newCapacity;
}

bool isEmpty(PriorityQueuePtr pq) {
    // Check if the priority queue is empty
    return pq->size == 0;
}

bool Compare(ElementType a, ElementType b){
    // Compare two elements based on their distance
    return a.dis < b.dis;
}

void pushBack(PriorityQueuePtr pq, ElementType value) {
    if (pq->size == pq->capacity) {
        // Resize the priority queue if it is full
        resizePriorityQueue(pq, pq->capacity * 2);
    }
    int i = ++pq->size;
    // Insert the new element at the end of the queue
    while ( i > 1 && Compare(value,pq->data[i / 2])) {
        pq->data[i] = pq->data[i / 2];
        i /= 2;
```

```
    }
    pq->data[i] = value;
}

ElementType popFront(PriorityQueuePtr pq, int index) {
    int i = index + 1;
    ElementType value = pq->data[i];
    while( i <= pq->size / 2) {
        int child = i * 2;
        // Find the larger child
        if (child < pq->size && Compare(pq->data[child + 1], pq->data[child])) {
            child++;
        }
        // If the current element is larger than the larger child, break
        if (Compare(pq->data[pq->size], pq->data[child])) break;
        pq->data[i] = pq->data[child];
        i = child;
    }
    // Insert the last element at the position of the removed element
    pq->data[i] = pq->data[pq->size];
    pq->size--;
    return value;
}

void printPriorityQueue(PriorityQueuePtr pq) {
    // For Debugging
    for (int i = 1; i <= pq->size; i++) {
        printf("%d ", pq->data[i].v);
    }
    printf("\n");
}

int searchHeap(PriorityQueuePtr pq, int target,int index) {
    // Search for a target element in the heap
    if(index > pq->size || pq->data[index].dis > pq->data[1].dis) {
        return -1;
    }
    if(pq->data[index].v == target) return index - 1;
    // Recursively search in the left and right subtrees
    int left = searchHeap(pq, target, index * 2);
    if(left != -1) return left;
    int right = searchHeap(pq, target, index * 2 + 1);
    return right;
}

int isHeapTop(PriorityQueuePtr pq, int data){
    // Check if the top element of the priority queue is a heap
    return searchHeap(pq, data, 1);
}
```

Declaration

I hereby declare that all the work done in this project titled "Dijkstra Sequence" is of my independent effort.