دانشگاه بوعلی سینا

پروژه نهایی ساختمان داده

استاد:

سمیرا خدابنده لو

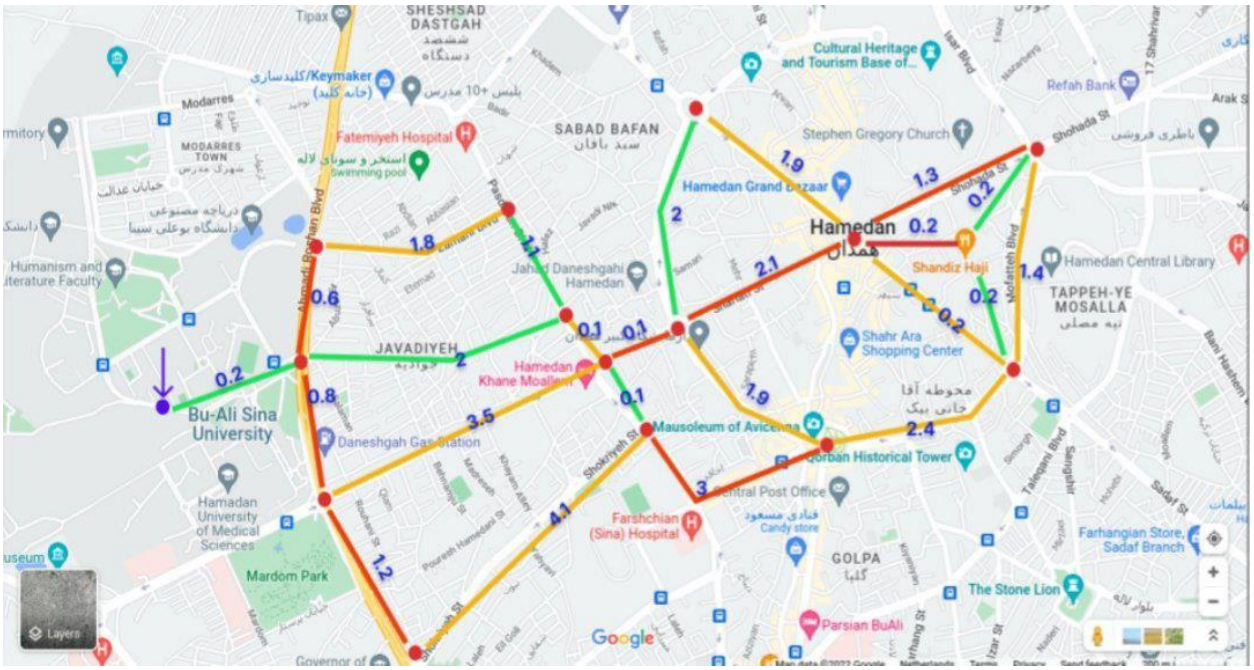
نرجس لاچین اف

1401

مقدمه:

در این پروژه با توجه به داده های مسئله کم هزینه ترین مسیر موجود ما بین خوابگاه دانشگاه بوعلی سینا و رستوران شاندیز حاجی را می یابیم.

روش حل:



شکل بالا گراف در نظر گرفته شده است.

راس ها و یال های آن در قالب استراکت تعریف شده اند که این استراکت ها در وکتور هایی نگهداری می شوند.

و توابع ای که خواسته های مسئله را حل می کند تعریف شده اند.

اجزای تشکیل دهنده:

1)

```
 struct node

{
    node(string n ,vector<string> a ) : name(n), adjacent(a)  {}
    string name;
    vector<string> adjacent;
    bool met  = false;
    float cost = INT_MAX ;
    string parent ;
};
```

این استراکت برای تعریف یک رأس ایجاد شده که strint name اسم آن رأس وvector<string> adjacent وکتوری از نام همسایه های آن راس است bool met که به طور پیش فرض false در نظر گرفته شده است زمانی true می شود که آن رأس درالگوریتم dijkstra خارج شود float costهزینه هر رأس در الگوریتم dijkstra است که به طور پیش فرض بی نهایت در نظر گرفته شده است و string parent نام راسی است که در نهایت هزینه این رأس را تعیین می کند و ازآن برای یافتن کم هزینه ترین مسیر استفاده می شود.

این استراکت کانستراکتوری دارد که با گرفتن نام یک راس و لیستی از نام همسایه های آن راس یک رأس را تعریف می کند.

(2

```
struct edage
{
    edage(string s , string d , float dis , float t) : node1(s) , node2(d) ,
distance(dis) ,  traffic_insesity(t) {}
    string node1;
    string node2;
    float distance;
    float traffic_insesity;
    float cost = traffic_insesity * distance;
};
```

این استراکت برای تعریف یک یال ایجاد شده که strintg node1 و string node2 اسم رأس های این یال و float distance و float traffic_insesity به ترتیب مسافت و ضریب ترافیک یک یال هستند و float costهزینه یک یال است.

این استراکت کانستراکتوری دارد که با گرفتن نام رأس های یک یال و مسافت و ضریب ترافیک آن یک یال را تعریف می کند.

(3

```
void creat_graf()
{
 keep_node = {

  node ("shandiz_haji"
,{"intersection_takhti","square_khomeyni","square_parvaneha"}),
                  …
               };
keep_edage = {edage("shandiz_haji", "square_khomeyni" ,0.2 ,3 ),….

   };



}
```

این تابع با فراخوانی کانستراکتور استراکت های بالا تمای رأس ها و یال های موجود را تعریف می کند و آن ها را در وکتور های keep_node

و keep_edage نگهداری می کند .

(4

```
node min_cost(node src , node des )
{
    for (int i =0  ; i < keep_edage.size()  ; i++)
    {
        if ( (keep_edage[i].node1 == src.name && keep_edage[i].node2 == des.name)
||(keep_edage[i].node1 == des.name && keep_edage[i].node2 == src.name)   )
        {
            if( des.cost == INT_MAX)
            {
                des.cost  = keep_edage[i].cost + src.cost;
                cost.push_back(des.cost);
                des.parent = src.name;
                return des;
            }
            else if(des.cost > (src.cost + keep_edage[i].cost) && des.cost -
(src.cost + keep_edage[i].cost) > 0.1 )
            {
                replace(cost.begin() , cost.end() , des.cost ,keep_edage[i].cost
+ src.cost );
                des.cost  = keep_edage[i].cost + src.cost;
                des.parent = src.name;
                return des;
            }
            return des;
        }
    }

}
```

این تابع تعیین کننده هزینه جدید راس مقصد و والد آن است به طوری که نام دو راس را می گیرد و یال ما بین آن های را پیدا می کند اگر هزینه رأس مقصد بی نهایت باشد هزینه آن را برابر با هزینه یال + هزینه راس مبدا می کند و parent رأس مقصد رأس مبدا می شود هزینه جدید رأس مقصد در وکتور cost ثبت می شود و تابع آن را برمی گرداند اما اگر هزینه رأس مقصد بزرگتر از هزینه یال + هزینه رأس مبدا باشد هزینه آن باز هم برابر هزینه یال + هزینه راس مبدا می شود و باز هم parent رأس مقصد رأس مبدا می شود

اما هزینه جدید رأس مقصد در وکتور cost جایگزین هزینه قبلی آن می شود و تابع آن را برمی گرداند و در غیر این صورت هزینه قبلی را برمیگرداند .

(5

```
void dijkstra( string src)
{
    int num1;
    int num2;
    for(int i = 0 ; i < keep_node.size() ; i++)
    {
        if(keep_node[i].name == src)
        {
          num1 = i;
        }
    }
    if(first)
    {
        keep_node[num1].cost  = 0;
    }
    keep_node[num1].met = true;
    first = false;

    for (int i = 0 ; i <keep_node[num1].adjacent.size()   ; i++)
    {
        for(int j = 0 ; j <keep_node.size() ; j++)
        {
            if(keep_node[j].name == keep_node[num1].adjacent[i] )
            {
                num2 = j;
            }
        }

        if(keep_node[num2].met == false)
        {
            keep_node[num2] =  min_cost(keep_node[num1] , keep_node[num2]  );
        }
    }

    sort(cost.begin(), cost.end());
    for (int i = 0  ; i < keep_node.size()   ; i++)
    {
        if(keep_node[i].cost == cost[0] )
        {
            if(keep_node[i].met == false )
            {
                cost.erase(cost.begin());
                dijkstra( keep_node[i].name);
```

```
            }
        }
    }
}
```

این تابع نام یک راس را به عنوان راس مبدا می گیرد و کمترین هزینه برای پیمایش از ان راس به باقی راس ها می یابد .

اولین بار که فراخوانی میشود هزینه این راس را صفر می گذارد می دانیم که هزینه باقی راس ها بی نهایت است.

سپس این تابع راسی که گرفته است را به همراه تک تک همسایه هایش اگر خارج نشده باشند به تابع بالایی می دهد تا هزینه جدید نود های همسایه اش تعیین شوند و راس مبدا را خارج می کند.

بعد ازآن این تابع با راس دیگری فراخوانی می شود که آن راس راسی است که خارج نشده و کمترین هزینه را دارد اینکار را تا جایی ادامه می دهیم تا همه راس ها به عنوان راس مبدا به این تابع داده شده باشند برای اینکار وکتور cost که حاوی هزینه های جدید است را مرتب می کنیم تا کمترین هزینه در اولین خانه آن قرار گیرد و نودی که دقیقا همان هزینه را دارد می یابیم و به این تابع می دهیم.

(6

```
string print_shortest_path(string src, string des)
{
    if(first)
    {
        dijkstra( src);
    }

    if(des != src)
    {
         for(int i = 0 ; i < keep_node.size() ; i++)
        {
            if(keep_node[i].name == des)
            {
                return print_shortest_path(src,keep_node[i].parent ) +"->" + des
;
            }
        }

    }
    else
    {
        first = false;
        return   des ;
    }
}
```

این تابع راس مبدا و مقصد را می گیرد و بهترین مسیر ما بین آن ها که همان کم هزینه ترین مسیر است را چاپ می کند.

ابتدا تابع بالایی را فراخوانی می کند که هزینه بهترین مسیر و والد هر راس مشخص شود.

برای یافتن بهرین مسیر از سمت راس مقصد به سوی والد آن می رویم سپس والد آن را مقصد در نظر می گیریم و به سوی والد والد می رویم تاجایی ادامه می دهیم تا به راس مبدا برسیم.

(7

```
float find_cost(string path)
{

    string src;
    string des;
    float total_cost = 0;
    int found = 0;
    int i = -2;
    int j;

    while(found != -1)
    {
        found = path.find("->" ,found + 1 );
        j = found;
        if( i == -2)
        {

            src = path.substr(0 , j );
        }
        else
        {
            des = path.substr(i + 2 , j -i - 2 );
            for (int k = 0; k <= keep_edage.size(); k++)
            {
                if( (keep_edage[k].node1 == src && keep_edage[k].node2 == des)
||(keep_edage[k].node1 == des && keep_edage[k].node2 == src))
                {

                    total_cost = total_cost +( keep_edage[k].traffic_insesity *
keep_edage[k].distance) ;
                }
            }
            src = des;
        }
        i = j;
    }
    return total_cost;
}
```

این تابع یک مسیر را می گیرد و هزینه آن را بر می گرداند.

مسیر ها به فرم زیر می باشند:

```
shandiz_hji->square_khomeyni->abdolah_shrine
```

این تابع با یافتن <- راس ها را از این مسیر بیرون می کشد به طوری که اولین بار اولین راس از سمت چپ را راس مبدا و دومی را راس مقصد در نظر می گیرد و هزینه یال ما بین آنها را در متغییری درج میکند سپس راس مقصد مبدا و راس بعدی راس مقصد می شود و مقدار هزینه یال ما بین آنها با هزینه یال قبلی جمع و در همان متغییر نگهداری می شود اینکار را تاجایی ادامه میدهیم که دیگر <- یافت نشود و راس ها تمام شده باشند.

(8

```
 float find_distance(string path)

{
    string src;
    string des;
    float total_distance = 0;
    int found = 0;
    int i = -2;
    int j;

    while(found != -1)
    {
        found = path.find("->" ,found + 1 );
        j = found;
        if( i == -2)
        {

            src = path.substr(0 , j );
        }
        else
        {
            des = path.substr(i + 2 , j -i - 2 );
            for (int k = 0; k <= keep_edage.size(); k++)
            {
                if( (keep_edage[k].node1 == src && keep_edage[k].node2 == des)
||(keep_edage[k].node1 == des && keep_edage[k].node2 == src))
                {
                    total_distance = total_distance + keep_edage[k].distance;

                }
            }
            src = des;
        }
        i = j;
    }
    return total_distance;


}
```