算法设计与分析实验3

计科2205 刘志垚 202208010512

1.Dijkstra求解单源最短路径问题

问题描述

给定一个图 $G=(V,E)$，其中 V 是顶点集，E 是边集，每条边都有一个非负权重 $w(u,v)$ 表示从顶点 u 到顶点 v 的边的权重。目标是从源点 s 出发，找到从源点到图中所有其他顶点的最短路径。

思路

核心思想是：**每次选择当前距离源点最近的未访问顶点，更新其邻接点的距离**。这种方法会在保证局部最优的同时，通过不断更新全局最短路径。

1. 初始化：

对于每个顶点 v（除了源点 s），将其最短路径初始化为无穷大（`INF`）。对于源点 s，最短路径初始化为 0。

使用一个布尔数组 `vis[]` 来记录哪些顶点已经被访问过。最初，所有顶点的 `vis` 状态为 `false`，源点被设置为 `true`。

2. 贪心选择：

在每一步，选择当前距离源点最近的未访问顶点作为当前顶点 u。

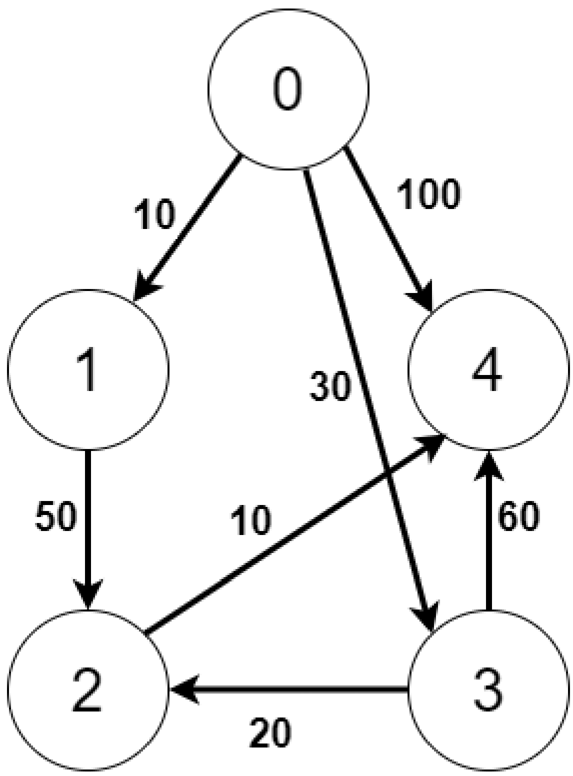
更新与 u 相邻的顶点的最短路径值。

更新规则：如果 $dist[v] > dist[u] + w(u, v)$，则更新 $dist[v]$ 为 $dist[u] + w(u, v)$。

3. 继续选择：

每次选择距离源点最近的顶点后，将其标记为已访问，并更新其邻接点的最短路径。继续选择下一个未访问的最短路径顶点。

4. 重复步骤2和3，直到所有顶点都被访问或没有可访问的顶点。



使用邻接矩阵来存储

```
INF, 10, INF, 30, 100
INF, INF, 50, INF, INF
INF, INF, INF, INF, 10
INF, INF, 20, INF, 60
INF, INF, INF, INF, INF
```

代码

```
//用Dijkstra贪心算法求解单源最短路径问题
#include <iostream>
#include <vector>
#include <bits/stdc++.h>
#define INF 0x3f3f3f3f
#define endl '\n'
const int N = 1e5 + 9;
using namespace std;
int dist[N];
bool vis[N];
vector<vector<int> >graph;
void dijkstra(int s) {
    int n = graph.size();
    memset(dist, 0x3f, sizeof(dist));

    dist[s] = 0;
    // 选择距离源点最近的未访问顶点。
    for(int i = 0; i < n; i++) {
        int minDist = INF, u = -1;
        for(int  j = 0; j < n; j++) {
            if(!vis[j] && dist[j] < minDist) {
                minDist = dist[j];
                u = j;
            }
        }

        if(u == -1) return;

        vis[u] = true;

        // 更新其相邻点的最短路径。
        for(int v = 0; v < n; v++) {
            if(!vis[v] && dist[v] > dist[u] + graph[u][v])
                dist[v] = dist[u] + graph[u][v];
        }
    }
}
int main() {
    int s;
    graph={
        {INF, 10, INF, 30, 100},
        {INF, INF, 50, INF, INF},
        {INF, INF, INF, INF, 10},
        {INF, INF, 20, INF, 60},
        {INF, INF, INF, INF, INF}
    };
    dijkstra(0);
```

```
    for(int i = 0; i < graph.size(); i++) {
        cout << "dist" << "[" << i << "]: " << dist[i] << endl;
    }
    return 0;
}
```

测试结果

```
dist[0]: 0
dist[1]: 10
dist[2]: 50
dist[3]: 30
dist[4]: 60
```

复杂度分析

每次都需要遍历所有未访问的顶点，寻找距离源点最近的顶点，更新相邻点的最短距离，因此时间复杂度是 $O(V^2)$。

1(续). 基于优先队列实现的Dijkstra算法

思路

因为在原始的实现中，每次都需要遍历所有未访问的顶点，寻找距离源点最近的顶点，这样的时间复杂度是 $O(n^2)$，对于稠密图来说效率较低。

使用 **优先队列** 后，可以在 $O(\log n)$ 的时间内提取出距离源点最近的顶点，从而优化 Dijkstra 算法的时间复杂度。

为什么是 $O(\log n)$？

优先队列是基于 最小堆 的数据结构实现的，队列中的元素会按照其值（在本题中是距离）进行排序，值最小的元素总是位于堆的根节点。最小堆的两个主要操作——**插入**和**提取最小值**

1. **插入操作（Push）**：将一个元素插入堆时，需要将其放到堆的底部，然后通过 **堆化**（heapify）过程将它移动到正确的位置。这一过程的时间复杂度为 $O(\log n)$， n 是堆中元素的数量。
2. **提取最小元素操作（Pop）**：当提取最小元素时，最小堆的根节点会被移除，并且最后一个元素会被放到根节点的位置，然后再次通过 **堆化** 将其调整到正确的位置。这个操作的时间复杂度同样是 $O(\log n)$。

如何实现优先队列?

C++ STL（标准模板库）中的一个容器适配器 `priority_queue`，底层默认使用 `vector` 存储数据，元素通过堆结构来维护顺序。

默认情况下，`priority_queue` 使用最大堆（Max Heap）来存储元素，最大元素放在队首，也就是根节点。

```
std::priority_queue<int> pq;
```

如果需要最小堆（Min Heap），可以使用 greater 作为比较器。

```
priority_queue<int, vector<int>, greater<int> > pq;
```

优先队列实现Dijkstra算法步骤:

1. 每次从优先队列中提取出 **当前距离源点最近的顶点**（即堆顶元素）。这就是所谓的 **贪心选择**。
2. 然后，更新该顶点的所有相邻顶点的最短路径，并将更新后的顶点重新插入到优先队列中。

代码

图的表示采用邻接表

```
0 -> [(1, 10), (3, 30), (4, 100)]
1 -> [(2, 50)]
2 -> [(4, 10)]
3 -> [(2, 20), (4, 60)]
4 -> []
```

c++代码

```
//优先队列实现Dijkstra
#include <iostream>
#include <vector>
#include <queue>
#include <bits/stdc++.h>
#define INF 0x3f3f3f3f
#define endl '\n'
#define PII pair<int, int>
const int N = 1e5 + 9;
using namespace std;
int dist[N];
bool vis[N];
vector<vector<PII> >graph;
priority_queue<PII, vector<PII>, greater<PII>> pq;

void dijkstra(int s) {
    int n = graph.size();
    memset(dist, 0x3f, n * sizeof(int));
    memset(vis, false, n * sizeof(bool));
    dist[s] = 0;
    pq.push({0, s});
```

```cpp
    while(!pq.empty()) {
        int u = pq.top().second;
        pq.pop();
        if(vis[u]) continue;
        vis[u] = true;
        for(auto edge : graph[u]) {
            int v = edge.first;
            int weight = edge.second;

            if(dist[v] > dist[u] + weight) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }

}
int main() {
    int s;
    graph={
        {{1, 10}, {3, 30}, {4, 100}},// 顶点0到其他顶点的边
        {{2, 50}},
        {{4, 10}},
        {{2, 20}, {4, 60}},
        {}
    };
    s = 0;
    dijkstra(s);
    // 输出每个顶点的最短路径
    for (int i = 0; i < graph.size(); i++) {
        if (dist[i] == INF)  // 如果dist[i]仍为无穷大，说明不可达
            cout << "dist[" << i << "]: INF" << endl;
        else
            cout << "dist[" << i << "]: " << dist[i] << endl;
    }
    return 0;
}
```

复杂度分析

最坏情况下，对于每个顶点，执行一次 **提取最小值操作**（即 $O(V \log V)$时间），对于每条边，执行一次 **插入操作**（即 $O(E \log V)$ 时间）。因此，总的时间复杂度为:

$$O((V + E) \log V)$$

2.收集样本问题（实现题3-15）

问题描述

3-15 收集样本问题。

问题描述：机器人 Rob 在一个有 $n×n$ 个方格的方形区域 F 中收集样本。(i, j)方格中样本的价值为 $v(i, j)$，如图 3-8 所示。Rob 从方形区域 F 的左上角 A 点出发，向下或向右行走，直到右下角的 B 点，在走过的路上,收集方格中的样本。Rob 从 A 点到 B 点共走 2 次，试找出 Rob 的 2 条行走路径，使其取得的样本总价值最大。

算法设计：给定方形区域 F 中的样本分布，计算 Rob 的 2 条行走路径，使其取得的样本总价值最大。

数据输入：由文件 input.txt 给出输入数据。第 1 行有 1 个正整数 n，表示方形区域 F 有 $n×n$ 个方格。接下来每行有 3 个整数，前 2 个数表示方格位置，第 3 个数为该位置样本价值。最后一行是 3 个 0。

结果输出：将计算的最大样本总价值输出到文件 output.txt。

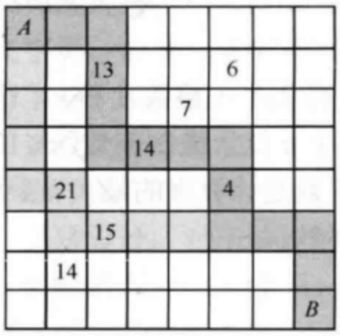
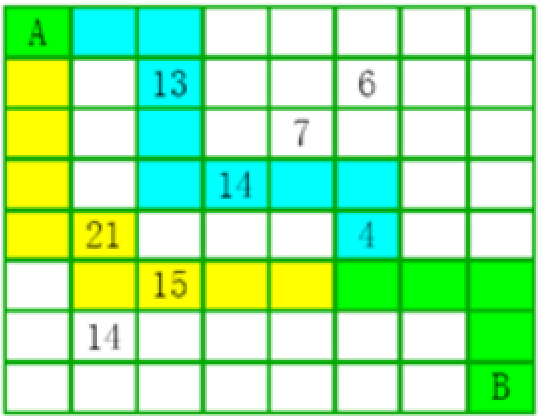


图 3-8 $n×n$ 个方格的方形区域 F

输入文件示例
input.txt
```
8
2 3 13
2 6 6
3 5 7
4 4 14
5 2 21
5 6 4
6 3 15
7 2 14
0 0 0
```

输出文件示例
output.txt
```
67
```

思路



给定一个由 $n \times n$ 个方格组成的地图，每个方格都有一个价值，每一步只能朝右或者朝下走，从左上角到右下角走两次，一个方格内的价值只能获取一次。

因此可以先走一遍，记下路径，然后回溯，把走过的格子清空，再走第二遍，输出两次价值的和。

$dp[i][j]$ 表示到达 (i, j) 的最大总价值，因为每一步只能朝右或者朝下走，以此状态转移方程为：

$$dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + v[i][j];$$

同时，用 bool 数组 $path[i][j]$ 记录路径选择，如果 $dp[i][j] = dp[i-1][j] + v[i][j]$ ，记 $path[i][j]$ 为 0，表明是由上方格子得到的，如果$dp[i][j] = dp[i][j-1] + v[i][j]$，记 $path[i][j]$ 为 1，表明是由左边格子得到的。

代码

```
#include <iostream>
#include <cstring>
using namespace std;
const int N = 55;
#define endl '\n'
#define up 0
#define left 1

int dp[N][N];
bool path[N][N];
int v[N][N];
// 回溯
void trace(int x, int y, int n) {
    if(x < 1 || y < 1) return;
    if(path[x][y] == up)
        trace(x - 1, y, n);// 由上方得来
    else if(path[x][y] == left)
        trace(x, y - 1, n);// 由左边得来

    if(x == 1 && y == 1) {
        cout << "A -> ";
        return;
    }
    else if(x == n && y == n)
        cout << "B" << endl << endl;
    else
        cout << "(" << x << "," << y << ") -> ";

    // 清空走过的格子
    v[x][y] = 0;
    return;
}
int solve(int n) {
    memset(dp, 0, sizeof(dp));
    memset(path, 0, sizeof(path));

    //第一列
    for(int i = 1; i <= n; i++) {
        path[i][1] = up;
        dp[i][1] = dp[i - 1][1] + v[i][1];
    }
```

```
    //第一行
    for(int j = 1; j <= n; j++) {
        path[1][j] = left;
        dp[1][j] = dp[1][j - 1] + v[1][j];
    }

    //动规实现
    for(int i = 2; i <= n; i++) {
        for(int j = 2; j <= n; j++) {
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]) + v[i][j];
            path[i][j] = dp[i - 1][j] > dp[i][j - 1] ? up : left;
        }
    }

    trace(n, n, n);

    return dp[n][n];
}
int main() {
    int n; cin >> n;
    int x, y;
    while(cin >> x >> y >> v[x][y]) {
        if(!x) break;
    }
    int ans1 = solve(n);
    int ans2 = solve(n);
    cout << ans1 + ans2 ;
    return 0;
}
```

测试结果

```
A -> (2,1) -> (3,1) -> (4,1) -> (5,1) -> (5,2) -> (6,2) -> (6,3) -> (7,3) -> (8,3) -> (8,4) -> (8,5) -> (8,6) -> (8,7) -> B
A -> (2,1) -> (2,2) -> (2,3) -> (3,3) -> (4,3) -> (4,4) -> (5,4) -> (5,5) -> (5,6) -> (6,6) -> (7,6) -> (8,6) -> (8,7) -> B
67
```

复杂度分析

回溯需要遍历整个路径，从 (n,n) 到 (1,1)，路径长度为 $2 \times n - 1$，回溯的时间复杂度为 $O(n)$，动态规划的时间复杂度为 $O(n^2)$，总的时间复杂度为：

$$O(n^2)$$

3.字符串比较问题（实现题3-17）

问题描述

3-17　字符串比较问题。

问题描述： 对于长度相同的两个字符串 A 和 B，其距离定义为相应位置字符距离之和。两个非空格字符的距离是它们的 ASCII 编码之差的绝对值。空格与空格的距离为 0，空格与其他字符的距离为一定值 k。

在一般情况下，字符串 A 和 B 的长度不一定相同。字符串 A 的扩展是在 A 中插入若干空格字符所产生的字符串。在字符串 A 和 B 的所有长度相同的扩展中，有一对距离最小的扩展，该距离称为字符串 A 和 B 的扩展距离。

对于给定的字符串 A 和 B，试设计一个算法，计算其扩展距离。

算法设计： 对于给定的字符串 A 和 B，计算其扩展距离。

数据输入： 由文件 input.txt 给出输入数据。第 1 行是字符串 A，第 2 行是字符串 B，第 3 行是空格与其他字符的距离定值 k。

结果输出： 将计算出的字符串 A 和 B 的扩展距离输出到文件 output.txt。

输入文件示例	输出文件示例
input.txt	output.txt
cmc	10
snmn	
2	

思路

ASCII
c：99
m：109
n：110
s：115

	0	s	n	m	n
0	0	2	4	6	8
c	2	4	6	8	10
m	4	6	5	6	8
c	6	8	7	8	10

用 $dp[i][j]$ 描述 $A[1:i]$ 和 $B[1:j]$ 两字符串之间的扩展距离，可以证明 $dp[i,j]$ 具有最优子结构的性质，满足如下状态转移方程：

$$dp[i][j]=\begin{cases}dp[i-1][j]+k\\dp[i][j-1]+k\\dp[i-1][j-1]+|A[i]-B[j]|\end{cases}$$

代码

```
#include <iostream>
#include <string>
#define endl '\n'
using namespace std;
string A, B;
int k;
int StrComp() {
    int len1 = A.length();
    int len2 = B.length();
    int dp[len1 + 1][len2 + 1];

    for(int i = 0; i <= len1; i++) dp[i][0] = i * k;
    for(int i = 0; i <= len2; i++) dp[0][i] = i * k;
    dp[0][0] = 0;

    for(int i = 1; i <= len1; i++) {
        for(int j = 1; j <= len2; j++) {
            dp[i][j] = min(dp[i - 1][j - 1] + abs(A[i - 1] - B[j - 1]), min(dp[i][j - 1] + k, dp[i - 1][j] + k));
        }
    }

    return dp[len1][len2];
}
int main() {
    cin >> A >> B >> k;
    cout << StrComp();
    return 0;
}
```

测试结果

```
cmc
snmn
2
10
```

复杂度分析

主要的复杂度在于 `dp()` 函数中的双重循环，因此时间复杂度为 $O(len1 \times len2)$