



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。

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 | 优化方向 | 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

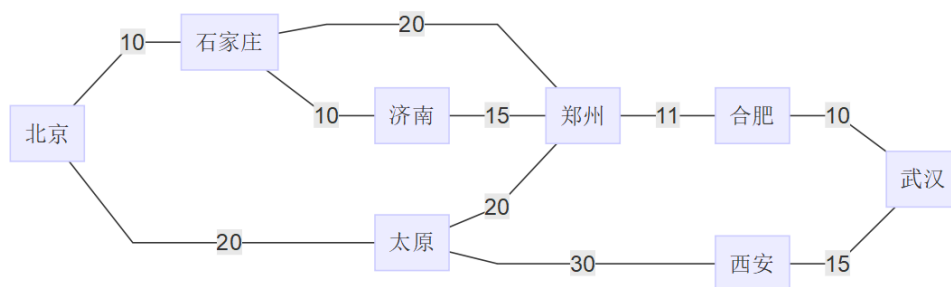
在实际生活中，我们常常会遇到关注点在于最短、最近、最省钱这些方面的问题，就比如下列问题：

- 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。

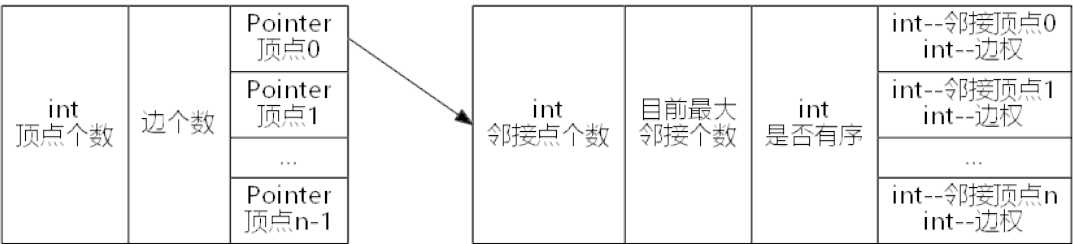
2.2 迪杰斯特拉算法步骤

- 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)

```

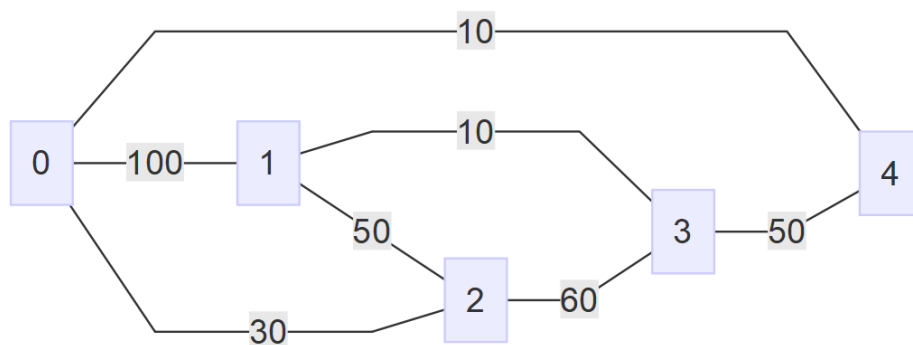
1  for  $v \in G.V$ 
2       $Dis(v) = G.W(v, v_0)$ 
3   $Set\ v_0 \in S$ 
4  while  $S \neq G.V$ 
5       $k = Dis(V).V_m$ 
6       $Set\ k \in S$ 
7      for  $v \notin S$ 
8          if  $Dis(k) + G.W(k, v) < Dis(v)$ 
9               $Dis(v) = Dis(k) + G.W(k, v)$ 
10              $Prev(v) = k$ 

```

第 4 章 实验结果

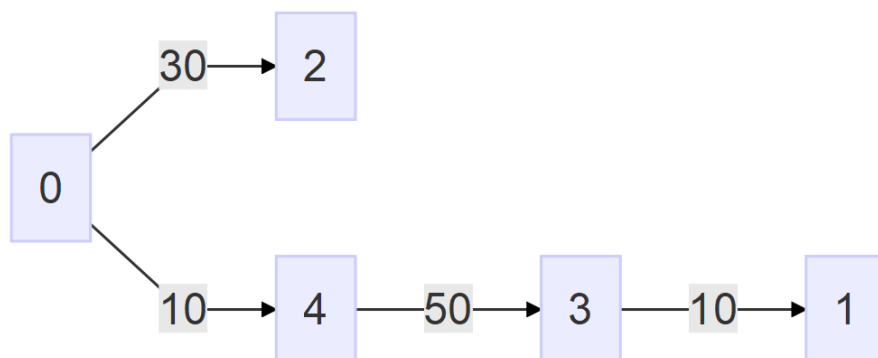
4.1 数据输入

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

```

```

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

```



```

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

```

```

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

```

```

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

```

B 最短路径实现

```

1 //
2 // Created by Along on 2017/5/13.
3 //
4
5 #include <stddef.h>
6 #include <assert.h>

```

```

7  #include <malloc.h>
8  #include <stdio.h>
9  #include "WeightGraph.h"
10 #include "Graph_tools.h"
11
12 struct min_len {
13     int n;
14     int vec;
15     struct list {
16         int dist;           //与所要求顶点的距离
17         int prev;          //前驱动点
18     } a_list[1];
19 };
20
21 //迪杰斯特拉算法
22 Min_len Dijkstra(WGraph g, int source) {
23     int i, j, *S;
24     Min_len res;
25
26     int vec_num = w_graph_vector_count(g);
27
28     assert(source >= 0);
29     assert(source < vec_num);
30
31     res = malloc(sizeof(struct min_len) +
32                 sizeof(struct list) * (vec_num - 1));
33     S = calloc((size_t) vec_num, sizeof(int));
34     assert(res);
35     assert(S);
36
37     res->n = vec_num;
38     res->vec = source;
39
40     //初始化各点
41     for (i = 0; i != vec_num; ++i) {
42         S[i] = 0;
43         if (w_graph_has_edge(g, source, i)) {
44             res->a_list[i].dist = w_graph_weight_edge(g, source, i);
45             res->a_list[i].prev = source;
46         } else {
47             res->a_list[i].prev = -1;
48             res->a_list[i].dist = INFINITY;
49         }
50     }
51

```

```

52     res->a_list[source].dist = 0;
53     res->a_list[source].prev = source;
54     S[source] = 1;
55
56     for (i = 1; i != vec_num; ++i) {
57         int min_dst = INFINITY;
58         int u = source;
59         //找出未使用过的点中 dist 最小的
60         for (j = 0; j != vec_num; ++j) {
61             if ((!S[j]) && res->a_list[j].dist < min_dst) {
62                 u = j;        //u是距离 source 最近的点
63                 min_dst = res->a_list[j].dist;
64             }
65         }
66
67         S[u] = 1;    //将u标记为已使用
68
69         for (j = 0; j != vec_num; ++j)
70             //j点未被使用且u,j之间有边
71             if ((!S[j]) && w_graph_has_edge(g, u, j)) {
72                 if (res->a_list[u].dist + w_graph_weight_edge(g, u, j)
73                     < res->a_list[j].dist) {
74                     res->a_list[j].dist = res->a_list[u].dist +
75                     w_graph_weight_edge(g, u, j);    //更新距离
76                     res->a_list[j].prev = u;          //更新路径
77                 }
78             }
79     }
80     free(S);
81     return res;
82 }
83
84 //数据遍历接口
85 void min_len_foreach(Min_len m,
86 void (*f)(Min_len, int, int, void *), void *data) {
87     int i;
88     for (i = 0; i != m->n; i++) {
89         f(m, m->a_list[i].dist, m->a_list[i].prev, data);
90     }
91 }
92
93 //顶点个数
94 int min_len_vector_count(Min_len m) {
95     return m->n;
96 }

```

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