



HARBIN INSTITUTE OF TECHNOLOGY

集合论实验报告：最短路径

作者: 冯云龙
学号: 1160300202

2017 年 5 月 20 日

摘要

生活中的许许多多看似不同的问题在本质上却是相同的，我们对于问题的关注的也往往都是最省时，最省钱... 这个时候，通过对图论问题的研究，我们就可以对这些问题做出解答，此报告主要回答关于图论中最短路径的问题。

关键字： *Dijkstra* 最短路径

目录

第一部分	正文	2
第 1 章	实验背景	2
1.1	实验目的	2
1.2	实验方法	2
第 2 章	实验原理	2
2.1	迪杰斯特拉算法思想	2
2.2	迪杰斯特拉算法步骤	3
第 3 章	代码实现	3
3.1	设计数据结构	3
3.2	设计数据操作	3
3.3	实现迪杰斯特拉算法	3
第 4 章	实验结果	4
4.1	数据输入	4
4.2	结果输出	4
第 5 章	实验分析	5
5.1	效率分析	5
5.2	同类算法比较	5
5.3	优化策略	5
第二部分	附录	6
A	带权图实现	6
B	最短路径实现	10

第一部分 正文

第 1 章 实验背景

1.1 实验目的

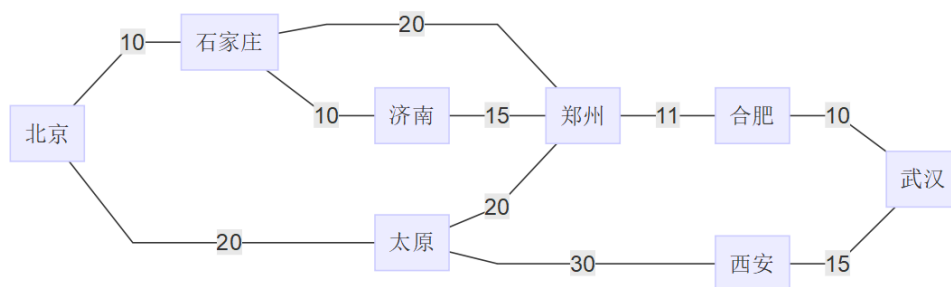
在实际生活中，我们常常会遇到关注点在于最短、最近、最省钱这些方面的问题，就比如下列问题：

- 一批货从北京到武汉的最快，或最省钱的走法。
- 在城市群中建一个仓储基地，建在什么位置可以让各个城市的送货速度都比较快。

而像这样的问题，我们都可以通过将其转化为图的问题来解决。

1.2 实验方法

诸如以上问题，我们都可以通过将其转化成图，而后使用求解图的方法解决它。例如，上述一个和距离有关的问题，我们就可以将其按如下方式转化：取图 $G(V, E, W)$ ，城市所对应的顶点集 $(V_0, V_1 \dots V_{n-1}) \in V$ ，若两个城市 V_i, V_j 邻接，距离为 w ，则有 $(V_i, V_j) \in E$ ， $W(V_i, V_j) = w$ 。



第 2 章 实验原理

2.1 迪杰斯特拉算法思想

- 设 $G = (V, E, W)$ 是一个带权有向图，把图中顶点集合 V 分成两组：
 1. 第一组为已求出最短路径的顶点集合（用 S 表示）
 2. 第二组为其余未确定最短路径的顶点集合（用 U 表示）
- 初始时 S 中只有一个源点，以后每求得一条最短路径，就将加入到集合 S 中，直到全部顶点都加入到 S 中，算法就结束了。
- 按最短路径长度的递增次序依次把第二组的顶点加入 S 中。在加入的过程中，总保持从源点 v 到 S 中各顶点的最短路径长度不大于从源点 v 到 U 中任何顶点的最短路径长度。
- 此外，每个顶点对应一个距离， S 中的顶点的距离就是从 v 到此顶点的最短路径长度， U 中的顶点的距离，是从 v 到此顶点只包括 S 中的顶点为中间顶点的当前最短路径长度。

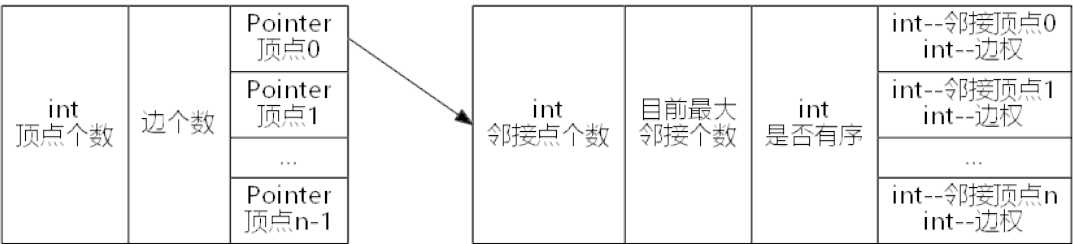
2.2 迪杰斯特拉算法步骤

- 1. 初始时, S 只包含源点, 即 $S = \{v\}$, v 到 v 的距离为 0。 U 包含除 v 外的其他顶点, 即: $U = V \setminus S$, 若 v 与 U 中顶点 u 有边, 则 $\langle u, v \rangle$ 正常有权值, 若 u 不是 v 的出边邻接点, 则 $\langle u, v \rangle$ 权值为 ∞ 。
- 2. 从 U 中选取一个距离 v 最小的顶点 k , 把 k , 加入 S 中 (该选定的距离就是 v 到 k 的最短路径长度)。
- 3. 以 k 为新考虑的中间点, 修改 U 中各顶点的距离; 若从源点 v 到顶点 u 的距离 (经过顶点 k) 比原来距离 (不经过顶点 k) 短, 则修改顶点 u 的距离值, 修改后的距离值为顶点 k 的距离加上边上的权。
- 4. 重复步骤 2 和 3 直到所有顶点都包含在 S 中。

第 3 章 代码实现

3.1 设计数据结构

参照了耶鲁大学的一位前辈的代码, 动态分配数组, 长度可以扩展, 既不浪费空间, 有不会带来性能损失。数据结构如下A:



3.2 设计数据操作

- 1. 创建一个顶点从 $0 \rightarrow n - 1$ 的带权图
- 2. 从内存中删去一个图
- 3. 添加边和权
- 4. 返回顶点个数
- 5. 返回边个数
- 6. 返回顶点的度
- 7. 判断是否邻接
- 8. 获取边的权
- 9. 提供读取顶点数据的接口

3.3 实现迪杰斯特拉算法

- 定义数据结构
- 迪杰斯特拉算法实现B

DIJKSTRA'S ALGORITHM(G, v_0)

```

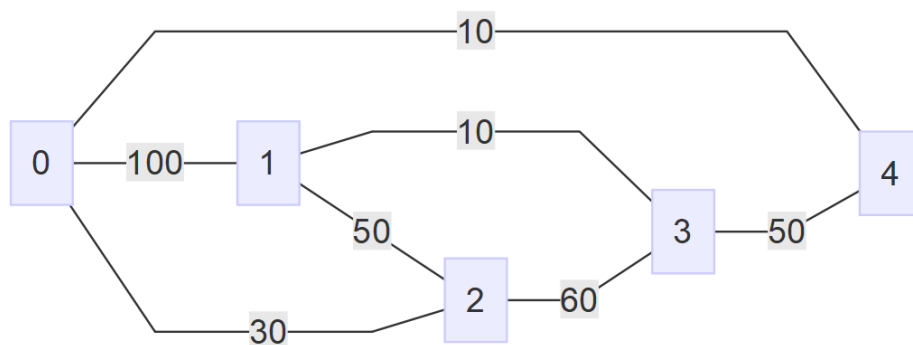
1  for  $v \in G.V$ 
2       $Dis(v) = G.W(v, v_0)$ 
3  Set  $v_0 \in S$ 
4  while  $S \neq G.V$ 
5       $k = Dis(V).V_m$ 
6      Set  $k \in S$ 
7      for  $v \notin S$ 
8          if  $Dis(k) + G.W(k, v) < Dis(v)$ 
9               $Dis(v) = Dis(k) + G.W(k, v)$ 
10              $Prev(v) = k$ 

```

第 4 章 实验结果

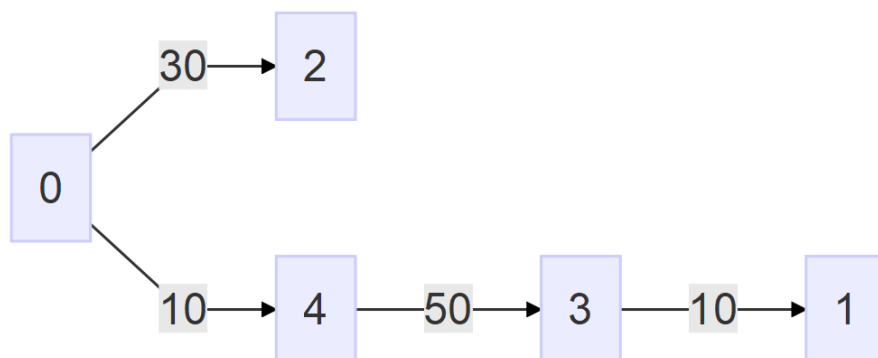
4.1 数据输入

输入如下无向带权图



4.2 结果输出

通过迪杰斯特拉算法可得到如下结果，既得出距离，又可以显示路径



第 5 章 实验分析

5.1 效率分析

迪杰斯特拉算法的时间复杂度是 $O(n^2)$, 空间复杂度则取决于数据结构, 使用矩阵则为 $O(n^2)$ 。

5.2 同类算法比较

相对于弗洛伊德算法迪杰斯特拉算法来说, 迪杰斯特拉算法不能处理边权为负值的情况。

5.3 优化策略

权值排序优化策略 将要扫描的结点按其对应弧的权值进行顺序排列, 每循环一次即可得到符合条件的结点, 大大提高了算法的执行效率。

A* 算法优化策略 采用改进的 *Dijkstra* 算法——A* 算法。A* 算法是人工智能运用在游戏中的一个重要实践, 它主要是解决路径搜索问题。A* 算法实际是一种启发式搜索。所谓启发式搜索, 就是利用一个估价函数 *judge* 评估每次决策的价值, 决定先尝试哪一种方案。这样可以极大地优化普通的广度优先搜索。从 *Dijkstra* 算法到 A* 算法是判断准则的引入, 如果这个判断条件不成立, 同样地, 只能采用 *Dijkstra* 算法。所以 A* 算法中的估价函数是至关重要。

扇形优化策略 从尽量减少最短路径分析过程中搜索的临时结点数量, 限制范围搜索和限定方向搜索考虑进行优化。那么这种有损算法是否可行呢? 我们知道, 现实生活中行进, 不会向着目的地的相反方向行进, 否则就是南辕北辙。所以, 当所研究的网络可以抽象化为平面网络的条件下, 也不必搜索全部结点, 可以在以源点到终点所在直线为轴线的扇形区域内搜索最短路径。这样, 搜索方向明显地趋向终点, 提高了搜索速度, 虽然抛弃了部分结点, 但基本上不影响搜索的成功率。

第二部分 附录

A 带权图实现

```

1  //
2  // Created by Along on 2017/5/13.
3  // https://github.com/AlongWY/Graph
4  //
5
6  #include <stdlib.h>
7  #include <assert.h>
8
9  #include "WeightGraph.h"
10
11 //基础带权图定义
12 //使用可变数组表示的临接矩阵
13
14
15 typedef struct __list {
16     int vec;           //临接顶点
17     int weight;        //权
18 } link_list;
19
20 struct w_graph {
21     int n;              //顶点个数
22     int m;              //边个数
23     struct successors {
24         int d;          //临接点个数
25         int len;        //最大临接点个数
26         char is_sorted; //
27         link_list list[1]; //临接列表
28     } *v_list[1];
29 };
30
31
32 //创建一个顶点从 0 ~ n-1 的带权图
33 WGraph w_graph_create(int n) {
34     WGraph g;
35     int i;
36
37     g = malloc(sizeof(struct w_graph) +
38               sizeof(struct successors *) * (n - 1));
39     assert(g);
40

```

```
41     g->n = n;
42     g->m = 0;
43
44     for (i = 0; i != n; i++) {
45         g->v_list[i] = malloc(sizeof(struct successors));
46         assert(g->v_list[i]);
47         g->v_list[i]->d = 0;
48         g->v_list[i]->len = 1;
49         g->v_list[i]->is_sorted = 1;
50     }
51
52     return g;
53 }
54
55 //释放内存
56 void w_graph_destroy(WGraph g) {
57     int i;
58
59     for (i = 0; i != g->n; i++) {
60         free(g->v_list[i]);
61     };
62     free(g);
63 }
64
65 //添加边和权
66 void w_graph_add_edge(WGraph g, int u, int v, int weight) {
67     assert(u >= 0);
68     assert(u < g->n);
69     assert(v >= 0);
70     assert(v < g->n);
71
72     //是否需要增长 list
73     while (g->v_list[u]->d >= g->v_list[u]->len) {
74         g->v_list[u]->len *= 2;
75         g->v_list[u] =
76             realloc(g->v_list[u], sizeof(struct successors) +
77                 sizeof(link_list) * (g->v_list[u]->len - 1));
78     }
79
80     //添加新临接点
81     g->v_list[u]->list[g->v_list[u]->d].vec = v;
82     g->v_list[u]->list[g->v_list[u]->d].weight = weight;
83
84     g->v_list[u]->d++;
85 }
```



```

86     g->v_list[u]->is_sorted = 0;
87
88     //边数+1
89     g->m++;
90 }
91
92 //返回顶点个数
93 int w_graph_vector_count(WGraph g) {
94     return g->n;
95 }
96
97 //返回边个数
98 int w_graph_edge_count(WGraph g) {
99     return g->m;
100 }
101
102 //返回顶点的度
103 int w_graph_out_degree(WGraph g, int source) {
104     assert(source >= 0);
105     assert(source < g->n);
106
107     return g->v_list[source]->d;
108 }
109
110 //是否需要进行二分搜索和排序
111 #define BSEARCH_THRESHOLD (10)
112
113 static int list_cmp(const void *a, const void *b) {
114     return ((const link_list *) a)->
115         vec - ((const link_list *) b)->vec;
116 }
117
118
119 #include <stdio.h>
120
121 //二者之间有边则返回1
122 int w_graph_has_edge(WGraph g, int source, int sink) {
123     int i;
124
125     assert(source >= 0);
126     assert(source < g->n);
127     assert(sink >= 0);
128     assert(sink < g->n);
129
130     if (w_graph_out_degree(g, source) >= BSEARCH_THRESHOLD) {

```

```

131         //确保已经被排序
132         if (!g->v_list[source]->is_sorted) {
133             qsort(g->v_list[source]->list,
134                 g->v_list[source]->d,
135                 sizeof(link_list),
136                 list_cmp);
137         }
138         //使用二分查找
139         link_list to_find;
140         to_find.vec = sink;
141         to_find.weight = 0;
142
143         return bsearch(&to_find,
144                       g->v_list[source]->list,
145                       g->v_list[source]->d,
146                       sizeof(link_list),
147                       list_cmp) != 0;
148     } else {
149         //数据量很少，直接遍历
150         int vec_degree = g->v_list[source]->d;
151         for (i = 0; i != vec_degree; i++) {
152             if (g->v_list[source]->list[i].vec == sink) {
153                 return 1;
154             }
155         }
156     }
157     return 0;
158 }
159
160 //返回权
161 int w_graph_weight_edge(WGraph g, int source, int sink) {
162     int i;
163
164     assert(source >= 0);
165     assert(source < g->n);
166     assert(sink >= 0);
167     assert(sink < g->n);
168
169     if (w_graph_out_degree(g, source) >= BSEARCH_THRESHOLD) {
170         //确保已经被排序
171         if (!g->v_list[source]->is_sorted) {
172             qsort(g->v_list[source]->list,
173                 g->v_list[source]->d,
174                 sizeof(link_list),
175                 list_cmp);

```

```

176     }
177     //使用二分查找
178     link_list to_find;
179     to_find.vec = sink;
180     to_find.weight = 0;
181     link_list *res = bsearch(&to_find,
182                             g->v_list[source]->list,
183                             g->v_list[source]->d,
184                             sizeof(link_list),
185                             list_cmp);
186     return res->weight;
187 } else {
188     //数据量很少，直接遍历
189     for (i = 0; i != g->v_list[source]->d; i++) {
190         if (g->v_list[source]->list[i].vec == sink) {
191             int res = g->v_list[source]->list[i].weight;
192             return res;
193         }
194     }
195     return INFINITY;
196 }
197 }
198
199 //提供数据 遍历接口
200 void w_graph_foreach(WGraph g, int source,
201 void (*f)(WGraph, int, int, int, void *), void *data) {
202     int i;
203
204     assert(source >= 0);
205     assert(source < g->n);
206
207     for (i = 0; i != g->v_list[source]->d; ++i) {
208         f(g, source, g->v_list[source]->list[i].vec,
209           g->v_list[source]->list[i].weight, data);
210     }
211 }

```

B 最短路径实现

```

1 //
2 // Created by Along on 2017/5/13.
3 // https://github.com/AlongWY/Graph
4 //
5

```

```

6 #include <stddef.h>
7 #include <assert.h>
8 #include <malloc.h>
9 #include <stdio.h>
10 #include "WeightGraph.h"
11 #include "Graph_tools.h"
12
13 struct min_len {
14     int n;
15     int vec;
16     struct list {
17         int dist;           //与所要求顶点的距离
18         int prev;          //前驱动点
19     } a_list[1];
20 };
21
22 //迪杰斯特拉算法
23 Min_len Dijkstra(WGraph g, int source) {
24     int i, j, *S;
25     Min_len res;
26
27     int vec_num = w_graph_vector_count(g);
28
29     assert(source >= 0);
30     assert(source < vec_num);
31
32     res = malloc(sizeof(struct min_len) +
33                 sizeof(struct list) * (vec_num - 1));
34     S = calloc((size_t) vec_num, sizeof(int));
35     assert(res);
36     assert(S);
37
38     res->n = vec_num;
39     res->vec = source;
40
41     //初始化各点
42     for (i = 0; i != vec_num; ++i) {
43         S[i] = 0;
44         if (w_graph_has_edge(g, source, i)) {
45             res->a_list[i].dist = w_graph_weight_edge(g, source, i);
46             res->a_list[i].prev = source;
47         } else {
48             res->a_list[i].prev = -1;
49             res->a_list[i].dist = INFINITY;
50         }

```

```

51     }
52
53     res->a_list[source].dist = 0;
54     res->a_list[source].prev = source;
55     S[source] = 1;
56
57     for (i = 1; i != vec_num; ++i) {
58         int min_dst = INFINITY;
59         int u = source;
60         //找出未使用过的点中 dist 最小的
61         for (j = 0; j != vec_num; ++j) {
62             if ((!S[j]) && res->a_list[j].dist < min_dst) {
63                 u = j;        //u是距离 source 最近的点
64                 min_dst = res->a_list[j].dist;
65             }
66         }
67
68         S[u] = 1;        //将u标记为已使用
69
70         for (j = 0; j != vec_num; ++j)
71             //j点未被使用且u,j之间有边
72             if ((!S[j]) && w_graph_has_edge(g, u, j)) {
73                 if (res->a_list[u].dist + w_graph_weight_edge(g, u, j)
74                     < res->a_list[j].dist) {
75                     res->a_list[j].dist = res->a_list[u].dist +
76                     w_graph_weight_edge(g, u, j);    //更新距离
77                     res->a_list[j].prev = u;           //更新路径
78                 }
79             }
80     }
81     free(S);
82     return res;
83 }
84
85 //数据遍历接口
86 void min_len_foreach(Min_len m,
87 void (*f)(Min_len, int, int, void *), void *data) {
88     int i;
89     for (i = 0; i != m->n; i++) {
90         f(m, m->a_list[i].dist, m->a_list[i].prev, data);
91     }
92 }
93
94 //顶点个数
95 int min_len_vector_count(Min_len m) {

```

```
96     return m->n;
97 }
98
99 //释放内存
100 void min_len_destroy(Min_len m) {
101     free(m);
102 }
103
104 //封装一次添加二条边
105 void w_graph_add_edge2(WGraph g, int source, int sink, int weight) {
106     w_graph_add_edge(g, source, sink, weight);
107     w_graph_add_edge(g, sink, source, weight);
108 }
109
110 //打印数据
111 static void w_graph_show_vec(WGraph g, int src, int sink,
112 int weight, void *data) {
113     printf(" %d:%d ", sink, weight);
114 }
115
116 //调用接口
117 void min_len_show(Min_len ml) {
118     int i, j;
119     for (i = 0; i != ml->n; ++i) {
120         printf("Dist:%d ", ml->a_list[i].dist);
121         for (j = i; j != ml->vec; j = ml->a_list[j].prev) {
122             printf("%d <- ", j);
123         }
124         printf("%d\n", ml->vec);
125     }
126 }
```