**最短路径算法研究**

**摘要**

最短路径算法是图论中重要的研究领域之一。本文总结了最短路径算法的研究成果，包括Floyd算法、Bellman-Ford算法与SPFA算法、A算法以及Dijkstra算法和Johnson算法。

Floyd算法是一种适用于有权图中任意两点之间最短路径问题的算法，通过动态规划的思想逐步求解每个节点之间的最短路径。它的优点在于简单易懂、正确性高，并且能够处理有向图和负权边。然而，Floyd算法的时间复杂度和空间复杂度较高，对稀疏图不适用。

Bellman-Ford算法和SPFA算法是解决带有负权边的图中最短路径问题的算法，其中SPFA算法对稠密图或存在正权边的图具有较好的表现。它们的优点是可以处理负权边，但也存在负环问题和时间复杂度较高的缺点。

A\*算法是一种启发式搜索算法，用于在图中找到两个节点之间的最短路径。它通过评估函数和优先级队列来提高搜索效率，具有最优性、完备性和启发式的特点。

最后，Dijkstra算法和Johnson算法也是常用的最短路径算法，它们各自适用于不同的问题场景。综上所述，根据具体情况选择最短路径算法是实际应用中的重要考虑因素。

**关键词：最短路径，Floyd、Dijkstra、Bellman-Ford、SPFA、A\*算法、Johnson算法**

**目录**

[**一、最短路径算法及相关概念 3**](#_Toc138609283)

[**二、Floyd算法 5**](#_Toc138609284)

[**1.算法简介 5**](#_Toc138609285)

[**2.适用情况 5**](#_Toc138609286)

[**3.优点 6**](#_Toc138609287)

[**4.缺点 6**](#_Toc138609288)

[**5.Floyd算法的使用选择 6**](#_Toc138609289)

[**6.深入理解 8**](#_Toc138609290)

[**7.代码实现 13**](#_Toc138609291)

[**三、BeIlman-Ford（贝尔曼-福特）算法与SPFA（Shortest Path Faster Algorithm）算法 13**](#_Toc138609292)

[**1.算法简介 13**](#_Toc138609293)

[**2.松弛操作 14**](#_Toc138609294)

[**3.适用情况 15**](#_Toc138609295)

[**4.各算法优缺点 16**](#_Toc138609296)

[**5.两种算法的联系和区别 16**](#_Toc138609297)

[**6.代码运行情况 17**](#_Toc138609298)

[**四、A\*算法 18**](#_Toc138609299)

[**1.算法简介 18**](#_Toc138609300)

[**2.适用情况 18**](#_Toc138609301)

[**3.优点 19**](#_Toc138609302)

[**4.缺点 19**](#_Toc138609303)

[**5.代码运行情况 20**](#_Toc138609304)

[**五、Dijkstra算法 21**](#_Toc138609305)

[**1.算法简介 21**](#_Toc138609306)

[**2.适用情况 21**](#_Toc138609307)

[**3.实现步骤 21**](#_Toc138609308)

[**4.优点 23**](#_Toc138609309)

[**5.缺点 23**](#_Toc138609310)

[**6.Dijkstra算法和A\*算法的比较 24**](#_Toc138609311)

[**6.代码运行情况 24**](#_Toc138609312)

[**六、Johnson算法 26**](#_Toc138609313)

[**1.算法简介 26**](#_Toc138609314)

[**2.适用情况 26**](#_Toc138609315)

[**3.优点 26**](#_Toc138609316)

[**4.缺点 27**](#_Toc138609317)

[**5.实现步骤 27**](#_Toc138609318)

[**6.代码实现情况 28**](#_Toc138609319)

[**七、小组收获 29**](#_Toc138609320)

[**八、参考文献 29**](#_Toc138609321)

[**附录 31**](#_Toc138609322)

# 一、最短路径算法及相关概念

最短路径问题是在图论中经常遇到的一个重要问题，它涉及在图中找到两个顶点之间的最短路径。该问题在实际生活中有广泛的应用，如网络路由、物流配送、航班规划等。

在最短路径问题中，常用的图模型是有向图或无向图，其中顶点表示位置或节点，边表示路径或连接。每条边可能带有权重或距离值，表示从一个顶点到另一个顶点的代价或开销。最短路径指的是从一个顶点到另一个顶点的路径中，使得路径上的边权重之和最小的路径。

**Ⅰ.以下是一些与最短路径问题相关的概念：**

1.单源最短路径问题（Single-Source Shortest Path Problem）：在图中给定一个起始顶点，求该顶点到其他所有顶点的最短路径。

2.单目标最短路径问题（Single-Destination Shortest Path Problem）：在图中给定一个目标顶点，求其他所有顶点到该目标顶点的最短路径。

3.单对最短路径问题（Single-Pair Shortest Path Problem）：在图中给定一个起始顶点和一个目标顶点，求两者之间的最短路径。

4.全局最短路径问题（All-Pairs Shortest Path Problem）：在图中求解任意两个顶点之间的最短路径。

**Ⅱ.常见的最短路径算法包括：**

1.Dijkstra算法：解决单源最短路径问题，适用于没有负权边的图。

2.Bellman-Ford算法：解决单源最短路径问题，适用于带有负权边的图。

3.Floyd-Warshall算法：解决全局最短路径问题，适用于有向图或无向图。

4.Johnson算法：解决全局最短路径问题，适用于带有负权边的图。

5.A\*算法：一种启发式搜索算法，通过估计函数来优化搜索路径，适用于单源最短路径问题。

这些算法在解决最短路径问题时具有不同的优势和适用条件，根据具体情况选择合适的算法能够有效地求解最短路径问题。

# 二、Floyd算法

## 1.算法简介

Floyd算法，也称为插点法，是一种用于解决所有最短路径问题的算法。该算法同样使用动态规划的思想，通过逐步插入中间节点，来逐步求解每个节点之间的最短路径。

时间复杂度：O(n^3)。

空间复杂度：O(n^2)。

## 2.适用情况

（1）*适用于有权图中任意两点之间最短路径的问题*，这种问题通常可以用一个矩阵来表示线路之间的距离或代价。所以，Floyd算法对于求解全源最短路径问题非常有效【1】。

1. 其时间复杂度为O(n^3)，适用于数据规模较小的情况。

（3）特别地，*当图的边权值为正整数时，Floyd算法比较适用，*因为它不会产生负元的问题，也不需要进行额外的处理。

（4）如果边权值为负数，则Floyd算法不再适用，因为在这种情况下就会出现产生负环的情况，因此需要使用其他算法，如Bellman-Ford算法或者Dijkstra算法等来解决最短路径问题。其时间复杂度为O(n^3)，适用于数据规模较小的情况。

## 3.优点

（1）简单易懂：Floyd算法的核心思想是动态规划，容易理解。

（2）正确性高：Floyd算法能够得到最优解，保证了算法的正确性。

（3）可以处理有向图和负权边：Floyd算法能够处理有向图和负权边的情况，具有较高的通用性。

## 4.缺点

（1）时间复杂度较高：Floyd算法的时间复杂度为O(n^3)，在图较大时计算量比较大。

（2）空间复杂度较高：Floyd算法需要使用二维数组存储任意两点之间的最短路径长度，占用空间较大。

（3）对于稀疏图不适用：如果图比较稀疏，使用Floyd算法会造成不必要的计算，影响算法效率。

## 5.Floyd算法的使用选择

在实际应用中需要根据具体情况来选择是否使用Floyd算法。如果图较大或者需要处理有向图、负权边等复杂情况，Floyd算法是一种很好的选择；如果图比较稀疏或者对时间复杂度和空间复杂度有较高要求，可以考虑使用其他最短路径算法来解决问题。

***下面是简单的C语言Floyd算法代码：***

#define N 100

#define INF 1000000

int dist[N][N]; //邻接矩阵存储图

int min(int a, int b) {

return a < b ? a : b;

}

void floyd(int n) {

for(int k=0; k<n; k++) {//遍历中间点

for(int i=0; i<n; i++) {//遍历出发点

for(int j=0; j<n; j++) {//遍历到达点

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);//更新最短路径

}

}

}

}

Floyd算法的主要思想是动态规划。假设我们要求i到j的最短路径，可以枚举中间点k，假设通过k点可以使得i到j的路径更短，那么就更新i到j的最短路径。

具体实现上，使用邻接矩阵来存储图，dist[i][j]表示i到j的最短路径。在遍历中间点、出发点、到达点的过程中不断更新最短路径即可。

需要注意的是，在初始化时，如果有两点之间没有边相连，则可以将它们的最短路径初始化为一个较大的值（这里使用了INF来表示）。初始化完毕后，就可以调用Floyd算法进行求解了。

## 6.深入理解

* **代码实现：**

1. *//MAXN为最大的点的数量*
2. int f[MAXN][MAXN];*//f[i][j]代表节点i到节点j的最短路*
3. int main()
4. {
5. int n,m;*//n为图中点的数量，m为图中边的数量*
6. scanf("%d%d",&n,&m);
7. for(int i=1;i<=n;i++)
8. for(int j=1;j<=n;j++)
9. f[i][j]=1e9;*//初始每个节点对之间都是不可达，设为最大值*
10. for(int i=1;i<=n;i++)f[i][i]=0;*//自己到自己最短路为0*
11. for(int i=0;i<m;i++){
12. int x,y,w;*//分别是起点，终点和边权*
13. scanf("%d%d%d",&x,&y,&w);
14. f[x][y]=w;
15. f[y][x]=w;*//如果图为无向图*
16. }
17. for(int k=1;k<=n;k++)*//枚举转移节点*
18. for(int i=1;i<=n;i++)
19. for(int j=1;j<=n;j++){
20. f[i][j]=min(f[i][j],f[i][k]+f[k][j]);
21. *//枚举k作为i->j的转移节点计算*
22. }
23. for(int i=1;i<=n;i++){
24. for(int j=1;j<=n;j++){
25. printf("%d ",f[i][j]);
26. *//输出i到j的最短路*
27. }
28. puts("");
29. }
30. return 0;
31. }

* **代码思想：**

我们记f[k][i][j]为在只允许通过1,2,..k的情况下，节点i到节点j的最短路径，显然我们有f[0][i][j]=原图给定的<i,j>之间的边权。转移过程:f[k][i][j] = min(f[k][i][j], f[k-1][i][k]+ f[k-1][k][j]) 即我们通过枚举中间点k进行转移，从而使状态的第一维扩展，最终答案即为f[n][i][j]。我们可以注意到f[k][i][j]的转移与第一维无关，从而优化空间复杂度为O(n^2)。

简要证明转移与第一维无关：f[k][i][j]的转移仅仅与k-1有关，而在转移f[k][i][j]之前,f[k][i][j]存的值就是上一次f[k-1][i][j]的答案，所以运用滚动数组的方式，我们可以取消掉第一维来节省空间。

* **应用场景【2】：**

1.给定一个正权无向图，寻找一个权值和最小的环：

由代码思想可得在第一个循环中我们规定了每次进行转移的最大的点的编号为k。在求最小环的思路中，我们同样可以认为环上的最大编号的点就是为k，在转移的过程中，记录枚举答案ans=f[i][j]+w[i][k]+w[k][j](注意到不是枚举f[i][k]+f[k][j]因为此时k还没进入到最短路更新中，先进行最小环寻找，不然最小环中可能会出现重点。)

**代码实现:**

1. int ans=1e9;
2. for(int k=1;k<=n;k++)*//枚举转移节点*
3. {
4. *//枚举小于k的点计算。*
5. for(int i=1;i<k;i++){
6. for(int j=i+1;j<k;j++){
7. ans=min(ans,f[i][j]+w[i][k]+w[k][j]);
8. }
9. }
10. for(int i=1;i<=n;i++)
11. {
12. for(int j=1;j<=n;j++)
13. {
14. f[i][j]=min(f[i][j],f[i][k]+f[k][j]);
15. }
16. }
17. }

2.已知一个有向图中任意两点之间是否有连边，要求判断任意两点是否连通：该问题即是求原图的传递闭包。我们只需要按照Floyd的过程，逐个加入点判断一下。只是此时的边的边权变为0/1，而取min操作变成了|(或)运算。再进一步用 bitset 优化，复杂度可以到O(n^3/**ω**)的级别

**代码实现：**

1. bitset<MAXN> f[MAXN];
2. for (k = 1; k <= n; k++)
3. for (i = 1; i <= n; i++)
4. if (f[i][k]) f[i] = f[i] | f[k];

3.求解恰好经过T条边的全源最短路

离散课中我们学习到了A^m中的节点A^m i,j即vi到vj有aij^m条长度为m的路。我们同样运用这个思想计算恰好经过k条边的最短路。

我们记a[i][j]为恰好经过x条路的两两最短路,b[i][j]为恰好经过y条路的两两最短路。

如果c[i][j]=min(c[i][j],a[i][k]+b[k][j]),c[i][j]中储存的就是经过中转点k的恰好经过x+y次的最短路。

在最开始我们拥有原图矩阵w[i][j]他代表恰好经过1次的最短路，而W^T即为最终答案。

如果T较大我们可以通过快速幂进行计算。时间复杂度为O(log(T)N^3)

至于如果只询问S到其他点的最短路，而非是全源最短路，开始的矩阵可以只需要压缩成一行即只包含第S行的矩阵。

**代码实现：(使用c++实现)**

1. int main()
2. {
3. int n, m;
4. cin >> n >> m;
5. vector<vector<vector<int>>> g(31, vector<vector<int>>(n + 1, vector<int>(n + 1, INF)));
6. *// vector<vector<vector<int>>> g(31, vector<vector<int>>(n + 1, vector<int>(n + 1, INF)));*
7. for (int i = 1; i <= m; i++)
8. {
9. int u, v, w;
10. cin >> u >> v >> w;
11. g[0][u][v] = w;
12. }
13. auto floyd = [=](vector<vector<int>> a, vector<vector<int>> b)
14. {
15. vector<vector<int>> dis(n + 1, vector<int>(n + 1, INF));
16. rep(k, 1, n) rep(i, 1, n) rep(j, 1, n) dis[i][j] = min(dis[i][j], a[i][k] + b[k][j]);
17. return dis;
18. };
19. for (int i = 1; i <= 30; i++)
20. g[i] = floyd(g[i - 1], g[i - 1]);
21. auto qpow = [&](int s, int k)
22. {
23. vector<int> ans(n + 1, INF);
24. ans[s] = 0;
25. int cnt = 0;
26. while (k)
27. {
28. if (k & 1)
29. {
30. vector<int> res(n + 1, INF);
31. for (int i = 1; i <= n; i++)
32. {
33. for (int j = 1; j <= n; j++)
34. {
35. res[j] = min(res[j], ans[i] + g[cnt][i][j]);
36. }
37. }
38. ans = res;
39. }
40. cnt++;
41. k >>= 1;
42. }
43. return ans;
44. };
45. int \_;
46. cin >> \_;
47. while (\_--)
48. {
49. int s, k;
50. cin >> s >> k;
51. vector<int> ans = qpow(s, k);
52. for (int i = 1; i <= n; i++)
53. {
54. if (ans[i] == INF)
55. cout << "-1 ";
56. else
57. cout << ans[i] << " ";
58. }
59. cout << "\n";
60. }
61. return 0;

## 7.代码实现

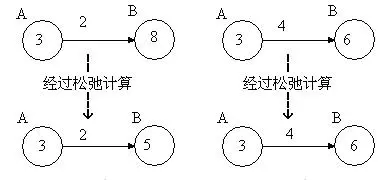
# 三、BeIlman-Ford（贝尔曼-福特）算法与SPFA（Shortest Path Faster Algorithm）算法

## 1.算法简介

Bellman-Ford算法的基本思想是将所有节点看作一个整体，进行n-1轮松弛操作，其中n为节点数。在每一轮操作中，对每一条边进行松弛操作，即检查是否可以通过当前路径可以获得更优的路径。如果在第n-1轮操作后仍然存在距离可以不断缩小的节点，那么说明存在负环。负环是指经过一条环路，可以一直减少路径长度，导致算法无法求出最短路径。

SPFA（Shortest Path Faster Algorithm）算法在Bellman-Ford算法的基础上进行了优化。它维护了一个队列，每次只对节点的邻接节点进行松弛操作，而不是像Bellman-Ford那样对所有边进行松弛操作。同时，如果某个节点的距离值在之前的操作中已经被修改过，那么把它加入队列就可以了，不需要每次都重新加入队列。这样可以大幅降低时间复杂度。

虽然SPFA算法在某些情况下有很好的表现，但是它也有一些问题。它可能出现负环的问题，而且在某些情况下其时间复杂度可能会退化到O(n^2)，甚至更高，因而在实际应用中需要特别注意。

2.松弛操作**——**不断更新最短路径和前驱结点的操作。

如上图所示，松弛计算之前，点B的值是8，但是点A的值加上边上的权重2，得到5，比点B的值（8）小，所以，点B的值减小为5。这个过程的意义是，找到了一条通向B点更短的路线，且该路线是先经过点A，然后通过权重为2的边，到达点B。

当然，如果出现右边这种情况，则不会修改点B的值，因为3＋4>6。

对边集合 E 中任意边，以 w(u,v) 表示顶点 u 出发到顶点 v 的边的权值，以 d[v] 表示当前从起点 s 到顶点 v 的路径权值

若存在边 w(u,v)，使得：d[v]> d[u]+w(u,v)，则更新 d[v] 值：d[v]=d[u]+w(u,v)。所以松弛函数的作用，就是判断是否经过某个顶点，或者说经过某条边，可以缩短起点到终点的路径权值。

为什么将缩短距离的操作称之为“松弛”，不妨理解为，选择某种方式后，到达目的的总代价降低了。什么名字无关紧要，不必纠结。

## 3.适用情况

（1）Bellman-Ford算法适用于一般图，包括带有负权边的图。

Bellman-Ford算法的基本思想是对所有边进行 V-1 次松弛操作，其中 V 表示图中的顶点数。如果在第 V-1 次松弛操作后仍然存在长度更短的路径，说明图中存在负权环，因为对于任意一个环，经过它的长度可以不断减小，直到负无穷。因此，Bellman-Ford算法可以应用于带有负权边的图，但由于需要 V-1 次松弛操作，因此时间复杂度为 O(VE)。

（2）SPFA算法适用于稠密图或者存在正权边的图。

SPFA算法（Shortest Path Faster Algorithm）的基本思想是利用队列进行顶点松弛操作，即从源点扩散更新到其他顶点时，如果当前顶点的最短路径估计值发生改变，则将其入队，直到队列为空。SPFA算法的优点在于它对于一般的图可以处理的比较快，不需要像BellmanFord算法那样对所有边进行松弛操作，但是由于它使用的是队列，当图的分支比较多时，队列的增长速度过快，可能会降低算法效率。因此，SPFA算法适用于稠密图或者存在正权边的图。

## 4.各算法优缺点

**（1）贝尔曼-福德算法优点：**

Ⅰ.可以处理负权边的图，而Dijkstra算法却不行。【3】

Ⅱ.在有些情况下比Dijkstra算法更快，如边权值随机分布的图。

**（2）贝尔曼-福德算法的缺点**：

Ⅰ.最坏情况下的时间复杂度为O(VE)，其中V是顶点数，E是边数，比Dijkstra算法慢。

Ⅱ.可能会产生负环，导致无法求解最短路径。

**（3）SPFA算法的优点：**

Ⅰ.在处理稠密图（边数接近于V^2）时比Dijkstra算法更快。

Ⅱ.可以处理负权边的图。

Ⅲ.可以在任意一种图上运行，不同于Dijkstra算法要求非负权边。

**（4）SPFA算法的缺点：**

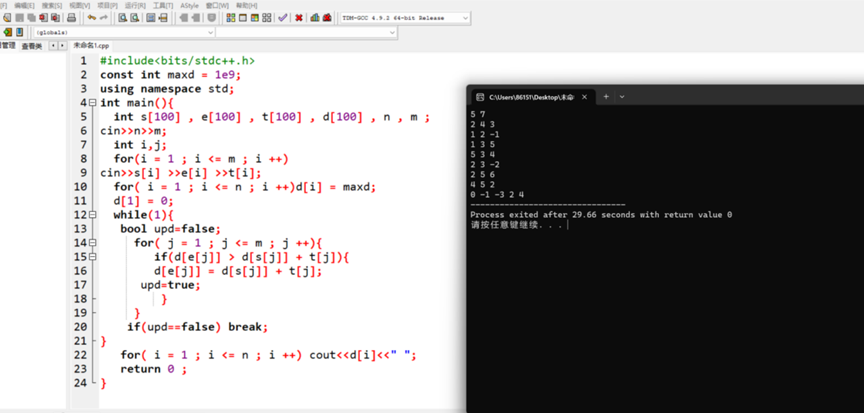
Ⅰ.在处理稀疏图时，由于其内部的队列操作，速度比Dijkstra算法慢。

Ⅱ.在最坏情况下，可能会产生负环，导致无法求解最短路径。

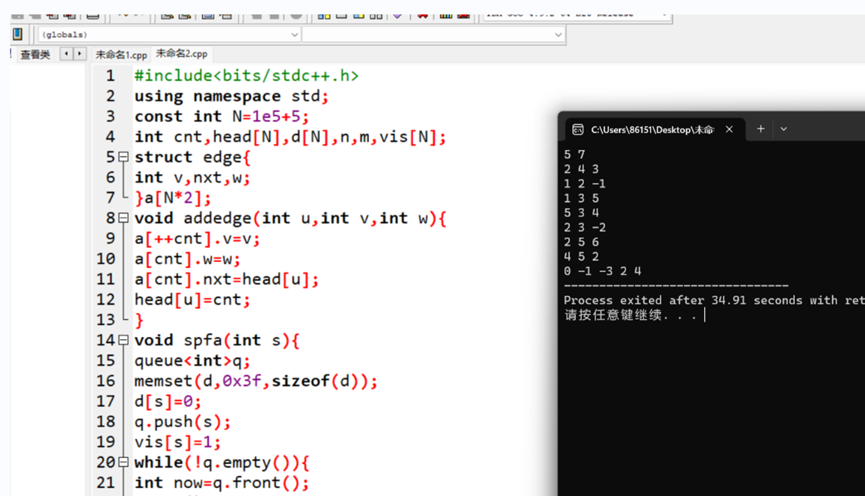
## 5.两种算法的联系和区别

Bellman-Ford 算法时间复杂度为 O (nm)，最多需要执行 n1 次循环，如果执行了n次循环，说明图中含有负圈。Spfa算法是为了改进Bellman-Ford算法的效率而提出来的，在最优的情况下，每个节点只入队一次，这时退化为广度优先搜索，在最坏情况下，每个节点都入队 n-1 次，此时 Spfa 算法退化为 Bell⁃man-Ford 算法，时间复杂度为 O(nm)。如果某个节点的入队次数超过 n 次，说明图中肯定含有负圈。由于这两种算法均采用邻接表的方式存储边的信息，因此他们的空间复杂度均为 O(m)。它们都可以解决负权图问题，并能够判断图中是否含有负圈，都适用于稀疏图。由以上分析可以看出，Bellman-Ford 算法可解决的问题Spfa 算法也都适用。因此，奥赛解题中更常用 Spfa 算法。但是，Bellman-Ford算法思想精妙，是Spfa算法的前置知识，在学习中先理解 Bellman-Ford 算法更有利于对 Spfa 算法的理解和应用。【4】

## 6.代码运行情况

Ⅰ. 贝尔曼-福德算法

Ⅱ. SPFA算法



# 四、A\*算法

## 1.算法简介

A\*算法是一种启发式搜索算法，用于在图中找到两个节点之间的最短路径。该算法使用一个评估函数，结合边的权重和从起点到当前节点的距离，来计算该节点到终点的估计距离，然后优先访问估计距离小的节点。这种优先级队列的排序方式可以保证优先考虑最优解路径，减少搜索次数，提高算法效率。【7】

## 2.适用情况

（1）问题可以表示为图形或网格。

（2）可以定义起点和终点。

（3）可以定义节点之间的代价或距离。

（4）可以定义一个估价函数，该函数可以评估从一个节点到终点的最小代价或距离。

## 3.优点

（1）最优性：A\*算法保证找到的路径是最短路径，即使在复杂的地图或网络中也能找到最优解。

（2）完备性：如果存在路径，A\*算法能够找到一条路径。

（3）启发式：A\*算法使用启发式函数来评估节点的价值，这样可以尽可能减少搜索的节点数，提高搜索效率。

（4）可扩展性：A\*算法可以轻松地扩展到多种问题和应用中，例如机器人导航、游戏开发、路径规划等。

（5）可定制性：A\*算法可以根据问题的特定需求进行修改和定制，例如更改估价函数或添加限制条件。

## 4.缺点

（1）启发式函数的设计需要一定的领域知识和经验，如果设计不好，可能会导致搜索效率低下。

（2）如果启发式函数不是严格单调递增的，A\*算法可能会在搜索过程中反复遍历相同的节点，从而降低搜索效率。

（3）对于具有高度分支因子的问题，A\*算法可能会导致搜索空间过大，导致计算时间和内存需求增加。

（4）对于一些特殊情况，例如存在大量的障碍物或者没有可行解的情况，A\*算法可能会陷入死循环或无限搜索。

## 5.代码运行情况

# 五、Dijkstra算法

## 1.算法简介

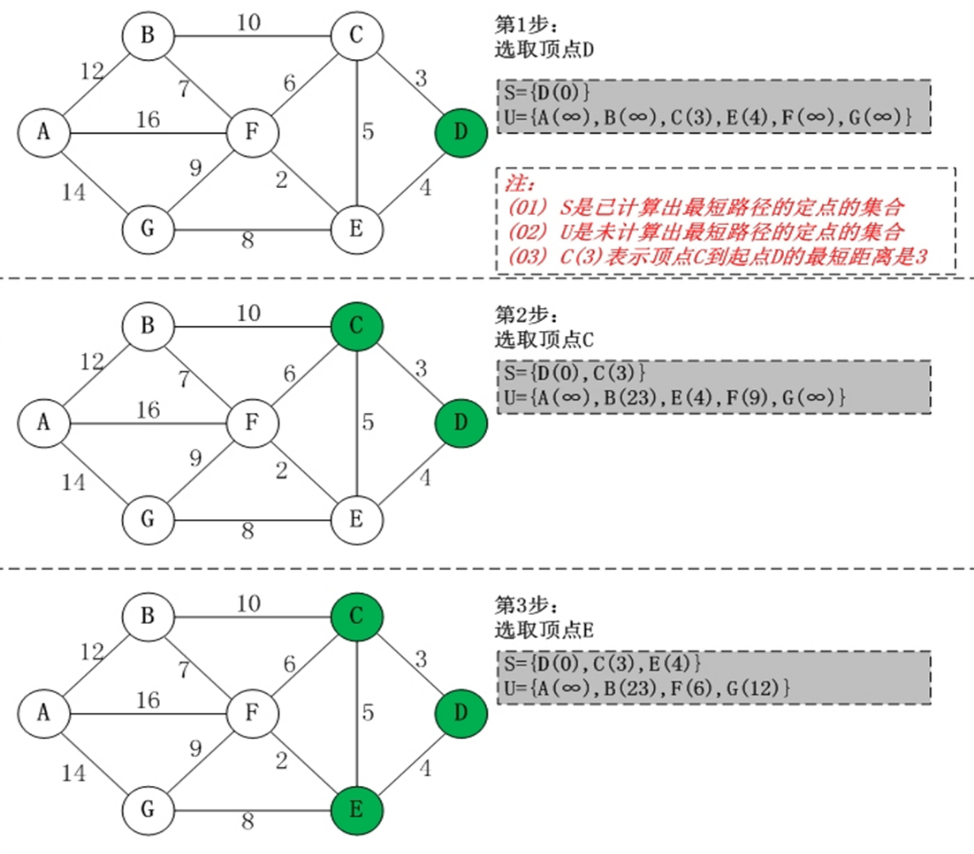
Dijkstra算法是一种用于解决带权重有向图或无向图的单源最短路径问题的贪心算法。该算法的基本思想是从源节点开始，依次对各个节点进行遍历，计算源节点到每个节点的最短路径，最终得到源节点到所有节点的最短路径。

## 2.适用情况

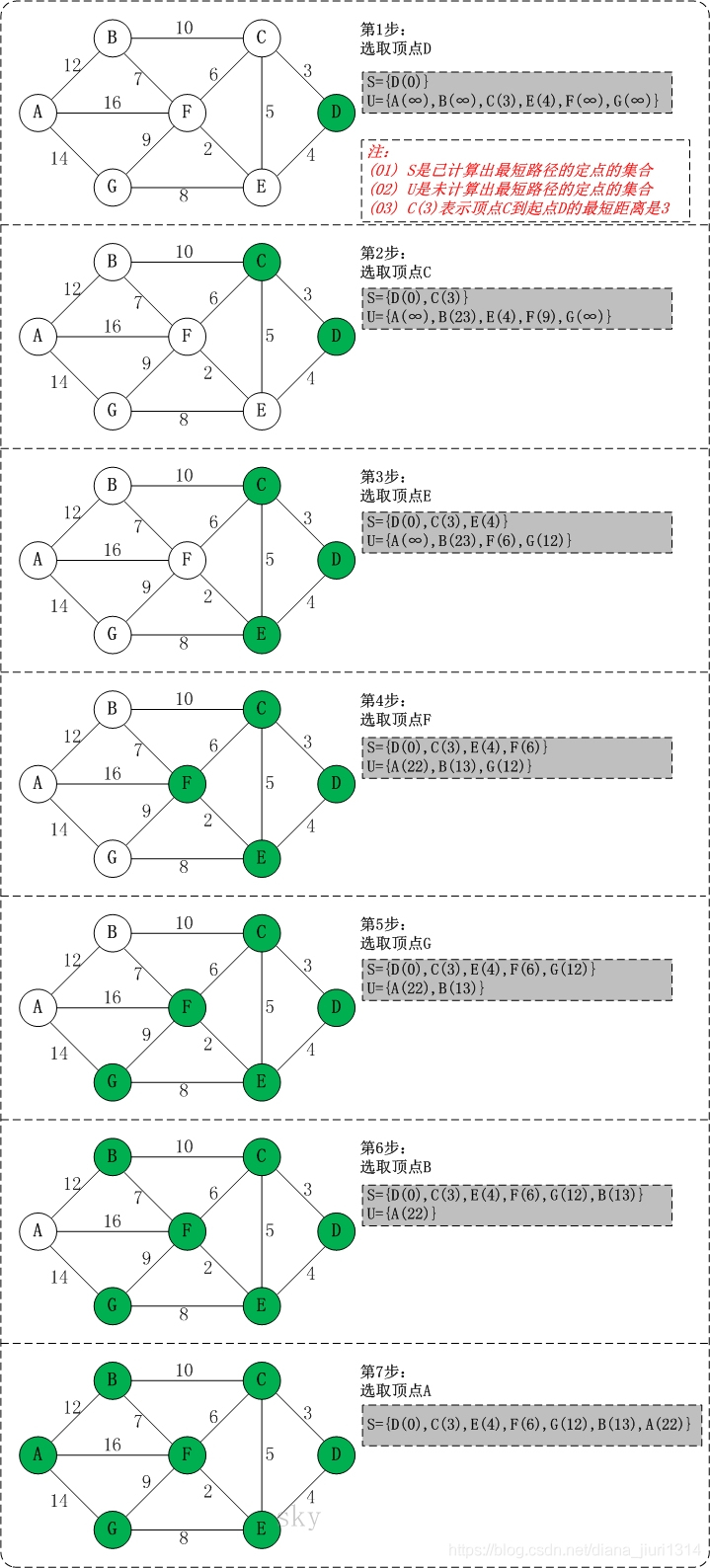
（1）Dijkstra算法适用于求解带权重的有向图或无向图中，单源最短路径问题，并且图中所有边的权重为非负数。

（2）适用于求解单源最短路径问题，即给定一个起点，求它到图中所有其他节点的最短路径。

（3）适用于稠密图和稀疏图，但是在稠密图中，时间复杂度会较高。

（4）当边权值不同，而且权值关系比较复杂时，可以使用Dijkstra算法。

## 3.实现步骤



## 4.优点

（1）算法保证能够找到最短路径。对于带权重的有向图或无向图中，求解单源最短路径的问题，Dijkstra算法能够保证找到最短路径。

（2）算法适用范围广。Dijkstra算法适用于带权重的有向图或无向图中，求解单源最短路径的问题，并且图中所有边的权重为非负数。

（3）算法时间复杂度较低。通过使用堆优化，Dijkstra算法的时间复杂度可以降低到O(E + VlogV)，其中E是边数，V是节点数，使得Dijkstra算法可以高效地解决大规模的图论问题。

（4）算法实现简单。Dijkstra算法的实现相对简单，容易理解和实现，因此被广泛应用于实际问题中。

## 5.缺点

（1）不能处理带有负权边的图：Dijkstra算法要求所有边的权值都为正数，否则可能会出现错误的结果。如果存在负权边，则可能会导致算法无法找到正确的最短路径，或者陷入死循环。

（2）空间复杂度高：Dijkstra算法使用了一个距离数组来存储源节点到其他节点的最短距离，如果节点数很大，这个数组的空间复杂度会很高，导致算法运行缓慢。

（3）时间复杂度高：在稠密图中，Dijkstra算法的时间复杂度为O(N^2)，其中N是节点数，这可能会导致算法运行缓慢。在稀疏图中，使用堆优化可以将时间复杂度降至O(MlogN)，其中M是边数，但是在实际应用中，堆操作的常数因子可能会很大，导致算法的实际运行时间不尽如人意。

（4）不能处理含有环的图：Dijkstra算法不能处理含有环的图，否则可能会导致算法无法找到最短路径或者陷入死循环。如果需要处理含有环的图，可以使用Bellman-Ford算法或者SPFA算法。【8】

## 6.Dijkstra算法和A\*算法的比较

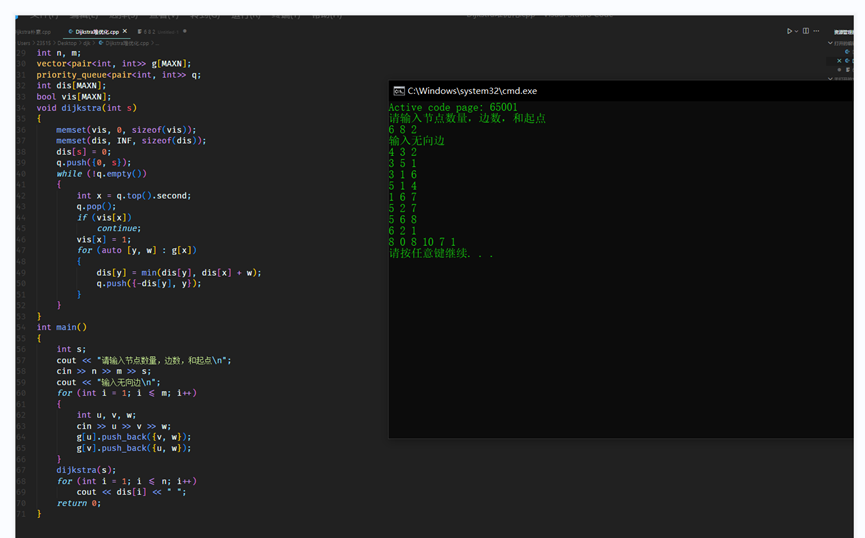
(1)Dijkstra算法计算源点到其他所有点的最短路径长度，A\*关注点到点的最短路径(包括具体路径)。

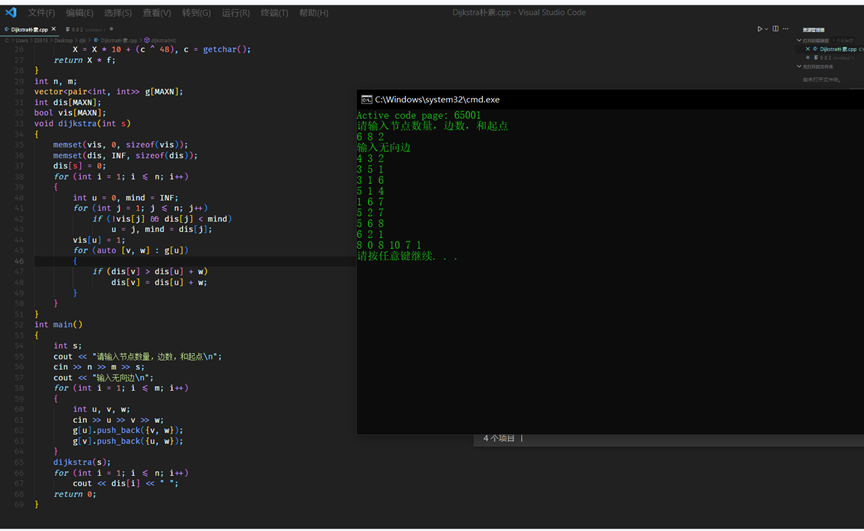
(2)Dijkstra算法建立在较为抽象的图论层面，A\*算法可以更轻松地用在诸如游戏地图寻路中。

(3)Dijkstra算法的实质是广度优先搜索，是一种发散式的搜索，所以空间复杂度和时间复杂度都比较高。对路径上的当前点，A\*算法不但记录其到源点的代价，还计算当前点到目标点的期望代价，是一种启发式算法，也可以认为是一种深度优先的算法。

(4)由第一点，当目标点很多时，A\*算法会带入大量重复数据和复杂的估价函数，所以如果不要求获得具体路径而只比较路径长度时，Dijkstra算法会成为更好的选择。【6】

## 6.代码运行情况

Ⅰ.**堆优化**

Ⅱ. **朴素**

# 六、Johnson算法

## 1.算法简介

Johnson算法则是一种用于解决带负权边的最短路径问题的算法。该算法首先对原图进行变换，使其不存在负权边，然后再利用Dijkstra算法求解每个源点到所有其他节点的最短路径。最后再根据变换前的结果得出最终结果。

## 2.适用情况

（1）带有负权边的稀疏图。在这种情况下，Dijkstra算法不能直接用于求解最短路径，因为它不能处理负权边。而Johnson算法通过将原图转换为一个不带负权边的图，再使用Dijkstra算法求解最短路径，从而解决了这个问题。

（2）需要求解稀疏图中所有节点对之间的最短路径。在这种情况下，Floyd算法的时间复杂度为O(V^3)，而Johnson算法的时间复杂度为O(VE + V^2logV)，在稀疏图中效率更高。

## 3.优点

（1）解决了带有负权边的稀疏图的最短路径问题。Johnson算法通过将原图转换为一个不带负权边的图，再使用Dijkstra算法求解最短路径，从而解决了这个问题。

（2）可以在稀疏图中高效地求解所有节点对之间的最短路径。在稀疏图中，Floyd算法的时间复杂度为O(V^3)，而Johnson算法的时间复杂度为O(VE + V^2logV)，在稀疏图中效率更高。

（3）算法的实现相对简单，只需要对原图进行一次边权重的调整和一次Dijkstra算法的执行。

（4）在实际应用中，由于负权边的存在，很多问题不能使用传统的最短路径算法求解，而Johnson算法提供了一种可行的解决方案。

## 4.缺点

（1）需要对原图进行一次边权重的调整，这个调整过程需要遍历整个图，因此在稠密图中，时间复杂度会比较高。

（2）算法的实现相对复杂，需要使用堆等数据结构来实现Dijkstra算法，因此需要一定的编程技巧。

（3）算法的空间复杂度比较高，需要额外的空间来存储转换后的图和每个节点的新权重值，因此在处理大规模图时，需要考虑内存的限制。

（4）算法对负环的处理比较麻烦，需要使用贝尔曼-福德算法来判断是否存在负环，并且在存在负环的情况下，Johnson算法无法求解最短路径。

## 5.实现步骤

（1）初始化，把一个node q添加到图G中，使node q 到图G每一个点的权值为0。

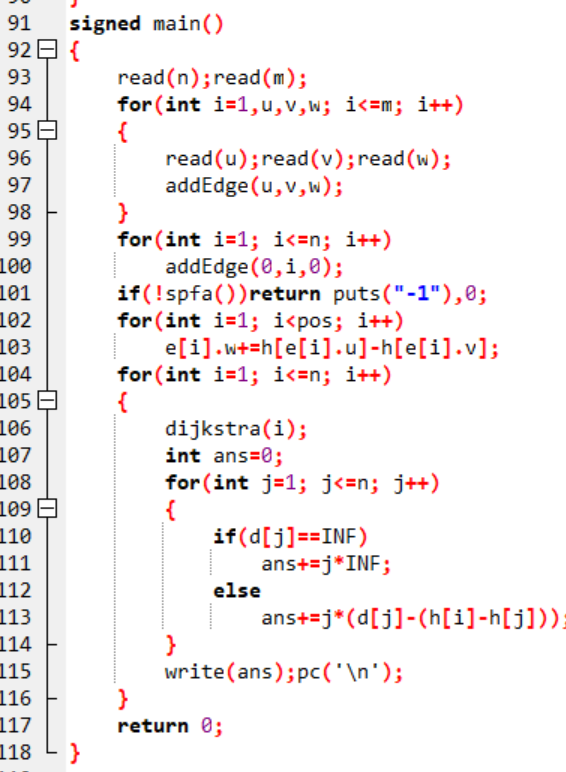
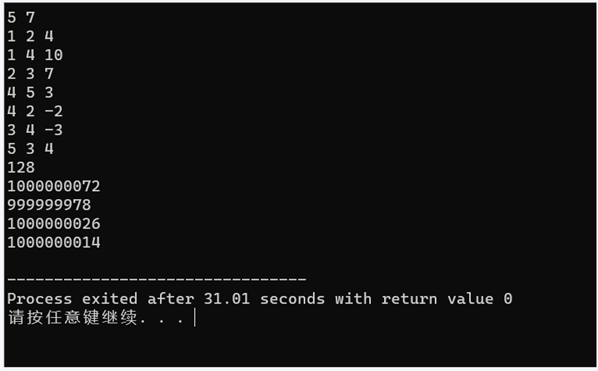
（2）使用Bellman-Ford算法，从源点为q，寻找每一个点 v从q到v的最短路径h(v)，如果存在负环的话，算法终止。

（3）使用第2步骤中Bellman-Ford计算的最短路径值对原来的图进行reweight操作（重赋值）：边<u,v>的权值w(u,v)，修改成w(u,v)+h(u)-h(v)。

（4）最后，移去q，针对新图（重赋值之后的图）使用Dijkstra算法计算从每一个点s到其余另外点的最短距离。

（5）使用dijstra计算完最短路后，需要d(i,j) = d'(i,j) + h(v) - h(u)才是最后结果【9】

## 6.代码实现情况



# 七、小组收获

在讨论最短路径算法的过程中，我们小组获得了解决最短路径问题的知识与团队间沟通协作的技能。首先，我们对Floyd算法的理解更加深入，了解了它在解决有权图中任意两点之间最短路径问题时的优点和限制。其次，我们学习了解决带有负权边的图中最短路径问题的Bellman-Ford算法和SPFA算法，对它们的原理和应用有了更清晰的认识。我们还探讨了启发式搜索算法A\*算法的特点和优势，了解了如何使用评估函数和优先级队列来提高搜索效率。此外，我们认识到Dijkstra算法和Johnson算法在不同场景下的适用性，了解了如何根据具体问题选择最适合的算法。这些讨论使我们对最短路径算法有了更全面的了解，为解决实际问题提供了宝贵的思路和方法。通过团队合作和交流，我们共同扩展了知识面，提高了问题解决能力，并培养了合作和沟通技巧。这次小组讨论的收获将对我们今后的学习和工作有所帮助。

# 八、参考文献

【1】[Floyd–Warshall algorithm - Wikipedia](https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm) Floyd–Warshall algorithm 2022.11.4

【2】<https://blog.csdn.net/weixin_45724872/article/details/122811602> Floyd算法思路以及扩展应用，荼白777，2022.02.10

【3】侃侃阳光，Bellman-ford算法，2022.10.14

【4】周鑫,张晶.实例解析Bellman-ford和Spfa算法[J].电脑知识与技术,2021,17(30):79-81.

【5】<https://blog.csdn.net/m0_463043 83/article/details/113457800>? spm=1001.2 014 .3001.5501 ，A91A981E ，A\*算法(A-star Algorithm)搜索最短路径，2022.07.17

【6】https://blog.csdn.net/dujuancao11/article/details/109749219，Clark-dj ，A\*算法，2022.09.20

【7】<https://zhuanlan.zhihu.com/p/51099376>，搬砖的旺财，路径规划——A\*算法，2018.11.28

【8】https://www.zhihu.com/question/591705246/answer/2952776552柒龍猪,在实际应用中，Dijkstra 算法有哪些局限性？为什么？,2023.03.25

【9】\_zhj ，johnson最短路径，2017-06-30 [https://blog.csdn.net/zhj\_fly/article/details/74009168?ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522168768785316800188579693%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fblog.%2522%257D&request\_id=168768785316800188579693&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~blog~first\_rank\_ecpm\_v1~rank\_v31\_ecpm-1-74009168-null-null.268^v1^koosearch&utm\_term=%20johnson%E6%9C%80%E7%9F%AD%E8%B7%AF%E5%BE%84&spm=1018.2226.3001.4450](https://blog.csdn.net/zhj_fly/article/details/74009168?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522168768785316800188579693%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fblog.%2522%257D&request_id=168768785316800188579693&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~blog~first_rank_ecpm_v1~rank_v31_ecpm-1-74009168-null-null.268%5ev1%5ekoosearch&utm_term=%20johnson%E6%9C%80%E7%9F%AD%E8%B7%AF%E5%BE%84&spm=1018.2226.3001.4450)，

# 附录

|  |
| --- |
| 1. **Floyd算法** 2. #include <iostream> 3. #include <cstdio> 4. #include <cstring> 5. #include <cmath> 6. #include <algorithm> 7. #include <set> 8. #include <map> 9. #include <queue> 10. #include <vector> 11. #define rep(a, b, c) for (int a = b; a <= c; a++) 12. #define IOS cin.tie(0)->sync\_with\_stdio(false) 13. typedef long long ll; 14. typedef unsigned long long ull; 15. using namespace std; 16. const int MAXN = 1e3 + 5; 17. const int INF = 0x3f3f3f3f; 18. int read() 19. { 20. int X = 0, f = 1; 21. char c = getchar(); 22. while (!isdigit(c) && c != '-') 23. c = getchar(); 24. if (c == '-') 25. c = getchar(), f = -1; 26. while (isdigit(c)) 27. X = X \* 10 + (c ^ 48), c = getchar(); 28. return X \* f; 29. } 30. int f[MAXN][MAXN]; *// f[i][j]代表节点i到节点j的最短路* 31. int main() 32. { 33. int n, m; *//n为图中点的数量，m为图中边的数量* 34. cin >> n >> m; 35. for (int i = 1; i <= n; i++) 36. { 37. for (int j = 1; j <= n; j++) 38. { 39. f[i][j] = 1e9; *//初始每个节点对之间都是不可达，设为最大值* 40. } 41. } 42. for (int i = 1; i <= n; i++) 43. f[i][i] = 0; *// 自己到自己最短路为0* 44. for (int i = 0; i < m; i++) 45. { 46. int x, y, w; 47. cin >> x >> y >> w;    *// 分别是起点，终点和边权* 48. f[x][y] = f[y][x] = w; *// 如果图为无向图* 49. } 50. for (int k = 1; k <= n; k++) *// 枚举转移节点* 51. for (int i = 1; i <= n; i++) 52. for (int j = 1; j <= n; j++) 53. { 54. f[i][j] = min(f[i][j], f[i][k] + f[k][j]); 55. *// 枚举k作为i>j的转移节点计算* 56. } 57. for (int i = 1; i <= n; i++) 58. { 59. for (int j = 1; j <= n; j++) 60. { 61. printf("%d ", f[i][j]); 62. *// 输出i到j的最短路* 63. } 64. puts(""); 65. } 66. return 0; 67. } 68. **BeIlman**-**Ford（贝尔曼-福特）算法** 69. #include<bits/stdc++.h> 70. const int maxd = 1e9; 71. using namespace std; 72. int main(){ 73. int s[100] , e[100] , t[100] , d[100] , n , m ; 74. cin>>n>>m; 75. int i,j; 76. for(i = 1 ; i <= m ; i ++) 77. cin>>s[i] >>e[i] >>t[i]; 78. for( i = 1 ; i <= n ; i ++)d[i] = maxd; 79. d[1] = 0; 80. while(1){ 81. bool upd=false; 82. for( j = 1 ; j <= m ; j ++){ 83. if(d[e[j]] > d[s[j]] + t[j]){ 84. d[e[j]] = d[s[j]] + t[j]; 85. upd=true; 86. } 87. } 88. if(upd==false) break; 89. } 90. for( i = 1 ; i <= n ; i ++) cout<<d[i]<<" "; 91. return 0 ; 92. } 93. **SPFA算法** 94. #include<bits/stdc++.h> 95. using namespace std; 96. const int N=1e5+5; 97. int cnt,head[N],d[N],n,m,vis[N]; 98. struct edge{ 99. int v,nxt,w; 100. }a[N\*2]; 101. void addedge(int u,int v,int w){ 102. a[++cnt].v=v; 103. a[cnt].w=w; 104. a[cnt].nxt=head[u]; 105. head[u]=cnt; 106. } 107. void spfa(int s){ 108. queue<int>q; 109. memset(d,0x3f,sizeof(d)); 110. d[s]=0; 111. q.push(s); 112. vis[s]=1; 113. while(!q.empty()){ 114. int now=q.front(); 115. q.pop(); 116. vis[now]=0; 117. for(int i=head[now];i;i=a[i].nxt){ 118. int to=a[i].v; 119. if(a[i].w+d[now]<=d[to]){ 120. d[to]=a[i].w+d[now]; 121. if(!vis[to]){ 122. q.push(to); 123. vis[to]=1; 124. } 125. } 126. } 127. } 128. } 129. int main(){ 130. cin>>n>>m; 131. for(int i=1;i<=m;i++){ 132. int x,y,w; 133. cin>>x>>y>>w; 134. addedge(x,y,w); 135. } 136. spfa(1); 137. for(int i=1;i<=n;i++)cout<<d[i]<<" "; 138. return 0; 139. }   **四、A\*算法**  **（1）算法实现：**   1. *#include <iostream>* 3. *#include "Astar.h"* 5. bool InPath(const int &row, const int &col, const std::list<Point \*> &path) { 6. for (const auto &p : path) { 7. if (row == (p->x) && col == p->y) { 8. return true; 9. } 10. } 11. return false; 12. } 14. int main() { 15. //初始化地图，用二维矩阵代表地图，1表示障碍物，0表示可通 16. std::vector<std::vector<int>> map = {{1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}, 17. {1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1}, 18. {1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1}, 19. {1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1}, 20. {1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1}, 21. {1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1}, 22. {1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1}, 23. {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}}; 24. Astar astar; 25. astar.InitAstar(map); 27. //设置起始和结束点 28. Point start(1, 1); 29. Point end(6, 10); 31. // A\*算法找寻路径 32. std::list<Point \*> path = astar.GetPath(start, end, false); 34. // 打印结果 35. for (auto &p : path) { 36. std::cout << "(" << p->x << "," << p->y << ") "; 37. } 38. std::cout << "\n"; 40. for (int row = 0; row < map.size(); ++row) { 41. for (int col = 0; col < map[0].size(); ++col) { 42. if (InPath(row, col, path)) { 43. if (map[row][col] != 0) { 44. std::cout << "e "; 45. } else { 46. std::cout << "\* "; 47. } 48. } else { 49. std::cout << map[row][col] << " "; 50. } 51. } 52. std::cout << "\n"; 53. } 54. return 0; 55. }   **（2）main**   1. *#include <math.h>* 2. *#include "Astar.h"* 4. void Astar::InitAstar(std::vector<std::vector<int>> &\_maze) 5. { 6. maze = \_maze; 7. } 9. int Astar::calcG(Point \*temp\_start, Point \*point) 10. { 11. int extraG = (abs(point->x - temp\_start->x) + abs(point->y - temp\_start->y)) == 1 ? kCost1 : kCost2; 12. int parentG = point->parent == NULL ? 0 : point->parent->G; *//如果是初始节点，则其父节点是空* 13. return parentG + extraG; 14. } 16. int Astar::calcH(Point \*point, Point \*end) 17. { 18. *//用简单的欧几里得距离计算H，这个H的计算是关键，还有很多算法，没深入研究^\_^* 19. return sqrt((double)(end->x - point->x)\*(double)(end->x - point->x) + (double)(end->y - point->y)\*(double)(end->y - point->y))\*kCost1; 20. } 22. int Astar::calcF(Point \*point) 23. { 24. return point->G + point->H; 25. } 27. Point \*Astar::getLeastFpoint() 28. { 29. if (!openList.empty()) 30. { 31. auto resPoint = openList.front(); 32. for (auto &point : openList) 33. if (point->F<resPoint->F) 34. resPoint = point; 35. return resPoint; 36. } 37. return NULL; 38. } 40. Point \*Astar::findPath(Point &startPoint, Point &endPoint, bool isIgnoreCorner) 41. { 42. openList.push\_back(new Point(startPoint.x, startPoint.y)); *//置入起点,拷贝开辟一个节点，内外隔离* 43. while (!openList.empty()) 44. { 45. auto curPoint = getLeastFpoint(); *//找到F值最小的点* 46. openList.remove(curPoint); *//从开启列表中删除* 47. closeList.push\_back(curPoint); *//放到关闭列表* 48. *//1,找到当前周围八个格中可以通过的格子* 49. auto surroundPoints = getSurroundPoints(curPoint, isIgnoreCorner); 50. for (auto &target : surroundPoints) 51. { 52. *//2,对某一个格子，如果它不在开启列表中，加入到开启列表，设置当前格为其父节点，计算F G H* 53. if (!isInList(openList, target)) 54. { 55. target->parent = curPoint; 57. target->G = calcG(curPoint, target); 58. target->H = calcH(target, &endPoint); 59. target->F = calcF(target); 61. openList.push\_back(target); 62. } 63. *//3，对某一个格子，它在开启列表中，计算G值, 如果比原来的大, 就什么都不做, 否则设置它的父节点为当前点,并更新G和F* 64. else 65. { 66. int tempG = calcG(curPoint, target); 67. if (tempG<target->G) 68. { 69. target->parent = curPoint; 71. target->G = tempG; 72. target->F = calcF(target); 73. } 74. } 75. Point \*resPoint = isInList(openList, &endPoint); 76. if (resPoint) 77. return resPoint; *//返回列表里的节点指针，不要用原来传入的endpoint指针，因为发生了深拷贝* 78. } 79. } 81. return NULL; 82. } 84. std::list<Point \*> Astar::GetPath(Point &startPoint, Point &endPoint, bool isIgnoreCorner) 85. { 86. Point \*result = findPath(startPoint, endPoint, isIgnoreCorner); 87. std::list<Point \*> path; 88. *//返回路径，如果没找到路径，返回空链表* 89. while (result) 90. { 91. path.push\_front(result); 92. result = result->parent; 93. } 95. *// 清空临时开闭列表，防止重复执行GetPath导致结果异常* 96. openList.clear(); 97. closeList.clear(); 99. return path; 100. } 102. Point \*Astar::isInList(const std::list<Point \*> &list, const Point \*point) const 103. { 104. *//判断某个节点是否在列表中，这里不能比较指针，因为每次加入列表是新开辟的节点，只能比较坐标* 105. for (auto p : list) 106. if (p->x == point->x&&p->y == point->y) 107. return p; 108. return NULL; 109. } 111. bool Astar::isCanreach(const Point \*point, const Point \*target, bool isIgnoreCorner) const 112. { 113. if (target->x<0 || target->x>maze.size() - 1 114. || target->y<0 || target->y>maze[0].size() - 1 115. || maze[target->x][target->y] == 1 116. || target->x == point->x&&target->y == point->y 117. || isInList(closeList, target)) *//如果点与当前节点重合、超出地图、是障碍物、或者在关闭列表中，返回false* 118. return false; 119. else 120. { 121. if (abs(point->x - target->x) + abs(point->y - target->y) == 1) *//非斜角可以* 122. return true; 123. else 124. { 125. *//斜对角要判断是否绊住* 126. if (maze[point->x][target->y] == 0 && maze[target->x][point->y] == 0) 127. return true; 128. else 129. return isIgnoreCorner; 130. } 131. } 132. } 134. std::vector<Point \*> Astar::getSurroundPoints(const Point \*point, bool isIgnoreCorner) const 135. { 136. std::vector<Point \*> surroundPoints; 138. for (int x = point->x - 1; x <= point->x + 1; x++) 139. for (int y = point->y - 1; y <= point->y + 1; y++) 140. if (isCanreach(point, new Point(x, y), isIgnoreCorner)) 141. surroundPoints.push\_back(new Point(x, y)); 143. return surroundPoints; 144. }   **五、Dijkstra算法**  **（1）Dijkstra朴素**   1. #include <iostream> 2. #include <cstdio> 3. #include <cstring> 4. #include <cmath> 5. #include <algorithm> 6. #include <set> 7. #include <map> 8. #include <queue> 9. #include <vector> 10. #define rep(a, b, c) for (int a = b; a <= c; a++) 11. #define IOS cin.tie(0)->sync\_with\_stdio(false) 12. typedef long long ll; 13. typedef unsigned long long ull; 14. using namespace std; 15. const int MAXN = 2e5 + 5; 16. const int INF = 0x3f3f3f3f; 17. int read() 18. { 19. int X = 0, f = 1; 20. char c = getchar(); 21. while (!isdigit(c) && c != '-') 22. c = getchar(); 23. if (c == '-') 24. c = getchar(), f = -1; 25. while (isdigit(c)) 26. X = X \* 10 + (c ^ 48), c = getchar(); 27. return X \* f; 28. } 29. int n, m; 30. vector<pair<int, int>> g[MAXN]; 31. int dis[MAXN]; 32. bool vis[MAXN]; 33. void dijkstra(int s) 34. { 35. memset(vis, 0, sizeof(vis)); 36. memset(dis, INF, sizeof(dis)); 37. dis[s] = 0; 38. for (int i = 1; i <= n; i++) 39. { 40. int u = 0, mind = INF; 41. for (int j = 1; j <= n; j++) 42. if (!vis[j] && dis[j] < mind) 43. u = j, mind = dis[j]; 44. vis[u] = 1; 45. for (auto [v, w] : g[u]) 46. { 47. if (dis[v] > dis[u] + w) 48. dis[v] = dis[u] + w; 49. } 50. } 51. } 52. int main() 53. { 54. int s; 55. cout << "请输入节点数量，边数，和起点\n"; 56. cin >> n >> m >> s; 57. cout << "输入无向边\n"; 58. for (int i = 1; i <= m; i++) 59. { 60. int u, v, w; 61. cin >> u >> v >> w; 62. g[u].push\_back({v, w}); 63. g[v].push\_back({u, w}); 64. } 65. dijkstra(s); 66. for (int i = 1; i <= n; i++) 67. cout << dis[i] << " "; 68. return 0; 69. }   **（2）Dijkstra堆优化**   1. #include <iostream> 2. #include <cstdio> 3. #include <cstring> 4. #include <cmath> 5. #include <algorithm> 6. #include <set> 7. #include <map> 8. #include <queue> 9. #include <vector> 10. #define rep(a, b, c) for (int a = b; a <= c; a++) 11. #define IOS cin.tie(0)->sync\_with\_stdio(false) 12. typedef long long ll; 13. typedef unsigned long long ull; 14. using namespace std; 15. const int MAXN = 2e5 + 5; 16. const int INF = 0x3f3f3f3f; 17. int read() 18. { 19. int X = 0, f = 1; 20. char c = getchar(); 21. while (!isdigit(c) && c != '-') 22. c = getchar(); 23. if (c == '-') 24. c = getchar(), f = -1; 25. while (isdigit(c)) 26. X = X \* 10 + (c ^ 48), c = getchar(); 27. return X \* f; 28. } 29. int n, m; 30. vector<pair<int, int>> g[MAXN]; 31. priority\_queue<pair<int, int>> q; 32. int dis[MAXN]; 33. bool vis[MAXN]; 34. void dijkstra(int s) 35. { 36. memset(vis, 0, sizeof(vis)); 37. memset(dis, INF, sizeof(dis)); 38. dis[s] = 0; 39. q.push({0, s}); 40. while (!q.empty()) 41. { 42. int x = q.top().second; 43. q.pop(); 44. if (vis[x]) 45. continue; 46. vis[x] = 1; 47. for (auto [y, w] : g[x]) 48. { 49. dis[y] = min(dis[y], dis[x] + w); 50. q.push({-dis[y], y}); 51. } 52. } 53. } 54. int main() 55. { 56. int s; 57. cout << "请输入节点数量，边数，和起点\n"; 58. cin >> n >> m >> s; 59. cout << "输入无向边\n"; 60. for (int i = 1; i <= m; i++) 61. { 62. int u, v, w; 63. cin >> u >> v >> w; 64. g[u].push\_back({v, w}); 65. g[v].push\_back({u, w}); 66. } 67. dijkstra(s); 68. for (int i = 1; i <= n; i++) 69. cout << dis[i] << " "; 70. return 0; 71. }   **六、Johnson算法**   1. #include <bits/stdc++.h> 2. using namespace std; 3. #define int long long 4. #define gc() getchar() 5. #define pc(a) putchar(a) 6. #define INF 1000000000 7. #define MAXN (int)(3e3+15) 8. template<typename T>void read(T &k) 9. { 10. char ch=gc();T x=0,f=1; 11. while(!isdigit(ch)){if(ch=='-')f=-1;ch=gc();} 12. while(isdigit(ch)){x=(x<<1)+(x<<3)+(ch^48);ch=gc();} 13. k=x\*f; 14. } 15. template<typename T>void write(T k) 16. { 17. if(k<0){pc('-');k=-k;} 18. if(k>9)write(k/10); 19. pc(k%10+'0'); 20. } 21. struct Edge 22. { 23. int u,v,w,next; 24. }e[MAXN<<2]; 25. struct node 26. { 27. int u,dis; 28. bool operator<(const node &o)const 29. {return dis>o.dis;} 30. }; 31. int n,m,pos=1,head[MAXN]; 32. int h[MAXN],d[MAXN],vis[MAXN],cnt[MAXN],ans; 33. void addEdge(int u,int v,int w) 34. { 35. e[pos]={u,v,w,head[u]}; 36. head[u]=pos++; 37. } 38. bool spfa() 39. { 40. queue<int> q; 41. fill(h+1,h+1+n,INF); 42. q.push(0); 43. vis[0]=1;h[0]=0; 44. while(!q.empty()) 45. { 46. int u=q.front();q.pop(); 47. vis[u]=0; 48. for(int i=head[u]; i; i=e[i].next) 49. { 50. int v=e[i].v; 51. if(h[v]>h[u]+e[i].w) 52. { 53. h[v]=h[u]+e[i].w; 54. if(!vis[v]) 55. { 56. q.push(v); 57. vis[v]=1; 58. if(++cnt[v]>n) *// n+1结点* 59. return 0; 60. } 61. } 62. } 63. } 64. return 1; 65. } 66. void dijkstra(int st) 67. { 68. priority\_queue<node> q; 69. fill(d+1,d+1+n,INF); 70. memset(vis,0,sizeof(vis)); 71. q.push({st,0}); 72. d[st]=0; 73. while(!q.empty()) 74. { 75. int u=q.top().u;q.pop(); 76. if(vis[u]) 77. continue; 78. vis[u]=1; 79. for(int i=head[u]; i; i=e[i].next) 80. { 81. int v=e[i].v; 82. if(d[v]>d[u]+e[i].w) 83. { 84. d[v]=d[u]+e[i].w; 85. if(!vis[v]) 86. q.push({v,d[v]}); 87. } 88. } 89. } 90. } 91. signed main() 92. { 93. read(n);read(m); 94. for(int i=1,u,v,w; i<=m; i++) 95. { 96. read(u);read(v);read(w); 97. addEdge(u,v,w); 98. } 99. for(int i=1; i<=n; i++) 100. addEdge(0,i,0); 101. if(!spfa())return puts("-1"),0; 102. for(int i=1; i<pos; i++) 103. e[i].w+=h[e[i].u]-h[e[i].v]; 104. for(int i=1; i<=n; i++) 105. { 106. dijkstra(i); 107. int ans=0; 108. for(int j=1; j<=n; j++) 109. { 110. if(d[j]==INF) 111. ans+=j\*INF; 112. else 113. ans+=j\*(d[j]-(h[i]-h[j])); 114. } 115. write(ans);pc('\n'); 116. } 117. return 0; |