Min Path in Library

Created by:
Bbiswabasu Roy (19EC30008)
Shaswata Dutta (19EC30042)

Motivation

When we are in a library, we often need to find a book on a particular topic of our requirement. Books of that particular topic may be present in various halls in the library. This problem attempts to guide a person to the nearest hall from his current location where he/she can find the book of required topic.

Formal Problem Statement

A library is defined as inter-connected halls with some distance from one hall to another. Each hall has several books. Each book has some tags associated with it. Given the current hall id where the user is present, he searches for a particular tag, say 't'. Our program will print the hall at shortest distance where such a book can be found along with the path which is to be followed to reach there.

Input

First line contains (n,c)-the no of halls & the no of connections

Next c lines contains 3 integers (u,v,w) - two hall id's (which have a connecting edge between them) & distance between these two halls

Each of the next n lines contains some space separated strings denoting that those tags are found in that hall.

Next line contains the hall id in the range [1,n] denoting that the person (searching for his required book) is currently in that hall.

The last line contains a single string t - the tag which he searching for

Output

First line displays two integers - the hall id denoting the nearest hall, and the distance (from current hall) to that hall.

The next line displays the path which he must follow to reach there.

Overview of the algorithm

Take all inputs

Let G be the graph of connectivity of halls having V nodes and E weighted edges

Compute shortest path to all reachable halls from the current hall irrespective of whether the tag searched for is present there or not. This can be efficiently done using Dijkstra’s algorithm. Also store parent of each node during the traversal.

In order to choose the optimal hall, we need to go through all the halls and if its distance from user’s hall is less than the distance to the last found valid hall, then we will binary search for the searched tag in the taglist of that list. If found, then this hall is the current optimal hall and the process continues until all the halls have been traversed.

Finally we display optimal hall and since we had already stored parent of each hall, we can trace the path to this hall.

Pseudocode of the algorithm

```
Dijkstra(G, w, src)
{
    d={}; parent={};
    for v in G.V:
    d[v]= infinity;
    parent[v]=NULL;
    d[src]=0;


     init(Q, n);            //initialize the priority queue Q with
    default distance value of each node equal to infinity and the
    parent of each node equal to NULL

    while(Q is not empty)
    {
    u = remove_min(Q); //remove hallId with least d[hallId] and
    update

    for each v in G.adj[u]:
    { if(d[v] > d[u]+w[u][v])
        { d[v] = d[u]+w[u][v]; parent[v] = u; }
    }
}
```

```
findHall(d[], parent[], src, n, booktag, books)
{
    leastdis=infinity; hallID = -1;
    for(i=1 to n)
    {
        if(leastdis > d[i])
        {
            if(search(booktag, book[i], size[i]))
            {
                leastdis = d[i];
                hallID = i;
            }
        }
    }
    return {hallID, leastdis};          //returns hallID and
    required corresponding least distance
}
```

Pseudocode of the algorithm (continued..)

```
compute_path(hallID, parent, src)
{
          Path = {} //Stores path to required hall
          size=0
          Hall, leastdis = findHall(...)
          if(hall = -1)
          {
                    return null;                        //book-tag not found
          }
          while(Hall has parent){
                    Path.insert(Hall); size++;
                    Hall = Parent of Hall //Tracing back the path to source
          }
          for(i = size to 1)
          {
                    print(Path[i]);
          }
}                                                       //end of compute_path pseudocode
```

Analysis of the Algorithm

1. Time complexity analysis:
   - Assume V halls, E edges, no of tags in a hall is atmost T
   - Dijkstra → O(V + Elog(V))
   - Search → O(logT) [We have assumed each tag length atmost 10 since they are generally small strings]
   - chooseHall → O(VlogT)
   - So, overall complexity → O(VlogT + ElogV)

Analysis of the Algorithm (continued..)

2. Space complexity analysis:

The distance, parent arrays as well as the priority queues need Θ(V) space. The adjacency list for all vertices will take Θ(E) in total. The books 2-D array of strings needs Θ(V*maximum number of book-tags in a hall*maximum length of each book-tag) space. Finally, the path array needs O(V) space. So total space complexity is O(V + E + V*maximum number of book-tags in a hall*maximum length of each book-tag).

Implementation of Heaps

```
typedef struct
{
        int first,second;
}Pair;
typedef struct
{
        Pair* data;
        int size,capacity;
}Heap;
void init(Heap* heap, int capacity){
        heap->capacity=capacity;
        heap->data=(Pair*)malloc(capacity*sizeof(Pair));
    for(int i=1;i<capacity;i++){
```

```
                (heap->data)[i].first=(1e8);
                (heap->data)[i].second=0;
        }
        heap->size=0;
}
void update_up(Heap* heap,int pos){
        if(pos==1)    return;
        if((heap->data)[pos/2]>(heap->data)[pos]){
                int t=(heap->data)[pos];
                (heap->data)[pos]=(heap->data)[pos/2];
                (heap->data)[pos/2]=t;
                update_up(heap,pos/2);
        }
}
```

Implementation of Heaps (contd.)

```
void insert(Heap* heap,int u,int w){
        ++(heap->size);
        if(heap->size > heap->capacity){
                heap->capacity=(heap->capacity)*2;
        heap->data=(Pair*)realloc(heap->data,heap->capacity*sizeof(Pair));
        }
        (heap->data)[heap->size].first=w;
        (heap->data)[heap->size].second=u;
        update_up(heap,heap->size);
}
void update_down(Heap* heap,int pos){
        if(2*pos>(heap->size) || ((heap->data)[pos].first<(heap->data)[2*pos].first&& (heap->data)[pos].first<(heap->data)[2*pos+1].first))
                return;
```

```
    if((heap->data)[2*pos].first<(heap->data)[2*pos+1].first){
                Pair t=(heap->data)[2*pos];
                (heap->data)[2*pos]=(heap->data)[pos];
                (heap->data)[pos]=t;
                update_down(heap,2*pos);
        }
        else{
                Pair t=(heap->data)[2*pos+1];
                (heap->data)[2*pos+1]=(heap->data)[pos];
                (heap->data)[pos]=t;
                update_down(heap,2*pos+1);
        }
```

Implementation of Heaps (contd.)

```
Pair remove_min(Heap* heap){
        Pair x=(heap->data)[1];
        (heap->data)[1]=(heap->data)[heap->size];
        (heap->size)--;
        update_down(heap,1);
        return x;
}
```

Implementation of Graph Adjacency List structure

```
struct AdjListNode

{

    int dest, weight;

    struct AdjListNode* next;

};

struct AdjList

{

    struct AdjListNode *head;

};

struct Graph

{

    int V;

    struct AdjList* array;

};
```

```
struct AdjListNode* newAdjListNode(int dest, int weight)

{

    struct AdjListNode* newNode =(struct
AdjListNode*)malloc(sizeof(struct AdjListNode));

    newNode->dest = dest;

    newNode->weight = weight;

    newNode->next = NULL;

    return newNode;

}
```

Implementation of Graph Adjacency List structure (contd.)

```
struct Graph* createGraph(int V)

{

    struct Graph* graph = (struct Graph*)malloc(sizeof(struct
Graph));

    graph->V = V;



    graph->array = (struct AdjList*)malloc(V * sizeof(struct
AdjList));



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

        graph->array[i].head = NULL;



    return graph;

}
```

```
void addEdge(struct Graph* graph, int src, int dest, int weight)

{



    struct AdjListNode* newNode = newAdjListNode(dest, weight);

    newNode->next = graph->array[src].head;

    graph->array[src].head = newNode;



    newNode = newAdjListNode(src, weight);

    newNode->next = graph->array[dest].head;

    graph->array[dest].head = newNode;

}
```

Implementation of Dijkstra

```
void dijkstra(struct Graph* graph, int src,int* dist,int* par)
{
        int V=graph->V;
        for(int i=0;i<V;i++)
                dist[i]=(int)(1e8);
        Heap *heap=(Heap*)malloc(sizeof(Heap));
        init(heap,V);
        insert(heap,src,0);
        par[src]=-1;
        dist[src]=0;
    while(heap->size > 0)
        {
                Pair top=remove_min(heap);
                int d=top.first,u=top.second;
```

```
        if(d!=dist[u])
                        continue;
                struct AdjListNode* node=graph-
>array[u].head;
                while(node!=NULL)
                {
                        int v=node->dest;
            if(dist[u]+(node->weight)<dist[v])
                        {

        dist[v]=dist[u]+(node->weight);

        par[v]=u;  insert(heap,v,dist[v]);
                        }
            node=node->next;;
        }
```

Implementation of choosing optimal hall

```
int search(char *s,char **list,int size)
{
        int l=0,r=size-1,mid;
        while(l<=r)
        {
                mid=(l+r)/2;
                int compare=strcmp(s,list[mid]);
                if(compare==0) return 1;
                if(compare>0) l=mid+1;
                else r=mid-1;
        }
        return 0;
}
```

```
int chooseHall(int V,char *tag, char ***tagList,int *dist,int *size)
{
        int hall=-1,minDist=1e8;
        for(int i=0;i<V;i++)
        {
                if(search(tag,tagList[i],size[i]) && dist[i]<minDist){
                        minDist=dist[i];
                        hall=i;
                }
        }
        return hall;
}
```

Implementation to display output

```
void compute_path(int *par,int V,int hall,int distToHall){
                    printf("Optimal hall: %d\nDistance to
hall:%d\n",hall+1,distToHall);
                    int *path=(int*)malloc(V*sizeof(int));
                    int node=hall,i=0;
          while(node!=-1){
                                        path[i++]=node;
                                        node=par[node];
                    }
          printf("Path to hall: %d",path[--i]+1);
          while(i>0)
                    printf(" -> %d",path[--i]+1);
          printf("\n");
}
```

Implementation of main() Function:

```
int main(){int n, c;                         printf("Enter the number of halls:");

                    scanf("%d", &n);                         struct Graph* g;

                    g = createGraph(n);                      //creation of graph
data-structure for the n halls

                    printf("Enter the number of connections among the
halls:");

                    scanf("%d", &c);

                    int i, j, k;

                    printf("\nEnter the 2 hall ID's of a pair of connected halls
and the distance between them: (for %d records)\n", c);

                    for(i=0;i<c;i++)

                    {printf("Enter three space separated integers denoting
the pair and the distance: ");

int u, v, w;
```

```
scanf("%d %d %d", &u, &v, &w); u--;v--;
                                        addEdge(g, u, v, w);

                    }
                    printf("Enter the book tags present in respective halls:
(space separated strings of total length not more than for 10 for each
book)\n");
                    char ***books = (char***)malloc(n*sizeof(char**));
                    int* size = (int*)malloc(n*sizeof(int));
          for(i=0;i<n;i++)
                    {

                    books[i]=(char**)malloc(20*sizeof(char*));
                    }
                    char str[301];
                    gets(str); //false input to initiate gets
                    for(i=0;i<n;i++)
                    {
                                        printf("Book: %d", i+1);
                                        gets(str);
                                        int len=strlen(str),pos=0; int cnt=0;
                                        for(j=0;j<=len;j++)
                                        {char w[21];
                                                            if(j==len || str[j]==' '){

                    w[pos]='\0';
                                                                                pos=0;

                    books[i][cnt]=(char*)malloc(10*sizeof(char));

                    strcpy(books[i][cnt],w);
                                                                                cnt++;
                                                            }
                                                            else

                    w[pos++]=str[j];
```

Continuation of main() Function:

```
        printf("Enter the Hall ID of the hall the person is currently in: (an
integer from 1 to n)");

                int src; scanf("%d", &src); src--;

                char searchstr[21];

                printf("Enter the book-tag he is searching for:");

                scanf("%s", searchstr);

                int *dist = (int*)malloc(n*sizeof(int)), *par =
(int*)malloc(n*sizeof(int));

                dijkstra(g,src,dist,par);

                int hall=chooseHall(n,searchstr,books,dist,size);

                if(hall==-1)

                                printf("Tag not found\n");

                else

                                compute_path(par,n,hall,dist[hall]);


}
```

```
static int comp(const void* a,const void* b)

 {

                return strcmp(*(const char**)a,*(const
char**)b)>0;

 }
```