以最短路问题讨论其相关算法及优化

xny

（同济大学 软件工程学院，上海 200028）

摘要： 最短路问题是网络理论解决的典型问题之一，不仅在理论上有较为重要的分析意义，在现实生活中也具有较高的实用价值，可以解决许多实际问题。本文通过对最短路问题的研究和分析，根据不同的思路推导出不同算法的实现方式，分析这些算法的特点和所运用的场景，并且对这其中部分算法的优化算法进行分析和研究，最终得出比较各个方法的适用范围，得出各个算法所适用的场景，帮助读者更深刻地了解最短路问题的处理方式。

关键词： 最短路问题；单源最短路径；多源最短路径；最短路算法

Discussing The Algorithm and Optimization of the Shortest Path Problem

Ny X

(Tongji University Software Engineering Institute, Shanghai 200028)

Abstract: The shortest path problem is one of the typical problems solved by network theory. It not only has important theoretical analysis significance, but also has high practical value in real life. It can solve many practical problems. In this paper, through the research and analysis of the shortest path problem, according to different ideas to deduce the implementation of different algorithms, analyze the characteristics of these algorithms and the scenarios used, and analyze and study the optimization algorithms of some of these algorithms, finally get the applicable scope of each method, get the applicable scenarios of each algorithm, and help readers understand more deeply How to deal with the shortest path problem.

Key words: Shortest path problem; single source shortest path; multi-source shortest path; shortest path algorithm

1 引言

最短路问题（short-path problem）网络理论解决的典型问题之一，题目一般描述如下：若网络中的每条边都有一个可供比较的数值，我们从中找出两结点之间总权值和最小的路径，这就是最短路问题。最短路问题也有其现实意义，比如运用解决最短路问题的思路进行 GPS 导航，获得能够到达目的地的最短路径；或者运用最短路的方法对管道铺设进行计算，以达到最低的成本；又或者采用最短路的思路进行设备更新等。

2 相关概念介绍

2.1 路径

路径，又称简单路径。指对于一条途径，同一条边不出现两次，并且除了起始点和终止点允许相同之外，其余途径上各点两两互不相同，若一条途径满足上述要求，我们则称其为路径。

2.2 最短路

最短路即为图上两点之间的最短距离。根据图的不同可分为有向图最短路及无向图最短路，有向图中的边是单向的，无向图中的边为双向的，虽然边的方向有所区分，但在求最短路时所用的方法相同。

2.3 单源最短路径及多源最短路径

求单源最短路径是指从某固定源点出发，求该结点到其他所有顶点的最短路径。而多源最短路径是指求任意两个顶点之间的最短路径。

3 相关算法介绍

3.1 Floyd 算法

Floyd 算法可以计算多源最短路径问题，适用于任何图，包括有向图、无向图、图中边权为正或边权为负，但图中最短路必须存在，即图中不能存在负环。

首先我们对存储图信息的数据结构进行选择，假设现在存在由n个顶点构成的图，我们选择用一个$n*n$的二维数组dis来存储当前结点之间的距离，即当前结点之间的边

权值。初始时$dis[i][j]$中所存储的值为由点i到点j直接相连的边的长度，当$i=j$时，即该点到自身的距离，我们约定其为0，当点i与点j之间没有直接相连的边时，我们约定两点之间的路径长度为∞。

通过以往经验我们可以得知，如果要让任意两点a和b之间的路程变短，只能引入第三个点k，并通过该顶点k进行中转，即将原本的路径$a \to b$转化为路径$a \to k \to b$，若$a \to k$的距离加$k \to b$的距离小于$a \to b$的距离，则该路径可以缩短点a与点b之间的距离。我们想要得知的即是当前所需的中转点k点，或需要得到最短路径所经过的多个中转点$k1, k2 \ldots\ldots kn$。

接下来我们探讨如何通过当前思路解决最短路径问题。当任意两个点之间不允许经过第三个点时，这些点之间的最短路程就是初始路程，假如现在只允许经过0号顶点，求任意两点之间的最短路程，我们只需要判断$dis[i][0]+dis[0][j]$与$dis[i][j]$之间的大小，$dis[i][0]+dis[0][j]$表示从i号顶点先到0号顶点，再从0号顶点到j号顶点的路程之和，$dis[i][j]$表示当前情况下从i号顶点到j号顶点之间的路程，代码实现如下。

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    if (dis[i][j] > dis[i][0] + dis[0][j])
      dis[i][j] = dis[i][0] + dis[0][j];
  }
}
```

图1 以0号顶点为中转点的路程表示

接下来求只允许经过0号和1号两个顶点情况下任意两点之间的最短路程，我们只需要在只允许经过0号顶点时任意两点的最短路程的结果下判断如果经过1号顶点是否可以使得i号顶点到j号顶点的路程变得更短，即判断$dis[i][1]+dis[1][j]$是否比$dis[i][j]$之间的距离短，其代码实现如下。

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    if (dis[i][j] > dis[i][0] + dis[0][j])
      dis[i][j] = dis[i][0] + dis[0][j];
  }
}
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    if (dis[i][j] > dis[i][1] + dis[1][j])
      dis[i][j] = dis[i][1] + dis[1][j];
  }
}
```

图2 允许经过0号和1号中转点的任意两点间最短路程

以此类推，任意两点之间的最终路径即为允许所有顶点作为中转点，经过判断所得到的最短路径，其代码实现如下。

```
for(int k=0;k<n;k++)
    for(int i=0;i<n;i++)
        for(int j=0;j<n;j++)
            if(dis[i][j]>dis[i][k]+dis[k][j])
                dis[i][j]=dis[i][k]+dis[k][j];
```

图3 Floyd算法核心代码

我们可以从代码实现推断出Floyd算法的时间复杂度为$O(N^3)$，空间复杂度为$O(N^2)$。

3.2 Bellman-Ford 算法

Bellman-Ford算法是一种基于松弛（relax）操作的最短路算法，是一种用于计算带权有向图中单源最短路径的算法。其中，无向图可以看作有边两点之间存在相互到达边的有向图。

Bellman-Ford算法和Dijkstra算法同为解决单源最短路径的算法，但相比于Dijkstra，Bellman-Ford算法采用动态规划（Dynamic Programming）思路进行设计，可以计算含有负权的带权有向图中最短路，或者报告某些最短路不存在。

假设最初的结点，也就是固定源点为S，创建源顶点到图中所有顶点的距离的集合为dis，为图中所有的顶点指定一个距离值，初始化均为INF（Infinite，即最大值）。为计算最短路径，需要对图中每条边进行松弛（relax）操作。定义松弛操作$relax(u,v)$为

$dis(v) = \min(dis(v), dis(u) + w(u,v))$，即如果起点$u$的距离$dis(u)$加上边的权值$w$小于终点$v$的距离$dis(v)$，则更新终点$v$的距离值$dis(v)$。可用三角形不等式来证明$relax$操作的正确性：$dis(v) \le dis(u) + w(u,v)$。

根据上述描述，可得出 Bellman-Ford 算法伪代码如下：

```
while (1) for each edge(u, v) relax(u, v);
```

图 4 Bellman-Ford 算法伪代码

当循环中没有松弛操作成功时停止。每次循环是$O(m)$的，m为边的数量，当存在一个S能走到的负环时，会循环∞次，但此时某些结点的最短路不存在。当仅考虑最短路存在的时候，由于一次松弛操作会使最短路的边数至少$+1$，而最短路的边数最多为$n-1$条，所以在寻找到最短路时，最多只需要执行$n-1$次松弛操作，即最多循环$n-1$次，其代码实现如下。

```
for (int i = 0; i < n; i++) dis[i] = INF;
dis[begin] = 0;
for (int k = 0; k < n - 1; k++) {
  for (int i = 0; i < edge.size(); i++) {
    int x = edge[i].u, y = edge[i].v;
    if (dis[x] < INF)
      dis[y] = min(dis[y], dis[x] + edge[i].w);
  }
}
```

图 5 Bellman-Ford 算法核心代码

Bellman-Ford 算法的时间复杂度为$O(MN)$，其中M为顶点数量，N为边的数量，空间复杂度根据实现方式不同而有所区别。

3.3 SPFA 算法

SPFA（Shortest Path Faster Algorithm）算法时求单源最短路径的算法，它是 Bellman-Ford 算法的队列优化，是一种非常高效的最短路算法。

上文提到的 Bellman-Ford 算法对每一条边都进行了松弛操作，但很多时候并不需要那么多无用的松弛操作，显然，只有上一次被松弛的结点所连接的边，才有可能引起下一次的松弛操作。根据这个思路，可以采用队列的方式维护“哪些结点可能会引起松弛操作”，来仅做必要边的方问。

首先建立一个队列，初始时队列里只有起始点，再建立一个表格记录起始点到所有点的最短路径，与 Bellman-Ford 算法一致，该表格的初始值赋为极大值，该点到它本身的路径权值为 0。初始化完成后进行松弛操作，用队列里存在的点刷新起始点到所有点的最短路，如果刷新成功且被刷新点不在队列中，则把该点加入队列末尾，按上述操作重复执行直到队列为空，假如一个点入队次数超过N，则图中含有负权环，不存在最短路。

SPFA 算法伪代码如下。

```
q = new queue();
q.push(S);
in_queue[S] = true;
while (!q.empty()) {
  u = q.pop();
  in_queue[u] = false;
  for each edge(u, v) {
    if (relax(u, v) && !in_queue[v]) {
      q.push(v);
      in_queue[v] = true;
    }
  }
}
```

图 6 SPFA 算法伪代码

SPFA 算法核心代码如下。

```
while (!Q.empty()) {
  t = Q.front();
  Q.pop();
  visited[t] = false;
  for (i = p[t]; i; i = e[i].next) {
    temp = e[i].to;
    if (Dis[temp] > Dis[t] + e[i].w) {
      Dis[temp] = Dis[t] + e[i].w;
      if (!visited[temp]) {
        Q.push(temp);
        visited[temp] = true;
        if (++In[temp] > n) return false;
      }
    }
  }
}
```

图七 SPFA 核心代码

SPFA 的复杂度大约是$O(kM)$其中，k是每个点的平均进队次数，M为边的数量。一般而言，k是一个常数，在稀疏图中小于 2。但是，SPFA 算法的稳定性较差，在稠密图中 SPFA 算法的时间复杂度会退化，其最坏情况下的时间复杂度为$O(MN)$。

除了队列优化（SPFA）以外，Bellman-Ford 还有其他形式的优化，这些优化在部分图上效果明显，但在某些特殊图上最坏的复杂度可能达到指数级。其他形式的优化包括堆优化、栈优化、LLL 优化、SLF 优化等。

3.4 Dijkstra 算法

Dijkstra 算法是一种用于计算带权有向图中单源最短路径的算法。相比于 Bellman-Ford 算法而言，Dijkstra 算法虽然不能计算存在权值为负边的图，但拥有比 Bellman-Ford 算法更优秀的时间复杂度。

Dijkstra 算法采用贪心算法（Greedy Algorithm）范式进行设计，主要思想为将结点分为两个集合：已确定最短路长度的顶点以及未确定最短路长度的顶点。最初以确定最短路长度顶点集合中只有 S。接下来重复下述操作：

1. 对刚刚被加入已确定最短路长度顶点集合的结点对其所有出边进行松弛（relax）操作。

2. 从未确定最短路长度顶点集合中选取一个最短路长度最小的结点，移至第一个集合中，直到未确定最短路长度顶点集合为空则全部顶点的最短路已知，算法结束。

接下来证明 Dijkstra 算法的正确性。通过两步进行证明，先证明任何时候第一个集合中的元素dis一定不大于第二个集合中的dis，再证明第一个集合中元素的最短路已经确定。

首先证明第一步，最开始时第一个集合中只有S，必然成立第一个集合中元素的dis不大于第二个集合中的dis，在每一步中，加入第二个集合的元素一定是最大值，且从第二个集合中取出并加入第一个集合中的元素是最小值，每次进行松弛操作时所加的数字为非负数，通过归纳法及非父权值的性质可知仍然成立。

然后证明第二步，考虑每次加进来的结点到它的最短路，上一步必然为第一个集合中的元素，否则它不会是第二个元素中的最小值并且具备第一步的性质，又因为第一个集合内的元素已经全部松弛过了，所以最短路显然确定了。

其代码实现如下。

```
void dijkstra(int u)
{
    memset(dis,88,sizeof(dis));
    int start=u;
    book[start]=1;
    for(int i=1;i<=n;i++)
    {
        dis[i]=min(dis[i],map[start][i]);
    }
    for(int i=1;i<=n-1;i++)
    {
        int minn=9999999;
        for(int j=1;j<=n;j++)
            if(book[j]==0 && minn>dis[j])
            {
                minn=dis[j];
                start=j;
            }
        book[start]=1;
        for(int j=1;j<=n;j++)
            dis[j]=min(dis[j],dis[start]+map[start][j]);
    }
}
```

图 8 Dijkstra 算法暴力方法实现

Dijkstra 算法时间复杂度如果只分析集合操作，则有n次$delete-min$，m次$decrease-key$，上述代码未被优化，我们称为暴力算法，其时间复杂度为$O(n^2+m)=O(n^2)$。

Dijkstra 算法也具有多种优化方式，是对其存储的数据结构进行优化，如用堆，其时间复杂度为$O(m\log n)$;用$priority_queue$的时间复杂度为$O(m\log m)$，应注意使用$priority_queue$时无法删除某一个旧的结点，只能插入一个权值更小的编号相同的结点，这样的操作会使得堆中元素为$O(m)$；当采用$Fibonacci$堆时，复杂度可以低至$O(m\log n+m)$，这也是其算法时间复杂度优秀的原因。

本文将会介绍用堆进行优化伪代码，思路与之前叙述的 Dijkstra 算法相同，不同

点仅为用于存储的数据的数据结构不同，进而导致了其时间复杂度的差异性。

```
H = new heap();
H.insert(S, 0);
dist[S] = 0;
for (i = 1; i <= n; i++) {
  u = H.delete_min();
  for each edge(u, v) {
    if (relax(u, v)) {
      H.decrease_key(v, dist[v]);
    }
  }
}
```

图 9 Dijkstra 算法堆优化伪代码

3.5 Johnson 算法

Johnson 算法和 Floyd 算法一样，是一种能求出无负环图上任意两点间最短路径的算法，该算法在 1977 年由 Donald B. Johnson 提出。

任意两点间的最短路可以通过枚举起点，通过运行n次 Bellman-Ford 算法进行问题的解决，此时时间复杂度为$O(n^2m)$，也可以直接采用 Floyd 算法解决，此时时间复杂度为$O(n^3)$。由之前论述可知，堆优化的 Dijkstra 算法求单源最短路径的时间比 Bellman-Ford 更优，如果枚举起点，运行n次 Dijkstra 算法，就可以在$O(nm\log m)$（取决于 Dijkstra 算法的实现）的时间复杂度内解决本问题，比上述运行n次 Bellman-Ford 算法的时间复杂度更为优秀，在稀疏图上的时间复杂度也比 Floyd 算法的时间复杂度更加优秀。

但 Dijkstra 算法不能正确求解带负边权的最短路，因此需要对原图上的边进行需处理，确保所有的边的边权均为非负。一种易想到的方式为给所有边的边权同时增加相同正数x，从而让所有边的权值非负，如果新图上起点到终点的最短路经过了k条边，则将最短路减去kx即可得到实际最短路。但这样的方法是错误的。下面举一个反例论证。

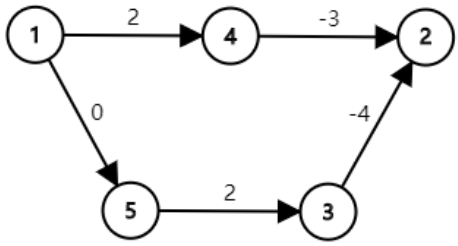

图 10 反例图 1

$1 \to 2$的最短路为$1 \to 5 \to 3 \to 2$，长度为-2。在把每条边的边权增加 5 后图变为下图。

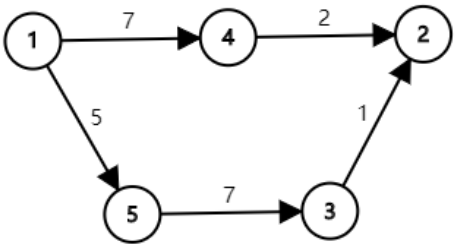

图 11 反例图 2

新图上$1 \to 2$最短路为$1 \to 4 \to 2$，已经不是实际的最短路了。

而 Johnson 算法则通过另外一种方法来给每条边重新标注边权。首先新建一个虚拟结点，设该虚拟结点为编号 0，并使其与所有顶点连通，新边赋权值为 0。然后使用 Bellman-Ford 算法求出 0 号点到其余各顶点的最短路，记作h_i，假如存在一条从u点到v点，边权为w的边，则将该边的边权重新设置为$w + h_u - h_v$。接下来以每个点

为起点，运行n次 Dijkstra 算法即可求出任意两点间的最短路。代码实现伪代码如下。

```
JOHNSON(G)
1  compute G', where V[G'] = V[G] ∪ {s} and
       E[G'] = E[G] ∪ {(s, v) : v ∈ V[G]}
2  if BELLMAN-FORD(G', w, s) = FALSE
3     then print "the input graph contains a negative-weight cycle"
4     else for each vertex v ∈ V[G']
5               do set h(v) to the value of δ(s, v)
                      computed by the Bellman-Ford algorithm
6          for each edge (u, v) ∈ E[G']
7               do ŵ(u, v) ← w(u, v) + h(u) − h(v)
8          for each vertex u ∈ V[G]
9               do run DIJKSTRA(G, ŵ, u) to compute
                      δ̂(u, v) for all v ∈ V[G]
10                for each vertex v ∈ V[G]
11                   do d_uv ← δ̂(u, v) + h(v) − h(u)
12 return D
```

图 12 Johnson 算法伪代码

开始所用的 Bellman-Ford 算法并不是时间上的瓶颈，若使用$priority_queue$实现 Dijkstra 算法，则该算法的时间复杂度是$O(nm\log m)$。

3.6 D´Esopo-Pape 算法

D´Esopo-Pape 算法与 Johnson 算法相似，是一个适用于无负环的带权图计算单源最短路的算法。

设d_i表示点s到点i的最短路长度。初始时$d_s=0$，其他都设为无限大，设p_i表示点i在最短路上的前一结点。

这个算法中需要维护三个点集，分别为M_0、M_1、M_2。M_0为已经计算出距离的点（即使不是最短距离）；M_1为正在计算距离的点；M_2为还没被计算距离的点。点集M_1采用双向队列来存储。每次从M_1中取一个点u，然后将u存入M_0。然后遍历从u触发的所有边。设另一端的顶点是点v，权值为w。如果$v\in M_2$，将v插入M_1队尾，并更新d_v: $d_v\leftarrow d_u+w$；如果$v\in M_1$，那我们就更新d_v: $d_v=\min(d_v,d_u+w)$；如果$v\in M_0$，并且d_v可以被改进到$d_v>d_u+w$，那我们就更新d_v，并将v插入到M_1的队首。同时，在每次更新数组d的时候，我们也必须更新p数组中相应的元素。

D´Esopo-Pape 算法的时间复杂度经过计算后可得出为$O(N\cdot 2^N)$。其代码实现如下图所示。

```cpp
struct Edge {
  int to, w;
};
int n;
vector<vector<Edge>> adj;
const int INF = 1e9;
void shortest_paths(int v0, vector<int>& d,
vector<int>& p) {
  d.assign(n, INF);
  d[v0] = 0;
  vector<int> m(n, 2);
  deque<int> q;
  q.push_back(v0);
  p.assign(n, -1);
  while (!q.empty()) {
    int u = q.front();
    q.pop_front();
    m[u] = 0;
    for (Edge e : adj[u]) {
      if (d[e.to] > d[u] + e.w) {
        d[e.to] = d[u] + e.w;
        p[e.to] = u;
        if (m[e.to] == 2) {
          m[e.to] = 1;
          q.push_back(e.to);
        } else if (m[e.to] == 0) {
          m[e.to] = 1;
          q.push_front(e.to);
        }
      }
    }
  }
}
```

图 13 D´Esopo-Pape 算法代码实现

4 小结

本文中介绍了有关最短路问题的算法，包括 Floyd 算法、Bellman-Ford 算法、SPFA 算法、Dijkstra 算法、Johnson 算法以及 D´Esopo-Pape 算法。下表为对这几种算法的总结与其适用范围的标注。

表 1 最短路问题各算法总结

Floyd	Johnson
每对结点之间的最短路	每对结点之间的最短路
没有负环的图	非负权图
$O(N^3)$	$O(NM\log M)$

续表 1

Bellman-Ford	SPFA
单源最短路	单源最短路

任意图（可以判定负环是否存在）	任意图（可以判定负环是否存在）
$O(MN)$	$O(kM)$

续表 1

Dijkstra	**D´Esopo-Pape**
单源最短路	单源最短路
非负权图	没有负环的图
$O(M\log M)$	$O(N\cdot 2^N)$

注：表中的 Dijkstra 算法计算复杂度时均用$priority_queue$实现

在需要计算单源最短路时，可根据时间复杂度的需要从 Bellman-Ford 算法、SPFA 算法、Dijkstra 算法以及 D´Esopo-Pape 中进行算法的选择。而当需要计算多元最短路时，可根据图中是否存在负权进行 Floyd 算法和 Johnson 算法的选择。

最短路问题算法是许多更深层算法的基础，只有掌握了这些算法才能探究更深层次的问题。

参考文献

[1] 曾庆红，杨桥艳.“最短路问题算法综述”，保山学院学报，2019,38(05),44-46 DOI:10.3969/j.issn.1674-9340.2019.05.011

[2] Dijkstra E W. A note on two problems in connexion with graphs[J]. Numerische mathematik,1959, 1(1):269-271

[3] Floyd R W. Algorithm 97:shortest path[J].Communications of the ACM, 1962, 5(6):345.

[4] Bellman R. On a routing problem[J]. Quarterly of applied mathematics, 1958, 16(1):87-90.