Fundamentals of Data Structures

Project 3 : Dijkstra Sequence

Date : 2025/04/05

2024–2025 Spring & Summer Semester

Chapter 1 : Introduction

Problem Description

Dijkstra's algorithm is a well-known greedy algorithm used to solve the single-source shortest path problem. Given a weighted graph, we are asked to check whether a given sequence of vertices matches a valid Dijkstra sequence, which means the sequence represents the order in which vertices are added to the shortest path tree. The graph is defined by a set of vertices and edges, no more than 1×10^3 and 1×10^5 respectively, and we must evaluate multiple sequences of vertex visits to determine whether they follow the rules of Dijkstra's algorithm, and print *YES* or *NO* accordingly.

Algorithm Background

Dijkstra's algorithm uses a greedy approach to find the shortest paths. The algorithm works by iteratively selecting the vertex with the smallest tentative distance from the source that hasn't yet been included in the shortest path tree, and updates the tentative distances to neighboring vertices and continues until all vertices are included in the shortest path tree. By using different approaches to select the vertex in each step, like simply scanning the table or uses min heap to store the distance, we can obtain time complexity of $O(|V|^2 + |E|)$ and $O(|E|log|V|)$ respectively, in general.

Additionally, for a given graph, there can be multiple valid Dijkstra sequences depending on the order of vertex selection when several vertices all have the smallest tentative distance. Our task is to validate whether a given sequence is a valid Dijkstra sequence by examine whether it adheres to the correct shortest path ordering.

Chapter 2 : Algorithm Specification

Algorithm 1 : Check Mininimun Distance

Input : Number of vertices *nv*, vertex *chosen* waiting to be checked, array stroing distance *dist[]*

Output : If *chosen* does have minimun tentative distance, return true; else return false

Main idea : The check_min algorithm simply checks if *chosen* has minimum tentative distance by traversing the *dist* array

Pseudo code :

```
int check_min(chosen, Nv)
    for each vertex i
        if known[i] == 0 and dist[i] < dist[chosen]
            return true  // chosen vertex is not the minimum
    return false  // chosen vertex is the minimum
```

Algorithm 2 : Dijkstra Sequence Validation

Input : Number of vertices *nv*, adjacency matrix *graph[][]*, and FILE pointer *file*

Output : If the sequence is a valid Dijkstra sequence, return true; if any vertex in the sequence doesn't follow the expected order, return false;

Main idea : The algorithm simulates Dijkstra's algorithm step-by-step and checks if the given sequence matches the order in which the vertices would be selected by Dijkstra's algorithm.

Pseudo code :

```
int Dijkstra(int nv, FILE *file){
    // Initialization
    initialize dist[] to infinity
    initialize known[] to 0
    initialize path[] to 0
    int s = read sequence[i] from file
    dist[s] = 0;  // distance to source is 0

    for i=1 to nv
    {    int chosen; // variable to mark current vertex which needs to be checked
        if i == 1 then
            chosen = s;  // choose the first vertex from sequence
        else
            chosen = read sequence[i] from file  // choose the next vertex from sequence

        if check_min(chosen, nv) then
            return false;  // sequence is invalid, as chosen vertex is not the minimum

        known[chosen] = 1;  // if valid, mark chosen vertex as known
        for each vertex j from 1 to nv{
            if j is adjacent to chosen then
                if dist[chosen] + weight < dist[j] then{
                    dist[j] = dist[chosen] + weight; // update the distance of j
                    path[j] = chosen;  // update the path to vertex j
                }
        }
    }
    return true  // sequence is valid
}
```

Chapter 3 : Testing Results

Two test cases, seperatively stored in file $case1.txt$ and $case2.txt$ are given to examine the correctness and efficiency of the program. Both results of these two cases are proved right, showing the program is correct and have strong robustness. Details of each case and their respective testing results are as follow :

Case1

Case1 is the sample shown in PTA system, which contains 5 vertices, 7 edges and 4 sequences, and is used for small graph validation. Both valid and invalid sequences are included and the algorithm can give right answers coresponding to sample output, as shown below

Input File (case1.txt)

```
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
2 3 4 5 1
3 2 1 5 4
```

Output File

```
Yes
Yes
Yes
No
```

Case2

Case2 tests input of maximun size, meaning that 1000 vertices, 100000 edges and 100 sequences are included. Data of case2 are generated by **datagen.c**, provided in appendix. This case examine the performance of the program in face of massive amount of data, and the result turns to be OK with output format is identical to given ones. However as 100 sequences are randmly generated and are too few regarding 1000 vertices, no valid sequences can be found. Due to that size of case2 is too big, here only offers a few contents.

Input File (case2.txt)

```
1000 100000
1 2 83
1 28 95
1 36 34
1 39 31
1 41 75
......
100
......
```

Output File

```
NO
NO
......
```

Chapter 4 : Analysis and Comments

Algorithm Analysis

The program involves several key steps which contribute to the overall time complexity, but it is maily dominated by the function **Dijkstra( )**. For each sequence, we need to examine all $|V|$ vertices, and for each vertex we need to check if it has the smallest tentative distance from the source using function **check_min( )** (which has a time complexity of $O|V|$) , while also find its neighbouring vertex in all $|V|$ vertices. Given that **Dijkstra( )** is called K times for K sequences, the overall time complexity will be $O(K \times |V|^2)$ for this algorithm.

On the other hand, the space complexity is $O(V^2)$ due to the adjacency matrix used to store the graph. Additional space is used for storing the distance, known, and path arrays, which are $O(V)$ each.

Comments

Through our analysis we can see that the Dijkstra algorithm has been efficiently implemented to check the validity of vertex sequences in graphs. However, while the algorithm performs well for typical graph sizes, it can still be optimized further in the following two aspects:

1. By using priority queues (min heap) we can reduce the complexity of finding the minimum distance vertex, resulting in an overall time complexity of $O(K \times |E|log|V|)$. This will have better time performances with graphs with sparse connectivity.
2. The algorithm uses adjacency matrix which is not space efficient and may consume much time in search of neighbouring vertex. By using adjacency list we can save some spaces and make the search faster.

Appendix : Source Code

main.c

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

static int graph[1005][1005];/*matrix to instore information of edges,
0 stands for no link and positive integer stands for weight of edges*/
int dist[1005];//length of shortest path
int known[1005];//mark the state of vertex
int path[1005];//record former vertex in the shortest pash

int check_min(int chosen,int nv)//check if 'chosen' has minimum distance
{
    for(int i=1;i<=nv;i++)
    {
        if((known[i]==0)&&(dist[i]<dist[chosen]))
            return 1;
    }
    return 0;
}

int Dijkstra(int nv, FILE *file)//function to check validity of a sequence
{
    int s,v;
    int flag=0;
    for(int i=1;i<=nv;i++){//initialization
        known[i]=0;//mark all vertex as unknown
        dist[i]=100000000;//set the distance to infinity
        path[i]=0;//set the former vertex to 0
    }
    fscanf(file, "%d", &s);
    dist[s]=0;//set the distance of the source vertex to 0
    for(int i=1;i<=nv;i++)
    {
        int chosen;//variable to store unkown vertex of minimum distance
        if(i==1)
            chosen=s;
        else{//use the next vertex in the sequence as 'chosen'
            fscanf(file, "%d", &chosen);
        }

        if(check_min(chosen,nv)){//if the vertex in the sequence does not have minimun distance
then flag=1
            flag=1;
        }
        if(flag){
```

```
            continue;
        }

        known[chosen]=1;//mark 'chosen' as known
        for(int j=1;j<=nv;j++)//check all vertex adjcent to 'chosen'
        {
            if(graph[chosen][j]>0)//if there exists an edge
            {
                if((known[j]==0)&&(dist[chosen]+graph[chosen][j]<dist[j]))//if the shortest path
can be updated
                {
                    dist[j]=dist[chosen]+graph[chosen][j];//update the shortest path
                    path[j]=chosen;//record the former vertex
                }
            }
        }
    }
    if(flag)
        return 0;
    else
        return 1;
}

int main()
{
    FILE *file;
    FILE *fp;
    fp = fopen("results.txt", "w+");//open file to write

    int choice;
    printf("Please enter the file number:\n");
    scanf("%d",&choice);
    switch(choice)//choose the file to read
    {
        case 1:file=fopen("case1.txt","r");break;
        case 2:file=fopen("case2.txt","r");break;
        case 3:file=fopen("case3.txt","r");break;
        default:printf("Invalid file number\n");return 0;
    }

    int nv,ne,k;//initialize the number of vertex, number of edges and number of sequences
    fscanf(file, "%d", &nv);
    fscanf(file, "%d", &ne);

    int v1,v2,weight;
    for(int i=0;i<ne;i++)//initialize the adjacent matrix
    {
        fscanf(file, "%d", &v1);
        fscanf(file, "%d", &v2);
        fscanf(file, "%d", &weight);
        graph[v1][v2]=graph[v2][v1]=weight;//set the weight of edges
    }

    fscanf(file, "%d", &k);
    for(int i=0;i<k;i++)
```

```
    {
        if(Dijkstra(nv, file))//if is a valid dijkstra sequence
            fprintf(fp,"Yes\n");
        else//if not a valid dijkstra sequence
            fprintf(fp,"No\n");
    }

    fclose(file);
    return 0;
}
```

datagen.c

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

static int graph[1001][1001];//adajacency matrix

// Function to generate 100 permutations of 1000 vertices
void permutation(FILE *fp, int n, int k) {
    int v[1001];
    for (int i = 1; i <= n; i++) {
        v[i] = i;
    }

    for (int seq = 0; seq < k; seq++) {
        // Shuffle the array to generate a random permutation
        for (int i = n; i > 1; i--) {
            int j = rand() % i + 1; // Random index between 1 and i
            int temp = v[i];
            v[i] = v[j];
            v[j] = temp;
        }

        for (int i = 1; i <= n; i++) { // Write the permutation to the file
            fprintf(fp, "%d ", v[i]);
        }
        fprintf(fp, "\n");
    }
}

int main()
{
    FILE *fp;
    fp = fopen("case2.txt", "w+");//open file to write
    srand((unsigned)time(NULL));

    // Create a graph which is guaranteed to be connected, 999 edges used
    for (int i = 1; i <= 999; i++) {
        graph[i][i + 1] = graph[i+1][i] = rand() % 100 + 1; // Set weight to 1-100
    }
```

```
    int cnt = 999; // Variable to count edges
    for (; cnt < 100000;) // Set 100000 edges
    {
        int m = rand() % 1000 + 1; // Row
        int n = rand() % 1000 + 1; // Column
        if ((n != m) && (graph[m][n] == 0))
        {
            graph[m][n] = graph[n][m] = rand() % 100 + 1; // Set edges
            cnt++;
        }
    }

    fprintf(fp, "%d %d\n", 1000, 100000); // Print the number of vertices and edges
    for (int i = 1; i <= 1000; i++) {
        for (int j = i; j <= 1000; j++) {
            if (graph[i][j] != 0) {
                fprintf(fp, "%d %d %d\n", i, j, graph[i][j]); // Print the edges
            }
        }
    }
    fprintf(fp, "%d\n", 100);

    // Generate 100 random permutations of 1000 vertices
    permutation(fp, 1000, 100);

    fclose(fp);
    return 0;
}
```

Declaration

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